Procedural Abstraction.
High-Order Programming.

Procedures

- Means of factorizing and reusing code.
- Inspired by mathematical functions.
- Resemble mathematical functional notation.
- Two parts: *definition* and *invocation* (or *call*).
- Definition: has *formal arguments*, that are also used in the body.
- Invocation: has *actual arguments*, to which the formal arguments are *bound* once the procedure is entered.
- May have a *return value*.

Abstraction

- ◇ Means of hiding details

- ◇ *Abstraction barrier:* defining an interface to a system.
  - Describes a set of operations without details on how the operations are implemented.
  - Allows freedom of changing the implementation later, as long as the high-level operations do not change their behaviour.

- ◇ Example: set implementation
  - Operations: union, intersection, difference, etc:
  - Could be implemented as a linked list, or as a bitmap
  - Implementor can change from one implementation to the other, as long as the operations do not change their meaning.

Procedural Abstraction

- Collections of procedures are assembled into *libraries* and *modules*
- We use the library as a *black box*: we learn the interface, and we don't care about the exact implementation.
- The implementor of the library has the freedom to change the implementation as long as the interface stays the same.
- The interface of the library acts as an *abstraction barrier* to the user.
- Devising good abstraction barriers is hard, but the benefit is huge!
  - Makes "using software" much easier than "implementing software"

Procedures in C

```
int p2(int a, int b, int c) {
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        a *= a ;
        b >>= 1 ;
    } else {
        b -- ;
        c *= a ;
    }
    return p2(a,b,c) ;
}
```

```
int p1(int a, int b) {
    if (b==0) return 1 ;
    else if ( b & 1 == 0) {
        int x = p1(a,b>>1) ;
        return x*x ;
    } else
        return a*p1(a,b-1) ;
}
```

Procedures in Python

```
def p1(a,b) :
    if b==0 :
        return 1
    elif b & 1 == 0 :
        x = p1(a,b>>1)
        return x*x
    else :
        return a*p1(a,b-1)
```

```
def p2(a,b,c) :
    if b == 0 :
        return c
    elif b & 1 == 0 :
        a = a*a
        b = b>>1
    else :
        b = b-1
        c = c*a
    return p2(a,b,c)
```

Procedures in Scheme

```
(define (p1 a b)
  (if (= b 0)
      1
      (if (= (remainder b 2) 0)
          (let ((x (p1 a (/ b 2))))
            (* x x))
          (* a (p1 a (- b 1))))))
```

```
(define (p2 a b c)
  (if (= b 0)
      c
      (if (= (remainder b 2) 0)
          (p2 (* a a) (/ b 2) c)
          (p2 a (- b 1) (* c a)))))
```

Procedures in Ocaml

```
let rec p1 a b =
    if b = 0 then 1
    else if b land 1 = 0 then
            let x = p1 a (b lsr 1) in x*x
         else a * (p1 a (b-1)) ;;
```

```
let rec p2 a b c =
    if b = 0 then c
    else if b land 1 = 0
         then p2 (a*a) (b lsr 1) c
         else p2 a (b-1) (c*a) ;;
```

Procedures in Haskell

```
import Data.Bits

p1 a 0 = 1
p1 a b | b .&. 1 == 0 = let x = p1 a (b `shiftR` 1) in x*x
p1 a b = a * (p1 a (b-1))
```

```
p2 _ 0 c = c
p2 a b c | b .&. 1 == 0 = p2 (a*a) (b `shiftR` 1) c
p2 a b c = p2 a (b-1) (c*a)
```

Procedures in Prolog

```
p1(_,0,1) :- !.
p1(A,B,R) :- 0 is B /\ 1, !, B1 is B>>1, p1(A,B1,X), R is X*X.
p1(A,B,R) :- B1 is B-1, p1(A,B1,X), R is A*X.
```

```
p2(_,0,C,C) :- !.
p2(A,B,C,R) :-
    0 is B /\ 1, !, A1 is A*A, B1 is B>>1, p2(A1,B1,C,R).
p2(A,B,C,R) :- B1 is B-1, C1 is C*A, p2(A,B1,C1,R).
```

Recursion

- Some languages do not have assignment (most notably: *Haskell*) ; thus they do not have iterative statements (i.e. looping statements)

- Recursion is then the only way to implement repetitive computation.

