CMPUT325 Extensions to Pure Lisp

Bob Price and Russ Greiner

5th October 2004

Extensions to Pure Lisp

- "Extensions" to Pure *Lisp*
  - Side Effects (setq, putprop, . . .)
  - Numbers
  - Dotted-Pair, Association & Property Lists
  - *Lisp qua* Procedural Language (i/o, do, ...)

No Side Effects

`(+ 11 23)` $\longrightarrow$

No Side Effects

`(+ 11 23)` $\longrightarrow$ `34`

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→
```

No Side Effects

```
(+ 11 23)  →  34
(CAR '(A B C))→  A
```

No Side Effects

```
(+ 11 23)  →   34
(CAR '(A B C))→   A

(+ 11 23)  →
```

No Side Effects

```
(+ 11 23)  →   34
(CAR '(A B C))→   A

(+ 11 23)  →    34
```

No Side Effects

```
(+ 11 23) →  34
(CAR '(A B C))→  A

(+ 11 23) →   34

(+ (* X 2) 5) →
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→ A

(+ 11 23) → 34

(+ (* X 2) 5) → X undefined
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→ A

(+ 11 23) → 34

(+ (* X 2) 5) → X undefined
(+ 11 23) →
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C)) → A

(+ 11 23) → 34

(+ (* X 2) 5) → X undefined
(+ 11 23) → 34
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→ A

(+ 11 23) → 34

(+ (* X 2) 5) → X undefined
(+ 11 23) → 34

(CAR '(A B C)) →
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C)) → A

(+ 11 23) → 34

(+ (* X 2) 5) → X undefined
(+ 11 23) → 34

(CAR '(A B C)) → A
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→ A

(+ 11 23) →  34

(+ (* X 2) 5) → X undefined
(+ 11 23) → 34

(CAR '(A B C)) → A
(+ (* X 2) 5)} →
```

No Side Effects

```
(+ 11 23) → 34
(CAR '(A B C))→ A

(+ 11 23) →  34

(+ (* X 2) 5) → X undefined
(+ 11 23) → 34

(CAR '(A B C)) → A
(+ (* X 2) 5)} → X undefined
```

Side-Effect Free — Def'n

- Form σ has NO side effects if
  Evaluating σ does not affect the value of any other expression τ.

Side-Effect Free — Def'n

- Form σ has NO side effects if
  Evaluating σ does not affect the value of any other expression τ.
- Hence: Value of form τ is the same
  whether or not σ was evaluated .

$\rightarrow \underline{\tau}$

$\langle \texttt{v1} \rangle$

$\rightarrow \underline{\sigma}$

$\langle \texttt{v2} \rangle$

$\rightarrow \underline{\tau}$

$\langle \texttt{v1} \rangle$

Side-Effect Free — Def'n

- Form σ has NO side effects if
  Evaluating σ does not affect the value of any other expression τ.
- Hence: Value of form τ is the same
  whether or not σ was evaluated .

  $\rightarrow$ $\underline{\tau}$

  $\langle$v1$\rangle$

  $\rightarrow$ $\underline{\sigma}$

  $\langle$v2$\rangle$

  $\rightarrow$ $\underline{\tau}$

  $\langle$v1$\rangle$

- Examples: Any form using only +, `CAR`, `CONS`, ...

Functions with Side Effects – SETQ

my-const →

Functions with Side Effects – SETQ

`my-const` $\longrightarrow$ `undefined variable`

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) →
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const →
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) →
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) → (t 4)
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) → (t 4)
my-const→
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) → (t 4)
my-const→ (t 4)
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) → (t 4)
my-const→ (t 4)
(CAR my-const) →
```

Functions with Side Effects – SETQ

```
my-const → undefined variable
(SETQ my-const '(A B C)) → (A B C)
my-const → (A B C)
(CAR my-const)→ A
(SETQ my-const '(t 4)) → (t 4)
my-const→ (t 4)
(CAR my-const) → t
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

The mysterious SETF

- SETF chooses a modifier function according to its first argument

  (SETF x '(1 2 3)) $\longrightarrow$

The mysterious SETF

- SETF chooses a modifier function according to its first argument

