CS383 Tutorial 8

Xiao Jia

xjia@cs.sjtu.edu.cn

Introduction to Scheme

Expressions and lists

- Arithmetic expressions
- Defining names
- Procedures
- Special forms: if, cond
- Applicative order versus normal order
- Example: square roots by Newton’s method
- Lists: cons, car, cdr

Arithmetic expressions

```
486                ➔ 486
(+ 137 349)        ➔ 486
(- 1000 334)       ➔ 666
(* 5 99)           ➔ 495
(/ 10 5)           ➔ 2
(+ 2.7 10)         ➔ 12.7
(+ 21 35 12 7)     ➔ 75
(* 25 4 12)        ➔ 1200
```

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

Defining names

```
(define size 2)

size         ➔ 2

(* 5 size)   ➔ 10

(define pi 3.14159)

(define radius 10)

(* pi (* radius radius))  ➔ 314.159

(define circumference (* 2 pi radius))
```

Defining procedures

```
(define (square x) (* x x))

(square 21)                    ➔ 441

(square (+ 2 5))               ➔ 49

(square (square 3))            ➔ 81

(define (sum-of-squares x y)

  (+ (square x) (square y)))

(sum-of-squares 3 4)           ➔ 25

(define (<name> <formal parameters>) <body>)
```

if

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

```
(if <predicate> <consequent> <alternative>)
```

cond

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      …
      (<pn> <en>))
```

Applicative-order evaluation

```
(sum-of-squares (+ 5 1) (* 5 2))

(sum-of-squares 6 10)

(+ (square 6) (square 10))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

Evaluate the arguments and then apply

Normal-order evaluation

```
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1))    (square (* 5 2))   )
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6)             (* 10 10))
(+ 36                  100)
136
```

Do not evaluate the operands until necessary

Square roots by Newton's method

sqrt(x) is the y such that y ≥ 0 and $y^2 = x$

Newton's method:

- given a guess y, average y and x/y to get a better guess

Guess	Quotient	Average
1	(2/1) = 2	((2 + 1)/2) = 1.5
1.5	(2/1.5) = 1.3333	((1.3333 + 1.5)/2) = 1.4167
1.4167	(2/1.4167) = 1.4118	((1.4167 + 1.4118)/2) = 1.4142
1.4142	…	…

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

(define (sqrt x)
  (sqrt-iter 1.0 x))
```

Lists

(cons x y) ➔ (x . y)

(car (cons x y)) ➔ x *"head"*

(cdr (cons x y)) ➔ y *"tail"*

A sequence 1, 2, 3, 4 ⇔ (cons 1 (cons 2 (cons 3 (cons 4 ()))))

where () is pronounced as "unit"

Shorter version: (list 1 2 3 4)

(cdr (list 1 2 3 4)) ➔ (2 3 4)

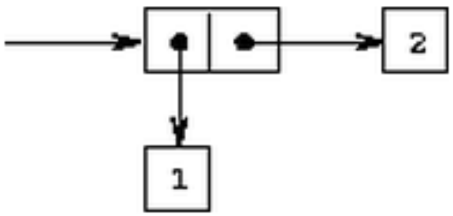


Figure 2.2: Box-and-pointer representation of `(cons 1 2)`.

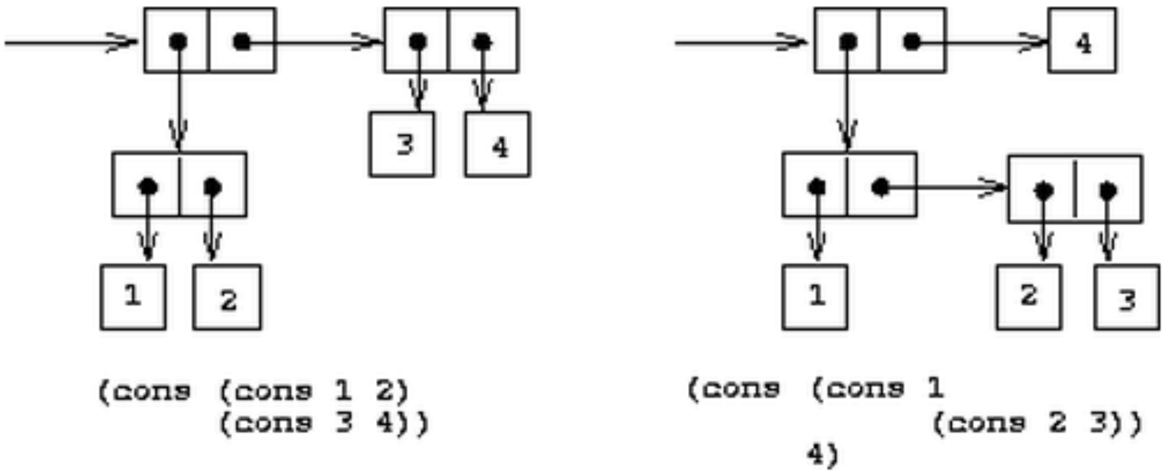


Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

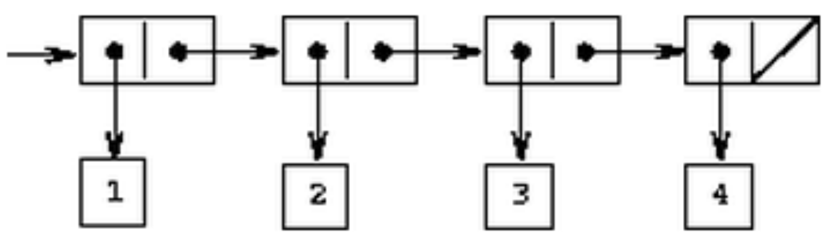


Figure 2.4: The sequence 1, 2, 3, 4 represented as a chain of pairs.

