Defining Scheme Functions

Handout written by Jerry Cain, Ben Newman, and David Hall.

Obviously Scheme wouldn't be of much use to us if all we ever did were compare strings and confirm that numbers like 222 are integers. Like all programming languages, Scheme allows us to build our own procedures and add them to the set of existing ones. Very few implementations of Scheme even distinguish between the built-in functions and the user-defined ones.

Using define

`define` is a special form used to define other functions. Typically, you equate a function symbol and a set of parameters with a parameterized expression. When you invoke the specified function, you're really evaluating the expression associated with the function name. Here's a simple function to convert a temperature from Celsius to Fahrenheit.

```
;;
;; Function: celsius->fahrenheit
;; -----------------------------
;; Simple conversion function to bring a Celsius
;; degree amount into Fahrenheit.
;;

(define (celsius->fahrenheit celsius)
   (+ (* 1.8 celsius) 32))
```

Our conversion function is a specific example of the more general form:

```
(define (procuedure-name <zero or more arguments>)
   <expression to be evaluated on procedure-name's behalf>)
```

If you want to feed the **`define`** expression right to the evaluator and then test it, you can do that:

```
#|kawa:1|# (define (celsius->fahrenheit celsius)
#|(---:2|#    (+ (* celsius 1.8) 32))
#|kawa:3|# (celsius->fahrenheit 0)
32.0
#|kawa:4|# (celsius->fahrenheit 100)
212.0
#|kawa:5|# (celsius->fahrenheit -40)
-40.0
```

More typically, you type up your code in a text file (by convention, the file extension is **`.scm`**), format and comment like good little coders, and then **`load`** the file. Scheme reads everything in the file as if typed out by hand. So, if your Scheme code sits in a file

called **"scheme-examples.scm"** and you want to play with it, you can load it using the **load** procedure.

```
jerry> kawa
#|kawa:1|# (load "scheme-examples.scm")
#|kawa:2|# (celsius->fahrenheit 100)
212.0
```

Regardless of how the new function got introduced, typing in **(celsius->fahrenheit 100)** at the command prompt is asking that **(+ (* 100 1.8) 32)** be evaluated. Functional paradigm purists don't speak in terms of return values; they speak of expression evaluations and how they help to synthesize a result. And pure Scheme programmers try to program without side effects, opting instead to think of their program as one huge, nested composition of function calls, where the primary function call evaluates to the answer of interest.

celsius->fahrenheit works to synthesize a floating point value from another one, but other functions can be defined to produce strings, or Booleans, or arbitrarily complex lists. Here's the Scheme version of a function that determines whether or not a year is a leap year:

```
;;
;; Predicate function: leap-year?
;; ------------------------------
;; Illustrates the use of the 'or, 'and, and 'not
;; special forms.  The question mark after the
;; function name isn't required, it's just customary
;; to include a question mark at the end of a
;; function that returns a true or false.
;;
;; A year is a leap year if it's divisible by 400, or
;; if it's divisible by 4 but not by 100.
;;

(define (leap-year? year)
  (or (and (zero? (remainder year 4))
           (not (zero? (remainder year 100))))
      (zero? (remainder year 400))))

#|kawa:1|# (load "scheme-examples.scm")
#|kawa:2|# (leap-year? 1968)
#t
#|kawa:3|# (leap-year? 2006)
#f
#|kawa:4|# (leap-year? 2000)
#t
#|kawa:5|# (leap-year? 2100)
#f
```

You may not be up on the set of Scheme built-ins, but you can probably intuit how **and**, **or**, **not**, **zero?**, and **remainder** work.

Recursion is big in Scheme. Actually, it's huge. That recursion is a big player in Scheme shouldn't be that surprising. After all, Scheme's primary data structure is the list, the list is inductively defined, and where there's an inductive definition there's sure to be recursion. Scheme supports iteration as well, but we're just going to stick with pure recursion. Most of the algorithms that could be implemented iteratively can just as easily be implemented recursively, and those recursive calls are generally the last expressions to be evaluated as part of evaluation. All Scheme interpreters can (and often will) replace this form of recursion (called tail recursion) with the iterative equivalent anyway.