  ```
  (SETF x '(1 2 3))→
  (1 2 3) ;;≡(SETQ x '(1 2 3))
  ```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X →
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) →
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →  (a 2 3)
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →  (a 2 3)

(SETF A (make-array '(2 2)))
→
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →  (a 2 3)

(SETF A (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →  (a 2 3)

(SETF A (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))

(SETF (aref A 0 0) 'q)
→
```

The mysterious SETF

- SETF chooses a modifier function according to its first argument

```
(SETF x '(1 2 3))→
(1 2 3) ;;≡(SETQ x '(1 2 3))
X → (1 2 3)

(SETF (car x) 'a) → (a 2 3) ;; ≡(RPLACA x 'a)
X →  (a 2 3)

(SETF A (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))

(SETF (aref A 0 0) 'q)
→ #2A((Q NIL) (NIL NIL))
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) →
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) →
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) →
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→ (C)
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→ (C)
(setf (symbol-function 'my-fn)
    (FUNCTION CAR) )
    →
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→ (C)
(setf (symbol-function 'my-fn)
    (FUNCTION CAR) )
    → #<complied-function car>
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→ (C)
(setf (symbol-function 'my-fn)
    (FUNCTION CAR) )
    → #<complied-function car>
 (my-fn '(A B C)) →
```

Functions with Side Effects – SETF

```
( (LAMBDA (X) (CDR X)) '(A B C) ) → (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) → (t)
(setf (symbol-function 'my-fn)
    (FUNCTION (LAMBDA (X) (CDR X))) )
→ (LAMBDA-CLOSURE ...(X) (CDR X)))
(my-fn '(A B C)) → (B C)
(my-fn '(Q t)→ (t)
(my-fn (my-fn my-const)))→ (C)
(setf (symbol-function 'my-fn)
    (FUNCTION CAR) )
    → #<complied-function car>
 (my-fn '(A B C)) → A
```

Functions with Side Effects – SETF

```
(setf (symbol-function '+)
      (symbol-function '-))
```

- Simultaneous assignment

```
(setf a 1 b 2 c 3)
```

a $\longrightarrow$1

b $\longrightarrow$2

c $\longrightarrow$3

Functions with Side Effects – SETF

```
(setf (symbol-function '+)
      (symbol-function '-)) ;; DON'T DO THIS!!
```

- Simultaneous assignment

```
(setf a 1 b 2 c 3)
```

a $\longrightarrow$1

b $\longrightarrow$2

c $\longrightarrow$3

Functions with Side Effects – SETF

```
(setf (symbol-function '+)
      (symbol-function '-)) ;; DON'T DO THIS!!

(setf (symbol-function 'bye)
```

- Simultaneous assignment

```
(setf a 1 b 2 c 3)
```

a $\longrightarrow$1

b $\longrightarrow$2

c $\longrightarrow$3

Functions with Side Effects – SETF

```
(setf (symbol-function '+)
      (symbol-function '-)) ;; DON'T DO THIS!!

(setf (symbol-function 'bye)
(FUNCTION (LAMBDA () "Not so quick bit brain!")))
```

- Simultaneous assignment

```
(setf a 1 b 2 c 3)
```

a $\longrightarrow$1

b $\longrightarrow$2

c $\longrightarrow$3

User-Defined Function with Side Effects

`(SETQ my-var 5)` $\longrightarrow$

User-Defined Function with Side Effects

(SETQ my-var 5) $\longrightarrow$ 5

User-Defined Function with Side Effects

$\texttt{(SETQ my-var 5)} \longrightarrow 5$

$\texttt{my-var} \longrightarrow$

User-Defined Function with Side Effects

`(SETQ my-var 5)` $\longrightarrow$ 5

`my-var` $\longrightarrow$ 5

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var →
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) →
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var →
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var → (A B C)
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var → (A B C)
(fn2 (LIST (+ 3 4)))→
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var → (A B C)
(fn2 (LIST (+ 3 4)))→ (7)
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var → (A B C)
(fn2 (LIST (+ 3 4)))→ (7)
my-var→
```

User-Defined Function with Side Effects

