



# Data-directed Design

---

Corky Cartwright  
Stephen Wong  
Rice University



# Sample Development of Sorting

Recall our definition of lists of numbers in Lecture 3:

```
; A list-of-number is either  
;   empty, or  
;   (cons n lon)  
; where n is a number and lon is a list-of-number.
```

and the corresponding template:

```
#| (define (f-lon ... alon ...)  
    (cond [(empty? alon) ... ]  
          [(cons? alon) ... (first alon) ...  
                            (f-lon ... (rest alon) ...) ... ])) |#
```

Our task is to define a function

```
; Type contract  
; sort: list-of-number -> list-of-number  
; Purpose: (sort lon) returns a list containing the  
; elements of lon in ascending (non-descending) order  
; Examples:  
(check-expect (sort empty) empty)  
(check-expect (sort '(1)) '(1))  
(check-expect (sort '(4 2)) '(2 4))  
(check-expect (sort '(4 10 2 -5 4)) '(-5 2 4 4 10))
```

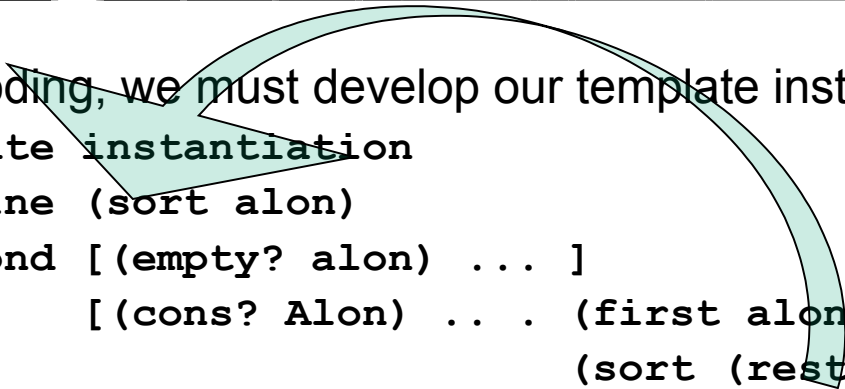


# Sorting cont.

---

Before coding, we must develop our template instantiation::

```
; Template instantiation
#| (define (sort alon)
    (cond [(empty? alon) ... ]
          [(cons? Alon) .. . (first alon) .. .
                               (sort (rest alon)) ... ])) |#
```



To write the code, we must fill in the ellipsis.

```
(define (sort alon)
  (cond [(empty? alon) empty ]
        [(cons? alon) (insert (first alon) (sort (rest alon))) ]))
```

What does `(insert n alon)` do? It inserts the element `n` in sorted position in `alon` assuming that `alon` is already sorted. We need to develop the function `insert` using our design recipe. We have already defined the list-of-number data type and provided a template for processing it.

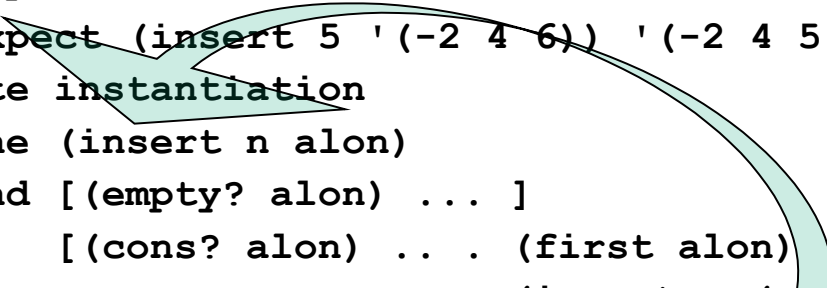


# Sorting cont.

---

Our task is to define a function

```
; Type contract
; insert: number list-of-number -> list-of-number
; Purpose: (insert n lon) returns a list containing n
; and the elements of lon in sorted (ascending) order,
; assuming that lon is already sorted.
; Examples:
(check-expect (insert 0 empty) '(0))
(check-expect (insert 0 '(1)) '(0 1))
(check-expect (insert 1 '(0)) '(0 1))
(check-expect (insert 5 '(-2 4 6)) '(-2 4 5 6))
; Template instantiation
#| (define (insert n alon)
    (cond [(empty? alon) ... ]
          [(cons? alon) ... (first alon) ...
                           (insert n (rest alon)) ... ])) |#
```



