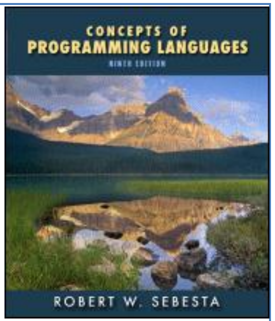
Scheme



1. Ch15 – Functional Programming Languages
   - 15.4 LISP
   - 15.5 Scheme
2. Scheme
   - The Revised^6 Report on the Algorithmic Language Scheme
   - Download Petite Chez Scheme
3. Reference
   - The Scheme Programming Language, 4th ed., Kent Dybvig

Read-eval-print loop

- REPL

```
> 2                       ; a constant evaluates to itself
2
> (define x 5)            ; unspecified value
> (+ x 6 7)               ; prefix notation, variable arguments
18
> (- (* 2 x) (/ x 2))     ; 2x - x/2, rational number
15/2
> (quotient x 2)          ; (remainder x 2) ⇒ 1;   (/ x 2.0) ⇒ 2.5
2
> (sqrt -1)               ; complex number
0+1i
```

Forms of expressions

- Self-evaluation forms (i.e. constants)

2	⇒ 2	#t	⇒ #t	#\c	⇒ #\c
3.4	⇒ 3.4	#f	⇒ #f	"str"	⇒ "str"
1/2	⇒ 1/2				
1+2i	⇒ 1+2i				

- Function applications

(+ 2 3)	⇒ 5	(+)	⇒ 0	(and)	⇒ #t
		(*)	⇒ 1	(or)	⇒ #f

- Special forms

(define x (* 2 3))	⇒ unspecified
(if (< x 3) (+ 2 3) (* 2 3))	⇒ 6
(if (< x 3) (+ 2 3)	⇒ unspecified

Define and set! expressions

- Define expression

  Primarily used for creating global bindings

  May also be used to modify global bindings

- Set! expression

  Set! is the assignment expression.

  The variable to be modified must exist.

  (define x 1)

  (define x "snoopy")

  (set! x "snoopy") ⇒ unspecified

  (set! y "pluto") ⇒ error (Chez Scheme fails)

  N.B. Scheme is an impurely, untyped functional language.

Function

- Lambda expression (e.g. $\lambda x.x+1$ in lambda calculus)
  - (lambda (formals) body)
  - (lambda (x) (+ x 1)) ⇒ #<procedure>

    ((lambda (x y) (+ x y)) 2 3) ⇒ 5

    ((lambda () 6)) ⇒ 6
- Named function
  - (define f (lambda (x y) (+ x y)))

    (define (f x y) (+ x y)) ; shorthand notation

    f ⇒ #<procedure:f>

    (f 2 3) ⇒ 5

Function

- Recursive function
  - (define f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))
  - (define sum ; digit-sum of a natural number

```
(define sum                      ; digit-sum of a natural number
  (lambda (n)
    (if (<= 0 n 9)               ; or, (and (<= 0 n) (<= n 9))
        n
        (+ (remainder n 10) (sum (quotient n 10))))))
```

N.B. (= x_1 x_2 x_3 …) ; equailty

(< x_1 x_2 x_3 …) ; increasing

(> x_1 x_2 x_3 …) ; decreasing

(<= x_1 x_2 x_3 …) ; nondecreasing

(>= x_1 x_2 x_3 …) ; nonincreasing

Conditional expression

- Cond expression
  - (cond (test1 exp1 ...) (test2 exp2 ...) ... (else exp ...))
  - (define pow

```
(define pow
    (lambda (a n)
        (cond ((= n 0) 1)
              ((odd? n) (* a (pow a (- n 1))))
              (else (pow (* a a) (quotient n 2))))))

; or, (not (even? n))        ; or, (/ n 2), since n is even
```

Let expression

- Let expression
  - (let ((var init) …) body)

    The scope of var covers the body only.

    For local variables and non-recursive functions
  - (define x 2)

    ```
    (let ((x 3) (y x)) (+ x y))            ⇒ 5
    ((lambda (x y) (+ x y)) 3 x)           ; equivalence
    ```

    ```
    (define f (lambda (x) (+ x 1)))
    (let ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
        (f 5))                             ⇒ 25
    ```

Let expression

- Let* expression
  - (let* ((var init) …) body)

    The scope of var covers the body and the init's to its right.

    For local variables and non-recursive functions
  - (define x 2)

```
(let* ((x 3) (y x)) (+ x y))            ⇒ 6
(let ((x 3)) (let ((y x)) (+ x y))      ; equivalence

(define f (lambda (x) (+ x 1)))
(let* ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
    (f 5))                              ⇒ 25
```

Let expression

- Letrec expression
  - (letrec ((var init) …) body)

    The scope of var covers the body and all the init's.

