λ

Chapter 10

(10.3)

Lists and functions

Building Lists

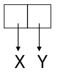
cons – Takes two arguments and returns a list formed by appending the first argument at the front of the second argument

```
(cons 'a '(b c))
(cons 'x '( ))
(cons '(a b) '(c))
```

Dot Notation

Scheme/Lisp lists are singly linked lists of two-part cells

`(X . Y)` denotes (cons 'x 'y)



`(1 2 3)` could be written `(1 . (2 . (3 . ())))`

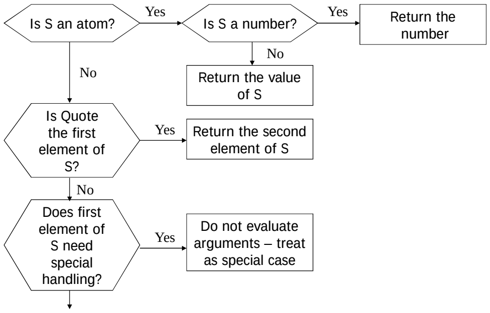
READ-EVAL-PRINT

Execution of LISP/Scheme code repeats a READ-EVAL-PRINT loop

The interpreter

- Reads an expression
- Evaluates the expression
- Prints the result

Expression Evaluation



Is S an atom?

Yes

Is S a number?

Yes

Return the number

No

Return the value of S

No

Is Quote the first element of S?

Yes

Return the second element of S

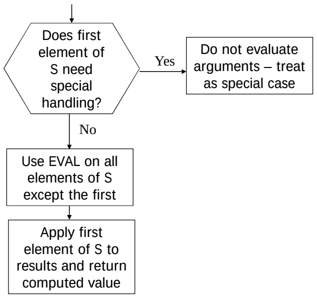
No

Does first element of S need special handling?

Yes

Do not evaluate arguments – treat as special case

Expression Evaluation, cont.



Does first element of S need special handling?

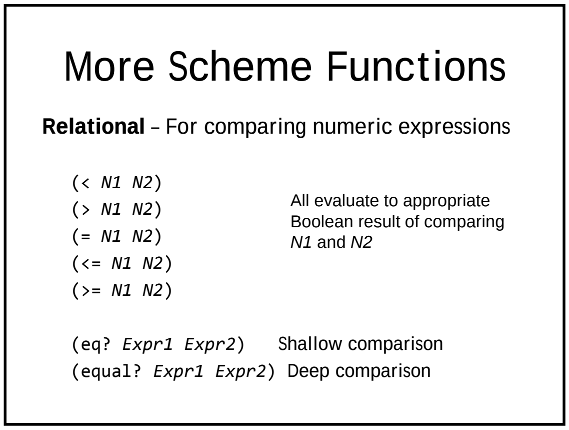
Yes

Do not evaluate arguments – treat as special case

No

Use EVAL on all elements of S except the first

Apply first element of S to results and return computed value



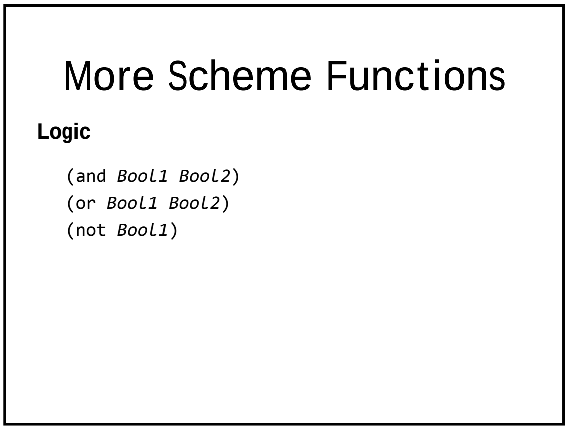
More Scheme Functions

Relational – For comparing numeric expressions

```
(< N1 N2)
(> N1 N2)
(= N1 N2)
(<= N1 N2)
(>= N1 N2)
```

