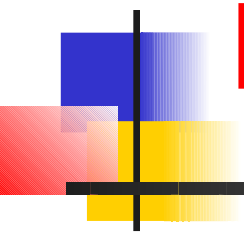
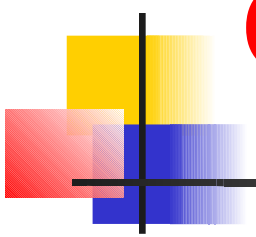


Data-directed Design

Corky Cartwright
Department of Computer Science
Rice University

Correction to Lecture 2

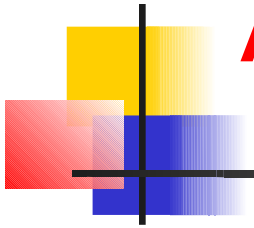


- In DrScheme, evaluate
  `(cond (else 17))`
- The evaluation rules for cond in Lecture 2 do not handle this case; the Lecture 2 slides have been revised to address this problem.
- Revised rules:

```
    (cond [false        result-1]
          ... )
=> (cond   ... )

    (cond [else         default-result])
=> default-result
```

- The new rules subsume the old ones (produce the same results for expressions handled by the old rules). Homework done using the old rules will still receive full credit

Addendum to Lecture 3

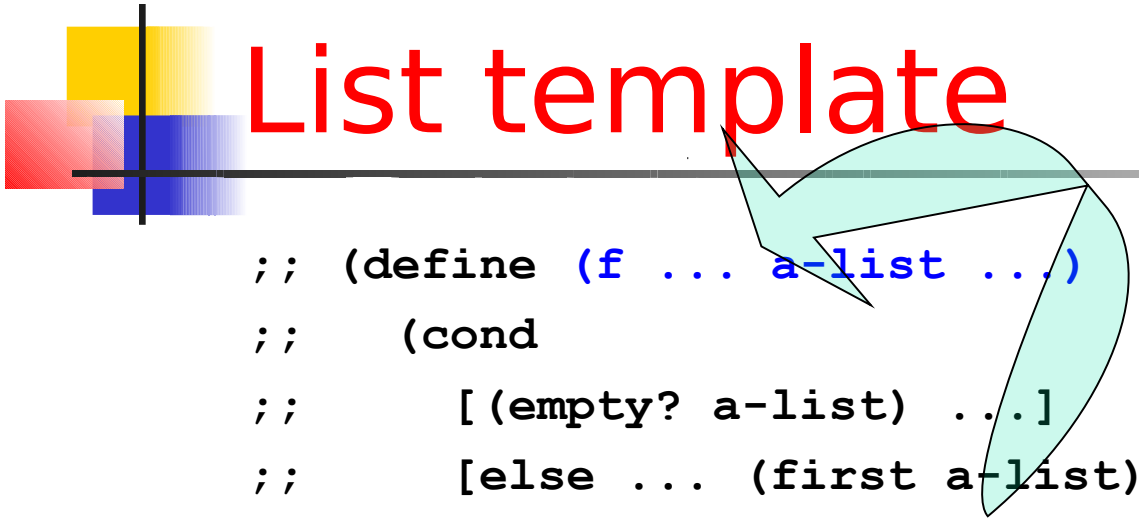


- Rules for evaluating **`if`** expressions

```
    (if true expr-1 expr-2) [
=> expr-1

    (if false expr-1 expr-2)
=> expr-2

    (if V expr-1 expr-2)   (V is non-boolean  value)
=> error: question result is not true or false
```

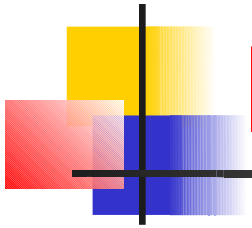
- Alternatively, we could expand an **`if`** expression into the equivalent **`cond`** expression, but this approach is clumsy.

From last lecture: List template



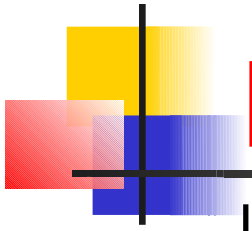
```
;; (define (f ... a-list ...)
;;   (cond
;;     [(empty? a-list) ...]
;;     [else ... (first a-list) ...
;;       ... (f ... (rest a-list) ...) ...]))
```

Template does not depend on element type. It applies to ***alpha*`-list`** 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 ***alpha*`-list`**. Henceforth, we will allow type variables like ***alpha*** in data definitions.



Plan for Today

- 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 ***e1***, ***e2***, …, ***en*** be Scheme expressions. Then

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

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

Let ***s1***, ***s2***, …, ***sn*** 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)
```

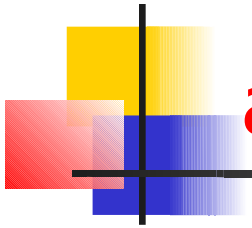
Do not nest quotation!

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

A simple list function of 2 list arguments

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

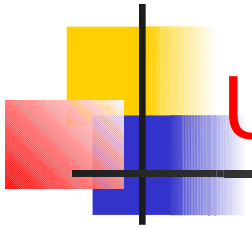
```
; app: list-of-alpha list-of-alpha -> list-of-alpha
; purpose: (app a b) concatenates the lists a and b.