    For local (mutual) recursive functions
  - (letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))

        (f 5))   ⇒ 120

    (letrec ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))

                (odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))))

        (even? 5))   ⇒ #f

    (letrec ((x x)) x)   ⇒ unspecified

Let expression

- Named let expression (shorthand for letrec)
  - (let fun ((formal actual) …) body)

    ≡ ((letrec ((fun (lambda (formal …) body)))

    fun) actual …)

    If fun doesn't occur free in actual's, it can be simplified to

    ≡ (letrec ((fun (lambda (formal …) body)))

    (fun actual …))
  - (let f ((n 5)) (if (= n 0) 1 (* n (f (- n 1)))))

    ≡ (letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))

    (f 5))

    ≡ ((letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))

    f) 5)

Let expression

- (define f 5)

```
(let f ((n f)) (if (= n 0) 1 (* n (f (- n 1)))))
≡ ((letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
       f)
    f)
≢ (letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
       (f f))
```

Type predicate

- Type predicate
  - Scheme, being an untyped language, has type predicates,
    e.g. number? complex? real? rational? integer?
    char? string? boolean? procedure? , and so on
  - ; A robust factorial function

```
(define f
   (lambda (n)
      (if  (and (integer? n) (>= n 0))
           (let f ((n n)) (if (= n 0) 1 (* n (f (- n 1)))))
           "Illegal argument")))
```

(f "snoopy") ⇒ "Illegal argument"

