Wellesley College ◊ CS251 Programming Languages ◊ Spring 2000

FIRST-CLASS FUNCTIONS

Data and procedures and the values they amass,
Higher-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, a control point for a catch ---
In the Lambda Order they are all first-class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

--Abstract for the *Revised^4 Report on the Algorithmic Language Scheme,*
MIT Artificial Intelligence Lab Memo 848b, November 1991

1. Functions as First-Class Values

The key feature that sets the functional programming paradigm apart from other paradigms is its treatment of functions as first-class values. A value is said to be first-class if it can be:

1. *named* by a variable;
2. *passed* as an argument to a procedure;
3. *returned* as the result of a procedure;
4. *stored* in a data structure;
5. *created* in any context.

You can tell a lot about a programming language by what its first class values are. For example, integers are first class in almost every language. But compound structures like records and arrays do not always satisfy all four first-class properties. For example, early versions of Fortran did not allow arrays to be stored in other arrays. Early versions of Pascal allowed records and arrays to be passed to a function as a value but not to be returned from a function as a result. Modern versions of C do not allow arrays to be passed as arguments or returned as results from functions, though they do allow *pointers* to arrays to be passed in this fashion. When combined with the fact that the lifetime of a local array ends when the procedure it was declared in is exited, this leads to numerous subtle bugs that plague C programmers.

In Pascal, functions and procedures satisfy the properties 2 and 5 but not the others. C functions (or more precisely, C function pointers) satisfy properties 1 through 4, but do not satisfy property 5, since all functions must be declared at top level.

Scheme functions satisfy all five first-class properties. Unlike C functions, they can be created anywhere in a program by a `lambda` expression. This is a source of tremendous power; it is hard to overemphasize the importance of `lambda` and first-class functions. Functions that take other functions as arguments or return them as results are known as ***higher-order functions***. We will see many examples of the power of higher-order functions in this course.

As we have seen in the substitution model, a `lambda` expression is just a notation for a function. For example, the lambda expression

```
(lambda (a b) (/ (+ a b) 2))
```

is pronounced "a function of two arguments that divides the sum of the arguments by 2". Such an expression can be used in the operator position of a function call:

```
> ((lambda (a b) (/ (+ a b) 2)) 3 7)
5
```

By the naming property of first-class function, we can attach a name to the averaging function using `define`:

```
> (define average (lambda (a b) (/ (+ a b) 2)))
average

> (average 8 10)
9
```

Note that `define` does not create a function, it just names one. Rather, it is `lambda` that creates the function. In fact, `lambda` is the *only* construct capable of creating a function. This fact is unfortunately obscured by the fact that Scheme supports syntactic sugar for function definition that hides the lambda. That is, the above definition can also be written as:

```
> (define (average a b) (/ (+ a b) 2))
average
```

Even though the lambda is not explicit in the sugared form of definition, it is important to remember for the substitution model that it is still there!

The fact that functions are values implies that the operator position of a function call can be an arbitrary expression. E.g. the expression `((if (= x 0) average +) 3 7)` returns 5 if `x` evaluates to 0 and otherwise returns 10.

Functions can be used as arguments to other functions. We can use the substitution model to evaluate the following expressions:

```
subst> (define apply-to-3-and-5 (lambda (p) (p 3 5))
; The global environment now contains a binding between
; apply-to-3-and-5 and (lambda (p) (p 3 5))

subst> (apply-to-3-and-5 +)
((lambda (p) (p 3 5)) +)
(+ 3 5)
8

subst> (apply-to-3-and-5 *)
((lambda (p) (p 3 5)) *)
(* 3 5)
15

subst> (apply-to-3-and-5 average)
((lambda (p) (p 3 5)) (lambda (a b) (/ (+ a b) 2)))
((lambda (a b) (/ (+ a b) 2)) 3 5)
(/ (+ 3 5) 2)
(/ 8 2)
4

subst> (apply-to-3-and-5 (lambda (a b) a))
((lambda (p) (p 3 5)) (lambda (a b) a))
((lambda (a b) a) 3 5)
3
```

```
subst> (apply-to-3-and-5 (lambda (a b) b))
((lambda (p) (p 3 5)) (lambda (a b) b))
((lambda (a b) b) 3 5)
5
```

