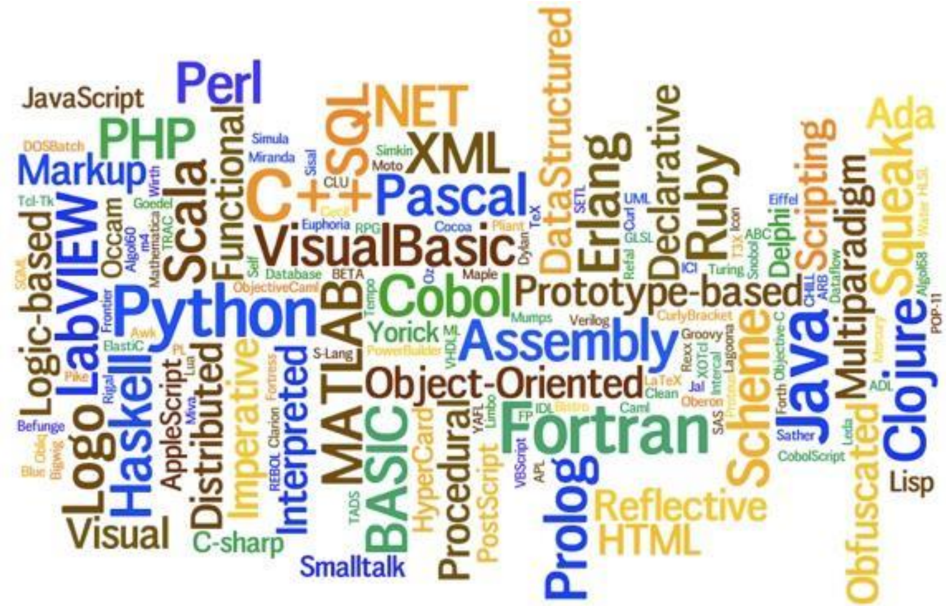


Programming Languages

Scheme Functional Programming

Theory of Programming Languages

Chapter 15 Appendix

Introduction to Scheme Syntax

Scheme: A Lisp dialect

- Basic syntax:

  expression $\rightarrow$ *atom* | *list*

  list $\rightarrow$ '(' { *expression* } ')'

  atom $\rightarrow$ *number* | *string* | *identifier* | *character* | *boolean*

- Everything is an expression: programs, data, anything.
- Lists are (almost) the only structure.
- A program consists of expressions.

Scheme Expressions: examples

`42`	a number
`"hello"`	a string
`#t or #T`	Boolean value "true"
`#f or #F`	false
`#\a`	the character 'a'
`a`	an identifier
`hello`	another identifier
`(2.1 2.2 -3)`	a list of numbers
`(1 (2 3) (a) )`	list containing other lists
`(+ 2 3)`	list consisting of the identifier "+" (a built-in procedure) and two numbers
`(* (+ 2 3) (/ 6 2))`	list consisting of an identifier and two lists

Scheme Operation

- Programs are executed by evaluating expressions.
- A Scheme program consists of a series of expressions.
- Usually, the expressions are **`define`**'s.
- Interpreter runs in "read-eval-print" loop.
- Programs can explicitly use **`eval`** to evaluate expressions.

```
(define pi 3.14159)

(define (area-of-circle rad) (* pi rad rad) )

(area-of-circle 10)
```

Expression Evaluation

Expression	Value
`10`	`10`
`3/5`	`0.6 (fractional form OK)`
`(+ a b c d e)`	`sum of values: (+) = 0`
`(* a b c d)`	`product of values: (*) = 1`
`(+ 3 4)`	`7`
`(* 3 4 5)`	`60`
`(+ (* 3 4) (* 5 6))`	`42`
`(= 10 (* 2 5))`	`"10 = 2*5"? #t` **(true)**
`(> 3 5 )`	`"3 > 5"? #f` **(false)**
`(and (= a b) (<> a 0))`	`(a == b) && (a != 0)`
`(not (= x 1) )`	`!(x==1)`
`(read-char)`	input char, like C `getchar()`

Defining Values

To define a symbol in Scheme, use "define".

```
(define pi 3.14159)
(define n  20)
(define n-square (* n n ) )
```

Defining Functions

Syntax:

(define (*function_name parameters*) *expression* { *expression* })

the value of the function is the value of the last expression.

- Area of a rectangle:

```
(define (area width height) (* width height) )
```

- Hypotenuse of a triangle:

```
(define (hypo side1 side2)
   (sqrt (+ (* side1 side1) (* side2 side2) ) ) )
```

or:

```
(define (hypo side1 side2)
   (sqrt (+ (square side1) (square side2) ) ) )
```

Input and Output Functions

Not available in beginner's level Dr. Scheme:

(**`read`**) read space-delimited value

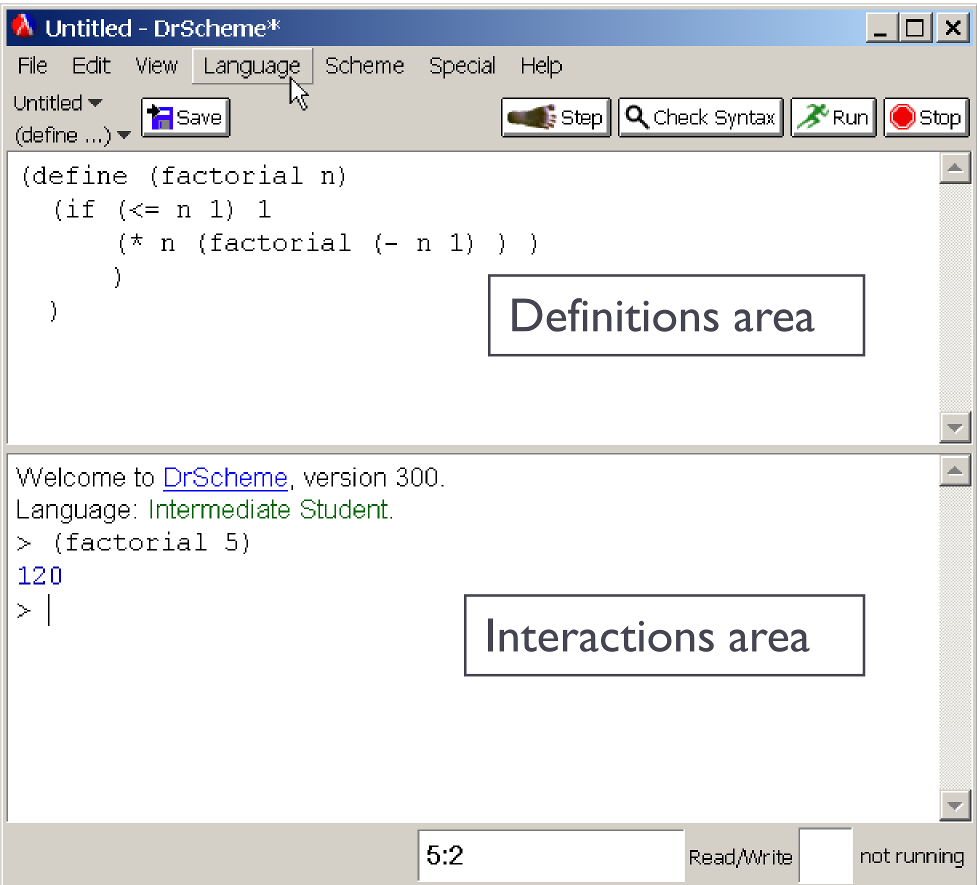
(**`display`** *expression*) output a value

(**`newline`**)