; Examples
; Test:
  (check-expect (app '(a b) '(c d)) '(a b c d))
  (check-expect (app empty '(c d)) '(c d))
  (check-expect (app '(a b) empty) '(a b))
; Template Instantiation (which argument is principal?)
|#
  (define (app x y)
    (cond [(empty? x) ...]
          [(cons? x) ... (first x) ...
             (app (rest x) y) ... ]))
#|
```



append cont.

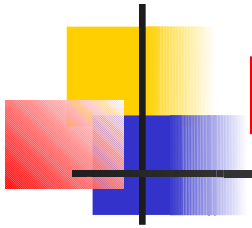
```
; Code:
  (define (app x y)
    (cond [(empty? x) y]
          [(cons? x)
           (cons (first x) (app (rest x) y)])))

; Test:
  (check-expect (app '(a b) '(c d)) '(a b c d))
  (check-expect (app empty '(c d)) '(c d))
  (check-expect (app '(a b) empty) '(a b))
```

- 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`**

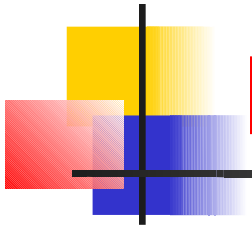


Defining `flatten`

```
;; 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))
```

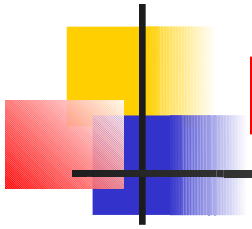
Recall that:

```
;; 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) ...) ...]))
```



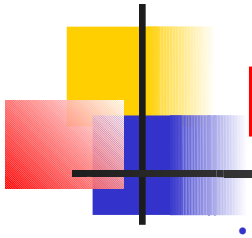
Defining `flatten`

```
;; 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)))]))
```



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

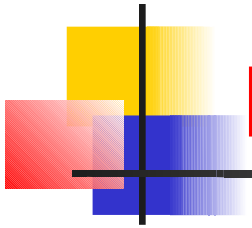
- Standard definition from mathematics

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

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

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

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

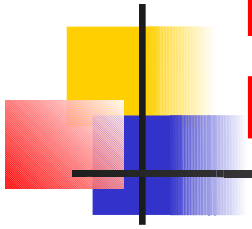


Examples and Basic Operations

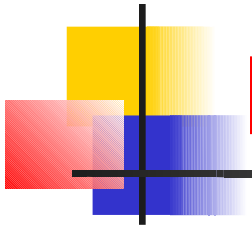
- Examples (using constructors)
  - Zero: `0`
  - One: `(add1 0)`
  - Four: `(add1 (add1 (add1 (add1 0))))`
- Accessors:
  - `sub1 : N -> N`

    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



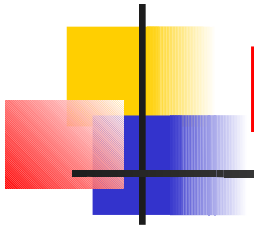
- Recall the ones for lists:
  - For all elements **`v`**, and lists **`l`**, we have
    - **`(empty? empty) = true`** ;; recognizer
    - **`(empty? (cons v l)) = false`**
    - **`(rest (cons v l)) = l`** ;; accessor
    - **`(first (cons v l)) = v`**
- Basic laws:
  - For all natural numbers `n`, 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) ...)
;;          ...]))
```



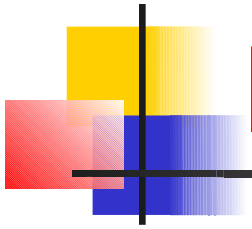
Example

- Write a function that repeats a symbol s several (n) times
- Examples

```
(repeat 'Rabbit 0) = empty
(repeat 'Rabbit (add1 (add1 0))) =
  (cons 'Rabbit (cons 'Rabbit empty))
```

- Code:

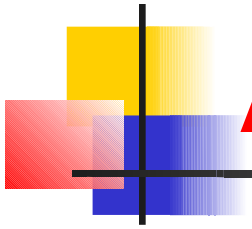
```
;; repeat : symbol natural-number -> list-of-symbol
(define (repeat s n)
 (cond [(zero? n) empty]
       [(positive? n)
        (cons s (repeat s (sub1 n)))]))
```



More Examples

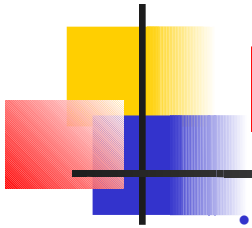
- **`add:  N N -> N`**
- **`multiply:  N N -> N`**
- **`factorial: N -> N`**

- Defining and using familiar functions on natural numbers helps us understand structural recursion (our design template)



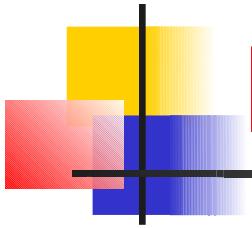
Add

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

(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 Friday
- Reading: Chs. 11-13
- Think about: what is the design template (structural recursion scheme) for integers? Hint: look at the inductive definition of integers on the previous slide.