Functions can be returned as results from other functions. E.g., assume `expt` is an exponentiation function:

```
(define raise-to
  (lambda (power)
    (lambda (base)
      (expt base power)))

(define square (raise-to 2))
(define cube (raise-to 3))

> (square 5)
25

> (cube 5)
125
```

Note that the function resulting from a call to `raise-to` must somehow "remember" the value of power that `raise-to` was called with. This "memory" is explained by the substitution model:

```
subst> (define square (raise-to 2))
(define square ((lambda (power) (lambda (base) (expt base power))) 2))
(define square (lambda (base) (expt base 2))

subst> (square 5)
((lambda (base) (expt base 2)) 5)
(expt 5 2)
25
```

Note that we can use the “memory” of substitution to create functions like `apply-to-3-and-5` via the following function:

```
(define kons
   (lambda (x y)
     (lambda (f) (f x y))))
```

For example, `(kons 17 23)` will return the function

```
(lambda (f) (f 17 23))
```

Because `kons` creates a function that remembers two values, it is effectively a pairing operator like Scheme's `cons` (hence its name). It is even possible to write versions of `car` and `cdr` that work on this kind of pair:

```
(define kar
  (lambda (pair)
    (pair (lambda (left right) left))))

(define kdr
  (lambda (pair)
    (pair (lambda (left right) right))))
```

For example:

```
subst> (kar (kons 17 23))
((lambda (pair) (pair (lambda (left right) left)))
 ((lambda (x y) (lambda (f) (f x y)))
  17 23))
((lambda (pair) (pair (lambda (left right) left)))
 (lambda (f) (f 17 23)))
((lambda (f) (f 17 23)) (lambda (left right) left))
((lambda (left right) left) 17 23)
17
```

Since any data structure can be made out of pairs, it is not surprising that any data structure can be implemented in terms of functions. In fact, you should start thinking of functions as just another kind of data structure! This term we will see many examples of how abstract data types can be elegantly represented by functions. See section 3 for an example.

Functions can be stored in data structures, like lists:

```
> (define procs (list list + kons average))

> ((first procs) 3 5)
(3 5)

> ((second procs) 3 5)
8

> (((third procs) 3 5) (fourth procs))
4
```

Finally, functions can be created in any context. In many programming languages, such as C, functions can only be defined at “top-level”; it is not possible to declare one function inside of another function. But as seen above in the kons example, the ability to specialize a function to “remember” values in its body hinges crucially on the ability to have one `lambda` nested inside another. For then, applying the outer `lambda` can cause values to be substituted into the body of the inner `lambda`.

2. Higher-Order List Functions

First-class functions are excellent tools for abstracting over common idioms. For example, consider the following `squares` function from the Introduction to Scheme handout.

```
(define squares
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (square (car lst))
              (squares (cdr lst))))))
```

If we wanted to double each element of the list rather then square it, we would write the function `doubles`:

```
(define doubles
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (double (car lst))
              (doubles (cdr lst))))))
```

One of the commandments of computer science is that you should attempt to capture idioms within functions so that you never have to write them more than once. The idiom of applying a function to each element of a list is so common that it deserves to be captured into a function, which is traditionally called `map`:

```
(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))
```

Given `map`, it is easy to define squares and doubles:

```
(define squares (lambda (lst) (map square lst)))

(define doubles (lambda (lst) (map double lst)))
```

In a similar vein, consider the following `evens` procedure, which returns a list of all the even elements in a given list:

```
(define evens
  (lambda (lst)
    (if (null? lst)
        '()
        (if (even? (car lst))
            (cons (car lst) (evens (cdr lst)))
            (evens (cdr lst))))))

(define even? (lambda (n) (= 0 (remainder n 2))))
```

This is an instance of a more general filtering idiom, in which a predicate determines which elements of the given list should be kept in the result:

```
(define filter
  (lambda (pred lst)
    (if (null? lst)
      '()
      (if (pred (car lst))
        (cons (car lst) (filter pred (cdr lst)))
        (filter pred (cdr lst))))))
```

For example,

```
> (filter even? '(3 8 -2 0 -1 1 10))
(8 -2 0 10)

> (filter (lambda (x) (> x 0)) '(3 8 -2 0 -1 1 10))
(3 8 1 10)

> (filter (lambda (x) (<= (abs x) 3)) ( '(3 8 -2 0 -1 1 10))
(3 -2 0 -1 1)
```

`map` and `filter` are list transformers: they take a list as an input and return another list as an output. In the case of map, the resulting list has exactly the same number of elements as the input list. In the case of filter, the resulting list has a length less than or equal to the input list, and the elements are a subset of the elements of the input list and in the same relative order.