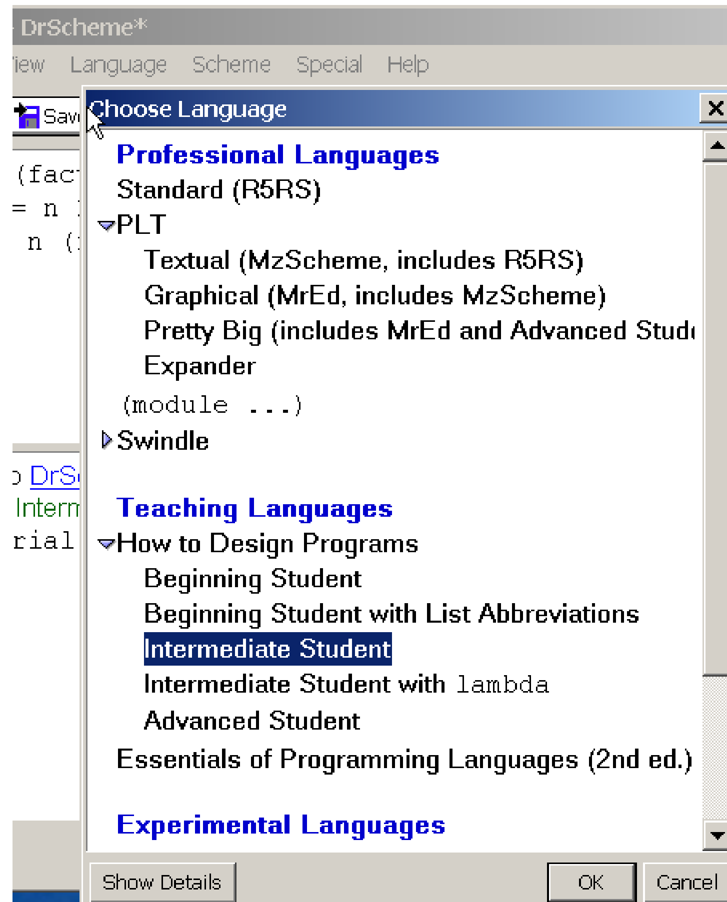
```
(display "Please input something: ")
(define x (read) )
(display (cons "you input: " (list x)))
```

```
Please input something:  4.5
you input 4.5
Please input something:  hello
you input hello
```