Imperative programming

- Imperative programming in Scheme
  - ```
    (define f
        (lambda (n)
            (let ((r 1))
                (let loop ()
                    (cond ((= n 0) r)
                          (else (set! r (* n r)) (set! n (- n 1))
                                (loop)))))))
    ; (if (= n 0) r (begin (set! r (* n r)) (set! n (- n 1)) (loop)))
    ```
  - In purely functional languages, a sequence of expressions is useless.
    ```

Quote expression

- Quote expression
  - (quote exp)

    'exp ; ' is a read macro

    Treat exp as a datum; don't evaluate it.
  - 'x ⇒ x ; x is a symbol, not variable

    ''x ⇒ 'x

    '(+ 2 3) ⇒ (+ 2 3) ; symbolic expression

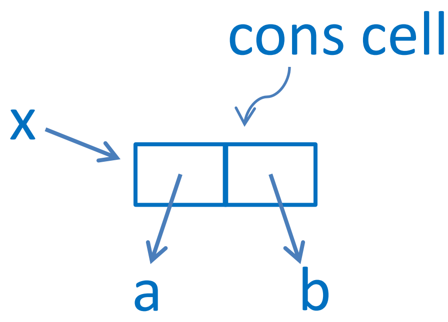
    '2 ⇒ 2 ; quote is redundant here
  - N.B. Expressions in Scheme are called symbolic expressions

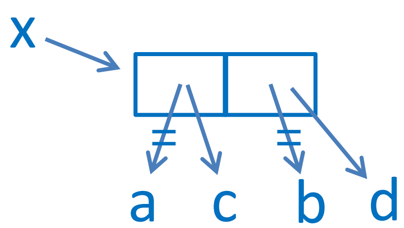
List processing

- Dotted pair

(define x '(a . b))

constructor	(define x (cons 'a 'b))	
selectors	(car x)	⇒ a
	(cdr x)	⇒ b
predicate	(pair? x)	⇒ #t
mutators	(set-car! x 'c)	⇒ unspecified
	(set-cdr! x 'd)	⇒ unspecified
	x	⇒ (c . d)





car and cdr (contents of address/decrement register) are two instructions of IBM 704 –the 1st target of LISP implementation

List processing

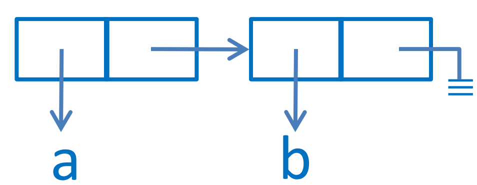
- Empty list '() ⇒ ()
- Abbreviation

A dot, the immediately followed (, and the corresponding) may be omitted together.

(a . (b . ()))

(a b . ()) (a . (b))

(a b)

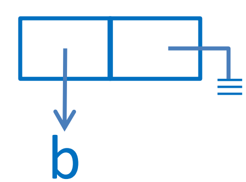


'(a . (b . ())) ⇒ (a b)

(cons 'a (cons 'b '())) ⇒ (a b)

(cons 'a '(b)) ⇒ (a b)

(b . ()) ⇒ (b)

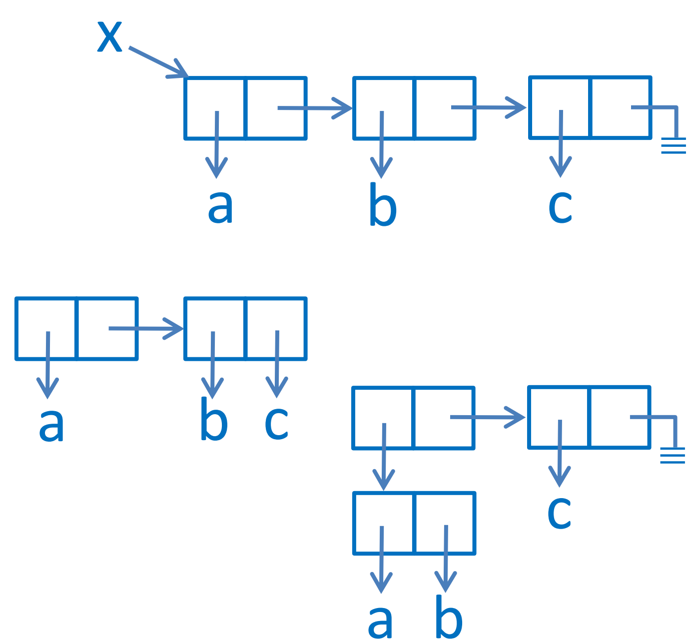


List processing

- (Proper) lists