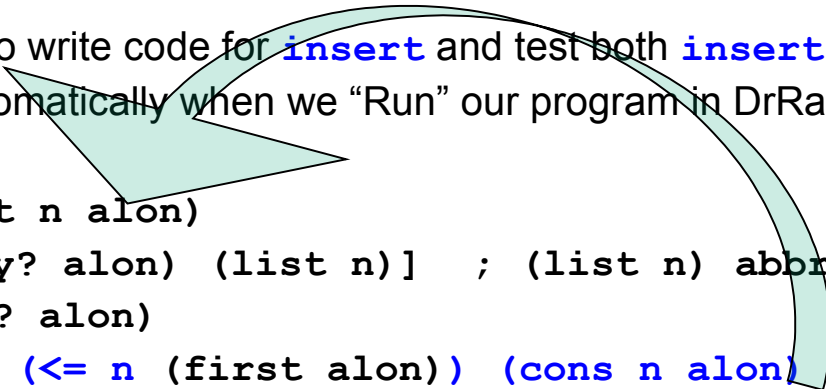


# Sorting cont.

---

All that remains is to write code for `insert` and test both `insert` and `sort` which happens automatically when we “Run” our program in DrRacket.

; Code



```
(define (insert n alon)
  (cond [(empty? alon) (list n)] ; (list n) abbreviates (cons n empty)
        [(cons? alon)
         (if (<= n (first alon)) (cons n alon)
             (cons (first alon) (insert n (rest alon))))]))
```



# Parameterized Data Definitions

---

In our definition of lists from Lecture 3 and Lab 2, we stipulated that the list elements were numbers. But we can use an unspecified type **alpha** for the element type and the definition looks essentially the same:

```
; A list-of-alpha is either  
;   empty, or  
;   (cons a loa)  
; where a is an alpha and loa is a list-of-alpha
```

In subsequent type contracts and template instantiations we can instantiate **alpha** as any type, such as **list-of-symbol**, **list-of-string**, or **list-of-number**.



# Parameterized List Template



The template for the preceding data definition is:

```
;; (define (f ... a-list ...)  
;;   (cond  
;;     [(empty? a-list) ...]  
;;     [else ... (first a-list) ...  
;;       ... (f ... (rest a-list) ...) ...]))
```

which is **identical** to the template for `list-of-number`. The form of the template does not depend on element type. It applies to `list-of-alpha` where `alpha` is any type. In fact, some functions like `length` (in HW01 under a different name and restricted to symbols), `reverse`, `append`, `first`, `rest` work for all types `list-of-alpha`. Henceforth, we will allow type variables like `alpha` in data definitions.



# Plan for this lecture

---

- List abbreviations
- Practice with the list template
  - Choosing the argument to process
  - Recognizing when help (auxiliary) functions are required/advisable.
- Data-directed design with numbers





# List Abbreviations

---

Let  $e_1, e_2, \dots, e_n$  be Scheme expressions. Then

`(list e1 e2 ... en)` abbreviates

`(cons e1 (cons e2 ... (cons en empty)))`

Let  $s_1, s_2, \dots, s_n$  be symbols, numbers, or unquoted lists (constructed in the same way).

`'(s1 ... sn)` abbreviates `(list 's1 ... 'sn)`

Examples (all equal):

`'((1 2) (3 four))`

`(list (list 1 2) (list 3 'four))`

`(cons (cons 1 (cons 2 empty))`

`(cons (cons 3 (cons 'four empty))) empty)`

Do not nest quotation! It does not work!

Do not use `true`, `false`, `empty` inside quotation. When in doubt, use `(list ...)` in preference to quotation.



## A simple list function of two list arguments

---

The append function that concatenates lists is built-in to Scheme.

; Type contract:

; app: list-of-**alpha** list-of-**alpha** -> list-of-**alpha**

; Purpose: (app a b) concatenates the lists a and b.

; Examples

```
(check-expect (app '(a) '(b c)) '(a b c))
```

```
(check-expect (app empty '(c d)) '(c d))
```

```
(check-expect (app '(a b) empty) '(a b))
```

```
(check-expect (app '(a b) '(c d)) '(a b c d))
```

; Template Instantiation:

```
|# (define (app x y)
      (cond [(empty? x) ...]
            [(cons? x) ... (first x) ...
                          (app (rest x) y) ... ]))
```

#|



## append cont.

---

- ; Code:  

```
(define (app x y)
  (cond [(empty? x) y]
        [(cons? x)
         (cons (first x) (app (rest x) y))]))
```
- Would recurring on the second argument work?