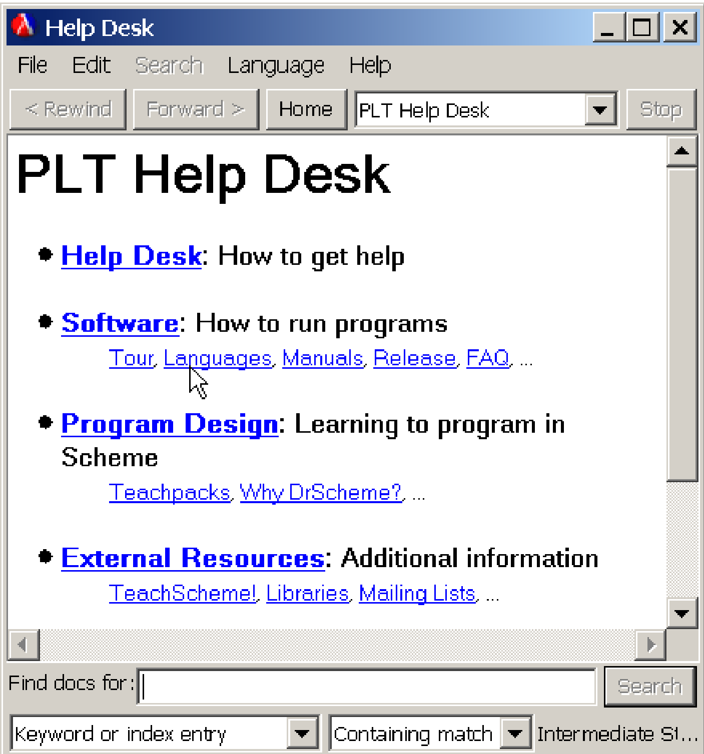
A Look at Dr.Scheme

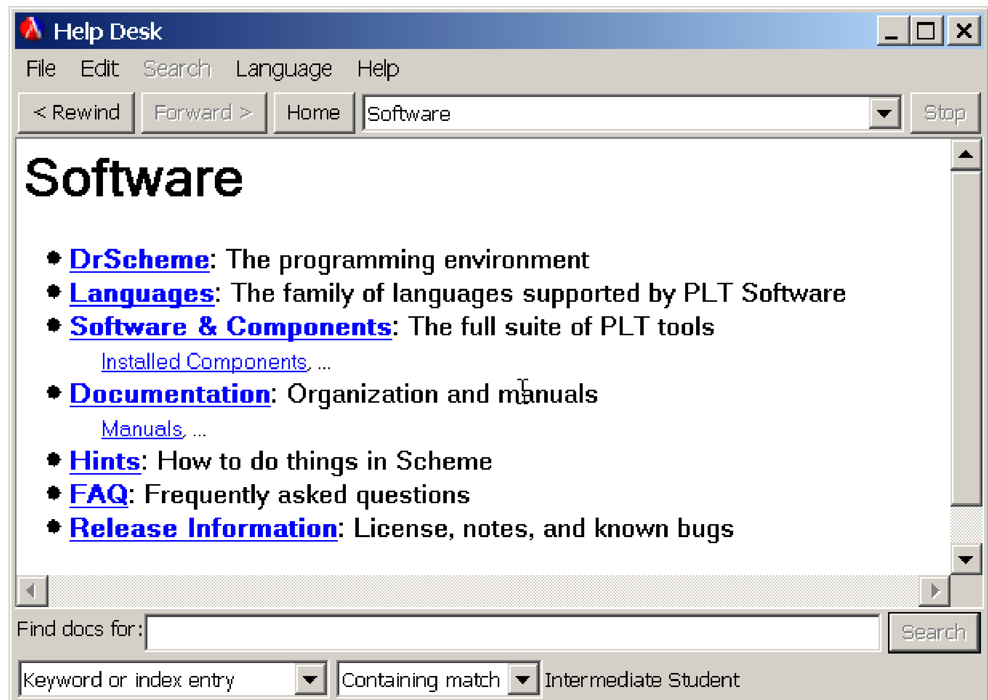


Language Choice Dialog

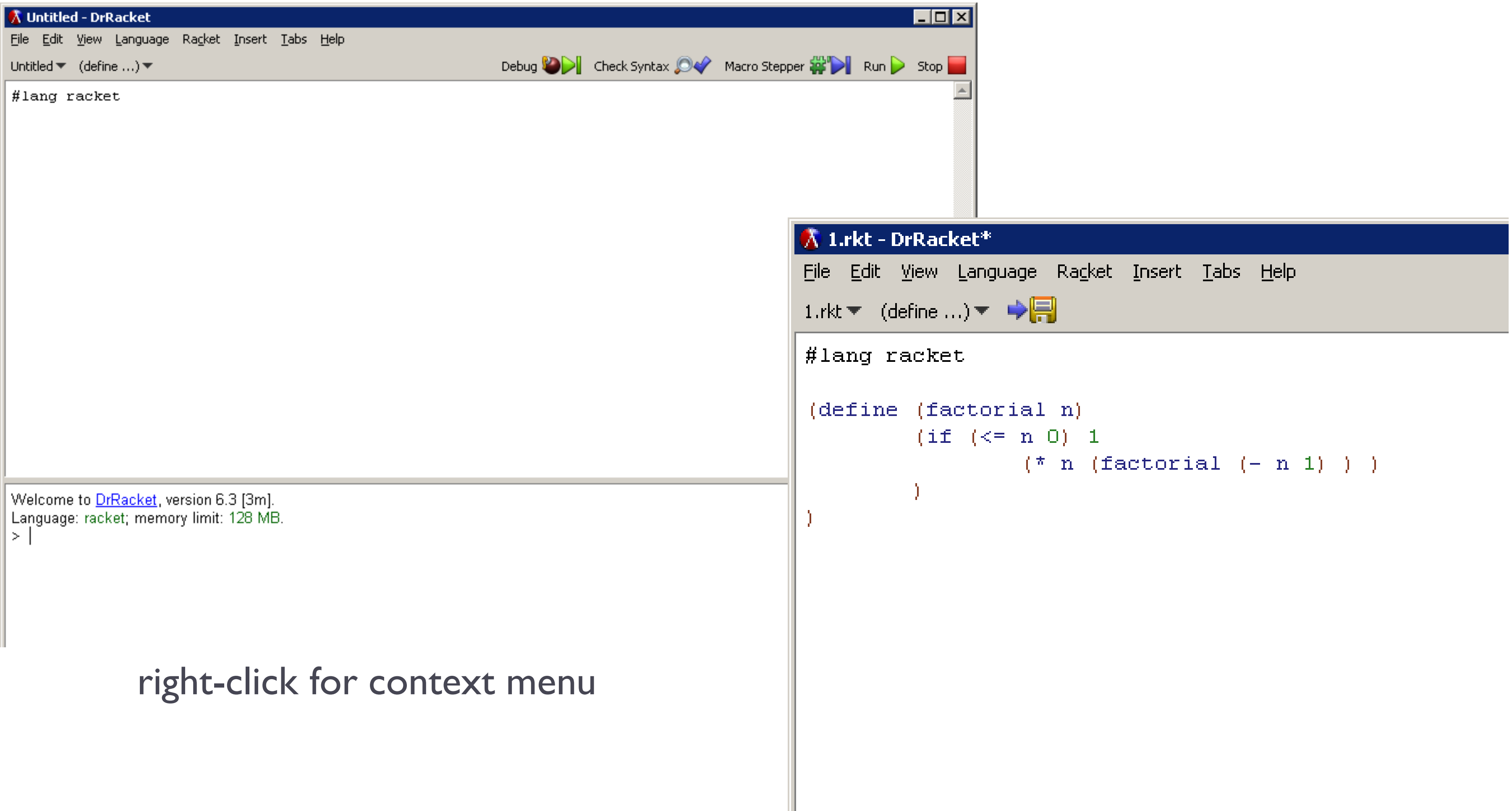


Dr.Scheme online help





Context-sensitive help, syntax checker



right-click for context menu

syntax checker and
easy-to-use debugging

Numeric Predicate Functions

- These functions compare numeric values and return true (`#t`) or false (`#f`).
- LISP (not Scheme) may return `()` for "false".

`(= a b)`	numeric equality, a == b
`(<> a b)`	`(not (= a b) )`
`(> a b)`	a > b
`(< a b)`	a < b
`(<= a b)`	`(or (< a b) (= a b) )`
`(>= a b)`	
`(even? x)`	is x an even number?
`(odd? x)`	is x an odd number?
`(zero? a)`	`(= a 0)`

`if` function

Syntax:

(if *predicate then_expression else_expression***)**

"**`if`**" always has 3 expressions.

```
(define (fabs x)
    (if (>= x 0) x (* -1 x) )
)
```

Note: Scheme has a built-in function named **"`abs`"**.

Control Flow using Predicates

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

- Scheme performs calculations using arbitrary precision:

  `(factorial 100)`

  ```
  93326215443944152681699238856266700490715968264
  31621468592963895217599993229915608941463976156
  5182862536979208272237582511852109168640000000000
  00000000000000
  ```

cond function

```
(cond                    true or false
  (predicate1 expression { expression })
  (predicate2 expression { expression })
  (predicate3 expression { expression })
  ...
  (else expression { expression } )
)
```

- Each of the predicates is evaluated (in order) until a predicate evaluates to True. Then, all expressions for that predicate are evaluated and the value of the <u>last</u> expression is the return value of **cond**.
- If no predicates are true, then the "else" clause is used and the value of the <u>last</u> expression is returned.

cond example

- define a function f(x) that returns:

$$f(x) = \begin{cases} 0 & x < 0 \\ x\sqrt{x} & 0 \le x \le 1 \\ x^2 & x > 1 \end{cases}$$

```
(define (f x)
   (cond
       ( (< x 0) 0 )
       ( (<= x 1) (* x (sqrt x) ) )
       ( else (* x x) )
   )
)
```

`if` and `cond` compared

- **`if`** contains one predicate and 2 cases ("then" and "else").
- **`cond`** contains an arbitrary number of predicates, evaluated conditionally.
- **`cond`** is similar to "if .. else if ... else if ... else ..." in C or Java.
- **`cond`** clauses may contain any number of expressions.
- In Dr. Scheme "student" level, each clause can contain only one predicate ("question") and one other expression ("answer").

Q: Can you replace "cond" with a series of "if" statements?

`cond` replaced by "`if`"

- define a function f(x) that returns:

$$f(x)=\begin{cases} 0 & x<0 \\ x\sqrt{x} & 0\le x\le 1 \\ x^2 & x>1 \end{cases}$$

```
(define (ff x)
   (if (< x 0) 0
        ( if (<= x 1) (* x (sqrt x) )
              (* x x)
         )
   )
)
```

Type Predicate Functions

- test the type of an argument and return #t or #f

`(boolean? x)`	is x a boolean?
`(char? x)`	is x a character?
`(list? x)`	is x a list?
`(number? x)`	is x a number? (int or floating point)
`(pair? x)`	is x a pair? (has car and cdr)
`(string? x)`	is x a string?
`(symbol? x)`	is x a valid symbol?

`(symbol? '$x-%y)`	→ **#t** (**$x-%y** is a valid symbol)
`(list? 5)`	→ **#f**

List Functions: `quote` and `list`

- **`quote`** or `'` prevents evaluation of a list
- **`quote`** is used to create data constants

Why is this needed?

```
(quote a)              → a
(quote (a b c))        → (a b c)
'(a b c)               → (quote (a b c))
```

- **`list`** makes a list of its arguments

```
(list 'a)                  → (a)
(list '(a b) '(c d))       → ((a b) (c d))
(list (* 5 2) (/ 6 2))     → (10 3)
```

Breaking down a list: `car` `cdr`

- essential list manipulation functions in Scheme
- **`car`** returns the first element from a <u>list</u>
- **`cdr`** returns everything after the first element of a <u>list</u>
- the argument *must* be a non-empty list

`(car '(a b c))`	→ `a`
`(cdr '(a b c))`	→ `(b c)`
`(car '((a b) c))`	→ `(a b)`
`(cdr '((a b) c))`	→ `(c)`
`(car '(a) )`	→ `a`
`(cdr '(a) )`	→ `()`
`(car '() )`	error (empty list)
`(cdr 'a )`	error (argument not a list)

Building a list: `cons`

- prepend a value to list, and return as a new list:

  (cons *expression list*)

- the second argument should be a list

```
(cons 'a '(b c))            → (a b c)
(cons '(a) '(b c))          → ((a) b c)
(cons '(a b) '(c d))        → ((a b) c d)
(cons '() '(b c) )          → (() b c)
(cons 'a 'b )               → (a . b)
```

- the last example is called a "dotted pair" and is usually an error.
- (a . b) means that a list element contains two atoms instead of an atom (or pointer) and a pointer

Building a list: `list` & `append`

- append several lists to create a single list:

  `(append` *`list1 list2`* `...)`

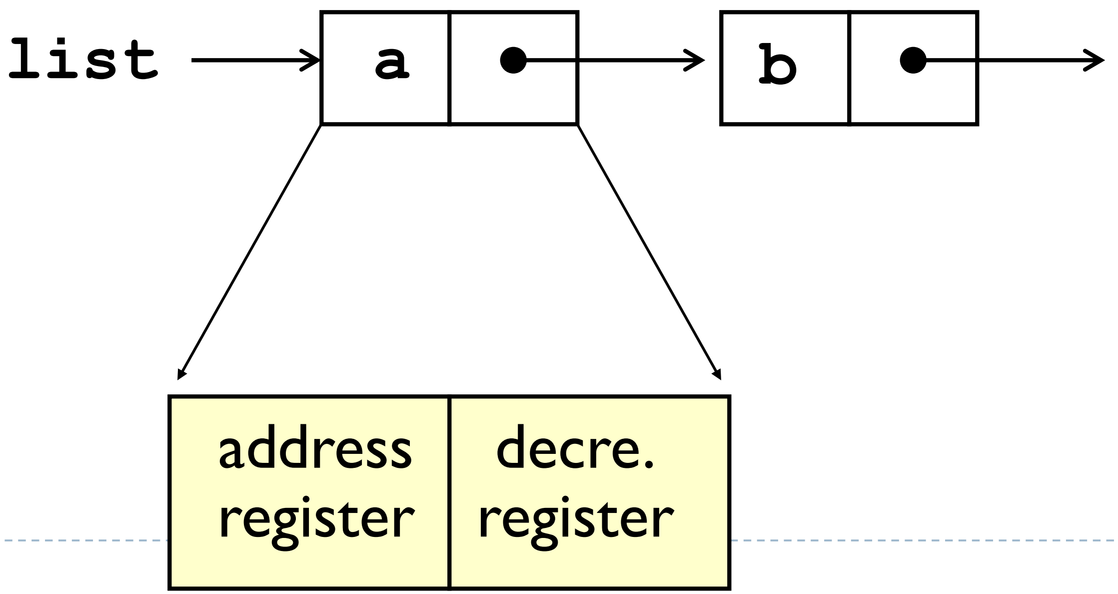
- the arguments should be lists

```
(append '(a b) '(c d))    → (a b c d)
(cons   '(a b) '(c d))    → ((a b) c d)
(list   '(a b) '(c d))    → ((a b) (c d))
(append '(a) '(b c) '(d)) → (a b c d)
(append '(a b) '() '(d))  → (a b d)
(append '(a b) 'c )         error:  'c is not a list
```

`append`, **`cons`**, and **`list`** are essential for building lists:
you should understand them!!