```
(SETQ my-var 5) → 5
my-var→ 5
(SETF (symbol-function 'fn2)
      '(LAMBDA (X) (SETQ my-var X)) ) →
(LAMBDA (X) ...)
my-var → 5
(fn2 '(A B C)) → (A B C)
my-var → (A B C)
(fn2 (LIST (+ 3 4)))→ (7)
my-var→ (7)
```

SETF symbol-function and DEFUN

- (DEFUN name (v_1 ... v_n) ⟨form⟩) is an ABBREVIATION for

```
(SETF (symbol-function name)
 (FUNCTION (LAMBDA (v1 ... vn) ⟨form⟩)))
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
B →
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
B → 5
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

  ```
  (SETQ b '5)
  ```

  B $\longrightarrow$ 5

  X $\longrightarrow$

The SETQ Function

- SETQ does NOT evaluate its first argument.

  ```
  (SETQ b '5)
  B → 5
  X → undefined
  ```

The SETQ Function

- SETQ does NOT evaluate its first argument.

  (SETQ b '5)

  B $\longrightarrow$ 5

  X $\longrightarrow$ undefined

  (setq x 'b) $\longrightarrow$

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
B → 5
X → undefined
(setq x 'b) → B ;; Not an error!
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
B → 5
X → undefined
(setq x 'b) → B ;; Not an error!
X→B
```

The SETQ Function

- SETQ does NOT evaluate its first argument.

```
(SETQ b '5)
B → 5
X → undefined
(setq x 'b) → B ;; Not an error!
X→B
B →5 ;; but B's value unchanged.
```

The `SET` Function

- `SET` DOES evaluate its first argument.

  `B` $\longrightarrow$

The SET Function

- SET DOES evaluate its first argument.

  B $\longrightarrow$ undefined

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X →
```

The SET Function

- SET DOES evaluate its first argument.

  B $\longrightarrow$ undefined
  X $\longrightarrow$ undefined

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) →
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
```

The SET Function

- SET DOES evaluate its first argument.

  `B` $\longrightarrow$ `undefined`

  `X` $\longrightarrow$ `undefined`

  `(set X '(foo bar))` $\longrightarrow$ `x undefined`

  `(set 'X '(foo bar))` $\longrightarrow$

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) →
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X →
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
```

The SET Function

- SET DOES evaluate its first argument.

  ```
  B → undefined
  X → undefined
  (set X '(foo bar)) → x undefined
  (set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
  (setq X 'B) → B ;; Now X ←B
  X → B
  (set X (+ 100 12))→
  ```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
(set X (+ 100 12))→
112  ;;Note: Changes value of X's value (B)
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
(set X (+ 100 12))→
112  ;;Note: Changes value of X's value (B)
X →
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
(set X (+ 100 12))→
112  ;;Note: Changes value of X's value (B)
X → B
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
(set X (+ 100 12))→
112  ;;Note: Changes value of X's value (B)
X → B
B →
```

The SET Function

- SET DOES evaluate its first argument.

```
B → undefined
X → undefined
(set X '(foo bar)) → x undefined
(set 'X '(foo bar)) →(foo bar) ;; Now X ←(foo bar)
(setq X 'B) → B ;; Now X ←B
X → B
(set X (+ 100 12))→
112  ;;Note: Changes value of X's value (B)
X → B
B → 112
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
;; don't need quote for numbers
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  5 $\longrightarrow$

  ```
  ;; don't need quote for numbers
  ```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  $5 \longrightarrow 5$

  ```
  ;; don't need quote for numbers
  ```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) →
;; don't need quote for numbers
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  ```
  5 → 5
  (list 5 'a) → (5 a)
  ;; don't need quote for numbers
  ```

- `Numberp` tests whether an s-expr is a numeric atom.

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  ```
  5 → 5
  (list 5 'a) → (5 a)
  ;; don't need quote for numbers
  ```

- `Numberp` tests whether an s-expr is a numeric atom.

  ```
  (numberp 12) →
  ```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  `5` $\longrightarrow$ `5`
  `(list 5 'a)` $\longrightarrow$ `(5 a)`
  `;; don't need quote for numbers`

- `Numberp` tests whether an s-expr is a numeric atom.