All evaluate to appropriate Boolean result of comparing *N1* and *N2*

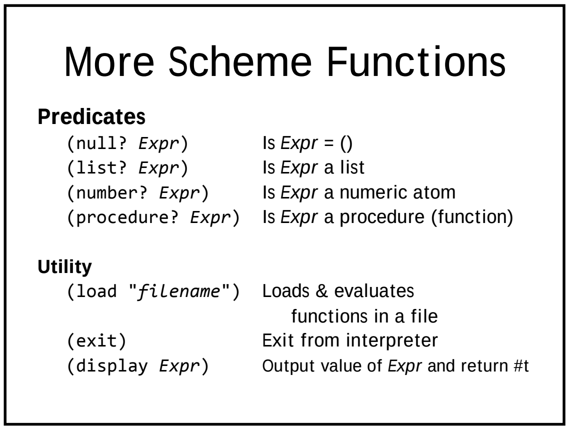
`(eq? Expr1 Expr2)`	Shallow comparison
`(equal? Expr1 Expr2)`	Deep comparison



More Scheme Functions

Logic

```
(and Bool1 Bool2)
(or Bool1 Bool2)
(not Bool1)
```



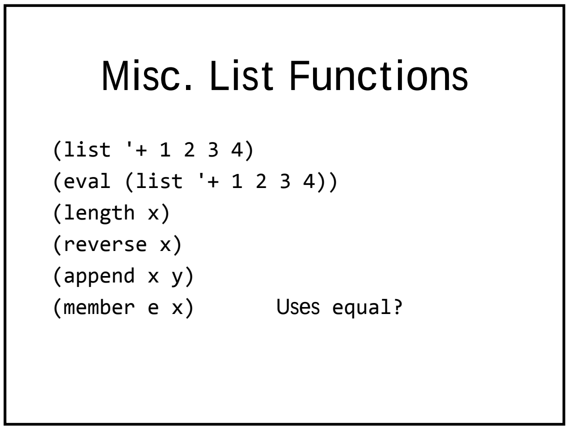
More Scheme Functions

Predicates

`(null? Expr)`	Is *Expr* = ()
`(list? Expr)`	Is *Expr* a list
`(number? Expr)`	Is *Expr* a numeric atom
`(procedure? Expr)`	Is *Expr* a procedure (function)

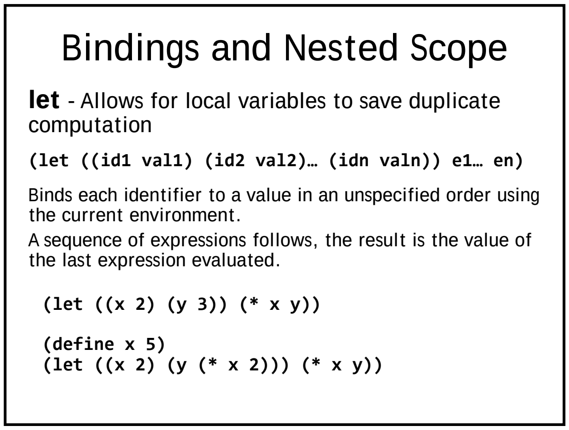
Utility

`(load "filename")`	Loads & evaluates functions in a file
`(exit)`	Exit from interpreter
`(display Expr)`	Output value of *Expr* and return #t



Misc. List Functions

```
(list '+ 1 2 3 4)
(eval (list '+ 1 2 3 4))
(length x)
(reverse x)
(append x y)
(member e x)       Uses equal?
```



Bindings and Nested Scope

let - Allows for local variables to save duplicate computation

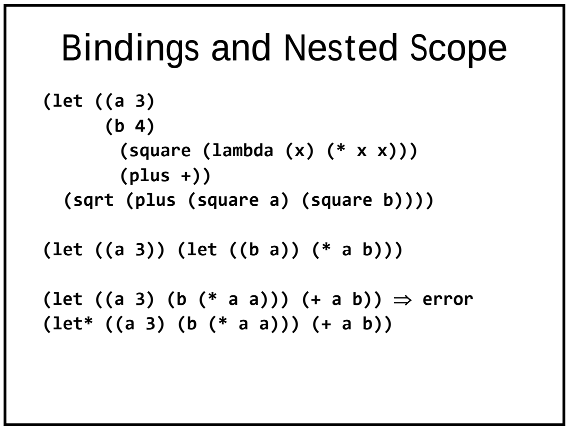
(let ((id1 val1) (id2 val2)… (idn valn)) e1… en)

Binds each identifier to a value in an unspecified order using the current environment.