Why "car" and "cdr"?

- The names CAR and CDR come from LISP, which was first implemented on an IBM 704 computer.
- IBM 704 words had two fields: *address* and *decrement*, that could each store a memory address.
- LISP used these 2 fields to store the 2 elements of each node in a list (value and pointer to next node).



Why "car" and "cdr"?

The 704 had two machine instructions to access these values:

Contents of Address Register (*CAR*)

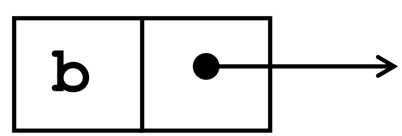
Contents of Decrement Register (*CDR*)

hence the name of the LISP commands!

... and LISP is supposedly "*machine independent*"... HA!

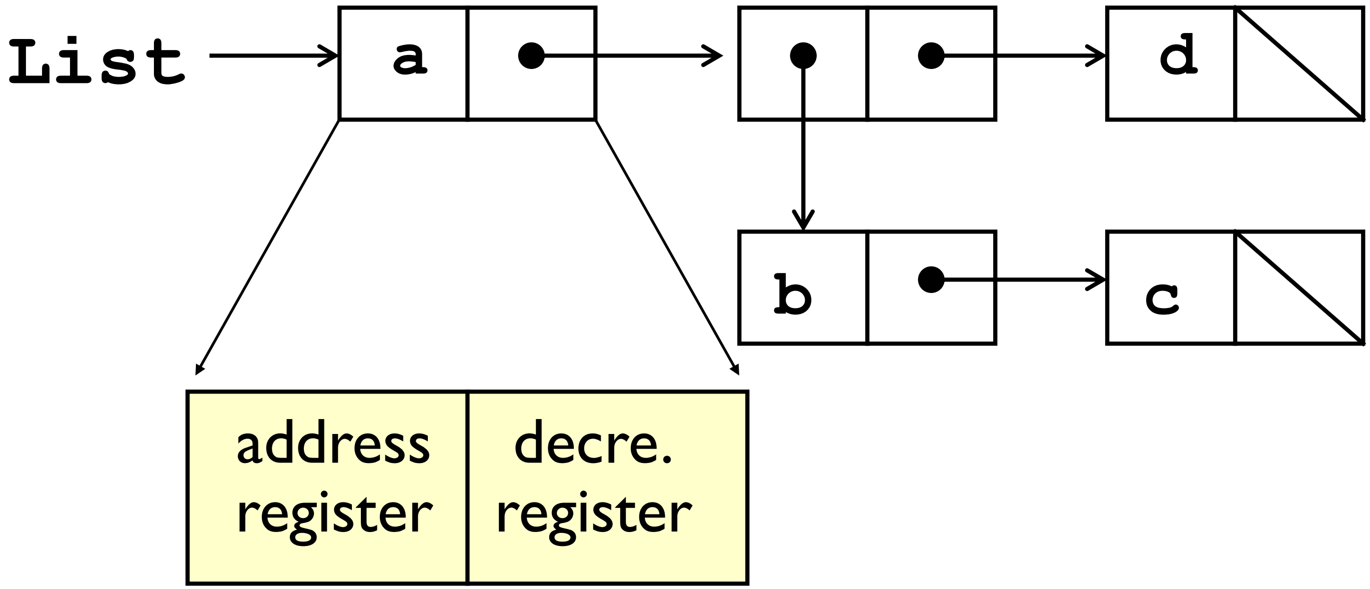
`(car list)` → **`a`**

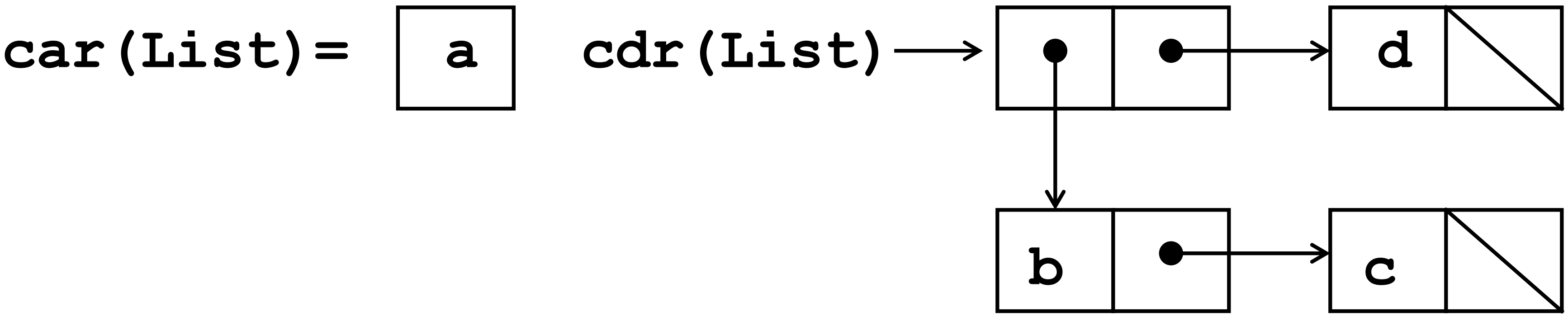
`(cdr list)` →



Why "car" and "cdr"?

How to store a list: **`List = ( a (b c) d )`**





Compound `car` and `cdr`

`car` and `cdr` are often used several times in succession:

`(car (cdr List) )` = get 2nd element of List

`(cdr (cdr List) )` = delete first 2 items from List

so, Scheme defines several compound car/cdr functions:

`(caar List )`	= (car (car List))
`(cadr List )`	= (car (cdr List))
`(caddr List )`	= (car (cdr (cdr List)))
`(cddddr List )`	= (cdr (cdr (cdr (cdr List))))
`(cdadadr List )`	= (cdr (car (cdr (car (cdr List)))))
`(cxxxxxr List )`	= any combination of up to 5 "a" and "d" refers to a composite of `car` and `cdr`

Tests

`(list?` *expression*`)`	true if *expression* is a list
`(null?` *expression*`)`	true if *expression* is a null list
`(eq?` *ex1* *ex2*`)`	true if *ex1* and *ex2* are both atoms and identical

```
(list? '(x y) )          → #T
(list? 'x)               → #F or ( )
(list? '())              → #T
(null? '(a b) )          → #F or ( )
(null? '() )             → #T
(null? 'A)               → #F or ( )
(eq? 'a 'a)              → #T
(eq? 'a '(a))            → #F or ( )
(eq? '(a b) '(a b))      → #T or #F (implementation dependent)
```

List Processing: *cdr down & cons up*

Many functions operate on a list and produce a new list as a result.