```scheme
(define (sum items)
  (+ (car items)
     (sum (cdr items))))

(sum (list 1 2 3))
(sum (1 2 3))
(+ (car (1 2 3)) (sum (cdr (1 2 3))))
(+ 1 (sum (2 3)))
(+ 1 (+ (car (2 3)) (sum (cdr (2 3)))))
(+ 1 (+ 2 (sum (3))))
(+ 1 (+ 2 (+ (car (3)) (sum (cdr (3))))))
(+ 1 (+ 2 (+ 3 (sum ()))))
(+ 1 (+ 2 (+ 3 (+ (car ()) (sum (cdr ()))))))
```

Error: Cannot apply car/cdr on the unit ()

```
(define (sum items)
  (+ (car items)
     (sum (cdr items))))

(define (sum items)
  (if (null? items)
      0
      (+ (car items)
         (sum (cdr items)))))

(sum (list 1 2 3)) ➔ 6
```

Recursive style:

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
```

Iterative style:

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter items 0))
```

How to represent a tree?

- A binary tree
- An arbitrary tree

Define some utility functions

- height
- leaves

Functions and streams

- Linear recursion
- Tree recursion
- Higher-order functions
- Anonymous functions
- Local names
- Infinite streams

Linear recursion

recursive style ➔ linear space

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Linear recursion

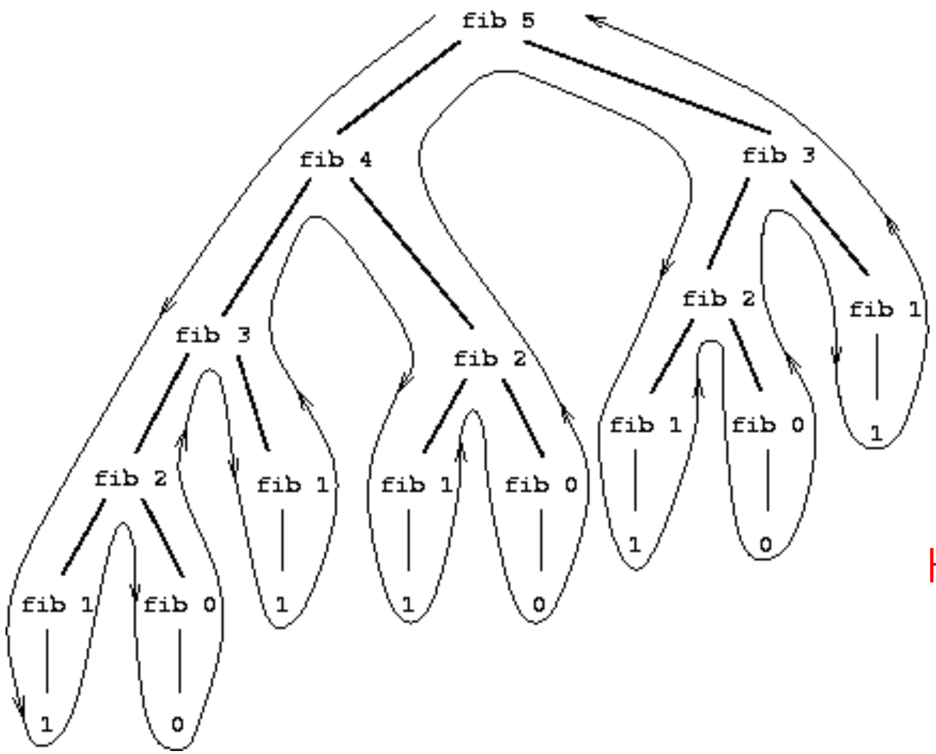
iterative style → constant space

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Tree recursion

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```



How to compute it with a linear iteration?

Higher-order functions