Here're the obligatory **`factorial`** and **`fibonacci`** functions. They work well enough, albeit slowly in the case of the doubly recursive **`fibonacci`** procedure.

```
;;
;; Function: factorial
;; -------------------
;; Traditional recursive formulation of the most obvious recursive
;; function ever.  Note the use of the built-in zero?
;; to check to see if we have a base case.
;;
;; What's more impressive about this function is that it demonstrates
;; how Scheme can represent arbitrarily large integers.  Type
;; in (factorial 1000) and see what you get.  Try doing *that* with
;; C or Java.
;;

(define (factorial n)
  (if (zero? n) 1
      (* n (factorial (- n 1)))))

#|kawa:2|# (factorial 0)
1
#|kawa:3|# (factorial 4)
24
#|kawa:4|# (factorial 10)
3628800
#|kawa:5|# (factorial 85)
281710411438055027694947944226061159480056634330574206405101912752560026159
795933451040286452340924018275123200000000000000000000

;;
;; Function: fibonacci
;; -------------------
;; Traditional recursive implementation of
;; the fibonacci function.  This particular
;; implementation is pretty inefficient, since
;; it makes an exponential number of recursive
;; calls.
;;

(define (fibonacci n)
  (if (< n 2) n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))
```

```
#|kawa:2|# (fibonacci 2)
1
#|kawa:3|# (fibonacci 10)
55
#|kawa:4|# (fibonacci 25)
75025
#|kawa:5|# (fibonacci 40)
102334155
```

Don't pretend you knew what the 40th Fibonacci number was, because no one will believe you.

A more clever implementation of the second function capitalizes on the understanding that the 40th element in the classic Fibonacci sequence (starting with 0 and 1) is also the 39th element in the Fibonacci-like sequence starting out with 1 and (+ 1 0). It's also the 38th element of the sequence starting out with 1 and (+ 1 1). And so on, and so on.

The implementation of the **`fast-fibonacci`** routine takes a dynamic programming approach to building up the answer from the base cases. The **`fast-fibonacci`** implementation is really just a wrapper around the call to **`fast-fibonacci-helper`**, which takes the two base case values in addition to the index and keeps reframing the computation in terms of a sequence that starts with different base cases.

```
;;
;; Function: fast-fibonacci
;; ------------------------
;; Relies on the services of a helper function to
;; generate the nth fibonacci number much more quickly.  The
;; key observation here: the nth number is the Fibonacci
;; sequence starting out 0, 1, 1, 2, 3, 5, 8 is the (n-1)th
;; number in the Fibonacci-like sequence starting out with
;; 1, 1, 2, 3, 5, 8.  The recursion basically slides down
;; the sequence n or so times in order to compute the answer.
;; As a result, the recursion is linear instead of binary, and it
;; runs as quickly as factorial does.
;;

(define (fast-fibonacci n)
  (fast-fibonacci-helper n 0 1))

(define (fast-fibonacci-helper n base-0 base-1)
  (cond ((zero? n) base-0)
        ((zero? (- n 1)) base-1)
        (else (fast-fibonacci-helper (- n 1) base-1 (+ base-0 base-1)))))
```

```
#|kawa:2|# (fast-fibonacci 40)
102334155
#|kawa:3|# (fast-fibonacci 80)
23416728348467685
#|kawa:4|# (fast-fibonacci 200)
280571172992510140037611932413038677189525
#|kawa:5|# (fast-fibonacci 500)
139423224561697880139724382870407283950070256587697307264108962948325571622
863290691557658876222521294125
```

List Recursion

We've asserted the list to be the central aggregate data type, but we've ignored lists so far. Code that reads and otherwise manipulates a list needs to visit every single one of its elements. Schemers use what's called **car-cdr** recursion, which recursively **cdr's** down the list until there's no list left. Each **cdr** has its own **car**, and you grab the car with each call and let it contribute to the answer you're trying to build. Think of the **car** of the list as the first element, and the **cdr** as everything else.

