Lecture 4 — August 12, 2015

- Today:
  - Abstracting Data
  - Lists
  - Types
  - Higher-Order Procedures
- Readings:
  - Finish *SICP* Section 2.1
  - Read *SICP* 2.2, 2.3, *&* 2.4 — including all footnotes and exercises

CSE130, Summer Session II

Compound Data

- "glue" together data elements
- "unglue" them to get the more basic components back out
- Ideally want the glue to have the closure property:
  - "The result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object"

The Node Abstraction

Let's use induction:

- we have a bAse case
- we have an inDuctive case

If we can hold a value in A, and have the option to have D point to another node, then we can hold as many values as we want.

Pairs, a.k.a. Cons Cells

- A regular procedure, **cons**(tructor)

```
(cons <x-exp>  <y-exp>)
```

  - Where *<x-exp>* evaluates to a value *<x-val>*,
  - and *<y-exp>* evaluates to a value *<y-val>*
  - Returns a "pair" *<P>*…
  - whose "car part" is *<x-val>*, and
  - whose "cdr part" is *<y-val>*

- Returns the car part of the pair *<P>*:

`(car <P>)` → *<x-val>*

- Returns the cdr part of the pair *<P>*:

`(cdr <P>)` → *<y-val>*

Pair Abstraction

- Constructor

; cons: A, B → Pair<A, B>

`(cons` *<x> <y>*`)` → *<P>*

- Accessors

; car: Pair<A, B> → A

`(car` *<P>*`)` → *<x>*

; cdr: Pair<A, B> → B

`(cdr` *<P>*`)` → *<y>*

- Predicate

; pair?: anytype → boolean

`(pair? <z>)` → `#t` if <z> is a pair; else `#f`

Pair Abstraction

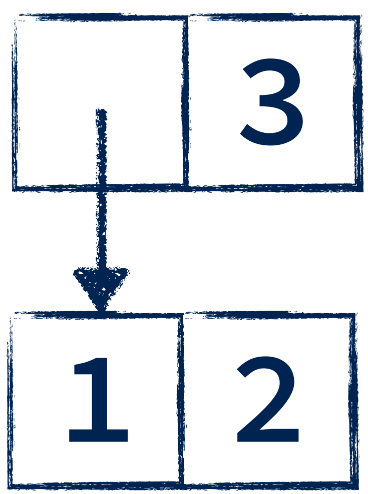
- There is a contract between the constructor and the selectors:

```
(car (cons <a> <b>)) → <a>
(cdr (cons <a> <b>)) → <b>
```

- Pairs have the property of closure; we can use the result of a pair as an element of a new pair:

```
(cons (cons 1 2) 3)
```

- Which produces the following “box and pointer” diagram:



Conventional Interfaces: Lists

- A list is a data object that can hold an arbitrary number of ordered items
- More formally, a list is a sequence of pairs with the following properties:
  - Car part of a pair in sequence – holds an item
  - Cdr part of a pair in sequence – holds a pointer to rest of list
  - Empty-list "`nil`" – signals no more pairs, or end of list
- In the book "`nil`" is used before it is "dispensed with"
  - Instead, use `'()` as the empty list

Box and Pointer Diagram Exercise

`(cons` *e1 e2*`)`

`(list` *e1 e2 … en*`)`

- car is always an element
- cdr is always the rest of the list

`(null?` <z>`)` → `#t` if <z> evaluates to empty list

Common Pattern: **cons**ing up a list

- Recursive structures naturally lead to recursive algorithms:

```
(define (enumerate-interval from to)
  (if (> from to)
      '()
      (cons from
            (enumerate-interval
             (+ 1 from)
             to))))
```

```
(enumerate-interval 1 5) → (1 2 3 4 5)
(enumerate-interval 1 1) → (1)
(enumerate-interval 1 0) → ()
```

Common Pattern: `consing up a list`

```
(define (e-i from to)
  (if (> from to) '()
      (cons from (e-i (+ 1 from) to))))
```

```
(e-i 2 4)
(if (> 2 4) '() (cons 2 (e-i (+ 1 2) 4)))
(if #f '() (cons 2 (e-i (+ 1 2) 4)))
(cons 2 (e-i (+ 1 2) 4))
(cons 2 (e-i 3 4))
 ;; ... omit some intermediate steps
(cons 2 (cons 3 (e-i 4 4)))
(cons 2 (cons 3 (cons 4 (e-i 5 4))))
(cons 2 (cons 3 (cons 4 '())))
  → (2 3 4)
```

Common Pattern: `cdr`ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1))))
```

- Let's try:

```
(define (length lst)
```

cdring and **cons**ing Examples

```
;; create a new list from the given one, squaring each element
(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst)))))

;; create a new list from the given one, doubling each element
(define (double-list lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst)
            (double-list (cdr lst))))))
```

cdring and **cons**ing Examples

```
(define (copy lst)
  (if (null? lst)
      '()
      (cons (car lst)
            (copy (cdr lst)))))
```

```
(copy (list "a" "b" "c" "d"))
 → ("a" "b" "c" "d")
```

cdring and **cons**ing Examples

```
;; create a new list from the given one, composed of only the even
;; elements
(define (filter-evens list)
  (cond
    [(null? list) '()]
    [(even? (car list))
     (cons (car list)
           (filter-evens (cdr list)))]
    [else (filter-evens (cdr list))]))
```

- Note: The `[]`s could also be just plain `()`s
- Inside and outside of Racket, using `()`s is always legal

cdring and **cons**ing Examples