Another list transformer is `zip`, which combines two lists elementwise into a single list of duples of the form `(a b)`, where *a* is from the first list and *b* is the corresponding element from the second list.

```
(define zip
  (lambda (lst1 lst2)
    (if (or (null? lst1) (null? lst2))
        '()
        (cons (list (car lst1) (car lst2))
              (zip (cdr lst1) (cdr lst2))))))
```

For example:

```
> (zip ‘(a b c) (1 2 3))
((a 1) (b 2) (c 3))

> (zip ‘(a b) (1 2 3))
((a 1) (b 2))

> (zip ‘(a b c) (1 2))
((a 1) (b 2))
```

Note that if the two lists differ in length, the result of `zip` has the length of the shorter input list. `zip` and `map` can be combined to perform binary operations elementwise on two lists:

```
> (map (lambda (duple) (+ (first duple) (second duple)))
     (zip ‘(1 2 3) (4 5 6))
(5 7 9)

> (map (lambda (duple) (* (first duple) (second duple)))
     (zip ‘(1 2 3) (4 5 6))
(4 10 18)
```

Since this combination is so useful, we can capture it via a `map2` abstraction:

```
(define map2
  (lambda (f lst1 lst2)
    (map (lambda (duple) (f (first duple) (second duple)))
         (zip lst1 lst2))))

> (map2 + ‘(1 2 3) (4 5 6))
(5 7 9)

> (map2 * ‘(1 2 3) (4 5 6))
(4 10 18)
```

In addition to transforming lists, there are useful abstractions for producing and consuming lists. A handy abstraction for list generation is the following generate function. It generates a sequence of values starting with an initial `seed` value, and uses the `next` function to generate the next value in the sequence from the current one. Generation continues until the `done?` predicate is satisfied. At that point, all the elements in the sequence (except for the one satisfying the `done?` predicate) are returned in a list.

```
(define generate
  (lambda (seed next done?)
    (if (done? seed)
      '()
      (cons seed (generate (next seed) next done?)))))
```

Here are some sample uses of `generate`:

```
> (define from-to
    (lambda (lo hi)
      (generate lo
                (lambda (x) (+ x 1))
                (lambda (x) (> x hi)))))

> (from-to 3 5)
(3 4 5 6 7)

> (define halves
    (lambda (n)
      (generate n
                (lambda (x) (quotient x 2))
                (lambda (x) (= x 0)))))

> (halves 26)
(26 13 6 3 1)

> (define tails
    (lambda (lst)
      (generate lst cdr null?)))

> (tails '(a b c d))
((a b c d) (b c d) (c d) (d))
```

A common way to consume a list is to recursively accumulate a value from back to front starting at the base case and combining each element with the result of processing the rest of the elements. This pattern is captured by the `foldr` function:

```
(define foldr
  (lambda (binop init lst)
    (if (null? lst)
      init
      (binop (car lst) (foldr binop init (cdr lst))))))
```

For example:

```
> (foldr + 0 '(3 1 2 5))
11

> (foldr * 1 '(3 1 2 5))
30

> (define minlist
  ; Assume lst is non-empty
    (lambda (lst) (foldr min (car lst) lst)))

> (minlist '(3 1 2 5))
1

> (foldr (lambda (x ans) (and (> x 0) ans)) #t '(3 1 2 5))
#t

> (foldr (lambda (x ans) (and (= 1 (remainder x 2)) ans)) #t '(3 1 2 5))
#f
```

```
> (foldr (lambda (x ans) (or (even? x) ans) #f '(3 1 2 5))
#t

> (foldr (lambda (x ans) (or (< x 0) ans)) #f '(3 1 2 5))
#f
```

The sorts of boolean combinations performed in the last four examples are better captured by the following forall? and exists? functions. Due to the use of the "short-circuit" boolean and and or special forms, these will not look at the entire list in cases where the answer can be determined earlier. This stands in contrast to foldr, which will always look at every element of the list:

```
(define forall?
  (lambda (pred lst)
    (if (null? lst)
        #t
        (and (pred (car lst))
             (forall? pred (cdr lst))))))

(define exists?
  (lambda (pred lst)
    (if (null? lst)
        #f
        (or (pred (car lst))
            (exists? pred (cdr lst))))))

> (forall? (lambda (x) (> x 0)) '(3 1 2 5))
#t

> (forall? (lambda (x) (= 1 (remainder x 2))) '(3 1 2 5))
#f

> (exists? even '(3 1 2 5))
#t

> (exists? (lambda (x) (< x 0)) '(3 1 2 5))
#f
```