- Distinguish between *tail recursion* (efficient) and *non-tail recursion* (less efficient)

Efficiency: Non-Tail Recursion

```
p1 a 0 = 1

p1 a b | b .&. 1 == 0 =
      let x = p1 a (b `shiftR` 1) in x*x

p1 a b = a * (p1 a (b-1))
```

```
p1 2 11

2*(p1 2 10)

2*(let x = p1 2 5 in x*x)

2*(let x = 2*(p1 2 4) in x*x)

2*(let x = 2*(let x = p1 2 2 in x*x) in x*x)

2*(let x = 2*(let x = (let x = p1 2 1 in x*x) in x*x) in x*x)

2*(let x = 2*(let x = (let x = (2*1) in x*x) in x*x) in x*x)

2*(let x = 2*(let x = (let x = 2 in x*x) in x*x) in x*x)

2*(let x = 2*16 in x*x)

2*(let x = 32 in x*x)

2*1024

2048
```

Efficiency: Tail Recursion

```
p2 2 11 1

p2 2 10 2

p2 4 5 2

p2 4 4 8

p2 16 2 8

p2 256 1 8

p2 256 0 2048

2048
```

```
p2 _ 0 c = c
p2 a b c | b .&. 1 == 0 = p2 (a*a) (b `shiftR` 1) c
p2 a b c = p2 a (b-1) (c*a)
```

Replacing Iteration with Recursion

```
while ( b != 0) {
    if ( b & 1 == 0 ) {
        a *= a ;
        b >>= 1 ;
    } else {
        b -- ;
        c *= a ;
    }
}

// c is the important
// value out of the loop
```

While loops can be turned into recursive functions by means of a *systematic translation scheme*

```
int p2(int a, int b, int c) {
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        a *= a ;
        b >>= 1 ;
    } else {
        b -- ;
        c *= a ;
    }
    return p2(a,b,c) ;
}
```

Replacing Iteration With Recursion

```
while ( b != 0) {
    if ( b & 1 == 0 ) {
        a *= a ;
        b >>= 1 ;
    } else {
        b -- ;
        c *= a ;
    }
}

// c is the important
// value out of the loop
```

```
int p2(int a, int b, int c) {
    int a1, b1, c1 ;
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        a1 = a*a ;
        b1 = b >> 1 ;
        c1 = c ;
    } else {
        a1 = a ;
        b1 = b - 1 ;
        c1 = c * a ;
    }
    return p2(a1,b1,c1) ;
}
```

Replacing Iteration with Recursion

```
int p2(int a, int b, int c) {
    int a1, b1, c1 ;
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        a1 = a*a ;
        b1 = b >> 1 ;
        c1 = c ;
    } else {
        a1 = a ;
        b1 = b - 1 ;
        c1 = c * a ;
    }
    return p2(a1,b1,c1) ;
}
```

```
int p2(int a, int b, int c) {
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        int a1, b1, c1 ;
        a1 = a*a ;
        b1 = b >> 1 ;
        c1 = c ;
        return p2(a1,b1,c1) ;
    } else {
        int a1, b1, c1 ;
        a1 = a ;
        b1 = b - 1 ;
        c1 = c * a ;
        return p2(a1,b1,c1) ;
    }
}
```

Replacing Iteration with Recursion

```
int p2(int a, int b, int c) {
    if ( b == 0 ) return c ;
    else if ( b & 1 == 0 ) {
        int a1, b1, c1 ;
        a1 = a*a ;
        b1 = b >> 1 ;
        c1 = c ;
        return p2(a1,b1,c1) ;
    } else {
        int a1, b1, c1 ;
        a1 = a ;
        b1 = b - 1 ;
        c1 = c * a ;
        return p2(a1,b1,c1) ;
    }
}
```

```
let rec p2 a b c =
 if b = 0 then c
 else
  if b land 1 = 0
  then
    let (a1,b1,c1) =
           (a*a,b lsr 1, c)
     in p2 a1 b1 c1
  else
    let (a1,b1,c1) =
           (a,b-1,c*a)
     in p2 a1 b1 c1 ;;
```