```
(define (square x) (* x x))
(define (map f a)
  (if (null? a)
      ()
      (cons (f (car a))
            (map f (cdr a)))))
(map square (list 1 2 3)) ➔ (1 4 9)
```

Treat a function as a value

Anonymous functions

```
(define (square x) (* x x))
```

… and functions are values …

```
(define square
  (lambda (x) (* x x)))
```

closures

```
(map (lambda (x) (+ x 10))
     (list 1 2 3))
```

➔ (11 12 13)

Local names

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

$$\begin{aligned} a &= 1 + xy \\ b &= 1 - y \\ f(x,y) &= xa^2 + yb + ab \end{aligned}$$

```
(define (f x y)
  ((lambda (a b)
     (+ (* x (square a))
        (* y b)
        (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

```
(define (f x y)
  (let ((a  (+ 1 (* x y)) )
        (b  (- 1 y)        ))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

Infinite streams

```
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))


(numbers-from 10)
(cons 10 (numbers-from (+ 10 1)))
(cons 10 (numbers-from 11))
(cons 10 (cons 11 (numbers-from (+ 11 1))))
```

Infinite streams

```
(define (numbers-from n)
  (cons n
        (lambda ()
          (numbers-from (+ n 1)))))
```

delay evaluation

```
(numbers-from 10) ➔ (10 . #<Closure>)
```

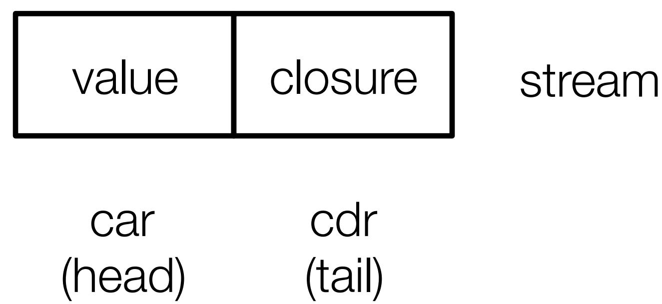
We need to define a tail function for infinite streams

Infinite streams

```
(define (tail stream)
  ((cdr stream)))
```

closure

closure application




```
(numbers-from 10)        ➔ (10 . #<Closure>)
(tail (numbers-from 10)) ➔ (11 . #<Closure>)
```

Infinite streams

```
(define (nth stream n)
  (if (= n 1)
      (car stream)
      (nth (tail stream) (- n 1))))
(nth (numbers-from 10) 1)   ➔ 10
(nth (numbers-from 10) 2)   ➔ 11
(nth (numbers-from 10) 100) ➔ 109
```

Infinite streams

```
(define (make-stream a)
  (if (null? a)
      ()
      (cons (car a)
            (lambda ()
              (make-stream (cdr a))))))
(make-stream (list 1 2 3)) ➔ (1 . #<Closure>)
```

Infinite streams

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (lambda ()
              (append (tail x) y)))))
(nth (append (make-stream (list 1 2 3))
             (numbers-from 10))
     4)
➔ 10
```

Infinite streams

```
(define (interleave x y)
  (if (null? x)
      y
      (cons (car x)
            (lambda ()
              (interleave y (tail x))))))
(nth (interleave (numbers-from 1)
                 (numbers-from 10))
     5)
➔ 3
```

Infinite streams

```
(define (map f stream)
  (if (null? stream)
      ()
      (cons (f (car stream))
            (lambda ()
              (map f (tail stream))))))
(map square (numbers-from 10))
➔ (100 . #<Closure>)
```

Infinite streams

```
(define (filter p stream)
  (if (null? stream)
      ()
      (if (p (car stream))
          (cons (car stream)
                (lambda ()
                  (filter p (tail stream))))
          (filter p (tail stream)))))
(filter (lambda (x) (> x 100)) (numbers-from 10))
➔ (101 . #<Closure>)
```

How to represent an infinite tree?