A sequence of expressions follows, the result is the value of the last expression evaluated.

```
(let ((x 2) (y 3)) (* x y))

(define x 5)
(let ((x 2) (y (* x 2))) (* x y))
```

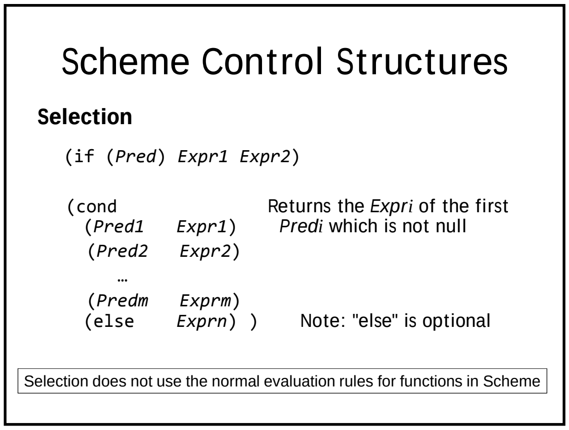


Bindings and Nested Scope

```
(let ((a 3)
      (b 4)
        (square (lambda (x) (* x x)))
        (plus +))
  (sqrt (plus (square a) (square b))))

(let ((a 3)) (let ((b a)) (* a b)))

(let ((a 3) (b (* a a))) (+ a b)) ⇒ error
(let* ((a 3) (b (* a a))) (+ a b))
```



Scheme Control Structures

Selection

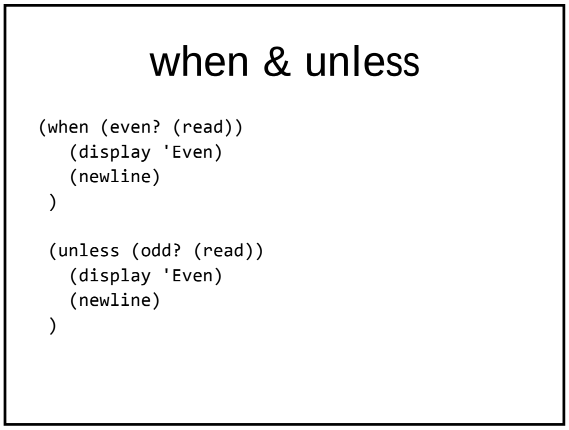
```
(if (Pred) Expr1 Expr2)
```

```
(cond
  (Pred1    Expr1)
  (Pred2    Expr2)
     …
  (Predm    Exprm)
  (else     Exprn) )
```

Returns the *Expri* of the first *Predi* which is not null

Note: "else" is optional

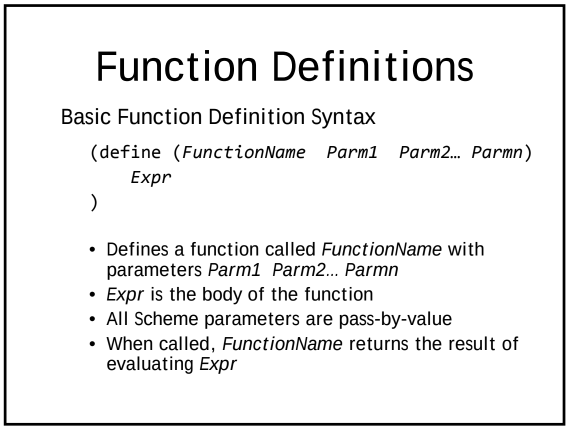
Selection does not use the normal evaluation rules for functions in Scheme



when & unless

```
(when (even? (read))
   (display 'Even)
   (newline)
 )

 (unless (odd? (read))
   (display 'Even)
   (newline)
 )
```