Here's an intentionally unspectacular function, just to help illustrate the idiom.

```
;;
; Function: sum
; -------------
; Computes the sum of all of the numbers in the specified
; number list.  If the list is empty, then the sum is 0.
; Otherwise, the sum is equal to the value of the car plus
; the sum of whatever the cdr holds.
;;

(define (sum ls)
  (if (null? ls) 0
      (+ (car ls) (sum (cdr ls)))))

#|kawa:2|#  (sum '(4.5  2.7  3.2  0.7))
11.1
#|kawa:3|#  (sum '(10  11  12  13))
46
#|kawa:4|#  (sum '(1/3  1/5  1/7  1/9  1/11  1/13))
43024/45045
```

The recursion manages to walk down the list, and as the recursion unwinds, the answer of interest accumulates up through a chain of evaluation results. Here are two more routines. The first takes a list of floating point values and generates a list where all of the original numbers have been tripled. The second takes a string list and generates another, where the new list holds all of the running concatenations of the original.

```
;;
;; Function: triple-everything
;; ---------------------------
;; Takes a list of integers (identified by sequence)
;; and generates a copy of the list, except that
;; every integer in the new list has been tripled.
;;

(define (triple-everything numbers)
  (if (null? numbers) '()
      (cons (* 3 (car numbers)) (triple-everything (cdr numbers)))))

#|kawa:2|# (triple-everything '(8 33.5 4/5 5-2i))
(24 100.5 12/5 15-6i)
```

```
;;
;; Function: generate-partial-concatenations
;; -----------------------------------------
;; Takes a list of strings and generates a new list of the
;; same length, where the nth element of the new list is
;; the running concatenation of the original list's first
;; n elements.
;;
;;    It takes '("a" "b" "c") and generates ("a" "ab" "acb").
;;    It takes '("CS" "107" "L") and generates ("CS" "CS107" "CS107L").
;;
;; This particular implementation relies on a helper function,
;; just like fast-fibonacci does.  The helper procedure not
;; only tracks what portion of the list remains to be seen, but
;; the accumulation of all strings seen so far as well.
;;

(define (generate-concatenations strings)
  (generate-concatenations-using strings ""))

(define (generate-concatenations-using strings accum)
  (if (null? strings) '()
      (cons (string-append accum (car strings))
            (generate-concatenations-using (cdr strings)
                                           (string-append accum (car strings))))))

#|kawa:2|# (generate-concatenations '("a" "bb" "cccc"))
(a abb abbcccc)
#|kawa:3|# (generate-concatenations '("CS" "107" "L"))
(CS CS107 CS107L)
#|kawa:4|# (generate-concatenations '("walla" "walla" "washington"))
(walla wallawalla wallawallawashington)
#|kawa:5|# (generate-concatenations '("marcia" "" ""))
(marcia marcia marcia)
```

Using let

Notice that the implementation of **`generate-concatenations-using`** called **`string-append`** twice, each time with the same exact arguments. **`(car strings)`** and **`accum`** retain their values for the lifetime of the evaluation, so we expect each of those **`string-append`** expressions to evaluate to the same thing each time. It's not like it's an expensive evaluation, so evaluating the same expression twice doesn't slow things down all that much. But we know very well how we'd fix this if we were coding in C, C++, or Java: we'd catch the result of the first evaluation in a local variable. Local variables are common in imperative languages like C and C++, where you build up a return value over a series of steps, using local variables to keep track of increasingly larger results until you finally have your answer.

Schemers don't do that quite as much, because assignment-based programming betrays the functional paradigm, which has you try as hard as possible to program without side effects. But occasionally you need to break from a purely functional approach, because it's foolish to stand firm by your paradigm if it dictates you calling, say,

`(fibonacci 1000)` three separate times. The functional paradigm is just a guiding principle, not some Communist dogma. You work within the framework of the functional paradigm to the extent it's practical, but occasionally you go astray if it makes sense to. C++ programmers mix paradigms all the time, opting for object orientation for some parts and procedural orientation for others.

If you'd like to pre-compute the result of a sub-expression because it gets used in two or more places, then go ahead and pre-compute it. Of course, you'll need to store the result somewhere, and that's where **`let`** bindings come in.