In the case of `exists?`, we sometimes want to know the first element in the list that satisfies the predicate. For this, we have the `some` function, which returns the first element in the list satisfying the predicate, or #f is there is no such element:

```
 (define some
  (lambda (pred lst)
    (if (null? lst)
        #f
        (if (pred (car lst))
            (car lst)
            (some pred (cdr lst))))))

> (some even '(3 1 2 5))
2

> (some (lambda (x) (< x 0)) '(3 1 5 2))
#f
```

Finally, there are cases where we want to accumulate the values in a list from left to right rather than from right to left. This is accomplished by `foldl`:

```
(define foldl
  (lambda (op init lst)
    (if (null? lst)
      init
      (foldl op (op (car lst) init) (cdr lst)))))

> (foldr cons '() '(3 1 2 5))
(3 1 2 5)

> (foldl cons '() '(3 1 2 5))
(5 2 1 3)

> (foldr list '() '(3 1 2 5))
(3 (1 (2 (5 ()))))

> (foldl cons '() '(3 1 2 5))
(5 (2 (1 (3 ()))))

> (define digits->int
    (lambda (digits)
       (foldl (lambda (x ans) (+ x (* 10 ans)))
              0
              digits)))

> (digits->int '(3 1 2 5))
3125
```

3. Functional Data Structures

As seen above, functions have "memory" in the sense that variables appearing within their bodies that are bound by enclosing `lambda`s can be replaced (according to the substitution model) by values. The fact that functions have memory and are first-class means that they are a suitable representation for implementing data structures.

To illustrate how functions can be used to implement data structures, we will study a simple Abstract Data Type (ADT) called an **environment**. An environment is essentially an immutable table that is a collection of bindings between keys and values. It turns out that environments are very useful for modeling the handling of names in programming language implementations, so we will make heavy use of environments throughout the rest of this course.

Here is a contract for a simple form of environments:

`(env-empty)`
: Returns an empty environment, i.e., an environment with no bindings. Looking up a name in an empty environment returns `#f`.

`(env-bind` *`name value env`*`)`
: Returns a new environment containing a binding of *`name`* to *`value`* in addition the bindings of *`env`*. Any binding involving *`name`* in *`env`* is superseded (or said to be **shadowed**) by new binding in the resulting environment. Environments are immutable, so `env-bind` is non-destructive. That is, `env-bind` returns a brand new environment and does not change *`env`*.

`(env-lookup` *`key env`*`)`
: Returns the value that is bound to *`key`* in the environment *`env`*, or `#f` if there is no binding keyed by *`key`*.

The fact that `#f` is used to indicate that a key is unbound in an environment means that these simple environments do not allow bindings where names are bound to the value `#f`. This can be fixed by extending the contract. Also, there are several more handy environment operations we will introduce later in the semester, but these three will suffice for now.

Here are some simple examples of the above contract in action:

```
>  (define e0 (env-empty))
e0

>  (define e1 (env-bind 'a 1 e0))
e1

> (define e2 (env-bind 'b 2 e1))
e2

> (define e3 (env-bind 'a 3 e2))
e3

>  (env-lookup 'a e3)
3

> (env-lookup 'a e2)
1

> (env-lookup 'a e1)
1
```

```
>  (env-lookup 'a e0)
#f

>  (env-lookup 'b e3)
2

> (env-lookup 'b e2)
2

> (env-lookup 'b e1)
#f

> (env-lookup 'b e0)
#f
```

There are many different ways to implement the above environment ADT. For example, below is a simple implementation where an environment is represented as a list of bindings, where each binding is a list of a key and a value. `env-bind` prepends a new binding to the front of the list, while `env-lookup` returns the first value associated with a given key in a linear search of the bindings.

```
(define env-empty (lambda () '()))

(define env-bind
  (lambda (key value env)
    (cons (list key value) env)))

(define env-lookup
  (lambda (key env)
    (let ((binding (some (lambda (bndng) (eqv? key (first bndng)))
                         env)))
      (if binding
          (second binding)
          #f))))
```