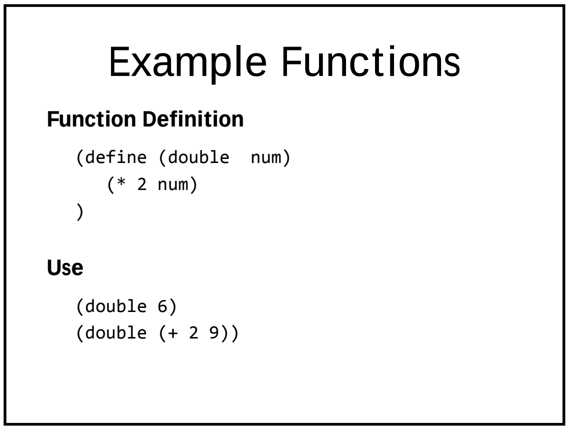


Function Definitions

Basic Function Definition Syntax

```
(define (FunctionName  Parm1  Parm2… Parmn)
     Expr
)
```

- Defines a function called *FunctionName* with parameters *Parm1 Parm2… Parmn*
- *Expr* is the body of the function
- All Scheme parameters are pass-by-value
- When called, *FunctionName* returns the result of evaluating *Expr*



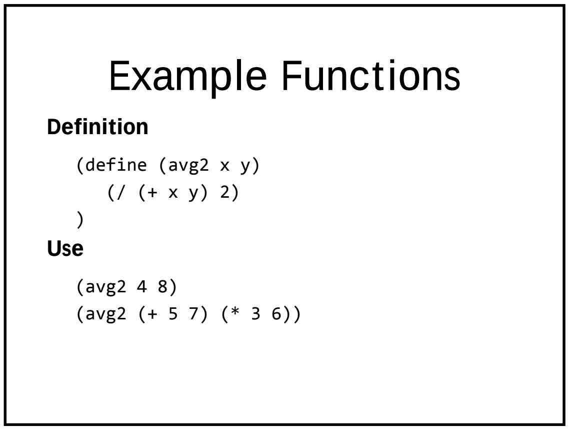
Example Functions

Function Definition

```
(define (double  num)
   (* 2 num)
)
```

Use

```
(double 6)
(double (+ 2 9))
```



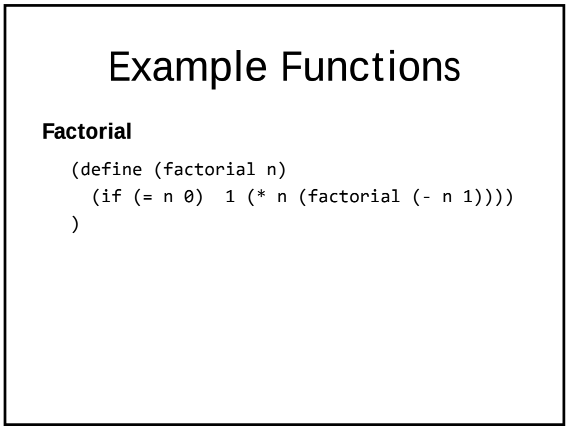
Example Functions

Definition

```
(define (avg2 x y)
   (/ (+ x y) 2)
)
```

Use

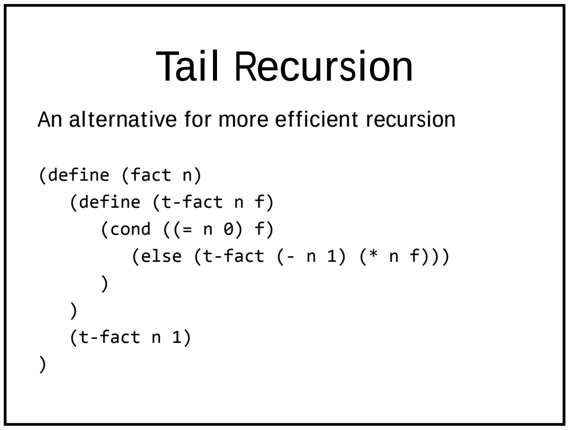
```
(avg2 4 8)
(avg2 (+ 5 7) (* 3 6))
```