  A (proper) list is either an empty list or a pair whose cdr is a list. Said differently, a chain of pairs ending in an empty list is called a proper list; otherwise, it is called an improper list.

constructor	(define x (list 'a 'b 'c))	
selectors	(list-ref x 1)	⇒ b
	(list-tail x 1)	⇒ (b c)
predicate	(null? x)	⇒ #f
	(list? x)	⇒ #t
	(list? '(a b . c))	⇒ #f
	(list? '((a . b) c))	⇒ #t



List processing

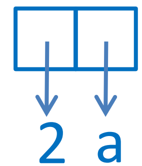
- Car and cdr of (proper) lists
  - car — take the 1^{st} element
    cdr — remove the 1^{st} element
    cxxxxr — where x = a or d, up to 4 x's
  - (define x '(a b (c (d e))))

(car x)	⇒ a		
(cdr x)	⇒ (b (c (d e)))		
(cadr x)	⇒ b		
(cddr x)	⇒ ((c (d e)))	(car (cdaddr x))	⇒ (d e)
(caddr x)	⇒ (c (d e))	(caar (cdaddr x))	⇒ d
(caaddr x)	⇒ c	(cdar (cdaddr x))	⇒ (e)
(cdaddr x)	⇒ ((d e))	(cadar (cdaddr x))	⇒ e

List processing

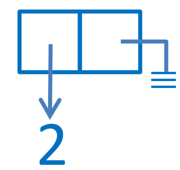
- Example

  (define a 2)

  (cons a 'a) ⇒ (2 . a)

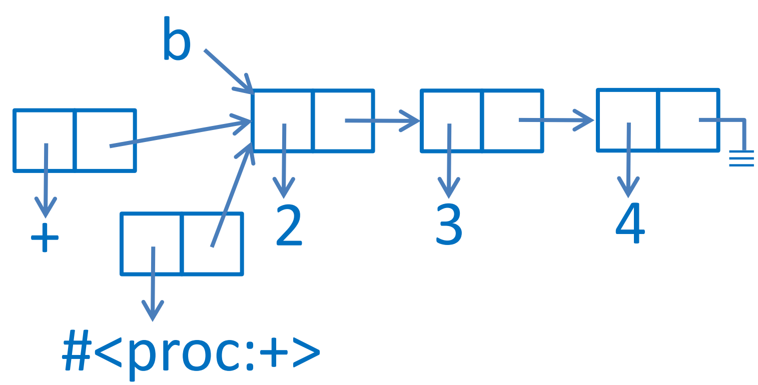
  (cons a '()) ⇒ (2) i.e. (2 . ())

  (define b '(2 3 4)) i.e. (2 . (3 . (4 . ())))

  (cons '+ b) ⇒ (+ 2 3 4)

  (cons + b) ⇒ (#<procedure:+> 2 3 4)



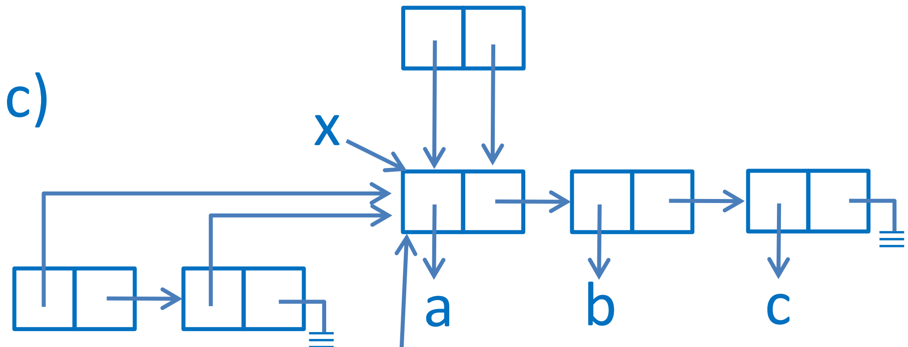
List processing

- Example

  (define x (list 'a 'b 'c)) i.e. (cons 'a (cons 'b (cons 'c '())))

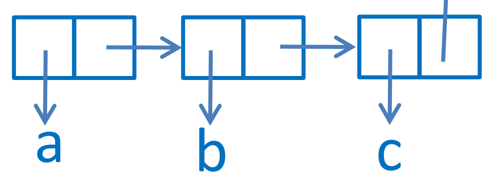
  x ⇒ (a b c)

  (cons x x) ⇒ ((a b c) a b c)



  (list x x) ⇒ ((a b c) (a b c))

  (cons x (cons x '()))



  (append x x) ⇒ (a b c a b c)

List processing

- Example

(define x '(a b c))

(define y x)

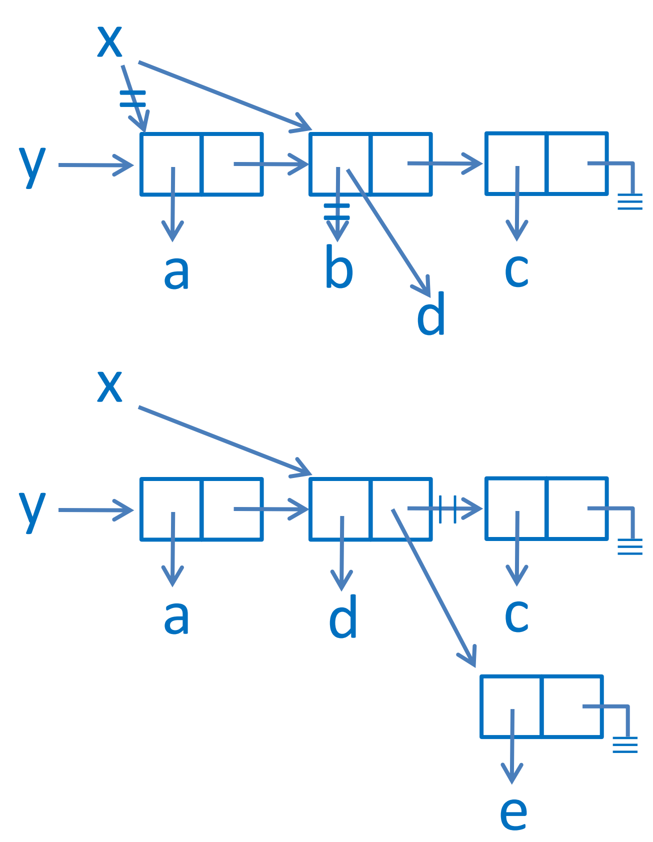
(set! x (cdr x))

(set-car! x 'd)

y ⇒ (a d c)

(set-cdr! (cdr y) '(e))

x ⇒ (d e)

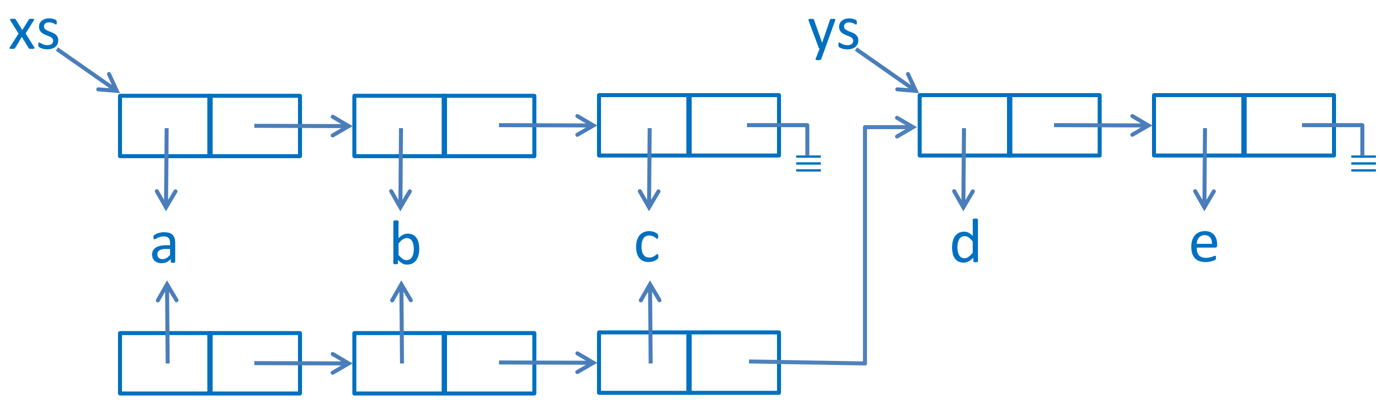


List processing

- Example - Appending lists