  `(numberp 12)` $\longrightarrow$ `t`

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

- `Numberp` tests whether an s-expr is a numeric atom.

```
(numberp 12) → t
(numberp 'a) →
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

- `Numberp` tests whether an s-expr is a numeric atom.

```
(numberp 12) → t
(numberp 'a) → nil
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

  ```
  5 → 5
  (list 5 'a) → (5 a)
  ;; don't need quote for numbers
  ```

- `Numberp` tests whether an s-expr is a numeric atom.

  ```
  (numberp 12) → t
  (numberp 'a) → nil
  (setq n 25)
  (numberp n) →
  ```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

- Numberp tests whether an s-expr is a numeric atom.

```
(numberp 12) → t
(numberp 'a) → nil
(setq n 25)
(numberp n) →
t    ;; numberp evaluates its arguments
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

- Numberp tests whether an s-expr is a numeric atom.

```
(numberp 12) → t
(numberp 'a) → nil
(setq n 25)
(numberp n) →
t     ;; numberp evaluates its arguments
(numberp '(1 2)) →
```

Numbers in Lisp

- *Numbers* are special atoms: (Each evaluates to itself.)

```
5 → 5
(list 5 'a) → (5 a)
;; don't need quote for numbers
```

- `Numberp` tests whether an s-expr is a numeric atom.

```
(numberp 12) → t
(numberp 'a) → nil
(setq n 25)
(numberp n) →
t     ;; numberp evaluates its arguments
(numberp '(1 2)) → nil
```

Types of Numbers in Lisp

- Rational
  - Integers
    - Fixnums
    - Bignums
  - Ratios

Types of Numbers in Lisp

- Rational
  - Integers
    - Fixnums
    - Bignums
  - Ratios
- Floats

Types of Numbers in Lisp

- Rational
  - Integers
    - Fixnums
    - Bignums
  - Ratios
- Floats
- Complex Floats

Types of Numbers in Lisp

- Rational
  - Integers
    - Fixnums
    - Bignums
  - Ratios
- Floats
- Complex Floats
- No irrationals!!

Integers

- There is no apriori limit on size of an integer

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` $\rightarrow$

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` $\longrightarrow$ 32

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` $\longrightarrow$ `32`

  `(expt 2 100)`

  $\longrightarrow$

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` → `32`

  `(expt 2 100)`

  → `1267650600228229401496703205376`

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` → `32`
  `(expt 2 100)`
  → `1267650600228229401496703205376`

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$

Integers

- There is no apriori limit on size of an integer

  ```
  (expt 2 5) → 32
  (expt 2 100)
  → 1267650600228229401496703205376
  ```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` $\longrightarrow$ 32

  `(expt 2 100)`

  $\longrightarrow$ 1267650600228229401496703205376

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer

Integers

- There is no apriori limit on size of an integer

```
(expt 2 5) → 32
(expt 2 100)
→ 1267650600228229401496703205376
```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

Integers

- There is no apriori limit on size of an integer

```
(expt 2 5) → 32
(expt 2 100)
→ 1267650600228229401496703205376
```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

```
#10r15 →
```

Integers

- There is no apriori limit on size of an integer

```
(expt 2 5) → 32
(expt 2 100)
→ 1267650600228229401496703205376
```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

```
#10r15 → 15
```

Integers

- There is no apriori limit on size of an integer

```
(expt 2 5) → 32
(expt 2 100)
→ 1267650600228229401496703205376
```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

```
#10r15 → 15   #2r1111 →
```

Integers

- There is no apriori limit on size of an integer

  `(expt 2 5)` → `32`
  `(expt 2 100)`
  → `1267650600228229401496703205376`

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

  `#10r15` → `15`   `#2r1111` → `15`

Integers

- There is no apriori limit on size of an integer