To illustrate that functions are data structures, we will also consider an alternative implementation in which environments are represented as lookup functions. A lookup function is a function that takes a key to be looked up, and returns the value bound to that key (or `#f` if there is no binding for the key). Here is an implementation of the environment ADT in terms of lookup functions:

```
(define env-empty
  (lambda ()
    ; The empty environment is a lookup function that always returns #f
    (lambda (key) #f)))

(define env-bind
  (lambda (name value env)
    ; Return a new lookup function that returns value for key
    ; and otherwise behaves like env.
    (lambda (key)
      (if (eqv? key name)
          value
          (env-lookup key env)))))

(define env-lookup
  (lambda (key env)
 ; Since env is a lookup function, it can just be applied to key!
```

On first encounter, many people find this implementation unsettling. What does it mean for an environment to be a function? The substitution model can readily answer this question. Below, we show how the substitution model explains several of the environment examples from above. To make the steps of the substitution model more readable, we will often use the name of a definition to stand for the value defined (even though technically such a name should be replaced by its value).

```
> (define e0 (env-empty))
; e0 is bound to (lambda (key1) #f)

> (define e1 (extend-env 'a 1 e0))
; e1 is bound to (lambda (key)
;                  (if (eqv? key 'a)
;                      1
;                      (env-lookup key e0)))

> (define e2 (extend-env 'b 2 e1))
; e2 is bound to (lambda (key)
;                  (if (eqv? key 'b)
;                      2
;                      (env-lookup key e1)))

> (define e3 (extend-env 'a 3 e1))
; e3 is bound to (lambda (key)
;                  (if (eqv? key 'a)
;                      3
;                      (env-lookup key e2)))

> (env-lookup 'a e0)
((lambda (key env) (env key)) 'a e0)
(e0 'a)
((lambda (key) #f) 'a)
#f

> (env-lookup 'a e1)
((lambda (key env) (env key)) 'a e1)
(e1 'a)
((lambda (key) (if (eqv? key 'a) 1 (env-lookup key e0))) 'a)
(if (eqv? 'a 'a) 1 (env-lookup 'a e0))
(if #t 1 (env-lookup 'a e0))
1

> (env-lookup 'a e2)
((lambda (key env) (env key)) 'a e2)
(e2 'a)
((lambda (key) (if (eqv? key 'b) 2 (env-lookup key e1))) 'a)
(if (eqv? 'a 'b) 2 (env-lookup 'a e1))
(if #f 1 (env-lookup 'a e1))
 (env-lookup 'a e1)
((lambda (key env) (env key)) 'a e1)
(e1 'a)
((lambda (key) (if (eqv? key 'a) 1 (env-lookup key e0))) 'a)
(if (eqv? 'a 'a) 1 (env-lookup 'a e0))
(if #t 1 (env-lookup 'a e0))
1

> (env-lookup 'a e3)
((lambda (key env) (env key)) 'a e3)
(e3 'a)
((lambda (key) (if (eqv? key 'a) 3 (env-lookup key e2))) 'a)
```

```
(if (eqv? ‘a ‘a) 3 (env-lookup ‘a e2))
(if #t 3 (env-lookup ‘a e2))
3

> (env-lookup ‘b e0)
((lambda (key env) (env key)) ‘b e0)
(e0 ‘b)
((lambda (key) #f) ‘b)
#f

> (env-lookup ‘b e1)
((lambda (key env) (env key)) ‘b e1)
(e1 ‘b)
((lambda (key) (if (eqv? key ‘a) 1 (env-lookup key e0))) ‘b)
(if (eqv? ‘b ‘a) 1 (env-lookup ‘b e0))
(if #f 1 (env-lookup ‘b e0))
(env-lookup ‘b e0)
((lambda (key env) (env key)) ‘b e0)
(e0 ‘b)
((lambda (key) #f) ‘b)
#f

> (env-lookup ‘b e2)
((lambda (key env) (env key)) ‘b e2)
(e2 ‘b)
((lambda (key) (if (eqv? key ‘b) 2 (env-lookup key e1))) ‘b)
(if (eqv? ‘b ‘b) 2 (env-lookup ‘b e1))
(if #t 2 (env-lookup ‘b e1))
2

> (env-lookup ‘b e3)
((lambda (key env) (env key)) ‘b e3)
(e3 ‘b)
((lambda (key) (if (eqv? key ‘a) 3 (env-lookup key e2))) ‘b)
(if (eqv? ‘b ‘a) 3 (env-lookup ‘b e2))
(if #f 3 (env-lookup ‘b e2))
(env-lookup ‘b e2)
((lambda (key env) (env key)) ‘b e2)
(e2 ‘b)
((lambda (key) (if (eqv? key ‘b) 2 (env-lookup key e1))) ‘b)
(if (eqv? ‘b ‘b) 2 (env-lookup ‘b e1))
(if #t 2 (env-lookup ‘b e1))
2
```