```
; Functional style takes O(|xs|) time and space.
(define append
    (lambda (xs ys)
        (if (null? xs) ys (cons (car xs) (append (cdr xs) ys)))))

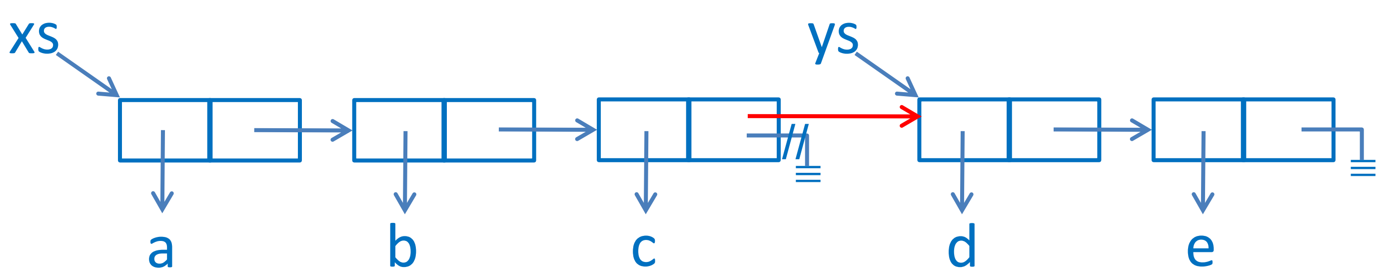
(append '(a b c) '(d e)) = (cons 'a (cons 'b (cons 'c '(d e))))
```



List processing

- Example - Appending lists

```
; Imperative style takes O(|xs|) time and O(1) space.
(define append!
    (lambda (xs ys)
        (cond ((null? xs) ys)
              ((null? (cdr xs)) (set-cdr! xs ys) xs)
              (else (append! (cdr xs) ys) xs))))
```



List processing

- Example - Reversing lists

```
; Naïve functional style takes O(|xs|²) time and space.
(define reverse
    (lambda (xs)
        (if (null? xs) '() (append (reverse (cdr xs)) (list (car xs))))))
```

let s(n) = # of cons cells allocated on reversing n-element list

Then,

$s(0) = 0$

$s(n) = s(n-1)+n, \ n \geq 1$; append: n-1 cells; list: 1 cell

Clearly, $s(n) = n(n+1)/2$

List processing

- Example - Reversing lists

```
; Accumulator-passing style takes O(|xs|) time and space
(define reverse
    (lambda (xs)
        (let rev ((xs xs)(acc '()))
            (if (null? xs) acc (rev (cdr xs) (cons (car xs) acc))))))
```

let s(n) = # of cons cells allocated on reversing n-element list

Then,

$s(0) = 0$

$s(n) = s(n-1)+1, \quad n \geq 1$; cons: 1 cell

Clearly, $s(n) = n$

List processing

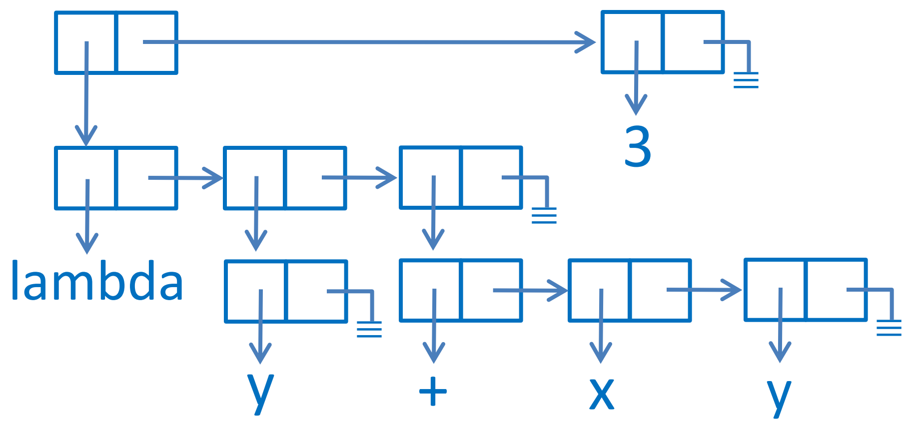
- Example – Insertion sort

```
(define isort
    (lambda (xs)
        (if (null? xs)
            xs              ; or, '()
            (let insert ((x (car xs))(ys (isort (cdr xs))))
                (cond ((null? ys) (list x))    ; (cons x ys)
                      ((<= x (car ys)) (cons x ys))
                      (else (cons (car ys) (insert x (cdr ys))))))))))
```

List processing

- Programs and data have the same syntax.
  - (define x 2)

    ((lambda (y) (+ x y)) 3) ⇒ 5



  - (define ie (interaction-environment))

    ie ⇒ #<environment *top*> ; ie may be omitted

    (eval '((lambda (y) (+ x y)) 3) ie) ⇒ 5 ; in Chez Scheme

    N.B. (eval ((lambda (y) (+ x y)) 3) ie) ⇒ (eval 5 ie) ⇒ 5

List processing

- Quote, quasiquote, unquote, unquote-splicing

```
(define x '(a b c))
'(x d e)      ⇒ (x d e)
`(x d e)      ⇒ (x d e)          ; ` is the same as ' if w/o , and ,@
`(,x d e)     ⇒ ((a b c) d e)    ; = (cons x '(d e))
`(,@x d e)    ⇒ (a b c d e)      ; = (append x '(d e))
'exp          ≡ (quote exp)
`exp          ≡ (quasiquote exp)
,exp          ≡ (unquote exp)
,@exp         ≡ (unquote-splicing exp)
```

List processing

- Program specialization