Search strategies and infinite lists

```
(define (depth-first next x)
  (define (dfs y)
    (if (null? y)
        ()
        (cons (car y)
              (lambda ()
                (dfs (@ (next (car y))
                        (cdr y)))))))
  (dfs (list x)))
```

@ appends two finite lists

next t is the list of the sub-trees of t

Search strategies and infinite lists

```
(define (breadth-first next x)
  (define (bfs y)
    (if (null? y)
        ()
        (cons (car y)
              (lambda ()
                (bfs (@ (cdr y)
                        (next (car y))))))))
  (bfs (list x)))
```

@ appends two finite lists

next t is the list of the sub-trees of t

Search strategies and infinite lists

Write versions of `depth-first` and `breadth-first` with an additional argument: a predicate to recognize solutions.

Solve the Eight Queens problem.

Questions?

Implementing Functional Data Structures

Stack

```
(define  empty-stack      ‘())
(define  (empty? st)      (null? st))
(define  (push elem st)  (cons elem st))
(define  (pop st)         (cdr st))
(define  (top st)         (car st))


(top (pop (push 1 (push 2 empty-stack))))
```

Queue in O(n) time

```
(define  empty-queue     ‘())
(define  (empty? q)      (null? q))
(define  (push elem q)   (append q (list elem)))
(define  (pop q)         (cdr q))
(define  (front q)       (car q))
(define  (rear q)        (car (reverse q)))
```

```
(define (reverse lst)
  (define (rev lst acc)
    (if (null? lst)
        acc
        (rev (cdr lst)
             (cons (car lst) acc))))
  (rev lst '()))
```

Queue in O(1) time

```
(define  empty-queue            (cons ‘() ‘()))
(define  (front-elements q)  (car q))
(define  (rear-elements q)   (cdr q))
(define  (empty? q)  (null? (front-elements q)))
```

Edge cases:

- Empty queue: `(() ())`
- A queue with 1 element: `((1) ())`
- A queue with 2 elements: `((1) (2))`

Queue in O(1) time

Edge cases:

- Empty queue: `(() ())`
- A queue with 1 element: `((1) ())`
- A queue with 2 elements: `((1) (2))`

Properties:

- `(front-elements q)` is empty iff `q` is empty
- `(rear-elements q)` is empty iff `q` is empty or only contains 1 element

First try (violate properties)

```
(define (push elem q)
  (cons (front-elements q)
        (cons elem (rear-elements q))))
(define (pop q)
  (cons (cdr (front-elements q))
        (rear-elements q)))
(define (front q) (car (front-elements q)))
(define (rear q)  (car (rear-elements q)))
```

Re-implement push

```
(define (push0 elem q)
  (cons (front-elements q)
        (cons elem (rear-elements q))))

(define (push elem q)
  (if (empty? q)
      (cons (list elem) '())
      (push0 elem q)))
```

Re-implement pop

```
(define (pop0 q)
  (cons (cdr (front-elements q))
        (rear-elements q)))

(define (pop q) (fix (pop0 q)))
```

Fix a queue state

```
(define (fix q)
  (define r (rear-elements q))
  (if (null? (front-elements q))
      (if (null? r) empty-queue
                    (if (null? (cdr r))
                        (cons r ‘())
                        (cons (reverse (cdr r))
                              (list (car r)))))
      q))
```

leads to an amortized complexity of O(1)

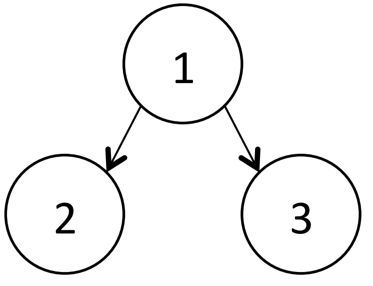
Re-implement rear

```
(define (rear0 q) (car (rear-elements q)))


(define (rear q)
  (if (null? (rear-elements q))
      (car (front-elements q))
      (rear0 q)))
```

q is empty or is of size 1

Binary tree

```
(define  (empty? t)    (null? t))

(define  (tree v l r)  (cons v (cons l r)))

(define  (value t)     (car t))

(define  (left t)      (car (cdr t)))

(define  (right t)     (cdr (cdr t)))


(tree 1 (tree 2 ‘() ‘())
        (tree 3 ‘() ‘()))
```