Procedures as First-Class Values

- *First class value:* entity that:
  - can become the value of a variable
  - can be used as an argument to, or return value from a function
  - can be created as an unnamed value

- Most modern languages allow *functions as first-class values*

- Exceptions:
  - C — only allows pointers to functions as function arguments
  - Prolog — allows dynamic modification of programs by adding and deleting rules, but not the creation of unnamed predicates

- Functions as unnamed entities:
  - Scheme: `(lambda (x) (+ x 1))` — `((lambda (x) (+ x 1)) 5)` ≡ 6
  - Ocaml: `fun x -> x+1` — `(fun x -> x+1) 5` ≡ 6
  - Haskell: `\ x -> x+1` — `(\ x -> x+1) 5` ≡ 6
  - Python: `lambda x: x+1` — `(lambda x: x+1)(5)` ≡ 6

Higher Order Programming: Haslkell

Solve $f(x) = 0$ by the *half-interval method*.

```
module Main where

solve f x1 x2 eps
    | abs(x1-x2) < eps = (x1+x2)/2
solve f x1 x2 eps
    | (f x1)*(f ((x1+x2)/2)) <= 0 =
                  solve f x1 ((x1+x2)/2) eps
solve f x1 x2 eps
    | (f x2)*(f ((x1+x2)/2)) <= 0 =
                  solve f ((x1+x2)/2) x2 eps
```

```
*Main> solve (\x -> x*x - 1.0) 0.0 3.0 0.00000000000001
1.0
*Main> solve cos 1.0 4.0 0.00000000000001
1.5707963267948966
*Main> solve sin 1.0 4.0 0.00000000000001
3.141592653589793 6
```

Higher-Order Programming: Python

```
from math import *

def solve(f,x1,x2,eps):
    if abs(x1-x2) < eps :
        return (x1+x2)/2
    elif f(x1)*f((x1+x2)/2) <= 0:
        return solve(f,x1,(x1+x2)/2,eps)
    elif f(x2)*f((x1+x2)/2) <= 0:
        return solve(f,(x1+x2)/2,x2,eps)
```

```
>>> solve(lambda x:x*x-1.0,1.0,4.0,0.000000000000001)
1.0000000000000004
>>> solve(lambda x:sin(x),1.0,4.0,0.000000000000001)
3.1415926535897936
>>> solve(lambda x:cos(x),1.0,4.0,0.000000000000001)
1.5707963267948966
```

Higher-Order Programming: Scheme

```
(define (solve f x1 x2 eps)
  (cond ((< (abs (- x1 x2)) eps)
         (/ (+ x1 x2) 2))
        ((<= (* (f x1) (f (/ (+ x1 x2) 2))) 0)
         (solve f x1 (/ (+ x1 x2) 2) eps))
        ((<= (* (f x2) (f (/ (+ x1 x2) 2))) 0)
         (solve f (/ (+ x1 x2) 2) x2 eps))))
```

```
> (solve (lambda (x) (- (* x x) 1.0)) 0.0 3.0 0.00000000000001)
1.0
> (solve sin 1.0 4.0 0.00000000000001)
3.141592653589793 6
> (solve cos 1.0 4.0 0.00000000000001)
1.5707963267948966
```

HOP Primitives

◇ Higher order programming simplifies programming over collections (lists, sets, bags, dictionaries)

◇ Primitives of higher order programming
  - `map` : apply a function to every element of a collection and create a similar collection of results
  - `fold` : combine all the elements of a collection via an operator
  - `filter` : remove from a collection the elements that do not satisfy a predicate
  - `zip` : create a collection of pairs, each pair being made up of elements of the same rank in two input collections

◇ They form a very useful *abstraction barrier*

Lists

- Languages without assignment are better off with recursive datatypes for data aggregation

- That is, *lists* are more suitable than *arrays*

- Haskell and Ocaml lists resemble Prolog lists

- Haskell:
  - `[]` — empty list
  - `h:t` — list with head `h` and tail `t`
  - `[1,2,3]` — list containing 1,2,3

