Lecture 05
Lists and recursion

T. METIN SEZGIN

1

Announcements

1. New etutor assignment coming (Oct 19^{th})
2. Reading SICP 1.2 (pages 79-126)

2

2

Lecture 04 – review
Structures and Patterns in Functional Programming

T. METIN SEZGIN

3

Lecture Nuggets

- Order of growth matters
- Support for compound data allows data abstraction
  - Pairs
  - Lists
  - Others
- Two main patterns when dealing with lists
  - Consing up – to build
  - Cdring down – to process

4

Nugget

Order of growth matters

5

Iterative and Recursive versions of fact

6

```
;; RECURSIVE

(define (fact-r x)
  (if (= x 0) 1 (* x (fact-r (- x 1)))))


;; ITERATIVE
(define (fact-i x)
  (fact-i-helper 1 1 x))

(define fact-i-helper
  (lambda (product counter n)
    (if(> counter n)
       product
       (fact-i-helper (* product counter) (+ counter 1) n))))
```

COMP 301 SICP 10/13/2021

6

Examples of orders of growth

- FACT
  - Space Θ(n) – linear
  - Time Θ(n) – linear

- IFACT
  - Space Θ(1) – constant
  - Time Θ(n) – linear

7

Nugget

Support for compound data allows data abstraction

8

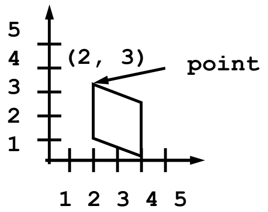
Pairs (cons cells)

- **(cons <x-exp> <y-exp>) ==> <P>**
  - 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>**
- **(car <P>) ==> <x-val>**
  - Returns the car-part of the pair **<P>**
- **(cdr <P>) ==> <y-val>**
  - Returns the cdr-part of the pair **<P>**

Compound Data

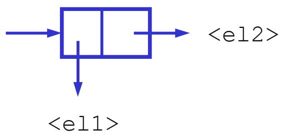
- Treat a PAIR as a single unit:
  - Can pass a pair as argument
  - Can return a pair as a value

```
(define (make-point x y)
  (cons x y))

(define (point-x point)
  (car point))

(define (point-y point)
  (cdr point))

(define (make-seg pt1 pt2)
  (cons pt1 pt2))

(define (start-point seg)
   (car seg))
```

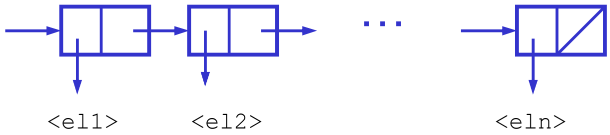


Conventional Interfaces - Lists

`(cons <el1> <el2>)`



`(list <el1> <el2> ... <eln>)`



Predicate
(null? <z>)
==> #t if **<z>** evaluates to empty list

6.001 SICP

11/38

11

Lecture 05
Lists and recursion

T. METIN SEZGIN

12

Lecture Nuggets

- Two main patterns when dealing with lists
  - Consing up – to build
  - Cdring down – to process
- Higher order procedures
- Three more patterns for lists
  - Transforming
  - Filtering
  - Accumulating

13

Nugget

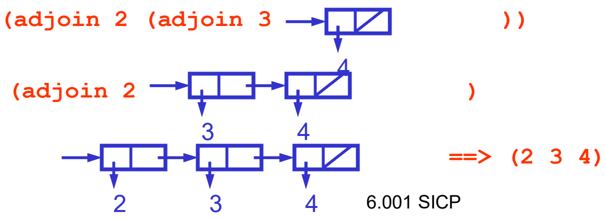
Two patterns for dealing with lists

14

Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
             (enumerate-interval
               (+ 1 from)
               to))))
```

```
(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))
```

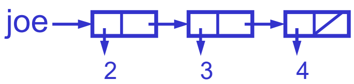


6.001 SICP

15/38

Common Pattern #2: cdr'ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (first lst)
      (list-ref (rest lst)
                (- n 1))))
```



`(list-ref joe 1)`

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

6.001 SICP

16/38

Nugget

Higher order procedures

17

Other common patterns

- 1 + 2 + … + 100 = (100 * 101)/2 ← $\sum_{k=1}^{100} k$
- 1 + 4 + 9 + … + 100^2 = (100 * 101 * 201)/6 ← $\sum_{k=1}^{100} k^2$
- 1 + $1/3^2$ + $1/5^2$ + … + $1/101^2$ = $\pi^2/8$ ← $\sum_{k=1,odd}^{101} k^{-2}$

```
(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)
  (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ a 2) b))))
```

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

10/13/2021 6.001 SICP 18/53

18

Let's check this new procedure out!

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

A higher order procedure!!

What is the type of this procedure?

$$\underbrace{(\underbrace{\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}, \underbrace{\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}) \rightarrow \text{number}}_{\text{procedure}}$$

Higher order procedures

- A higher order procedure:
  takes a procedure as an argument or returns one as a value

```
(define (sum-integers1 a b)
        (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (sum-squares1 a b)
        (sum square a (lambda (x) (+ x 1)) b))
(define (pi-sum1 a b)
      (sum (lambda (x) (/ 1 (square x))) a
           (lambda (x) (+ x 2)) b))
```

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

Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (cons (square (car lst))
            (square-list (cdr lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (cons (* 2 (car lst))
            (double-list (cdr lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (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 #2: Filtering a List

```
(define (keep-it-odd lst)
 (cond ((null? lst) nil)
       ((odd? (car lst))
        (cons (car lst) (keep-it-odd (cdr lst))))
       (else (keep-it-odd (cdr lst)))))

(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

Common Pattern #3: Accumulating Results

```
(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 (REDUCE op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (reduce op init (cdr lst)))))

(define (add-up lst)
  (reduce + 0 lst))
```