Binary search tree

```
(define (member? t x)
  (if (empty? t)
      #f
      (cond ((< x (value t))
              (member? (left  t) x))
             ((> x (value t))
              (member? (right t) x))
             (else #t))))
```

Binary search tree

```
(define (insert t x)
  (if (empty? t)
      (tree x ‘() ‘())
      (cond ((< x (value t))
             (tree (value t) (insert (left t) x)
                             (right t)))
            ((> x (value t))
             (tree (value t) (left t)
                             (insert (right t) x)))
            (else t))))
```

BST as a set

```
(define  empty-set      ‘())
(define  (member? s x)  …)
(define  (insert s x)   …)
```

From sets to maps

Values of the tree nodes can be a pair (key, value)

Function as a set

```
(define  empty-set        (lambda (x) #f))
(define  (member? s x)  (s x))
(define  (insert s x)   (if (member? s x)
                             s
                             (lambda (y)
                               (if (= x y)
                                   #t
                                   (s y))))))
```

Function as a map

```
(define  empty-map          (lambda (x) error))
(define  (lookup s x)       (s x))
(define  (insert s x v)     (lambda (y)
                              (if (= x y)
                                  v
                                  (s y))))
```

Questions?

Introduction to Prolog

Prolog can be separated in two parts:

1. The Program
2. The Query

Program:

```
sunny.
```

Query:

```
?- sunny.
```

How to query

```
eats(fred, oranges).
eats(tony, apple).
eats(john, apple).
?- eats(fred, oranges).
    yes
?- eats(john, apple).
    yes
?- eats(mike, apple).
    no
```

Variables

```
eats(fred, oranges).
eats(tony, apple).
eats(john, apple).
?- eats(fred, What).
    What = oranges
    yes
?- eats(Who, oranges).
    Who = fred
    yes
```

Example: Book Database

```
book(1, title1, author1).

book(2, title2, author1).

book(3, title3, author2).

book(4, title4, author3).

?- book(_, _, author2).     /* If we have a book from author2 ? */

      yes

?- book(_, X, author1).     /* Which book from author1 we have? */

      X = title1 ;

      X = title2 ;
```

Arithmetic

```
?- X is 3 + 4.

    X = 7

    yes
```

Predefined operators:

=, is, <, >, =<, >=, ==, =:=, =/=, /, *, +, -, mod, div

Rules

```
mortal(X) :- human(X).  /* X is mortal if X is human */

human(alice).

human(bob).

?- mortal(alice).

    yes

?- mortal(X).

    X = alice ;

    X = bob ;
```

Recursion

```
parent(john, paul).

parent(paul, tom).

parent(tom, mary).

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

?- ancestor(john, tom).

    yes
```

Example: Factorial

```
factorial(0, 1).

factorial(X, Y) :-   X1 is X - 1,
                     factorial(X1, Z),
                     Y is Z*X, !.

?- factorial(5, W).
    W = 120
    yes
```

*If X1 is X-1, Z is the factorial of X1, and Y is Z*X, then Y is the factorial of X.*

Lists

[item1, item2, item3, item4]

[Head | Tail]

?- [X | Y] = [a, b, c, d, e].

X = a

Y = [b, c, d, e]

?- [X | Y] = [ ].

no

?- [Fst, Snd | Rest] = [a,b,c,d].

Fst = a

Snd = b

Rest = [c, d]

member

```
member(T, [T | Q]).

member(X, [T | Q]) :- member(X, Q).

?- member(X, [apple, banana, peach]).

    X = apple ;

    X = banana ;

    X = peach ;

    no
```

append

```
append([ ], X, X).

append([T | Q1], X, [T | Q2]) :- append(Q1, X, Q2).

?- append([a, b, c], [d, e, f], X).

     X = [a, b, c, d, e, f]
```

Questions?

https://sites.google.com/site/prologsite/prolog-problems