- Ocaml
  - `[]` — empty list
  - `h::t` — list with head `h` and tail `t`
  - `[1;2;3]` — list containing 1,2,3

Map

Haskell:

```
map f []     = []
map f (x:xs) = f x : map f xs

> map (\x->x*x) [1,2,3,4]
[1,4,9,16]
```

Ocaml:

```
let rec map f l =
match l with []    -> []
      |      x::xs -> f(x) :: map f xs;;

# map (fun x -> x*x) [1;2;3;4]  ;;
- : int list = [1; 4; 9; 16]
```

Fold Left

Haskell:

```
foldl f z []      =  z
foldl f z (x:xs) =  foldl f (f z x) xs

> foldl (+) 0 [1,2,3,4]
10
> foldl div 32768 [16,8,4] -- ((32768/16)/8)/4
64
```

Ocaml:

```
let rec fold_left f z l =
match l with []     -> z
      |      x::xs -> fold_left f (f z x) xs);;

# fold_left (+) 0 [1;2;3;4];;
- : int = 10
# fold_left (/) 32768 [16;8;4] ;;
- : int = 64
```

Fold Right

Haskell:

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

> foldr (+) 0 [1,2,3,4]
10
> foldr div 1 [16,8,4] -- 16/(8/(4/1))
8
```

Ocaml:

```
let rec fold_right f l z =
match l with []     -> z
      |      x::xs -> f x (fold_right f xs z);;

# fold_right (+) [1;2;3;4] 0;;
- : int = 10
# fold_right (/) [16;8;4] 1 ;;
- : int = 8
```

Filter

Haskell:

```
filter p []                  = []
filter p (x:xs) | p x        = x : filter p xs
                | otherwise = filter p xs

> filter (\x-> x `mod` 2 == 0) [1,2,3,4]
[2,4]
```

Ocaml:

```
let filter p =
       fold_right
          (fun x acc -> if p x then x::acc else acc)
          [];;

# filter (fun x -> x mod 2 = 0) [1;2;3;4];;
- : int list = [2; 4]
```

zipWith

Haskell:

```
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []

> zipWith (+) [1,2,3,4] [10,20,30,40]
[11,22,33,44]
```

Ocaml:

```
let rec zipWith f lx ly =
   match lx,ly with
   | (x::xs),(y::ys) -> (f x y)::(zipWith f xs ys)
   | _ -> []

# zipWith (+) [1;2;3,4] [10;20;30,40] ;;
- : int list = [11; 22; 33; 44]
```

More In-Depth Haskell

- Function application:
  - we write `f x` instead of `f(x)`
  - `f a b c` same as `((f a) b) c`
  - `f a` is a function which can be applied to `b`, and in turn returns a function that can be further applied to `c` to yield a result.
  - Can be thought of as an *invisible* application operator

- Cuts:
  - `(3+)` same as `\x -> 3 + x`

- Infix operators:
  - Can declare operators as `infix` (left associativity) or `infixr` (right associativity)
  - Regular binary functions can be written infix if enclosed in back quotes: `div x y` same as ``x `div` y``

More Haskell

◇ Function composition: . (the period)

```
> ((\x->x+1) . (\x->x*x)) 3
10

> (\x->x+1) ( (\x->x*x) 3 )
10
```

◇ List append:

```
> [1,2,3]++[10,20,30]
[1,2,3,10,20,30]
```

◇ List length:

```
> length [1,2,3,4]
4
```

◇ List comprehensions

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
> [1,3..10]
[1,3,5,7,9]
> [x|x<-[1,3..10], x*(x-1)>10]
[5,7,9]
```

◇ List manipulation

```
> head [1,2,3,4]
1
> tail [1,2,3,4]
[2,3,4]
> [1,2,3,4] !! 3
4
> [1,2,3,4] !! 0
1
> take 3 [1..10]
[1,2,3]
> drop 3 [1..10]
[4,5,6,7,8,9,10]
```

HOP in Haskell

Matrix transposition

```
transpose l = map (\i->map (!!i) l) [0 .. (length l - 1)]

> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
```