```
; metaprogram – a program that generates another program
(define pow
    (lambda (n)
        `(lambda (x)
            ,(let loop ((n n))
                (if (= n 0) `1 `(* x ,(loop (- n 1))))))))

(pow 5)                  ⇒ (lambda (x) (* x (* x (* x (* x (* x 1))))))
((eval (pow 5) ie) 2)    ⇒ 32
((pow 5) 2)              ⇒ Error
```

Higher-order functions

- Higher-order functions
  - An ordinary function is a first-order function.
  - An n^{th} order function is one that takes an $(n - 1)^{th}$ order function as an argument or as a function value.
  - A higher order function (or functional) is an n^{th} order function for $n \geq 2$.
  - (lambda (x) (+ x 1)) $\Rightarrow$ 1st order
    (lambda (x) (lambda (y) (+ x y))) $\Rightarrow$ 2nd order
    (lambda (x) (lambda (y) (lambda (z) (+ x y z)))) $\Rightarrow$ 3rd order
  - A curried function takes one argument at a time.
    Haskell Curry
    Two languages named after him, Haskell and Curry

Higher-order functions

- Example

```
(map (lambda (x) (+ x 1)) '(1 2 3))             ⇒ (2 3 4)
(map (lambda (x y) (+ x y)) '(1 2 3) '(4 5 6)) ⇒ (5 7 9)
(define powerset
    (lambda (xs)
        (if (null? xs)
            '(())
            (let ((ys (powerset (cdr xs))))
                (append ys
                        (map (lambda (s) (cons (car xs) s)) ys))))))
(powerset '(a b c)) ⇒ (() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

Continuation-passing style

- Continuation
  - (define f (lambda (n) (if (= n 0) 1 (* n $\boxed{\text{(f (- n 1))}}$))))

    What should we do after the evalaution of the boxed exp?

    i.e. What is the continuation of the program point?

    or, What is the continuation of the computation?

    Well, the continuation includes the following actions:

    1. Receive the value v of the program point
    2. Evaluate n*v
    3. Return the value of n*v to the continuation of (f n)

    This continuation can be represented by the function

    (lambda (v) (c (* n v)))

    where c is the continuation of (f n)

Continuation-passing style

- Continuation
  - > (f 3)

    Continuation of this program point

    1. Receive the value v of the program point
    2. Print v
    3. Execute REPL

    Since the last two steps are done by the underlying REPL, we shall ignore them and represent the continuation as (lambda (v) v)

Continuation-passing style

- Continuation-passing style
  - (define f

```
    (lambda (n c)
        (if (= n 0) (c 1) (f (- n 1) (lambda (v) (c (* n v)))))))
```

(f 3 (lambda (v) v)) ⇒ 6

How does it work?

(f 3 λv.v)

⇒ (f 2 $\lambda v.c_1(3*v)$) where $c_1 = \lambda v.v$

⇒ (f 1 $\lambda v.c_2(2*v)$) where $c_2 = \lambda v.c_1(3*v)$

⇒ (f 0 $\lambda v.c_3(1*v)$) where $c_3 = \lambda v.c_2(2*v)$

⇒ (($\lambda v.c_3(1*v)$) 1)

⇒ $c_3(1*1) \Rightarrow c_2(2*1) \Rightarrow c_1(3*2) \Rightarrow 6$

Continuation-passing style

- Continuation-passing style
  - Example (Escaping from deep recursion)

```
(define product
  (lambda (xs c)
    (cond ((null? xs) (c 1))
          ((= (car xs) 0) 0)
          (else (product (cdr xs)
                         (lambda (v) (c (* (car xs) v))))))))
> (product '(2 3 0 4 5) (lambda (v) v))
0
```

Continuation-passing style

- Compare with C++ exception handling
  - ```
    int product(int* begin,int* end)
    {
        if (begin==end) return 1;
        else if (*begin==0) throw 0;   // raise an exception
        else return *begin*product(begin+1,end);
    }
    int main()
    {
        int a[5]={2,3,0,4,5};                // exception handler
        try { cout << product(a,a+5);} catch (int r) { cout << r; }
    }
    ```
    ```

Continuation in Scheme

- Call-with-current-continuation
  - Scheme allows the continuation of a computation to be captured by call/cc.
  - (call/cc proc)
    where proc is a one-parameter function.
  - Let κ be the current continuation of the call/cc expression. Then,
    (call/cc proc) $\Rightarrow$ (proc κ)
    That is, proc is called with the current continuation of the call/cc expression.

Continuation in Scheme

- Call-with-current-continuation
  - Example

> (* 3 (call/cc (lambda (c) (+ 4 5)))) ; $\kappa = \lambda v.3*v$

27

(* 3 (call/cc (λc.(+ 4 5))))

$\Rightarrow$ (* 3 ((λc.(+ 4 5)) κ)) $\Rightarrow$ (* 3 (+ 4 5)) $\Rightarrow$ 27

> (* 3 (call/cc (lambda (c) (c (+ 4 5))))) ; $\kappa = \lambda v.3*v$

27

(* 3 (call/cc (λc.(c (+ 4 5)))))

$\Rightarrow$ (* 3 ((λc.(c (+ 4 5))) κ)) $\Rightarrow$ (* 3 (κ (+ 4 5))) $\Rightarrow$ 27

Continuation in Scheme

- Call-with-current-continuation
  - Example (Cont'd)

    > (* 3 (call/cc (lambda (c) (+ (c 4) 5))))   ; $\kappa = \lambda v.3*v$

    12

    (* 3 (call/cc (λc.(+ (c 4) 5))))

    $\Rightarrow$ (* 3 ((λc.(+ (c 4) 4 5)) κ)) $\Rightarrow$ (* 3 (+ (κ 4) 5)) $\Rightarrow$ 12

    > (define get-back 'any)

    > (* 3 (call/cc (lambda (c) (set! get-back c) (+ 4 5))))

    27

    > (get-back k)

    3*k

Continuation in Scheme

- Call-with-current-continuation
  - Example (Escaping from deep recursion)