```
(expt 2 5) → 32
(expt 2 100)
→ 1267650600228229401496703205376
```

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

```
#10r15 → 15   #2r1111 → 15   #3r120 →
```

Integers

- There is no apriori limit on size of an integer

  (expt 2 5) $\longrightarrow$ 32
  (expt 2 100)
  $\longrightarrow$ 1267650600228229401496703205376

- Smaller numbers are more efficient
  - Called "fixnums" and guaranteed to range at least $(-2^{16}, 2^{16})$
- Storage is automatically added as required
  - Large integers are called "bignums"
- Generally transparent to programmer
- Can use arbitrary (well 2 to 36 anyway) radices to enter a number

  #10r15 $\longrightarrow$ 15 #2r1111 $\longrightarrow$ 15 #3r120 $\longrightarrow$ 15

Ratios, Floats

- Exact ratios can be represented without roundoff error

Ratios, Floats

- Exact ratios can be represented without roundoff error

  `(expt (/ 2 3) 2)` $\rightarrow$

Ratios, Floats

- Exact ratios can be represented without roundoff error

  `(expt (/ 2 3) 2)` → `4/9`

Ratios, Floats

- Exact ratios can be represented without roundoff error

  `(expt (/ 2 3) 2)` → `4/9`

- As in other languages, floating point numbers are represented as follows

Ratios, Floats

- Exact ratios can be represented without roundoff error

  `(expt (/ 2 3) 2)` $\rightarrow$ `4/9`

- As in other languages, floating point numbers are represented as follows

```
5.2
6.02E+23
5E-22
```

Ratios, Floats

- Exact ratios can be represented without roundoff error

  `(expt (/ 2 3) 2)` → `4/9`

- As in other languages, floating point numbers are represented as follows

```
5.2
6.02E+23
5E-22
```

- Control over precision of floating point numbers is available

Complex Numbers

- Complex numbers have their own notation in Lisp

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

- Many Lisp functions will take complex arguments

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

- Many Lisp functions will take complex arguments

```
(* #c(0 -1) #c(0 -1)) →
```

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

- Many Lisp functions will take complex arguments

```
(* #c(0 -1) #c(0 -1)) → 1
```

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

- Many Lisp functions will take complex arguments

```
(* #c(0 -1) #c(0 -1)) → 1

pi →3.1415926535897932385L0
(exp (* #c(0 -1) pi))
→
```

Complex Numbers

- Complex numbers have their own notation in Lisp

```
#C( real imaginary )
1-2i = #C(1 -2)
```

- Many Lisp functions will take complex arguments

```
(* #c(0 -1) #c(0 -1)) → 1

pi →3.14159265358979323851L0
(exp (* #c(0 -1) pi))
→ #C(-1.0L0 5.0165576136843360246L-20) ≈ -1
```

(i.e., the Euler identity $e^{i\pi}$=-1)

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: `+ * - /`

  `(+ 1 2 3 4 5 6 7 8 9 10)` $\longrightarrow$

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: `+ * - /`

  `(+ 1 2 3 4 5 6 7 8 9 10)` $\longrightarrow$ `55`

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

  (+ 1 2 3 4 5 6 7 8 9 10) $\longrightarrow$ 55
  (* 2 2 2) $\longrightarrow$

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

(+ 1 2 3 4 5 6 7 8 9 10) $\longrightarrow$ 55
(* 2 2 2) $\longrightarrow$ 8

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) →
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) → 9
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) → 9
(- 10 1 3) →
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) → 9
(- 10 1 3) → 6
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) → 9
(- 10 1 3) → 6
(/ 12 3 4) →
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

```
(+ 1 2 3 4 5 6 7 8 9 10) → 55
(* 2 2 2) → 8
(- 10 1 ) → 9
(- 10 1 3) → 6
(/ 12 3 4) → 1
```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

  (+ 1 2 3 4 5 6 7 8 9 10) $\rightarrow$ 55
  (* 2 2 2) $\rightarrow$ 8
  (- 10 1) $\rightarrow$ 9
  (- 10 1 3) $\rightarrow$ 6
  (/ 12 3 4) $\rightarrow$ 1

- Binary Functions: `MOD`

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: + * - /