Example Functions

Factorial

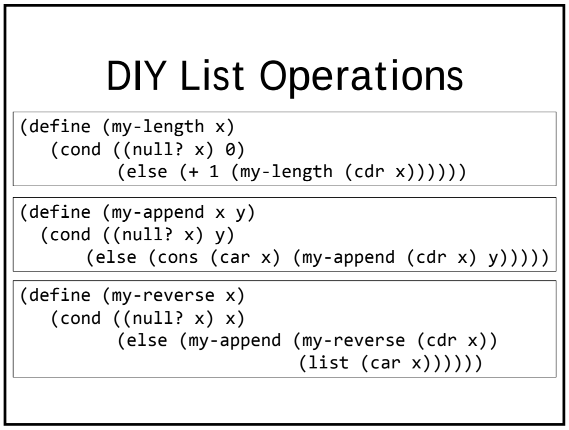
```
(define (factorial n)
  (if (= n 0)  1 (* n (factorial (- n 1))))
)
```



Tail Recursion

An alternative for more efficient recursion

```
(define (fact n)
   (define (t-fact n f)
      (cond ((= n 0) f)
         (else (t-fact (- n 1) (* n f)))
      )
   )
   (t-fact n 1)
)
```

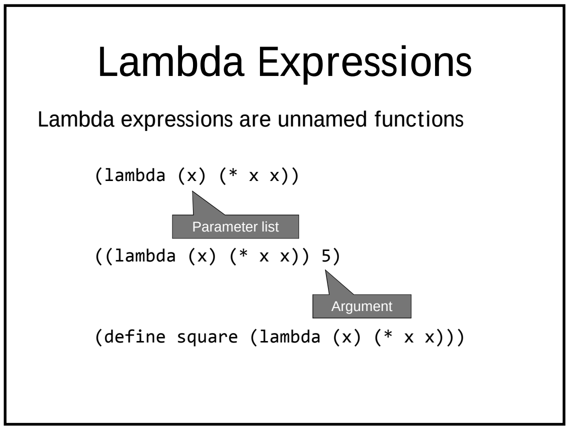


DIY List Operations

```
(define (my-length x)
   (cond ((null? x) 0)
         (else (+ 1 (my-length (cdr x))))))
```

```
(define (my-append x y)
  (cond ((null? x) y)
      (else (cons (car x) (my-append (cdr x) y)))))
```

```
(define (my-reverse x)
   (cond ((null? x) x)
         (else (my-append (my-reverse (cdr x))
                          (list (car x))))))
```



Lambda Expressions

Lambda expressions are unnamed functions

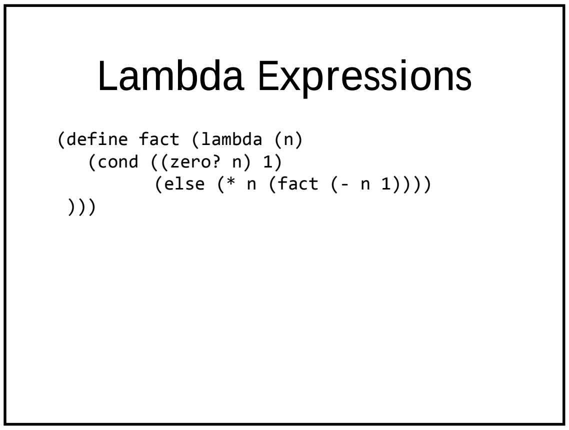
```
(lambda (x) (* x x))
```

Parameter list

```
((lambda (x) (* x x)) 5)
```

Argument

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



Lambda Expressions

```
(define fact (lambda (n)
   (cond ((zero? n) 1)
         (else (* n (fact (- n 1))))
 )))
```