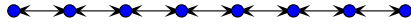


## Combining Functions

**Key idea**

building functions with functions (as opposed to calling functions from functions)

**Combination of functions**

Most procedural languages allow for the combination of data. For example, Pascal has records, C has structs, and C++ has classes. Since functions are first class objects in Scheme, they also can be combined much like data. Consider the function  $f(x) = x^3 - 2x^2 + 1$ . We can easily represent that function in Scheme.

```
(define (f x)
  (+ (* 1 x x x) (* -2 x x) 1)
)
```

Note the new notation for `+` and `*`. We normally think of these operators as binary, but in Scheme they are multi-ary or *variadic*. Now suppose we wish to calculate the derivative of this function at a point. Numerically, we can shift right a tiny amount and evaluate the function at this new point as well as the original point. Then the value of the derivative at the original point is roughly equal to  $\frac{f(x+\Delta)-f(x)}{\Delta}$ , where  $\Delta$  is the amount of the shift. Using this equation, we can naturally define a Scheme function:

```
(define (derivative f point delta)
  (/
    (- (f (+ point delta)) (f point))
    delta
  )
)
```

We would call our general purpose function thusly:

```
(derivative f 5 .00001)
```

However, our natural inclinations are wrong in this case because of our procedural upbringing. The proper approach is to create a new function out of old functions. Consider a rewrite which returns the function that computes the derivative:

```
(define (deriv f delta)
  (lambda (x) (/ (- (f (+ x delta)) (f x)) delta))
)
```

Now we can bind this new function to a name:

```
(define fprime (deriv f .00001))
```

and use it:

```
(fprime 6)
```

to compute the slope of  $f$  at  $x = 6$ .

The function *fprime* is a function which is built from a simpler function. The analogy in C++ and Java (using data) is a more complex class has been built from a simpler class. The first attempt, *derivative*, uses the passed function as a client (analogous to composition in C++ and Java) while the second version, *deriv*, returns a function that is literally built on the passed function (analogous to inheritance in C++ and Java). Now consider writing a function that finds the second derivative. Using a procedural style, we are forced to write

```
(define (derivative2 f point delta)
  (/
    (-
      (derivative f (+ point delta) delta)
      (derivative f point delta)
    )
    delta
  )
)
```

Note how similar the body of *derivative2* is to *derivative*. Using a functional approach, we do not need to write *derivative2* at all; *deriv* suffices:

```
(define fp (deriv f .00001))  
(define fpp (deriv fp .00001))
```

Scheme allows us to treat the derivative of a function as it really is, a function. In the procedural example, we treat the derivative as a single point.