Structure and Interpretation of Computer Programs

COMP200
HIGHER ORDER PROCEDURES

Using HOPs on Data Structures

(define first car)
(define rest cdr)
(define adjoin cons)
COMMON PATTERNS

cdr’ing and cons’ing examples

(define (copy lst)
  (if (null? lst) nil ; base case
   (adjoin (first lst) ; recursion
     (copy (rest lst))))))
COMMON PATTERNS

cdr’ing and cons’ing examples

(define (append list1 list2)
  (cond ((null? list1) list2) ; base case
        (else
          (adjoin (first list1); recursion
           (append (rest list1) list2))))))

(append (list 1 2) (list 3 4)) ==> (1 2 3 4)
COMMON PATTERN #1
Transforming a List

(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst))))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst))))))
COMMON PATTERN #1
Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst))))
)

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst))))))
```
COMMON PATTERN #1

Transforming a List

(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))
COMMON PATTERN #1
Transforming a List

(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
               (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
               (double-list (rest lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (adjoin (proc (first lst))
               (map proc (rest lst))))
COMMON PATTERN #1
Transforming a List

(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (adjoin (proc (first lst))
              (map proc (rest lst)))))

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

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
COMMON PATTERN #2
Filtering a List
COMMON PATTERN #2
Filtering a List

(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (first lst))
          (adjoin (first lst)
                   (filter pred (rest lst)))))
        (else (filter pred (rest lst))))
COMMON PATTERN #2
Filtering a List

(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (first lst))
         (adjoin (first lst)
                  (filter pred (rest lst))))
        (else (filter pred (rest lst))))

(filter even? (list 1 2 3 4 5 6)) ;Value: (2 4 6)
COMMON PATTERN #3

Accumulating Results
COMMON PATTERN #3

Accumulating Results

(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
          (add-up (rest lst))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
          (mult-all (rest lst))))
COMMON PATTERN #3
Accumulating Results

(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
          (add-up (rest lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
          (mult-all (rest lst)))))

(define (FOLD-RIGHT op init lst)
  (if (null? lst)
      init
      (op (first lst)
           (fold-right op init (rest lst))))
COMMON PATTERN #3
Accumulating Results

(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
          (add-up (rest lst))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
          (mult-all (rest lst))))

(define (FOLD-RIGHT op init lst)
  (if (null? lst)
      init
      (op (first lst)
           (fold-right op init (rest lst))))

(define (add-up lst)
  (fold-right + 0 lst))

(define (mult-all lst)
  (fold-right * 1 lst))
HIGHER ORDER PROCEDURES

Using HOPs on Data Structures

\[ \sum_{i=0}^{n} f(a + i\delta) = f(a) + f(a + \delta) + f(a + 2\delta) + \cdots + f(a + n\delta) \]
\[ \sum_{i=0}^{n} f(a + i \delta) \]

**HIGHER ORDER PROCEDURES**

*Using HOPs on Data Structures*

```
(define (generate-interval a b)
  (if (> a b)
      nil
    (cons a (generate-interval (+ 1 a) b))))
```
\[ \sum_{i=0}^{n} f(a + i\delta) \]

**HIGHER ORDER PROCEDURES**

*Using HOPs on Data Structures*

```scheme
(define (generate-interval a b)
  (if (> a b)
      nil
      (cons a (generate-interval (+ 1 a) b))))

(define (sum f start inc terms)
  (fold-right + 0
    (map (lambda (x) (f (+ start (* x inc))))
         (generate-interval 0 terms))))
```