Consider a well-intentioned implementation of **`power`**, which recognizes that any number is equal to the square of its square root. For simplicity, we'll assume the exponent is always a nonnegative integer.

```
;;
;; Function: power
;; ---------------
;; Assumes that exponent is a non-negative integer.  This
;; particular implemnentation is the realization of the following
;; inductive definition (all divisions are integer divisions)
;;
;;     n^m = 1                if m is 0
;;         = (n^(m/2))^2      if m is even
;;         = n * (n^(m/2))^2  if m is odd
;;

(define (power base exponent)
  (cond ((zero? exponent) 1)
        ((zero? (remainder exponent 2)) (* (power base (quotient exponent 2))
                                           (power base (quotient exponent 2))))
        (else (* base
                 (power base (quotient exponent 2))
                 (power base (quotient exponent 2))))))

#|kawa:2|# (power 12 2)
144
#|kawa:3|# (power 3 3)
27
#|kawa:4|# (power 2 9)
512
#|kawa:5|# (power 2 1000)
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

Provided you behave and pass in a nonnegative exponent, this works just great. The **`quotient`** procedure truncates, which is why the odd-exponent scenario includes another factor of **`base`** in the product. But otherwise this is a straightforward implementation. But there's a problem.

The **(power base (quotient exponent 2))** expression appears four times, and any nonzero exponent commands two of the four to be evaluated. For large exponents, this means the same exact expression is evaluated twice, even though both generate the same answer. Scheme programmers would typically use a **let** statement here to evaluate the recursive call just once and use it where needed.

Here's the updated code, which requires some reorganization now that **let** has arrived.

```
;;
;; Function: pwr
;; -------------
;; Functionally identical to the implementation of power, except
;; that pwr uses a let expression to bind a local symbol called root
;; to the result of the recursive call.
;;

(define (pwr base exponent)
  (if (zero? exponent) 1
      (let ((root (pwr base (quotient exponent 2))))
        (if (zero? (remainder exponent 2))
            (* root root)
            (* root root base)))))

#|kawa:2|# (pwr 4 5)
1024
#|kawa:3|# (pwr 2 101)
2535301200456458802993406410752
#|kawa:4|# (pwr 1i 4)
1
```

Notice that the else expression is actually a **let** expression, which invents a local variable called **root** and then refers to **root** inside. The general structure of a **let** expression is:

```
(let ((<name-1> <expression-1>)
      (<name-2> <expression-2>)
      ...
      (<name-k> <expression-k>))
   <expression making use of names 1 through k>)
```

Our **pwr** function relies on a single **let** binding, but had it needed to it could have used several. Typically you use **let** bindings to limit the number of expensive expression evaluations: things like **(pwr base (remainder exponent 2))** or **(fibonacci (- n 1))**, but not **(car sequence)** or **(* 2 n)**. Also, understand that the let bindings aren't guaranteed to be evaluated in any particular order, so <expression-2> can't use <name-1>.[1]

---

[1] In fact, you can rely on the following behavior instead: regardless of where a variable is bound in the sequence of **let** bindings, any other bindings which refer to that variable will use the **original** value of the variable, not the newly-bound value. If there was no original value, then the Scheme environment will choke.

Take a second to "let" the syntax sink in: the **let** operator takes a list of bindings, so there will always be two open-parentheses following the **let** symbol. If you forget, you'll almost certainly make the mistake of passing the bindings one after another, as separate arguments to **let**.

More examples

Let's write a function called **flatten**, which takes an arbitrarily complicated list of primitives and sublists and generates a list with all the same primitives, in the same order, but without all the intervening parentheses. Here's how the programmer wants **flatten** to work:

```
#|kawa:2|# (flatten '(1 (2) 3))
(1 2 3)
#|kawa:3|# (flatten '((1) (2 3 4) (5 6)))
(1 2 3 4 5 6)
#|kawa:4|# (flatten '(a (b (c d (e) f (g h))) (i j)))
(a b c d e f g h i j)
#|kawa:5|# (flatten '())
()
#|kawa:6|# (flatten '("nothing" "to" "flatten"))
(nothing to flatten)
```

Of course, the empty list is vacuously flat, so that's the base case. Otherwise, we need to flatten the **cdr** of the list (it'll contribute to the answer regardless of what the **car** is), and then figure out how to prepend the **car** to the recursively generated result. If the **car** is a primitive, then we just **cons** it onto the front of the flattened **cdr**. Otherwise, the **car** is a list that itself needs to be flattened and then prepended to the front of the recursively generated flattened **cdr**.

