Lecture 04
Structures and Patterns in Functional Programming

T. METIN SEZGIN

1

Announcements

1. Assignment due on Friday
2. Reading SICP 1.2 (pages 31-50)
3. Etutor assignment due Friday 8th
4. Labs (PSes) start this week -- Online

2

2

Lecture 3 – Review
Functional Programming

T. METIN SEZGIN

3

Lecture Nuggets

- Lambda expressions creates procedures
  - Formal parameters
  - Body
  - Procedures allow creating abstractions
- We can solve problems by creating functions
- The substitution model is a good mental model of an interpreter

4

Controlling the process

```
(define sqrt
    (lambda (x)
        (sqrt-loop 1.0 x)))

(define sqrt-loop (lambda G X)
   (if (close-enuf? G X)
       G
       (sqrt-loop (improve G X) X ) ) )
```


5

Nugget

The substitution model is a good mental model of an interpreter

6

Iterative and Recursive versions of fact

```
;; 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))))
```



```
(define fact(lambda (n)
  (if (= n 1)1(* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step

first row handles 0! cleanly

product	counter	N
1	1	4
1	2	4
2	3	4
6	4	4
24	5	4

product * counter

counter +1

answer

- The last row is the one where counter > n
- The answer is in the product column of the last row

Iterative factorial in scheme

- (define ifact (lambda (n) (ifact-helper 1 1 n)))

initial row of table

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

compute next row of table

answer is in product column of last row

at last row when counter > n

Partial trace for `(ifact 4)`

```
(define ifact-helper (lambda (product count n)
      (if (> count n) product
          (ifact-helper (* product count)
                        (+ count 1) n))))

(ifact 4)
(ifact-helper 1 1 4)
(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))
(ifact-helper 1 2 4)
(if (> 2 4) 1 (ifact-helper (* 1 2) (+ 2 1) 4))
(ifact-helper 2 3 4)
(if (> 3 4) 2 (ifact-helper (* 2 3) (+ 3 1) 4))
(ifact-helper 6 4 4)
(if (> 4 4) 6 (ifact-helper (* 6 4) (+ 4 1) 4))
(ifact-helper 24 5 4)
(if (> 5 4) 24 (ifact-helper (* 24 5) (+ 5 1) 4))
24
```


Iterative = no pending operations when procedure calls itself

- Recursive factorial:

```
(define fact (lambda (n)
    (if (= n 1) 1
        (* n (fact (- n 1)) )
    )))
```

pending operation

- `(fact 4)`

```
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
```

- Pending ops make the expression grow continuosly

Iterative = no pending operations

- Iterative factorial:

```
(define ifact-helper (lambda (product count n)
     (if (> count n) product
        (ifact-helper (* product count)
                      (+ count 1) n))))
```

- (ifact-helper 1 1 4)
  (ifact-helper 1 2 4)
  (ifact-helper 2 3 4)
  (ifact-helper 6 4 4)
  (ifact-helper 24 5 4)

no pending operations

- Fixed size because no pending operations

COMP 301 SICP

13

13

Lecture 04
Structures and Patterns in Functional Programming

T. METIN SEZGIN

14

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

15

Nugget

Order of growth matters

16

Orders of growth of processes

- Suppose `n` is a parameter that measures the size of a problem
- Let `R(n)` be the amount of resources needed to compute a procedure of size `n`.
- We say `R(n)` has order of growth Θ(`f(n)`) if there are constants k_1 and k_2 such that $k_1 f(n) \le R(n) \le k_2 f(n)$ for large `n`
- Two common resources are space, measured by the number of deferred operations, and time, measured by the number of primitive steps.

Examples of orders of growth

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

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

Nugget

Support for compound data allows data abstraction

19

Language Elements

- Primitives
  - prim. data: numbers, strings, booleans
  - primitive procedures
- Means of Combination
  - procedure application
  - compound data (today)
- Means of Abstraction
  - naming
  - compound procedures
    - block structure
    - higher order procedures (next time)
  - conventional interfaces – lists (today)
  - data abstraction

6.001 SICP

20/38

20

Compound data

- Need a way of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of getting the pieces back out

•Need a contract between the “glue” and the “unglue”

•Ideally want the result of this “gluing” to have the property of **closure:**

•“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”

6.001 SICP 21/38

21

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>`

6.001 SICP 22/38

22

Compound Data

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

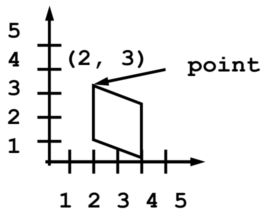
```
(define (make-point x y)
  (cons x y))

(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))
```



23

Pair Abstraction

- Constructor

```
; cons: A,B -> A X B
; 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> evaluates to a pair, else #f
```


24

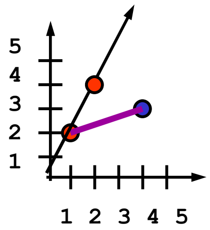
Pair abstraction

- Note how there exists a contract between the constructor and the selectors:
  - *(car (cons <a> <b>))* ➔ <a>
  - *(cdr (cons <a> <b>))* ➔ <b>
- Note how 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)*

Using pair abstractions to build procedures

- Here are some data abstractions

```
(define p1 (make-point 1 2))
(define p2 (make-point 4 3))
(define s1 (make-seg p1 p2))
```




```
(define stretch-point
  (lambda (pt scale)
    (make-point (* scale (point-x pt))
                (* scale (point-y pt)))))
(stretch-point p1 2) ➔ (2 . 4)
p1 ➔ (1 . 2)
```

Grouping together larger collections

- Suppose we want to group together a set of points. Here is one way

```
(cons (cons (cons (cons p1 p2)
                  (cons p3 p4))
            (cons (cons p5 p6)
                  (cons p7 p8)))
      p9)
```

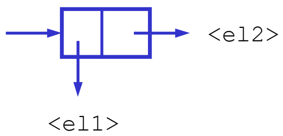
- **UGH!!** How do we get out the parts to manipulate them?

Conventional interfaces -- lists

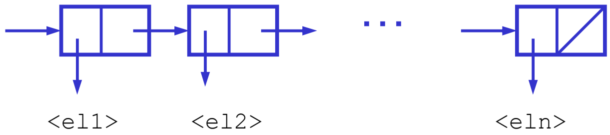
- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
  - Car-part of a pair in sequence – holds an item
  - Cdr-part of a pair in sequence – holds a pointer to rest of list
  - Empty-list **`nil`** – signals no more pairs, or end of list
- Note that lists are closed under operations of **`cons`** and **`cdr`**.

Conventional Interfaces - Lists

(cons <el1> <el2>)



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



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

6.001 SICP

29/38

29

… to be really careful

- For today we are going to create different constructors and selectors for a list
  - **(define first car)**
  - **(define rest cdr)**
  - **(define adjoin cons)**
- Note how these abstractions inherit closure from the underlying abstractions!

6.001 SICP

30/38

30

Nugget

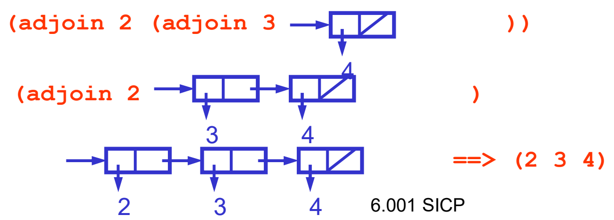
Two patterns for dealing with lists

31

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)))
```



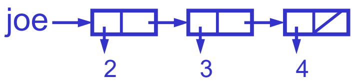
6.001 SICP

32/38

32

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)))))
```