Formulating Abstractions with Higher Order Procedures (part 2)

Abelson & Sussman & Sussman sections:

1.3.2-4

Lecture contents

- In this lecture we will look at:
- Formulating abstractions with higher-order procedures(A&S 1.3)
  - (procedures as arguments: previous lecture)
  - constructing procedures using lambda
  - procedures as general methods
  - procedures as returned values

Constructing procedures using `lambda`

- So far, we've used `define` when we want a new procedure
  - Cumbersome if we use the procedure only as an argument
- But procedures are values
  - We can write expressions that denote procedures
  - Scheme provides the special form `lambda`
  - Notation from the λ-calculus
- An example and how to read it
  - `(lambda (x) (+ x 4))`
  - `the procedure of an argument x that adds x and 4`
  - equivalent to `λx.x+4`
- General form is
  - `(lambda (<formal-parameters>) <body>)`

Procedure definition

- Procedure definition actually uses `lambda`

```
(define (plus4 x) (+ x 4))
```

is equivalent to


```
(define plus4
  (lambda (x)
            (+ x 4)
  )
)
```


- Scheme transforms the first form into the second

Local variables: `let`

- Local definitions are useful to limit the scope of, for example, temporary variables
- Scheme provides `let` for this purpose (ex. adapted from book)

```
(define (f x y)
  (let ( ( a        5    )
         ( b    (+ 1 y) )
       )
    (+ (* x (square a))
       (* y b)
       (* a b) )
  )
)
```

... let

- General form

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varn> <expn>))
  <body>
)
```


- Is syntactic shorthand for the `lambda` form

```
( (lambda (<var1> ... <varn>)
    <body>
  )
  <exp1>
  ...
  <expn>
)
```


- Note how the bindings are established, and scope
  - Simplify by hand expressions in Exercise 1.34

Procedures as general methods

- The *half-interval* method:
  - Find a zero by looking in *smaller and smaller* intervals
  - Here is an iterative, logarithmic procedure

```
(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))))))


(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

… half-interval

- We wrap it into a procedure that sanity-checks arguments

```
(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 "Values are not of opposite sign" a b)))))
```


For example

```
(half-interval-method sin 2.0 4.0)
```

- Note the “computational process” generated here
  - Cf iteration, recursion, tree recursion from earlier

Control abstractions

- We are now going to develop a series of *abstractions* that embody behaviour similar to that seen in the iterative convergence process of half-interval `search`
- We will express these abstractions as *higher-order procedures* i.e. procedures that take procedures as arguments, and often also *return* procedures as results
- Such abstractions express generally useful *control* patterns
- Other examples of control abstractions:
  - Recursion, as embodied in the Scheme interpreter
  - Iteration, as expressed via tail recursion
  - Tree recursion, as seen earlier
  - “Aggregate data” primitives (see later)
    - *map reduce scan*

Fixed points

- Sometimes we can solve *f(x) = x* by making an initial guess and computing *f(x), f(f(x)), f(f(f(x))), …*
- If *x* satisfies the equation, it is a *fixed point* of the function *f*
- We can define and use the process thus

```
(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))
(fixed-point cos 1.0)
```

Square root as a fixed point

- More interestingly, we can define `sqrt` as f.p. *y* of

  y → x/y

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```


- And using *average damping* as


```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```


- The technique of average damping aids convergence
  - Simplify by hand `(sqrt 2)` by each method above
  - Try Exercise 1.36

Procedures as returned values

- Average damping is a useful concept in its own right
- We can define an abstraction for it thus:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

- Note that the result is itself a procedure
- Then re-define square root

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

- Which makes clear the three important *ideas*
  - Finding a fixed point;
  - Use of average damping
  - The function $y \rightarrow x/y$

Newton's Method as a fixed-point process

- A higher-order procedure for derivatives

```
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
```


- To solve *g(x)=0* we find a fixed point of $x = f(x)$ where
  - `f(x) = x - ( g(x) / Dg(x) )`
  - and *D* indicates derivative

- And we can define an abstraction for Newton's Method
  - And then define square root again using that abstraction
- …

```
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))

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


- Again we express high-level concepts:
  - Find a zero of `y = y^2 - x`
  - Transform it to “fixed point” form
  - Apply fixed point abstraction

Abstractions and first-class procedures

- We can go further and combine the notions of *transform* and *fixed point* into one abstraction, as a higher-order procedure

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

  - And define square root i.t.o both avg damping and Newton's method

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (/ x y))
                            average-damp
                            1.0))

(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

First-class

- Procedures are *first-class* elements in Scheme
  - This is not common in programming languages
- First class programming language elements are characterized by the fact that they can be:
  - Named by identifiers;
  - Passed as arguments;
  - Returned as results;
  - Included in data structures.
- Trade-off:
  - Cost in implementation
  - Gain in expressive power
- Do Exercises 1.41 and 1.42