```
;;
;; Function: flatten
;; -----------------
;; Takes an arbitrary list and generates a another list where all atoms of the
;; original are laid down in order as top level elements.
;;
;; In order for the entire list to be flattened, the cdr of the
;; list needs to be flattened.  If the car of the entire list is a primitive
;; (number, string, character, whatever), then all we need to do is
;; cons that primitive onto the front of the recursively flattened cdr.
;; If the car is itself a list, then it also needs to be flattened.
;; The flattened cdr then gets appended to the flattened car.
;;

(define (flatten sequence)
  (cond ((null? sequence) '())
        ((list? (car sequence)) (append (flatten (car sequence))
                                        (flatten (cdr sequence))))
        (else (cons (car sequence) (flatten (cdr sequence))))))
```

Quicksort

Let's see how Scheme stacks up to other languages when it comes to sorting. Let's stick with numeric sequences and see how much work it is to sort it from low to high. Quicksort seems to be the Holy Grail of sorting algorithms, so let's work on that.

Let's write a function called **`partition`**, which distributes the specified list across two smaller lists—one containing all those elements smaller than the specified pivot, and another with everything else.

```
;;
;; Function: partition
;; -------------------
;; Takes a pivot and a list and produces a pair two lists.
;; The first of the two lists contains all of those element less than the
;; pivot, and the second contains everything else.  Notice that
;; the first list pair every produced is (() ()), and as the
;; recursion unwinds exactly one of the two lists gets a new element
;; cons'ed to the front of it.
;;

(define (partition pivot num-list)
  (if (null? num-list) '(() ())
      (let ((split-of-rest (partition pivot (cdr num-list))))
        (if (< (car num-list) pivot)
            (list (cons (car num-list) (car split-of-rest))
                  (cadr split-of-rest))
            (list (car split-of-rest) (cons (car num-list)
                  (car (cdr split-of-rest))))))))
```

The intent here is to split the list of numbers into a pair of lists of the form **`(<small-nums-list> <big-nums-list>)`**. The first pair of lists to ever be produced is **`(() ())`**, and as each recursive evaluation exits, exactly one of the pair gets a new element prepended to the front of it. If the number is smaller than the pivot, then it gets prepended to the front of the first of the pair; otherwise, it's prepended to the front of the second.

Check out the following trace to see how the answer is built up from **`(() ())`**:

```
#|kawa:2|# (partition 5 '(6 4 3 7 8 2 1 9 11))
call to partition (5 (4 3 7 8 2 1 9 11))
  call to partition (5 (3 7 8 2 1 9 11))
    call to partition (5 (7 8 2 1 9 11))
      call to partition (5 (8 2 1 9 11))
        call to partition (5 (2 1 9 11))
          call to partition (5 (1 9 11))
            call to partition (5 (9 11))
              call to partition (5 (11))
                call to partition (5 ())
                return from partition => (() ())
              return from partition => (() (11))
            return from partition => (() (9 11))
          return from partition => ((1) (9 11))
        return from partition => ((2 1) (9 11))
```

```
      return from partition => ((2 1) (8 9 11))
    return from partition => ((2 1) (7 8 9 11))
  return from partition => ((3 2 1) (7 8 9 11))
return from partition => ((4 3 2 1) (7 8 9 11))
((4 3 2 1) (6 7 8 9 11))
```

Of course, now that we can partition a sequence around a specific pivot, we can implement **`quicksort`** without much fuss:

```
;;
;; Function: quicksort
;; -------------------
;; Implements the quicksort algorithm to sort lists of numbers from
;; high to low.  If a list is of length 0 or 1, then it is trivially
;; sorted.  Otherwise, we partition to cdr of the list around the car
;; to generate two lists: those in the cdr that are smaller than the car,
;; and those in the cdr that are greater than or equal to the car.
;; We then recursively quicksort the two lists, and then splice everything
;; together in the proper order.
;;

(define (quicksort num-list)
  (if (<= (length num-list) 1) num-list
      (let ((split (partition (car num-list) (cdr num-list))))
        (append (quicksort (car split)) ;; recursively sort first half
                (list (car num-list))   ;; package pivot as a list
                (quicksort (cadr split)))))) ;; recursively sort second half

#|kawa:12|# (quicksort '(4 1 5 3 8 7 0 12))
(0 1 3 4 5 7 8 12)
#|kawa:13|# (quicksort '(9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9))
(0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9)
#|kawa:14|# (quicksort '(4.5))
(4.5)
#|kawa:15|# (quicksort '(1/2 2/3 1/3 3/4 1/4 4/5 3/5 2/5 1/5 5/6 1/6))
(1/6 1/5 1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6)
```