```
(define (append list1 list2)  ; recursive form
```

Types

- Addition is not defined for strings:

```
(+ 5 10) → 15
(+ "hi" 5)
 → +: contract violation
   expected: number?
   given: "hi"
   argument position: 1st
```

- The addition procedure has associated with it an expectation of what kinds of arguments it will get
  - Here, the expectation is that the type of each argument is a number

Types: Simple Data

- Number
  - complex (predicate `complex?` usually the same as `number?`)
  - real (predicate `real?` usually the same as `rational?`)
  - rational
  - integer
- String
- Boolean
- Names (symbols)

Types: Compound Data

Pair<A, B>

- A compound data structure formed by a **`cons`** pair, in which the first element is of type A, and the second of type B:
  - e.g., **`(cons 1 2)`** has type *Pair<number,number>*

List<A> = Pair<A, List<A> **or** *nil>*

- A compound data structure that is recursively defined as a pair:
  - Whose first element is of type A, and
  - Whose second element is either a list of type A or the empty list.
  - e.g., **`(list 1 2 3)`** has type *List<number>*;
  - and **`(list 1 "string" 3)`** has type *List<number or string>*

Types: Procedures

- Because procedures operate on object, and return values, we can define their types as well.
- We will denote a procedure type by indicating the types of each of its parameters, and the type of the returned value, plus the symbol $\rightarrow$ to indicate that the arguments are mapped to the return value
- E.g.,

$$number \rightarrow number$$

specifies a procedure that takes a number as input, and returns a number as output

Type Examples

Expression:	Evaluates to a value of type:
`15`	number
`"hi"`	string
`square`	number → number
`>`	number, number → boolean

- The type of a procedure is a ***contract***:
  - If the operands have the specified types, the procedure will result in a value of the specified type
  - Otherwise, its behavior is undefined; maybe an error is signaled, maybe random behavior

Types, precisely

- A type describes a set of Scheme values
  - E.g.,
    $number \rightarrow number$
    describes the set of: All procedures, whose result is a number, which require one argument that must be a number
- Every Scheme value has a type
- Some values can be described by multiple types
  - If so, choose the type which describes the largest set
  - For example, addition maps two integers to an integer, but it also maps two numbers (e.g. reals) to a number
- Special-form keywords, like `define`, do not name values, therefore special-form keywords have no type

What are the types?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))

(lambda (p) (if p "hi" "bye"))

(* 3.14 (* 2 5))

(+ car cdr)
```

Motivating higher-order procedures…

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

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

(define (pi-sum a b)   ; approximates pi*pi/8
  (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ a 2) b))))
```

$$\sum_{k=1}^{100} k$$

$$\sum_{k=1}^{100} k^2$$

$$\sum_{k=1,odd}^{101} 1/k^2$$

What are the patterns?

Higher-Order Procedures

- What's the type of this function?

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

- A ***higher-order procedure*** takes a procedure as an argument and/or returns one as a value

Higher-Order Procedures

```
(define (sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))

(define (pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))

; Or, another way to write sum-integers…
(define (id x) x)    ; identity function
(define (add1 n) (+ n 1))
(define (sum-integers a b)
  (sum id a add1 b))
```

```scheme
;; create a new list from the given one, squaring each element
(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst)))))

;; create a new list from the given one, doubling each element
(define (double-list lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst)
            (double-list (cdr lst))))))
```

The pattern is:

- we take a list as input,
- "walk down" the list an element at a time,
- do "something" to each element, and
- construct a new list of the results

Pattern: Transforming a list

```
;; create a new list from the given one, applying the given procedure
;; to each element
(define (map proc lst)
  (if (null? list)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

"But how is this different from Hadoop?"

(same "map" as in "map-reduce"!)

Abstracting away the commonality

```
;; create a new list from the given one, applying the
;; given procedure to each element
(define (map proc lst)
  (if (null? list)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

Common Pattern: Filtering a List

```
;; pred must be a procedure that returns a boolean
(define (filter pred lst)
  (cond
    [(null? lst) '()]
    [(pred (car lst))
     (cons (car lst)
           (filter pred (cdr lst)))]
    [else (filter pred (cdr lst))]))
```

```
(filter even? (list 1 2 3 4 5 6))
```

→ `(2 4 6)`

Pattern: Result accumulation

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (car lst) (add-up (cdr lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (car lst) (mult-all (cdr lst)))))
```

```
(define (fold-right op init lst)
```

Lambdas encapsulate their environment

A lambda remembers the values of the variables in its environment:

```
(define (make-adder a)
  (lambda (n) (+ n a)))

(define successor (make-adder 1))
(define add-5 (make-adder 5))

(successor 2)
(add-5 10)
(add-5 (add-5 (successor 24)))
```

The **cons**-**car**-**cdr** Conspiracy

```
(car (cons a b)) → a
(cdr (cons a b)) → b
```

A group of functions can “conspire” with each other. What we want is to define our <u>own</u> **cons**, **car**, and **cdr** functions:

```
(define (cons a b)
  (lambda (selector)
    (if selector a b)))

(define (car p) (p #t))

(define (cdr p) (p #f))
```

The **`cons-car-cdr`** Conspiracy

- Section 2.1.3 “mind boggling” definition
- We don’t need Booleans or `cond` or `if`!

```
(define (cons a b)
  (lambda (selector)
    (selector a b)))

(define (car p)
  (p (lambda (a b) a)))

(define (cdr p)
  (p (lambda (a b) b)))
```