1. operate on the first element: (car List)

2. recursive call for rest of the list: (fun (cdr List))

3. put the results together (cons *first-result recursive-result*)

cdr down the list

```
; square each element of a list
(define (square-all L)
   (if (null? L) '()
      (cons (* (car L)(car L)) (square-all (cdr L)))
 ))
```

termination condition

cons join results together

List Manipulation: *cdr down & cons up*

"*cdr down and cons up*" is a common technique for applying operations to all elements of a list.

```
; append two lists
(define (append L M)
   (if (null? L) M
      (cons (car L) (append (cdr L) M)))
 )
; reverse a list
(define (reverse L)
   (if (null? L) '( )
   (append (reverse (cdr L)) (list (car L)))
   )
 )
```

cdr down the list

cons up the result

create a list from element

Textbook example: member

Write a "member" function that returns true if the first argument is a member of the second arg (a list):

`(member 'B '(A B C))`	returns true, #T
`(member 'B '(A C D))`	returns false, #F
`(member 'B '(A (B C) D))`	returns false, #F

Study textbook example

```
(define (member atm lis)
  (cond
    ((null? lis) #F )
    ((eq? atm (car lis)) #T) ; compare first elem
    (else (member atm (cdr lis))) ; compare rest
  )
)
```

`member` built-in function

Scheme has a member function. It returns the *tail* of the list beginning with the search element.

`(member 'B '(A B C))`	returns (B C)
`(member 'B '(A C D))`	returns #F
`(member '(B) '(A ((B) D)))`	returns ((B) D)

Scoping

- Scheme uses *static scope*.
- "let" creates a new scope:

```
(define a 10)
(define (f x )
      (let (a 10)    # create a new a
            (* a x)
      )
)
```

Filter Function

- A more interesting example: define a filter for lists.

```
> (filter odd? '(2 3 4 5 6 7 8) )

(3 5 7)
```

```
; extract elements from List that satisfy f
(define (filter f List)
 (cond
  ( (null? List) '() )
  ( (f (car List))      ; filter function is true
       (cons (car List) (filter f (cdr List))) )
  ( else
       (filter f (cdr List)) )
  )
)
```

filter example (1)

```
(filter even? '(1 2 4 7) )
(cond
   ( (null? '(1 2 4 7) ...)    #F
   ( (even? 1) ... )           #F
   ( else (filter even? '(2 4 7) ) ) #T
```

```
(filter even? '(2 4 7) )
(cond
   ( (null? '(2 4 7) ...) #F
   ( (even? 2) (cons 2 (filter even? '(4 7)) ) #T
)
```

```
(filter even? '(4 7) )
(cond
   ( (null? '(4 7) ...) #F
   ( (even? 4) (cons 4 (filter even? '(7)) ) #T
)
```

```
(filter even? '(7) )
```

filter example (2)

```
(filter even? '(7) )
(cond ...
     (else (filter even? '() ) )
```

returns: '( )

```
(filter even? '(4 7) )
(cond ...
     ( (even? 4) (cons 4 (filter even? '(7)) ) #T
```

returns: (cons 4 '()) => '(4)

```
(filter even? '(2 4 7) )
(cond ...
     ( (even? 2) (cons 2 (filter even? '(4 7)) ) #T
```

returns: (cons 2 '(4)) => '(2 4)

```
(filter even? '(1 2 4 7) )
(cond ...
        ( else (filter even? '(2 4 7) ) ) #T
```

returns: '(2 4)

Boolean and Numeric Functions

- **Boolean Functions**

(not *expression*)	logical negation
(and *expr1 expr2*)	logical and
(or *expr1 expr2*)	logical or

- **Numeric Functions**

(sqrt a)	returns $(a)^{1/2}$
(sqr a)	returns a^2
(expt A B)	returns A^B
(remainder A B)	remainder of integer division *A*/*B*
(log A)	natural logarithm of A
(sin A), (cos A), ...	trigonometric functions, A is in radians
(max a b c d ...)	max of several values
(min a b c d ...)	min of several values
(random n)	returns a random integer value 0, 1, ..., n-1

String Functions

(**string-length** "hello")	number of characters in string
(**substring** *string start end*)	return a substring
(**string->list** "hello there")	convert to list of characters
(**list->string** (#\h #\i))	convert list of chars to a string
(**string** #\h #\e #\l #\l #\o)	concatenate char args to new string
(**string-copy** *string*)	create a new string as copy of old
(**string?** *arg*)	true if argument is a string
(**string=?** *string1* ...)	there are also < > <= >= functions
(**string=?** "bye" "BYE")	false
(**string-ci=?** "bye" "BYE")	true. there are also < > <= >=
(**string-null?** *string*)	
(**string-index** *string char*)	
(**string-rindex** *string char*)	

#\C#\h#\a#\r#\a#\c#\t#\e#\r Functions

char? *obj*	true of obj is a character
char=? *char …*	equals comparison. also < <= > >=
char-ci=? *char …*	case insensitive =. also < <= > >=
(**char=?** #\A #\a)	false
(**char-ci=?** #\A #\a)	true
char-alphabetic? *char*	true if alphabetic character
char-numeric? *char*	true if numeric character
char-whitespace? *char*	true if whitespace
char-upper-case? *char*	
char-lower-case? *char*	
char->integer *char*	char to int, (char->integer #\a) is 50
integer->char *char*	integer char sequence num. to char
char-upcase *char*	
char-downcase char	

Special String and Char Values

String	Character	C/Java Equivalent
`\0`	`#\null`	`\0 or null`
`\f`	`#\ack`	`\f`
`\n`	`#\newline`	`\n`
`\r`	`#\return`	`\r`
`\t`	`#\tab`	`\t`
`\a`	`#\bel`	`\a`
`\v`	`#\vt`	`\v`

```
> (display "hello\tdog\n woof woof!")
hello   dog
 woof woof!
```

Code blocks using (begin ...)

begin is used to insert several expressions where a Scheme command expects one, like Pascal begin...end.

syntax: `(begin (expression1) (expression2) ... )`

```
(* Pascal *)
if (x < 0) then begin
   statement;
   statement;
   ...
end
else begin
   statement;
   statement;
   ...
end;
```

```
; Scheme
(if (< x 0) ( begin
   expression
   expression
   ...
)
( begin   ; else part
   expression
   expression
   ...
)  ) ; end of "(if..."
```

Testing Equality in Scheme

Scheme has many different equality functions:

- **`=`** applies only to numbers: `(= x y)`
- **`char=?`** applies only to characters: `(char=? #\a #\b)`
- **`string=?`** applies only to strings
- Each non-numeric data type has its own equality function.

General Equality Functions

There are three "generic" equality functions:

(eq? a b) test if a and b *refer* to same object, like == in Java

(eqv? a b) test if a and b are equivalent

(equal? a b) test atom-by-atom equality of lists. like obj.equals( ) in Java.

```
> (define L1 '(a b c) )
> (define L2 '(a b c) )
> (eq? L1 L2)
false
> (equal? L1 L2)
true
```

Scheme evaluation rules

1. Constant atoms, such as numbers and strings, evaluate to themselves: 4.2, "hello"

2. Identifiers are looked up in the current environment and replaced by the value found there.

3. A list is evaluated by recursively evaluating each element in the list (in an unspecified order).

4. The first expression in the list must evaluate to a function. This function is applied to the remaining values in the list. Thus, all expressions are in prefix form.

Example: Equals (1)

Write an "Equals" function that returns true if two lists are equal:

`(Equals '(B C) '(B C) )` returns true

`(Equals '(B C) '(B C D) )` returns false

```
(define (Equals L1 L2)
   (cond
       ( (null? L1) (null? L2) )
       ( (null? L2) #F )
       ( (eq? (car L1) (car L2))
              (Equals (cdr L1) (cdr L2) ) )
       ( else #F )
   )
)
```

Example: Equals (2)

Equals (1) doesn't work if the arguments are atoms

`(Equals 'a  'a )` Error

```
(define (Equals L1 L2)
   (cond
      ( (not (list? L1)) (eq? L1 L2) )
      ( (not (list? L2)) #F )
      ( (null? L1) (null? L2) )
      ( (null? L2) #F )
      ( (eq? (car L1)(car L2))
           (Equals (cdr L1) (cdr L2) ) )
      ( else #F )
   )
)
```

Example: Equals (3)

Equals (2) doesn't work if the list contains other lists

```
(Equals '((a b) c) '((a b) c) )
```

Fix this using more recursion...

```
(define (Equals L1 L2)
   (cond
      ( (not (list? L1)) (eq? L1 L2) )
      ( (not (list? L2)) '() )
      ( (null? L1) (null? L2) )
      ( (null? L2) '() )
      ( (Equals (car L1) (car L2))
             (Equals (cdr L1) (cdr L2) ) )
      ( else '() )
   )
)
```

More Examples

The boring GCD function:

```
(define (gcd u v) ; gcd of u and v
    (if (= v 0) u
        (gcd v (remainder u v))
    )
)
```

A "power" function to compute x^n, for integer n.

```
(define (power x n)
    (cond
        ((< n 0) (/ 1 (power x (- 0 n)))) ;1/x^(-n)
        ((= n 0) 1 )                      ;x^0 = 1
        (else (* x (power x (- n 1))))
    )
)
```

for-each

Syntax:

```
(for-each  function  list1 [list2 ...] )
```

function is called repeatedly; on the *k-th* call it is given the *k-th* element from *list1*, *list2*, ...

