Data Abstractions

Providing an interface to a data type. Actual implementation of data type can change.

Binary Tree abstract data type, provide *constructor* and *selectors*:

- `(make-tree <data> <left-child> <right-child>)` ⟶ `<tree>`
- `(empty-tree)` ⟶ `<tree>`
- `(tree-data <tree>)` ⟶ `<data>`
- `(tree-left-child <tree>)` ⟶ `<left-child>`
- `(tree-right-child <tree>)` ⟶ `<right-child>`

Someone can provide binary search tree facilities `insert`, `lookup`, `delete`, and `is-empty?` using this abstract binary tree.

Example Binary Tree

A binary tree implementation:

```
(define make-tree list)
(define (emptytree) 'empty-tree)
(define tree-data car)
(define tree-left-child cadr)
(define tree-right-child caddr)
```

Can define `is-empty?` using these:

```
(define (is-empty tree) (equal? tree (empty-tree)))
```


First Class Functions

Like numbers, Scheme functions are *first class* citizens.

They

- can put stored in variables or lists,
- can be passed to other functions, and
- can be returned from other functions.

1st Class Func Examples

Consider:

```
guile> (define x 10)
guile> +
#<primitive-procedure +>
guile> x
10
```

What is special about `+`?

```
guile> (define plus +)
guile> (plus 1 2 3 4 5)
15
```

Procedural Abstractions

Recall `double-list`. Other functions, e.g. `sqrt-list`, have same structure.

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)

(define (sqrt-list lst)
   (if (null? lst)
       '()
       (cons (sqrt (car list) (double-list (cdr lst)))
   )
)
```

What is different?

Procedural Abstraction (cont)

Like *data abstraction*, we can do *procedural abstraction*:

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map func (cdr lst)))
   )
)
```

This `map` is already defined in the global environment.

First-Order Funcs

First-Order Functions take other functions as arguments. E.g. `map`.

What do these do?

```
(map double '(1 2 3))
(map sqrt '(1 4 9))
(map symbol? '(3 a 6 b 9 c))
(map + '(1 2 3) '(4 5 6))
```

What if no existing function does exactly what you want?

Lambdas

Anonymous Functions, the `lambda` special form.

```
(map (lambda (x) (* x x)) '(1 2 3))
```

Compare

```
(define (square x) (* x x))
(define square (lambda (x) (* x x)))
```

Rename this `mystery` function:

```
(define (mystery lst)
   (sum (map (lambda (x) 1) lst))
)
```

Abstracting Computations

Abstracting computational patterns gives, flexibility, power, and glue.

Can we abstract `sum` so that we can easily implement `product`?

```
(define (sum lst)
   (if (null? lst)
       0                              ; base case
       (+ (car lst) (sum (cdr lst)))     ; recursive call
   )
)
```


Abstract `sum`

Fold

Fold is useful for many things.

```
(define (mystery lst)
   (foldr (lambda (x y) (+ 1 y)) 0 lst)
)

(define (product lst)
   (foldr * 1 lst)
)

(define (append first second)
   (foldr cons second first)
)
```

We defined `foldr` (fold right) which starts at the rear and works forwards. `foldl` (fold left) does the opposite and is also useful.

Funcs as Return Values

Recall constant function `(lambda (x) 1)`.

In general we might want a constant function to return any constant.

```
(define (const-func val)
   (lambda (x) val)
)
```

Now, rewrite `mystery` as:

```
(define (length lst)
   (sum (map (const-func 1) lst))
)
```

Benefits . . .

Compare:

```scheme
(define (length lst)
   (sum (map (const-func 1) lst))
)

(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))
   )
)
```

Which is easier to understand?

Not Convinced?

How about:

```scheme
(define (sum lst)
   (foldr + 0 lst)
)

(define (sum lst)
   (if (null? lst)
       0
       (+ (car lst) (sum (cdr lst)))
   )
)
```


What does this do?

```scheme
(define x 10)
(define y 20)

(define (add x y) (+ x y))

(let ((x 50))
     (add x y)
)
```

What gets printed?

Lexical Scoping

Scheme does *lexical* or *static scoping*.

static ⇒ “at compile time”

Can look at scheme code and tell which variables are which.

What about C?