## Using append as an auxiliary function

---

- **append** is included in the Scheme library
- concatenation is the common string (a form of list of char) “construction” operation
- *Problem:* cost of operation is not constant; it is proportional to size of first argument (or, in case of strings, size of constructed list)
- Example of function that uses **append** to construct its result: **flatten**



# Defining flatten

---

```
;; Type contract
;; flatten: list-of-list-of-alpha -> list-of-alpha
;; Purpose: concatenates all of the lists of elements in the
;; input to form a list of elements
;; Tests  WARNING: empty, true, false do NOT work inside '
(check-expect (flatten '((a b) (c d) (e f))) '(a b c d e f))
(check-expect (flatten empty) empty)
(check-expect (flatten '((a b) () (c d))) '(a b c d))
(check-expect (flatten '(() (a b) (c d) ())) '(a b c d))
```

## Recall that:

```
;; A list-of-alpha is either:
;;   empty, or
;;   (cons a aloa) where a is an alpha and aloa is a list-of-alpha
;; Template:
;; (define (f ... aloa ...))
;;   (cond [(empty? aloa) ...]
;;         [(cons? aloa) ... (first aloa)
;;                           ... (f ... (rest aloa) ...) ...]))
```



# Defining flatten

---

```
;; Template Instantiation:
```

```
#|
```

```
(define (flatten aloloa)
  (cond [(empty? aloloa) ... ]
        [(cons? aloloa) ... (first aloloa)
                             ... (flatten (rest aloloa)) ... ]))
```

```
|#
```

```
;; Code:
```

```
(define (flatten aloloa)
  (cond [(empty? aloloa) empty]
        [(cons? aloloa)
         (append (first aloloa)
                  (flatten (rest aloloa))) ]))
```

This is not the standard operation that is defined in some Lisp/Scheme libraries; it has a more restrictive input type.



# Examples of Algebraic Data

---

- Files on your computer
  - Simple File, or
  - Folder, which contains a list of Files
- XML
  - General format for representing algebraic data as ASCII text
- Internet domain names
- Natural numbers
- Arithmetic expressions
- Syntax trees



# Natural Numbers: Data definition

---

- Standard definition from mathematics

```
;; A natural-number (N for short) is either  
;; 0, or  
;; (add1 n)  
;; where n is a natural-number
```

- Comments:

- In mathematics, **add1** is usually called **succ** or **S**, for *successor*.
- Principle of mathematical induction for the natural numbers is based on this definition (using **S** for successor):

$$P(0), \forall x [P(x) \rightarrow P(S(x))]$$

-----

$$\forall x P(x)$$

- Is there an analogous induction principle for other forms of inductively defined data? Yes!





# Examples and Basic Operations

---

- Examples (using constructors)
  - Zero: `0`
  - One: `(add1 0)`
  - Four: `(add1 (add1 (add1 (add1 0))))`

- Accessors:

- `sub1 : N -> N`

Note: `sub1` is typically called **pred** or **P** in mathematics; using `sub1` instead is a bit of a cheat because `(sub1 0)` behaves incorrectly.

- Recognizers:

- `zero? : Any -> bool`
  - `positive?: Any -> bool ; ; not add1?`



# Basic Laws (Reductions) for Natural Numbers

---

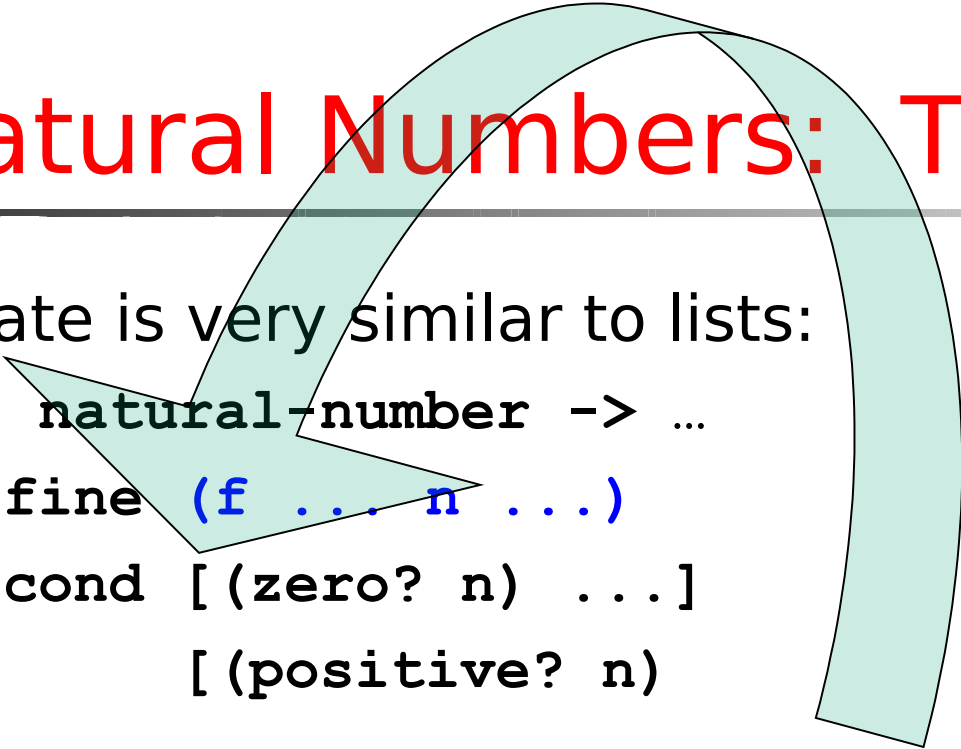
- Recall the ones for lists:
  - For all elements  $v$ , and lists  $l$ , we have
    - `(empty? empty) = true` ;; recognizer
    - `(empty? (cons v l)) = false`
    - `(rest (cons v l)) = l` ;; accessor
    - `(first (cons v l)) = v`
- Basic laws:
  - For all natural numbers  $n$ , we have
    - `(zero? 0) = true` ;; recognizer
    - `(zero? (add1 n)) = false`
    - `(positive? (add1 n)) = true`
    - `(positive? 0) = false`
    - `(sub1 (add1 n)) = n` ;; accessor
- Similar rules exist for **all** inductively-defined data types
- What about laws for `(equal? ...)`



# Natural Numbers: Template

---

Template is very similar to lists:



```
;; f : natural-number -> ...  
;; (define (f ... n ...)  
;;   (cond [(zero? n) ...]  
;;         [(positive? n)  
;;          ... (f ... (sub1 n) ...) ...])  
;;
```



# Example

---

- Write a function `repeat` that given a symbol `s` and number `n` constructs a list containing `n` copies of `s`.

```
; Type contract
; repeat : symbol natural-number -> list-of-symbol
; Purpose: (repeat s n) returns a list containing n copies of s
```

```
; Examples
```

```
(check-expect (repeat 'Rabbit 0) empty)
(check-expect (repeat 'Goose 1) '(Goose))
(check-expect (repeat 'Rabbit 2) '(Rabbit Rabbit))
```

```
; Template instantiation:
```

```
; f : natural-number -> ...
```

```
; (define (repeat s n)
```

```
;   (cond [(zero? n) ...]
```

```
;         [(positive? n) ... (repeat s (sub1 n)) ...]))
```

```
; Code
```

```
(define (repeat s n)
```

```
  (cond [(zero? n) empty]
```

```
        [(positive? n) (cons s (repeat s (sub1 n)))]))
```



# More Examples

---

- `add: N N -> N`
- `multiply: N N -> N`
- `factorial: N -> N`
- Defining and using familiar functions on natural numbers helps us understand structural recursion (our design template for recursive mixed data definitions)



# Add

---

```
; Template Instantiation
```

```
(define (add m n)
  (cond [(zero? m) ...]
        [(positive? m) ... (add (sub1 m) n) ...]))
```

```
; Code
```

```
(define (add m n)
  (cond [(zero? m) n]
        [(positive? m) (add1 (add (sub1 m) n))]))
```

```
; Template Instantiation
```

```
(define (right-add m n)
  (cond [(zero? n) .. ..]
        [(positive? n) .. . (right-add m (sub1 n)) .. ..]))
```

```
; Code
```

```
(define (right-add m n)
  (cond [(zero? n) m]
        [(positive? n) (add1 (right-add m (sub1 n)))]))
```



# Defining Integers

---

An integer is either:

- 0; or
- (`add1` `n`) where `n` has the form 0 or (`add1` ...) [non-negative]; or
- (`sub1` `n`) where `n` has the form 0 or (`sub1` ...) [non-positive].

Recognizers:

- `zero?: any -> bool`
- `positive?: any -> bool`
- `negative?: any -> bool`

In Scheme, `add1` and `sub1` have been extended to all integers by defining for all integers `n` :

- `(add1 (sub1 n)) = n`
- `(sub1 (add1 n)) = n`



# For Next Class

---

- Homework due 10am, Friday. Submit it via OwlSpace.
- Reading: Chs. 11-13