Contents

1 SICP

1.1 2a.1

A few programs that have a lot in common with each other. Make some abstractions that are not easy to make in most languages.

First is the sum:

$$\sum_{i=a}^{b} i$$

(It has a closed form and it is easy to compute, but we are not interested in that kind of solution here).

You have a base case and a recursive case

you have an easy case you know the answer to or you have to reduce it to simpler problem.

The subproblem - add up the integers, one fewer integer, and one fewer again, and once it is solved add a to it and the result is the answer to the whole problem.

```
(define (sum-int a b)
  (if (> a b)
    0
    (+ a (sum-int (+ a 1) b))))
```

Recursive function but no tail position

```
const sumInt = function(a, b) {
    if(a > b) { return 0 }
    return a + sumInt(a+1, b);
};
```

Sum as reduce on a list

```
const range = function(a, b) {
    return [...Array(b - a + 1).keys()].map(i => i + a);
};

const sumInt = function(a, b) {
    return range(a, b).reduce((acc, x) => acc + x, 0);
};
```

$$\sum_{i=a}^{b} i^2$$

```
(define (sum-sq a b)
  (if (> a b)
      0
      (+ (square a)
      (sum-sq (+ 1 a) b))))
```

Now those two programs are almost identicle, the same first clause, the same predicate, the same consequence, and the alternatives are very similar too. The only difference is the A and the square of A.

What is similar here has to do with the Sigma notation and not depending upon what is it adding up.

When you design complex systems and you want to be able to understand them, it's crucial to divide the things up into as many pieces as you can, each of which you understand separately.

$$\sum_{i=aby4}^{b} \frac{1}{i(i+2)}$$

$$\frac{1}{1\dot{3}} + \frac{1}{5\dot{7}} + \frac{1}{9\dot{1}1}$$

converges to $\frac{\pi}{8}$

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (* a (+ a 2)))
         (pi-sum (+ 4 a) b))))
```

When you learn a language you also leearn common patterns of usage. You learn ideoms, useful things to know at a flash (they are often hard to think out on your self).

In scheme you can not only know that, but you can also give the knowledge of that a name.

The pattern:

```
;;(define (<name> a b)
;;  (if (> a b)
;;      0
;;      (+ (<term> a)
;;         (<name> (<next> a) b))))
```

Numbers are not special, they are just one kind of data. You must be able to give all sorts of names to all kinds of data, like procedures. Many languages allow procedural arguments.

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term
              (next a)
              next
              b))))

(sum (lambda (x) (* x x)) 1 (lambda (x) (+ 1 x)) 4)

;; as in the video lecture
(define (sum-int a b)
  (define (identity a) a)
  (sum identity a (+ 1 a) b))

;; helpers diverging from the original
(define (identity x) x)
(define (square x) (* x x))
(define (inc x) (+ 1 x))

(define (sum-sq a b)
  (sum square a inc b))
```

```
(define (pi-sum a b)
  (sum (lambda (i) (/ 1 (* i (+ i 2))))
       a
       (lambda (i) (+ 4 i))
       b))
```

With js, but not in tail position

```
const sum = function(term, a, next, b) {
    if(a > b) { return 0; }
    return term(a) + sum(term, next(a), next, b);
};

const identity = x => x;
const square = x => x * x;
const inc = x => x + 1;

const sumInt = function(a, b) {
    return sum(identity, a, inc, b);
};

const sumSq = function(a, b) {
    return sum(square, a, inc, b);
};
```

As a reduce on list

```
const range = function(a, b, next) {
    return [...Array(b - a + 1).keys()].map( x => next(x));
};

const sum = function(term, a, next, b) {
    return range(a, b, next).reduce((acc, x) => {
        return acc + term(x);
    });
};
```

The invention of the procedure that takes a procedural argument, allows you to compress a lot of these procedures into one thing.

Iterative implementation:

```
(define (sum term a next)
```

```
(define (iter j ans)
  (if (> j b)
      ans
      (iter (next j)
            (+ (term j) ans))))
(iter a 0))
```

Iterative implementation for some reasom might be better than the recursive, but the important thing is that it is different. But the recursive way allows for decomposition. To independantly change one part of the program without affecting the other part that was written for some other cases.

1.2 2a.2

"Computers to make people happy, not people to make computers happy."

Bybilonian method for finding square root

```
(define (sqrt x)
  (define tolerance 0.00001)
  (define (good-enuf? y)
    (> (abs (- (* y y) x)) tolerance))
  (define (improve y)
    (average (/ x y) y))
  (define (try y)
    (if (good-enuf? y)
        y
        (try (improve y))))
   (try 1))
```

Look complicated, it is not obvious by looking at it what it is computing.

If y is a guess for a square root, then what we want is a function f (this is a means of improvement):

$$y \to f \to \frac{y + \frac{x}{y}}{2}$$

Such that:

$$f(\sqrt{x}) = \sqrt{x}$$

If you subsitute y with $\sqrt{x}$ you get $\sqrt{x}$. We are looking for a fixed point of the function f.

A fixed point is a place which has the property that if you put it into the function, you get the same value out. Some functions have the property that you can find their fixed point by iterating the function.