The lists (list1 list2 ...) must have the same lengths.

```
> (for-each * '(3 4 5) '(5 2 20) )
>                               ; nothing returned
> (for-each (lambda (a b)
        (display (* a b )) (newline))
     '(3 4 5) '(5 2 20))
15                              ; 3*5
8                               ; 4*2
100                             ; 5*20
```

Evaluating Expressions: `eval`

- The Scheme interpreter "executes" your program by invoking the **`eval`** function.
- The interpreter is said to run a "read - eval - print" loop.
- You can use **`eval`** in your code, too.

Usage **`(`** `eval` ***`list`*** **`)`**

```
> (define hi '( display "hello" ) )
>
> (eval hi )
Hello
```

How to...

- How would you use eval to create a "sum" function:

Usage `( sum` *`list-of-numbers`* `)`

```
> ( sum '( 1 2 3 4 5 ) )
15
```

This won't work:

`(+` *`list-of-numbers`* `)`

What we want is:

```
 '+ (     1 2 3 4 5 )
```

Evaluating Expressions: `eval`

Example: Sum the elements in a list

```
> (define data '( 12 3 7 99 23 17 88 ) )
> (cons '+ data )
( + 12 3 7 99 23 17 88 )
> (eval (cons '+ data ) )
249
```

Exercise using `eval`

Exercise:

1. Define a "sum" function that sums a list: **`(sum list)`**
2. Define a "square" function: **`(square x)`** is **`x`**^2
3. Define an **`average`** function that computes the average of a list of numbers. Use **`length`** .
4. Define a **`variance`** function that computes the variance of a list of numbers. The variance can be computed as:

   variance = (average data^2) - (average data)^2

Ex: (variance '(0 1)) is 1/4

   (variance '(20 20 20 20 20)) is 0

Use of `eval`

- **`eval`** enables your program to write its own instructions and then run them!
- this is another way to create programs that learn. In comparison:
  - **`lambda`** returns a new, unnamed function
  - **`eval`** evaluates a list.

Applicative Order Evaluation

- ***Applicative*** (aka *eager*) ***order evaluation***: arguments are evaluated at the call location (caller) before they are passed to the called function.

  Example: `(* (sin x) (/ x 2) )`

  compute `sin(x)` and `(x/2)` first, then call * to multiply
- *Applicative* evaluation is used in most languages, including C, Java, Pascal, and Scheme... *with exceptions*.

Q: In what situations might applicative order evaluation be a problem?

Delayed Evaluation

Consider the use of "if" in C:

```
if ( x > 0 ) y = log(x);
else printf("x must be positive");
```

- `y = log(x)` is only evaluated if (x>0) is true
- `printf(...)` is only evaluated if (x>0) is false

- This is an example of *delayed evaluation*:
  - delayed evaluation is an essential part of "if"
  - *then* part is only evaluated if test condition is true.
- Also called *normal order evaluation*

Delayed Evaluation (2)

- Consider "if" and "and" in Scheme:

  `(if (> x 0) (log x) (display "x must be pos") )`

  if *applicative order evaluation* is used here, then all three expressions would be evaluated *before* the "if" function is called! Result:

  - `(log x)` would *always* be computed
  - `(display "...")` would *always* be executed

- `(if a b c)` must use *delayed evaluation*.

Lazy Evaluation

- An expression is evaluated the *first time it is needed*.
- It is not evaluated again.
- This is not the same as "normal order" evaluation.

Short-circuit Evaluation in C and Java

- The && and || operators use ***short-circuit evaluation***:

```
if ( x != 0 && y/x < 1 ) ...;
```

  - `y/x` is only evaluated if (`x != 0`) is true

```
if ( x == 0 || log(x) < 1 ) ...;
```

  - `log(x)` is only evaluated if (`x == 0`) is *false*

- C, C#, and Java *guarantee* this property of && and ||

Short-circuit Evaluation in Scheme

- What about "**`and`**" in Scheme?

```
(if (and (f x) (g x))
     (display "True")   ;; then statement
     (display "False") ;; else statement
)
```

does Scheme *always* evaluate `(g x)` ?

- Write a test program:

```
(define (f x) (display "F called") #f )
(define (g x) (display "G called") #t )
```

evaluate:

```
(and (f 1) (g 1) )        (or (f 1) (g 1) )
(and (g 1) (f 1) )        (or (g 1) (f 1) )
```

Delayed Evaluation (3)

Other situations where delayed evaluation is needed:

- **`cond`**

  ```
  (cond
       (condition1 expression1)
       (condition2 expression2) ...
       (else       expression3)
  )
  ```

  evaluation of arguments is *delayed* until they are needed by "cond".

- Producer - consumer relationship (described next).

Delayed Evaluation (4)

- Applications can also benefit from delayed evaluation:

```
; generate a list of integers from m to n
(define (intlist m n)
   (if (> m n) '()
       (cons m (intlist (+ 1 m) n)) )
)
; extract first n items from list
(define (head n List)
   (if (= n 0) '()
       (cons (car List)(head (- n 1)(cdr List)))
   )
)
; extract first 10 items from list of integers
(head 10 (intlist 1 100000) )
```

Delayed Evaluation (5)

- **`intlist`** is evaluated first, but only the first 10 items are used, so the rest of the work was *unnecessary*.

```
; extract first 10 items from list
(head 10 (intlist 1 100000) )
```

delay and force

- `delay` requests that an expression be evaluated later.
- Scheme creates a "promise" to evaluate it at a later time:

```
(delay (+ 3 4))
#<struct:promise>
```

- `force` causes the "promise" to be fulfilled.
- The expression is evaluated and the result returned.
- The promise is not turned into a value -- it stays a promise.

```
> (define E (delay (+ 3 4)))
> E
#<struct:promise>
> (force E)
7
> (* E 2)      ; multiply E by 2
*: expects type <number> as 1st argument,
   given: #<struct:promise>
```

E is still a promise, not a value

Uses of delay

- avoid a long calculation that may not be necessary
- to enable a function to generate an infinite amount of data
  - each time the caller wants the next term he forces the tail of the function.
  - this causes the calculation to perform one increment

```
;; generate all the integers starting from n
(define (integers n)
     (cons n (integers (+ n 1) ) )
     )
```

This recursion will never stop!

Applying `delay` and `force`

- Rewrite integers to use **`delay`** :

```
; generate a list of all integers starting at n
(define (integers n)
    (cons n (delay (integers (+ 1 n)) ) )
 )
; extract first n items from list
(define (head n list)
    (if (= n 0) '()
       (cons (car (force list))
          (head (- n 1)(cdr (force list))))
    )
 )
; example usage:
(head 100 (delay (integers 1) ) )
```

Applying `delay` and `force` (2)

- Each reference to the list generator is *wrapped* in (**`delay ...`**) or (**`force`** ...)

```
; generate a list of all integers
(define (integers n)
    (cons n (delay (integers (+ 1 n)) ) )
 )
; consume first n items from list
(define (head n list)
    (if (= n 0) '()
       (cons (car (force list))
          (head (- n 1)(cdr (force list))))
    )
 )
```

The generator *delays* building the list;
The consumer *forces* each next step of list building.

Is this inefficient?