```
(define product
    (lambda (xs)
        (call/cc (lambda (exit)
                    (let loop ((xs xs))
                        (cond ((null? xs) 1)
                              ((= (car xs) 0) (exit 0))
                              (else (* (car xs) (loop (cdr xs)))))))))))
> (product '(2 3 0 4 5))            ; exit = λv.v
> (+ 1 (product '(2 3 0 4 5)))      ; exit = λv.1+v
```

Continuation in Scheme

- Call-with-current-continuation
  - Example (Returning to deep recursion)

```
(define get-back 'any)
(define f
   (lambda (n)
      (cond ((= n 0) (call/cc (lambda (c) (set! get-back c) 1)) )
            (else (* n (f (- n 1)))))))
> (f 5)
120
> (get-back k) ⇒ 120*k      ; get-back = λv.5*4*3*2*1*v
> (get-back k) ⇒ 120        ; get-back = λv.5*4*3*2*1*1
```

SKI combinators

- Combinator
  - A combinator is a λ-expression without free variables.
  - S, K, and I combinators

    S = λx.λy.λz.x z (y z)   distributor

    K = λx.λy.x   canceller

    I = λx.x = S K K   ∵ S K K = λz.K z (K z) = λz.z
  - An expression in the λ-calculus may be **compiled** to code consisting of only S, K, I, built-in constants and functions. (Note: All built-in functions, e.g. +, *, etc. are curried.)
  - Example

    (λx.+xx) 2   may be compiled to   S (S (K +) I) I 2

    or, in Scheme, (((S ((S (K +)) I)) I) 2)

SKI combinators

- Reduction (evaluation)
  - $\underline{(\lambda x.+xx)\ 2}$

    $= \underline{+\ 2\ 2}$ reducible expression

    $= 4$

    N.B. Each underlined expression is called a redex.
  - Eager evaluation, i.e. call by value

    $(\lambda x.+xx)\ \underline{(+\ 2\ 3)} = \underline{(\lambda x.+xx)\ 5} = \underline{+\ 5\ 5} = 10$
  - Lazy evaluation, i.e. call by need

    $\underline{(\lambda x.+xx)\ (+\ 2\ 3)} = +\ \underline{(+\ 2\ 3)}\ (+\ 2\ 3) = +\ 5\ \underline{(+\ 2\ 3)} = \underline{+\ 5\ 5} = 10$

SKI combinators

- Reduction (evaluation)
  - With lazy evaluation, we have

<u>S (S (K +) I) I 2</u>	call to S	
= <u>S (K +) I 2</u> (I 2)	call to S	(S (K +) I) 2 (I 2)
= <u>K + 2</u> (I 2) (I 2)	call to K	(K +) 2 (I 2) (I 2)
= + <u>(I 2)</u> (I 2)	call to +, argument (I 2) is reduced	
= + 2 <u>(I 2)</u>		
= <u>+ 2 2</u>		
= 4		

Lazy evaluation always reduces the leftmost redex.

SKI combinators

- Compilation algorithm

compile x ⇒ x, if x is a variable, built-in constant, or built-in function

compile (e1 e2) ⇒ compile e1 (compile e2)

compile λx.e ⇒ abstract x (compile e)

abstract x x ⇒ I

abstract x y ⇒ K y, if y is a variable (≠ x), built-in constant or built-in function

abstract x (e1 e2) ⇒ S (abstract x e1) (abstract x e2)

SKI combinators

- Compilation algorithm
  - Notice that the three cases of abstract correspond to:

    $\lambda x.x = I$

    $\lambda x.y = K\ y$, since $K\ y = (\lambda a.\lambda x.a)\ y = \lambda x.y$

    $\lambda x.e1\ e2 = S\ (\lambda x.e1)\ (\lambda x.e2)$,

    since

    $S\ (\lambda x.e1)\ (\lambda x.e2) = (\lambda a.\lambda b.\lambda x.a\ x\ (b\ x))\ (\lambda x.e1)\ (\lambda x.e2)$

    $= \lambda x.(\lambda x.e1)\ x\ ((\lambda x.e2)\ x) = \lambda x.e1\ e2$
  - Example

    compile $((\lambda x.+xx)\ 2)$

    $\Rightarrow$ compile $(\lambda x.+xx)$ (compile 2)

    $\Rightarrow$ compile $(\lambda x.+xx)\ 2$

SKI combinators

- Example

  compile (λx.+xx)

  ⇒ abstract x (compile (+ x x))

  ⇒ abstract x (compile (+ x) (compile x))

  ⇒ abstract x (compile + (compile x) (compile x))

  ⇒ abstract x (+ (compile x) (compile x))

  ⇒ abstract x (+ x (compile x))

  ⇒ abstract x (+ x x)

