Contents

1 Building Abstractions with Procedures

1.1 The elements of programming

We can combine simple ideas into a complex one using this:

- Primitive expressions: Simplest entities the language is concerned with
- Means of combination: Compound elements built from simpler ones
- Means of abstraction: How compound elements can be named and manipulated as units

1.1.1 The substitution model

The interpreter evaluates the element of the combination and applies the procedure to the arguments. We can do this by two means. Assuming the following procedures.

```
(define (square a) (* a a))
(define (sum-of-squares a b) (+ (square a) (square b)))
(define (f a) (sum-of-squares (+ a 1) (* a 2)))
```

1.1.2 Applicative order

Evaluate the arguments then apply, the method the interpreter actually uses.

```
;; When we evaluate
(f 5)
;; The reductions are as follows
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

1.1.3 Normal order

Fully expand then reduce, an alternative method.

```
;; When we evaluate
(f 5)
;; We expand everything
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
;; Then reduce
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

1.2 Procedures and the processes they generate

1.2.1 Recursive processes

Consider:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (n-1)))))
```

If we apply the substitution model we end up with:

```
(factorial 4)
(* 4 (factorial 3))
```

```
(* 4 (* 3 (factorial 2)))
(* 4 (* 3 (* 2 (factorial 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
```

We can observe the process expands and reduces as it's evaluating the expressions, the state is held within the chain of deferred operations, this is a *recursive process*, since the number of expansions and reductions grows linearly with n, this is called a *linear recursive process*.

1.2.2 Iterative processes

Now consider the following:

```
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count))
```

It grows like so:

```
(factorial 4)
(fact-iter 1 1 4)
(fact-iter 1 2 4)
(fact-iter 2 3 4)
(fact-iter 6 4 4)
(fact-iter 24 5 4)
24
```

This does not grow and shrink, all the state is contained within the arguments to the procedures, we could stop and resume the chain at any point in time if we count with the right arguments to pass to the parameters.

1.2.3 Tree Recursion

Consider:

```
(define (fib n)
  (cond ((=n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

It evolves like this

```
                              (fib 4)
                 (+ (fib 3)                (fib 2))
         (+ (+ (fib 2) (fib 1))     (+ (fib 1) (fib 0)))
(+ (+ (+ (fib 1) (fib 0)) (fib 1)) (+ (fib 1) (fib 0)))
(+ (+ (+    1       0   )    1   ) (+    1       0))
```

The process evolves into a tree of calls to different procedures, hence it's name.

1.2.4 Orders of Growth

$R(n)$ has an order of growth $R(n) = \Theta(f(n))$ if there are positive constants k_1 and k_2 independent of n such that:

$$k_1 f(n) \leq R(n) \leq k_2 f(n) \tag{1}$$

The order of growth only provides a crude description of the behavior of the process, a process with n^2 steps and a process with $1000n^2$ steps will all have $theta(n^2)$ order of growth.

1.3 Formulating abstractions with higher-order procedures

1.3.1 Procedures as arguments

Procedures that manipulate procedures are called higher-order procedures, they are useful when the same programming pattern is used with a number of different procedures. For example, the following can abstract the pattern of "summing" numbers, and we can use it to create new procedures that sum cubes, integers, etc.

```
(define (identity (a) a))
(define (square a) (* a a))
(define (cube a) (* a a a))
(define (1+ a) (+ a 1))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)))
(define (sum-cubes a b)
  (sum cube a 1+ b))
(define (sum-squares a b)
  (sum square a 1+ b))
(define (sum-ints a b)
  (sum identity a 1+ b))
```

1.3.2 Procedures using lambda

For trivial procedures, it's often more convenient to directly specify them without any names, rather than defining them. This is what lambdas are for, really useful when passing them as arguments to higher-order procedures.

```
(define (sum-cubes a b)
  (sum (lambda (x) (* x x x)) a 1+ b))
(define (sum-squares a b)
  (sum (lambda (x) (* x x) a 1+ b))
(define (sum-ints a b)
  (sum (lambda (x) x) a 1+ b))
```

1.3.3 Using let to create local variables

Imagine we wanted to express:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

Which could be also expressed as:

$$a = 1 + xy$$

$$b = 1 - y$$

$$f(x,y) = xa^2 + yb + ab$$

We could do:

```
(define (f x y)
  (define (f-helper a b)
```

```
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

Which binds a and b to the computed values before applying the rest of the procedure, but it's somewhat inconvenient, for this, we can use the special form called *let*.

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        ( b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

1.3.4 Finding roots of equations by the half-interfal method

The half-interval method is a technique for finding the roots of an equation $f(x) = 0$, where f is a continuous function and we count with points a and b such that $f(a) < 0 < f(b)$, we do this by averaging a and b many times, reducing the interval we're searching on each time. (We're basically binary searching).

```
;; Helper for the tolerance
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
;; Actual search
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
;; Checks if they're actually of opposite signs
;; and picks on which value passed is the negative
```

```
;; one and the positive one in order to pass
;; them correctly to search
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Valeus are not of opposite sign" a b)))))
;; We can then use it to approximate pi
;; (pi is the point where sin(x) = 0)
(half-interval-method sin 2.0 4.0) ;; 3.14111328125
```

1.3.5 Finding fixed points of functions

A number x is the *fixed point* of a function if $f(x) = x$, for some functions we can find this with an initial guess and applying f repeatedly until the value does not change very much (a tolerance)

$$f(x), f(f(x)), f(f(f(x))), ...,$$

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
;; Trying it on cos
(fixed-point cos 1.0)
;; We can even define a sqrt function in terms of this
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

1.3.6 Procedures as returned values

The average damping used in *sqrt* is useful in general, we can abstract that by returning it as a procedure.

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
;; We can redefine sqrt in term of this,
;; now the idea that we're really using the average damp
;; is explicit.
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
;; We can reuse ideas easily
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

We could for example approximate the derivative of a function $g(x)$, the derivative is given (in the limit of a small dx) by:

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

Thus, we can convert this function into the following procedure:

```
(define dx 0.00001)
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
;; We can derive a function
(define (cube x) (* x x x))
;; Dg(x) = 3(x^2)
;; Dg(5) = 3(5^2) = 75
((deriv cube) t) ;; 75.0001499...
```

We can use this to represent Newton's method, which is a way of finding the roots of an equation. Newton's method states that if $g(x)$ is a differentiable function, a solution to the equation $g(x) = 0$ is a fixed point of the function $f(x)$ where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

Thus, we can express this as

```
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

Then, we can redefine *sqrt* once again, since what we did before was just a special case of Newton's method.

```
(define (sqrt x)
  (newtons-method
   (lambda (y) (- (square y) x)) 1.0))
```

1.3.7 Abstractions and first-class procedures

We've seen how higher-order procedures allow us to represent abstractions explicity as elements, so that they can be haldled just like any other computational elements. In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the "rights and privileges" of first-class elements are:

- They may be named by variables
- They may be passed as arguments to procedures
- They may be returned as the results of procedures
- They may be included in data structures

Lisp awards procedures full first-class status, allowing for the expressiveness described before.