- "head" (list consumer) forces the list 2 times.
- Does that cause the generator to do the work twice?

```
; generate a list of all integers
(define (integers n)
    (cons n (delay (integers (+ 1 n)) ) )
 )
; consume first n items from list
(define (head n list)
    (if (= n 0) '()
       (cons (car (force list))
          (head (- n 1)(cdr (force list))))
    )
 )
```

Memoization

- Promises would be inefficient if they are evaluated every time a delayed expression is forced:

```
(define E (delay (+ 1 2 3 4 5 6 7 8 9 10)))
(define (f-square x) (* (force x) (force x)))
```

- **`(f-square E)`** would require the sum to be computed twice.
- To avoid this, promises are **memo-ized**:
  - when a promise is **`force`**-d, the value is stored
  - subsequent "**`force`**" simply return the stored value
- Thus, a **`delay`** expression is evaluated *at most* once.

Memoization *Exposed*

- Memoization can be seen in Scheme using a definition that involves I/O:

```
(define SayHello (delay (display "Hello There")))
> SayHello
# <struct:promise>
> (force SayHello)
Hello There
> (force SayHello)
>
```

No output from the second "force".

Functions as 1st class entities

- Result of a **`lambda`** can be manipulated as ordinary data:

```
> ( define f (lambda (x) (* x x)) )
> f
(#<procedure>)
> (f 5)
25
```

```
(define (scale-by f) ( lambda(x) (* x f) )  )
(define inch2cm (scale-by 2.54) ) ; inches to cm
(inch2cm 20 )                     ; use the func.
50.8
```

Higher-order functions

- A *higher-order* function returns a function as its value, or takes a function as a parameter.
- **`eval`** and **`map`** are higher-order
- The use of higher-order functions is a characteristic of functional programs

Higher-order functions

```
; apply function to all values in a list
(define apply-to-all  ( fun  values )
    (if (null? values) '()
        (cons (fun (car values) )
         (apply-to-all fun (cdr values) ) ) ; recursion
    )
)
```

The Scheme **map** function performs apply-to-all. Example:

```
;; compute factorial of all values in a list
> (map factorial '(1 2 3 4 5 6) )
(1 2 6 24 120 720)
```

Higher-order functions

```scheme
; apply a filter (p) to elements of a list L
(define (filter p L)
  (cond
    ( (null? L) L )
    ( (p (car L)) (cons (car L) (filter p (cdr L))) )
    ( else (filter p (cdr L)) )
  )
)
```

Example:

```scheme
> (filter even? '(1 2 4 7) )
```

see next slide for step-by-step evaluation.

`power` function generator

- Define a square function and a cube function:

```
(define (square x) (* x x) )
(define (cube x) (* x x x) )
> (square 3)
9
> (cube 3)
27
```

- Can we define a function that can generate *any* power function?
  (for integer powers) That is:

```
(define square (power 2) ) ; square is a function
(define cube (power 3) )   ; cube is a function
(define inverse (power -1) )
```

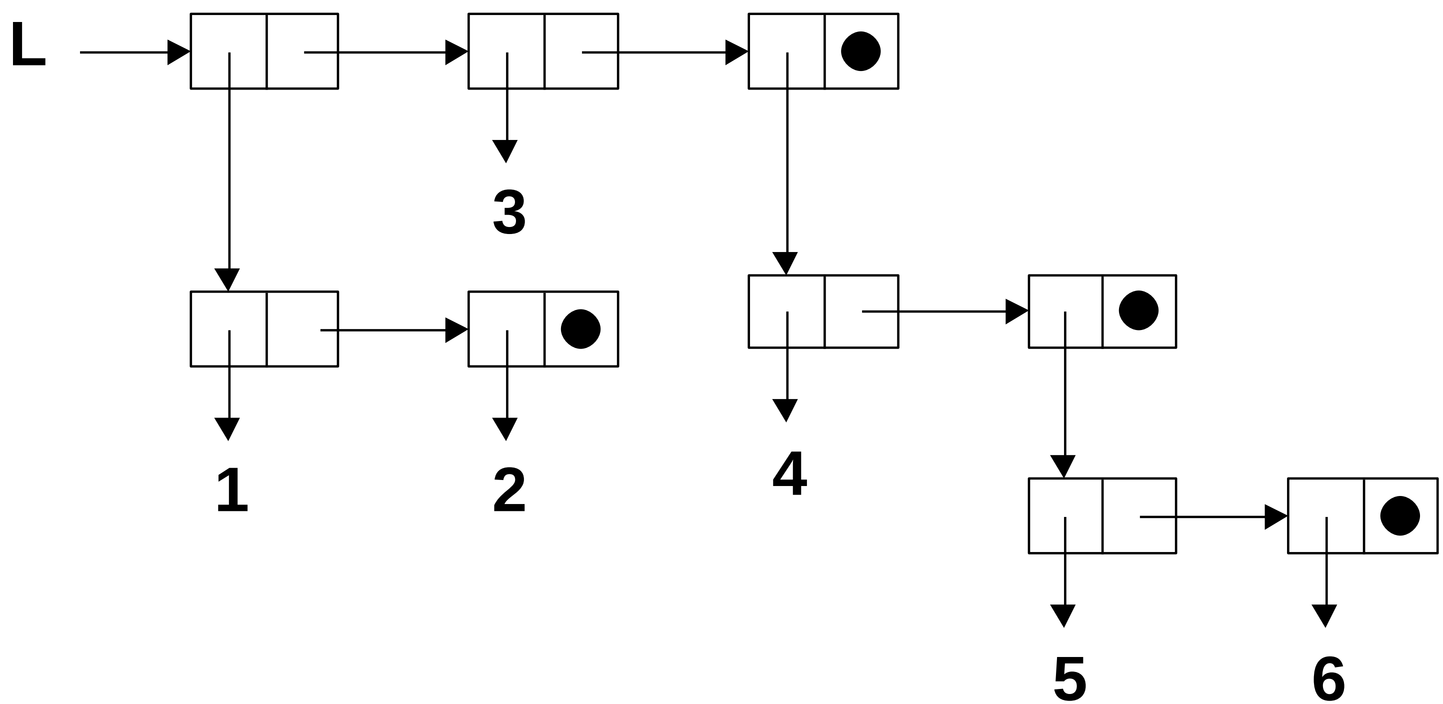
power function generator (cont'd)

```
(define (power n)
   (cond
      ; n = 0 is a constant function: x^0 = 1
      ( (= n 0) (lambda(x) 1 ) )
      ; n = 1 is the identity function
      ( (= n 1) (lambda(x) x ) )
      ; n < 0 use the property: x^n = 1/x^(-n)
      ( (< n 0) (lambda(x)
                (/ 1 ((power (- 0 n)) x) ) ) )
      ; n > 1 define recursively: x^n = x*x^(n-1)
      ( else    (lambda(x)
              ...complete this as an exercise
   )
)
```

Organization of Lists in Scheme

- Lists are stored as … linked lists.
- The "car" of each node contains a type identifier (number, string, list) and pointer to value.

Example: L = ((1 2) 3 (4 (5 6)))



Question: *alternative to* car and cdr

- In Scheme:

  `(car '(a b c d e ...) )` = a = first element

  `(cdr '(a b c d e ...) )` = '(b c d e) = remainder

- What would be the effect of replacing **`car`** and **`cdr`** with a "head" and "tail" like this:

  `(head '(a b c ... m n) ) = '(a b c ... m)`

  `(tail '(a b c ... m n) ) = n` = last element

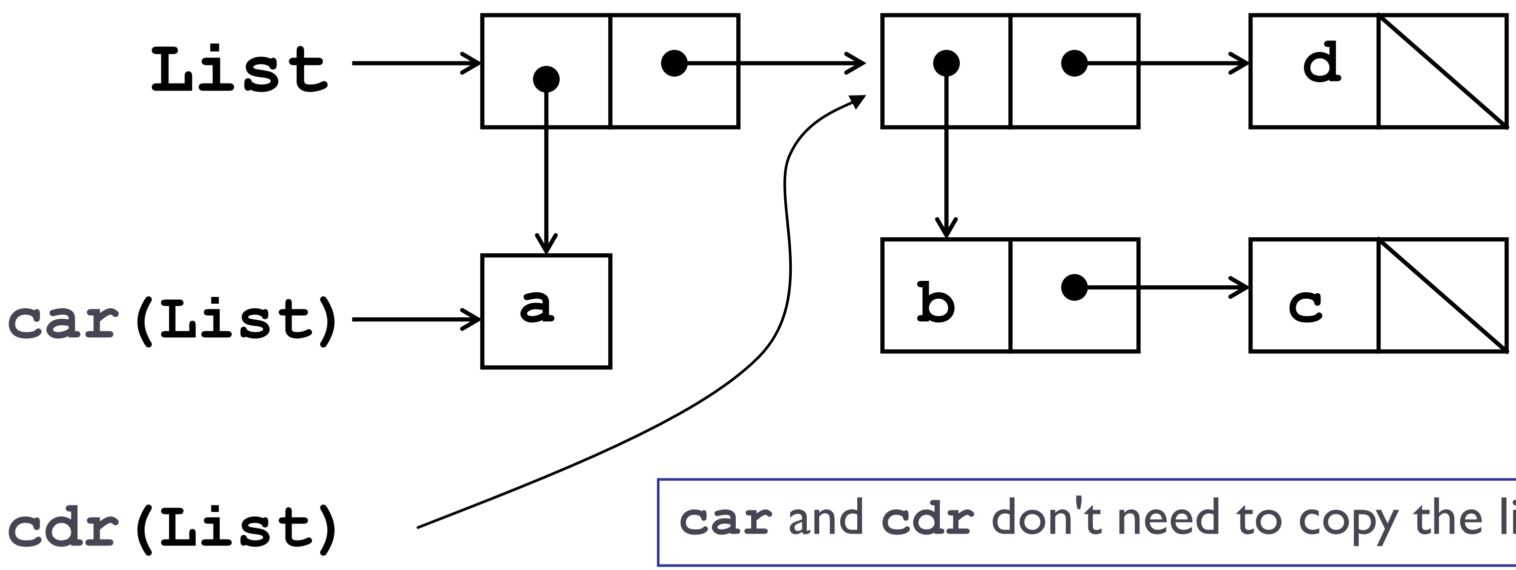
- For a linked list structure, is one more efficient? **`car-cdr`** or **`head-tail`**

Question: *alternative to* car and cdr

Consider: `List = ( a (b c) d )`

`car(List) = a`

`cdr(List) = ( (b c) d )`



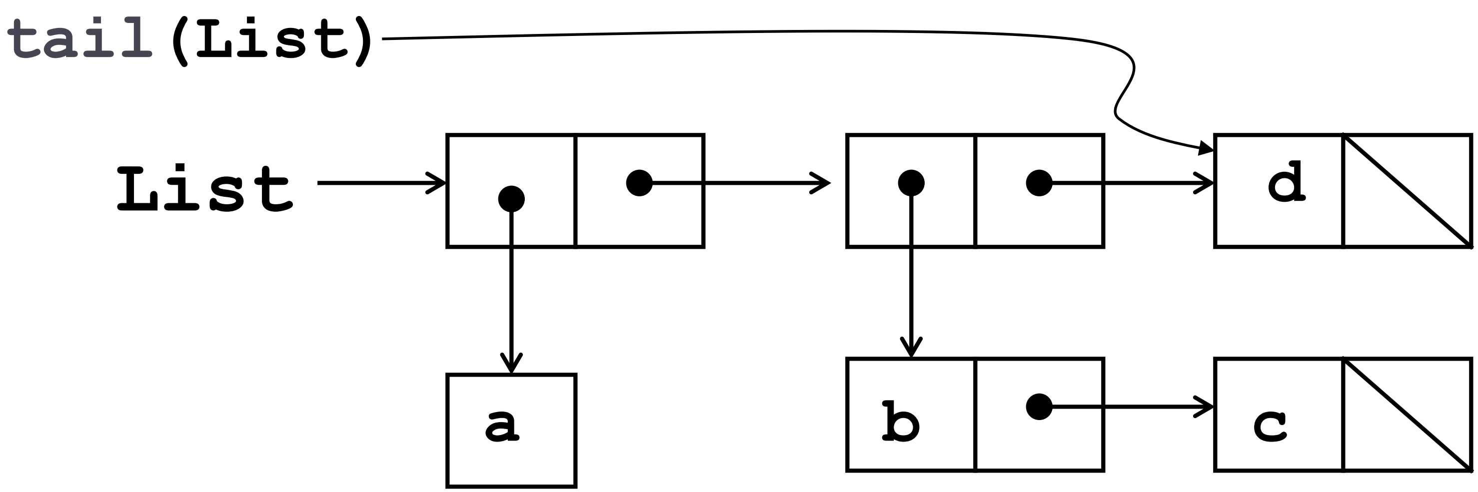
`car` and `cdr` don't need to copy the list.

Question: *alternative to* car and cdr

Consider: `List = ( a (b c) d )`

`head(List) = ( a (b c) )`

`tail(List) = d`



`head(List) = ?`

Data structures in Scheme

Scheme has a set of cludgy commands for managing data structures.

- `define-struct` defines a new structure

```
(define-struct person '(name telephone))
```

- `make-`*structname* creates a new instance

```
(define shin (make-person "Taksin" '01-5551212))
```

- ***`struct-attribute`*** commands are accessors:

```
> (person-name shin)
"Taksin"
> (person-telephone shin)
01-5551212
```

- **`make-person`**, **`person-name`**, ... are defined automatically when you use "**`define-struct`**"

Object-oriented programming

- Since functions are first class objects, an element of a structure can be a function.

Binary search trees

- Represent each node as a 3-item list: (data left-child right-child)

  '(mydata (left-child-list) (right-child-list))

```
(define (data B) (car B))
(define (leftchild B) (cadr B))
(define (rightchild B) (caddr B))
```

- Example - see Figure 11.8, page 487:

```
("horse"("cow"()("dog"()()))
        ("zebra"("yak"()())()))
```

- Now we can write traversals such as

```
(define (tree-to-list B)
  (if (null? B) B
      (append (tree-to-list (leftchild B))
         (list (data B))
         (tree-to-list (rightchild B)))))
```

`eval` and symbols

- The **`eval`** function evaluates an expression in an environment; many Scheme versions have an implied current environment:
  `(eval (cons max '(1 3 2)) => 3`
- Symbols are virtually unique to Lisp: they are runtime variable names, or unevaluated identifiers:
  `'x => x`
  `(eval 'x)` => the value of **`x`** in the environment
- Use symbols for enums (they are more efficient than strings)

Functions and objects

- Functions can be used to model objects and classes in Scheme.
- Consider the simple Java class:

```
public class BankAccount
{ public BankAccount(double initialBalance)
    { balance = initialBalance; }
    public void deposit(double amount)
    { balance = balance + amount; }
    public void withdraw(double amount)
    { balance = balance - amount; }
    public double getBalance()
    { return balance; }
    private double balance;
}
```

Functions and objects (cont'd)

- This can be modeled in Scheme as:

```
(define (BankAccount balance)
   (define (getBalance) balance)
   (define (deposit amt)
     (set! balance (+ balance amt)))
   (define (withdraw amt)
     (set! balance (- balance amt)))
   (lambda (message)
     (cond
       ((eq? message 'getBalance) getBalance)
       ((eq? message 'deposit) deposit)
       ((eq? message 'withdraw) withdraw)
       (else (error "unknown message" message))) )
)
```

Functions and objects (cont'd)

- This code can be used as follows:

```
> (define acct1 (BankAccount 50))
> (define acct2 (BankAccount 100))
> ((acct1 'getbalance))
50
> ((acct2 'getbalance))
100
> ((acct1 'withdraw) 40)
> ((acct2 'deposit) 50)
> ((acct1 'getbalance))
10
> ((acct2 'getbalance))
150
> ((acct1 'setbalance) 100)
. unknown message setbalance
```

Imperative Commands in Scheme

- In the BankAccount code **`set!`** enables us to treat a symbol as a memory location and alter its value. **`set!`** is *not* purely functional programming:

  `(set! balance (+ balance amount) )`

- In Scheme, any function ending in "!" is a non-functional, imperative-style operation. These include:

  `set!`

  `set-car!`

  `set-cdr!`

  `string-set!`

  etc.

  All of these are versions of assignment.