  ⇒ S (abstract x (+ x)) (abstract x x)

  ⇒ S (abstract x (+ x)) (abstract x x)

  ⇒ S (S (abstract x +) (abstract x x)) (abstract x x)

  ⇒ S (S (K +) I) I

Fixed-point combinators

- Fixed point
  - x is a fixed point of f iff f x = x
- Can a recursive function be anonymous?

  f = λn.if n=0 then 1 else n*f(n-1)
  - What is the solution for f?

    Analog

    Given 2x+3 = x+8

    Q: What is the solution for x?

    A: x = 5 is a solution

    ∵ 2·5+3 = 5+8

Fixed-point combinators

- Equation: f = λn.if n=0 then 1 else n*f(n-1)
  - What is the solution for f? (Cont'd)

    Answer

    f = ! (i.e the *factorial* function) is the solution.

    Claim

    ! = λn.if n=0 then 1 else n*(n-1)!

    Proof

    The proof is based on the fact: f = g iff f x = g x ∀x

    Applying n to functions on both sides yields

    lhs = n!

    rhs = (λn.if n=0 then 1 else n*(n-1)!) n

    = n!

Fixed-point combinators

- Equation: f = λn.if n=0 then 1 else n*f(n-1)
  - How to obtain the solution for f?

    Answer

    It is a fixed point of the functional (i.e. higher-order function)

    F = λf.λn.if n=0 then 1 else n*f(n-1)   ← not recursive

    since

    F ! = λn.if n=0 then 1 else n*(n-1)! = !
  - How to obtain a fixed point of F?
- Fixed-point combinators
  - A fixed-point combinator fix is a combinator such that, for any f, fix f = f (fix f)
  - In other words, (fix f) a fixed point of f.

Fixed-point combinators

- Y combinator with lazy evaluation
  - $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
  - Y is a fixed-point combinator, because

    $Y\ f = (\lambda x.f(xx))\underline{(\lambda x.f(xx))}$

    $\quad = f\ ((\underline{\lambda x.f(xx))}\ \underline{(\lambda x.f(xx))})$

    $\quad = f\ (Y\ f)$
  - However, it fails to terminate with eager evaluation.

    $Y\ f = (\lambda x.f(xx))(\lambda x.f(xx))$

    $\quad = f\ (\underline{(\lambda x.f(xx))\ (\lambda x.f(xx))})$; must delay the argument

    $\quad = f\ (f\ (\underline{(\lambda x.f(xx))(\lambda x.f(xx))}))$

    $\quad = f\ (f\ (f\ \ldots)))$

Fixed-point combinators

- Y combinator with eager evaluation
  - $Y = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$ N.B. $\lambda y.xxy = xx$
  - Y is a fixed-point combinator, because $\because\ (\lambda y.xxy)y = xxy$

    $Y\ f = (\lambda x.f(\lambda y.xxy))\underline{(\lambda x.f(\lambda y.xxy))}$

    $= f\ (\lambda y.\underline{(\lambda x.f(\lambda y.xxy))}\ \underline{(\lambda x.f(\lambda y.xxy))}\ y)$

    $= f\ (\lambda y.Y\ f\ y)$

    $= f\ (Y\ f)$ $\because\ \lambda y.Y\ f\ y = Y\ f$
  - Note that the argument is in effect delayed.

    $Y\ f = (\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$

    $= f\ \underline{(\lambda y.(\lambda x.f(\lambda y.xxy))\ (\lambda x.f(\lambda y.xxy))\ y)}$

    pass the λ-function to f

Fixed-point combinators

- Y combinator in Scheme
  - ```
    (define Y
        (lambda (f)
            (let ((h (lambda (x) (f (lambda (y) ((x x) y))))))
                (h h))))
    (define F
        (lambda (f)
            (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
    (define ! (Y F))
    (! 10)  ⇒  3628800
    ```
    ```

Fixed-point combinators

- Y combinator in Scheme
  - Finally, we may compute the factorial without using any function name:

```
(((lambda (f)
      ((lambda (x) (f (lambda (y) ((x x) y))))
       (lambda (x) (f (lambda (y) ((x x) y))))))
  (lambda (f)
      (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
 10)
⇒ 3628800
```