In the above examples, the result of extending environment *E* with a binding between key *K* and value *V* is represented as the specialized lookup function

```
(lambda (key) (if (eqv? key K) V (env-lookup key E)))
```

An environment is essentially a sequence of such lookup functions that are tried in turn. In the above example:

```
E3 = (lambda (key) (if (eqv? key ‘a) 3 (env-lookup key E2)))
E2 = (lambda (key) (if (eqv? key ‘b) 2 (env-lookup key E1)))
E1 = (lambda (key) (if (eqv? key ‘a) 1 (env-lookup key E0)))
E0 = (lambda (key) #f)
```

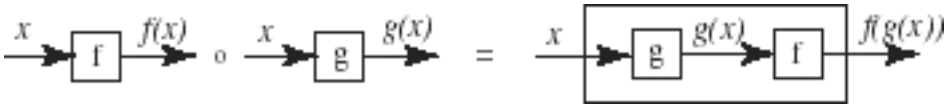
This collection of lookup functions encodes a linear search over the sequence of bindings in the environment. Rather than separating the data (such as a list of bindings) and the linear search function, the functional representation combines these.

4. Function Composition

Just as there are standard ways of combining two integers to yield another integer (e.g., `+`, `*`) and standard ways of combining two booleans to yield a boolean (e.g., `and`, `or`), there are standard ways of combining two functions to yield a function. Perhaps the most important of these is composition. In mathematics, if *f* and *g* are two functions, then the composition of *f* and *g*, written $f \circ g$, is defined as follows:

$$(f \circ g)(x) = f(g(x))$$

If we depict functions as boxes that take their inputs from their left and produce their outputs to the right, composition would be depicted as follows:



Note that the left-to-right nature of the graphical depiction of the function boxes requires inverting the order of the function boxes when they are composed. In contrast, the right-to-left nature of the textual notation requires no inversion.

Composition is straightforward to define in Scheme:

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

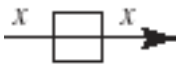
For example:

```
(define 1+ (lambda (x) (+ x 1)))

(define 2* (lambda (x) (* 2 x)))

> ((compose 1+ 2*) 10)
21

> ((compose 2* 1+) 10)
22
```

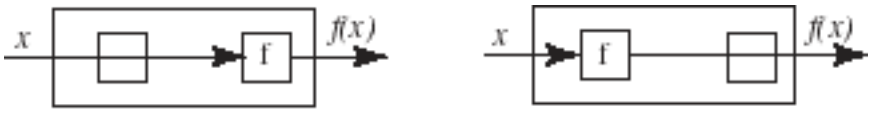
Just as addition, multiplication, conjunction, and disjunction all have identity values (respectively, 0, 1, #t, #f), so too does composition have an identity value --- the identity function:

```
(define identity (lambda (x) x))
```

Graphically, the identity function is a function box that passes its argument unaltered:



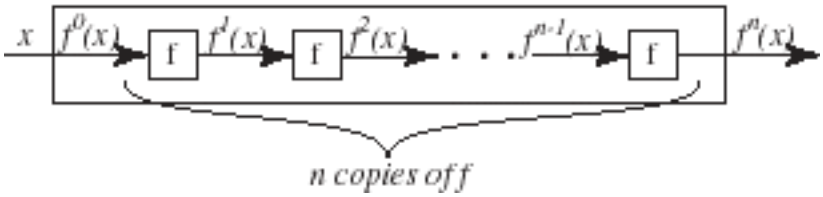
You should convince yourself that `(compose f identity)` and `(compose identity f)` are functions that are behaviorally indistinguishable from `f`. This is easy to see from the graphical representation:



It is common to compose functions with themselves. For example:

```
(define twice (lambda (f) (compose f f)))
; twice behaves like (lambda (f) (lambda (x) (f (f x))))

> ((twice 1+) 10)
; This is the same as (1+ (1+ 10))
12

((twice 2*) 10)
; This is the same as (2* (2* 10))
40

(define thrice (lambda (f) (compose f (twice f))))
; thrice behaves like (lambda (f) (lambda (x) (f (f (f x)))))

> ((thrice 1+) 10)
; This is the same as (1+ (1+ (1+ 10)))
13

> ((thrice 2*) 10)
; This is the same as (2* (2* (2* 10)))
80

> ((twice (twice 1+)) 0)
; This is the same as ((twice 1+) ((twice 1+) 10)),
4

> ((twice (thrice 1+)) 0)
; This is the same as ((thrice 1+) ((thrice 1+) 10)),
6

> ((thrice (twice 1+)) 0)
; This is the same as ((twice 1+) ((twice 1+) ((twice 1+) 10))),
6

> ((thrice (thrice 1+)) 0)
; This is the same as ((thrice 1+) ((thrice 1+) ((thrice 1+) 10))),
9

(((twice twice) 1+) 0)
; This is the same as ((twice (twice 1+)) 0)
4

> (((twice thrice) 1+) 0)
; This is the same as ((thrice (thrice 1+)) 0)
9

> (((thrice twice) 1+) 0)
; This is the same as ((twice (twice (twice 1+))) 0)
8

> (((thrice thrice) 1+) 0)
; This is the same as ((thrice (thrice (thrice 1+))) 0)
27
```

More generally, the *n*-fold composition of a function *f*, written f^n, is the result of composing *n* copies of *f*. (f^0, the zero-fold composition of *f* , is just the identity function.) Here is a graphical depiction of f^n :



In Scheme, *n*-fold composition can be expressed as the following `repeated` function:

```
(define repeated
  (lambda (f n)
    (if (= n 0)
        identity
        (compose f (repeated f (- n 1))))))
```

For example:

```
> ((repeated 1+ 5) 0)
5

> ((repeated 2* 3) 1)
8

> ((repeated square 3) 2)
256

> ((repeated square 0) 17)
17
```

Note that the `twice` and `thrice` functions from above can be defined in terms of `repeated`:

```
(define twice (lambda (f) (repeated f 2)))

(define thrice (lambda (f) (repeated f 3)))
```

In fact, we can use repeated to define *n*-fold composition functions for any *n*:

```
(define n-fold (lambda (n) (lambda (f) (repeated f n))))
```

For instance, we can define twice and thrice yet again as:

```
(define twice (n-fold 2))

(define thrice (n-fold 3))
```

The celebrated logician Alonzo Church (who invented the lambda calculus, a formal mathemtical system upon which functional programming is based) observed that *n*-fold composition functions can be viewed as numerals in the sense that it is possible to perform arithmetic on them (e.g. addition, multiplication, exponentiation, etc.). Such numerals are called **Church numerals** in his honor. We shall use `int->church` and `church->int` to convert between Scheme integers and church numerals:

```
(define int->church
  (lambda (int)
    (lambda (f) (repeated f int))))
```

```
(define church->int
  (lambda (church-numeral)
    ((church-numeral 1+) 0)))
```

`int->church` is just a synonym for `n-fold`. `church->int` finds the integer corresponding to an *n*-fold composition function by incrementing *n* times starting at 0.
For example:

```
(define two (int->church 2)) ; a synonym for twice

(define three (int->church 3)) ; a synonym for thrice

> (church->int two)
2

> (church->int three)
3
```

As an example of arithmetic on Church numerals, consider the successor function `succ` that adds one to a Church numeral. Since the Church numeral corresponding to *n* is the *n*-fold composition function, applying `succ` to such a church numeral should yield an *(n+1)*-fold composition function. Here's one definition:

```
(define succ
  (lambda (c)
    (lambda (f) (compose f (c f)))))

> (church->int (succ two))
3

> (church->int (succ three))
4

> (((succ three) 2*) 1)
16
```

It is also possible to define functions `plus`, `times`, and `raise` that perform addition, multiplication and exponentiation on Church numerals. Example calls of these functions are shown below, but their definition is left as an exercise. (*Hints*: (1) carefully study the `twice`/`thrice` examples from above; (2) think in terms of the function box notation.)

```
> (church->int (plus two three))
5

> (church->int (plus three two))
5

> (church->int (times three two))
6

> (church->int (times three two))
6

> (church->int (raise two three))
8

> (church->int (raise three two))
9
```