  ```
  (+ 1 2 3 4 5 6 7 8 9 10) → 55
  (* 2 2 2) → 8
  (- 10 1 ) → 9
  (- 10 1 3) → 6
  (/ 12 3 4) → 1
  ```

- Binary Functions: MOD

  ```
  (MOD 11 2) →
  ```

Numerical Operations

- Unlike most languages, basic arithmetic op's are n-ary: `+ * - /`

  (+ 1 2 3 4 5 6 7 8 9 10) → 55
  (* 2 2 2) → 8
  (- 10 1) → 9
  (- 10 1 3) → 6
  (/ 12 3 4) → 1

- Binary Functions: `MOD`

  (MOD 11 2) → 1

Numerical Operations

- Unary Functions:

Numerical Operations

- Unary Functions:

  `(1+ 3)` $\longrightarrow$

Numerical Operations

- Unary Functions:

  (1+ 3) $\longrightarrow$ 4

Numerical Operations

- Unary Functions:

```
(1+ 3) → 4
(1- 3) →
```

Numerical Operations

- Unary Functions:

  (1+ 3) $\longrightarrow$ 4

  (1- 3) $\longrightarrow$ 2

Numerical Operations

- Unary Functions:

```
(1+ 3) → 4
(1- 3) → 2
(ABS -2) →
```

Numerical Operations

- Unary Functions:

  (1+ 3) $\longrightarrow$ 4
  (1- 3) $\longrightarrow$ 2
  (ABS -2) $\longrightarrow$ 2

Numerical Operations

- Unary Functions:

```
(1+ 3) → 4
(1- 3) → 2
(ABS -2) → 2
(SIN (/ pi 2)) →
```

Numerical Operations

- Unary Functions:

```
(1+ 3) → 4
(1- 3) → 2
(ABS -2) → 2
(SIN (/ pi 2)) → 1.0L0 ;; returned  a float
```

Numerical Operations

- Unary Functions:

  (1+ 3) → 4
  (1- 3) → 2
  (ABS -2) → 2
  (SIN (/ pi 2)) → 1.0L0 ;; returned a float
  (COS (/ pi 2)) →

Numerical Operations

- Unary Functions:

  (1+ 3) $\longrightarrow$ 4
  (1- 3) $\longrightarrow$ 2
  (ABS -2) $\longrightarrow$ 2
  (SIN (/ pi 2)) $\longrightarrow$ 1.0L0 ;; returned a float
  (COS (/ pi 2)) $\longrightarrow$ -2.5082788076048218878L-20 $\approx$0

Numerical Operations

- Unary Functions:

  ```
  (1+ 3) → 4
  (1- 3) → 2
  (ABS -2) → 2
  (SIN (/ pi 2)) → 1.0L0 ;; returned  a float
  (COS (/ pi 2)) → -2.5082788076048218878L-20 ≈0
  ```

- Binary Predicates: < > >= <=

Numerical Operations

- Unary Functions:

  ```
  (1+ 3) → 4
  (1- 3) → 2
  (ABS -2) → 2
  (SIN (/ pi 2)) → 1.0L0 ;; returned  a float
  (COS (/ pi 2)) → -2.5082788076048218878L-20 ≈0
  ```

- Binary Predicates: <   >   >=   <=
- Unary Predicates: ZEROP

Numbers Are Not Always EQ!

- Numbers are atoms:

Numbers Are Not Always EQ!

- Numbers are atoms:

`(atom 5)` →T `(atom 4.0)` →T `(atom #C(1 -1))` →T

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` →T `(atom 4.0)` →T `(atom #C(1 -1))` →T

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)

Numbers Are Not Always EQ!

- Numbers are atoms:

`(atom 5)` $\rightarrow$`T` `(atom 4.0)` $\rightarrow$`T` `(atom #C(1 -1))` $\rightarrow$`T`

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` →T `(atom 4.0)` →T `(atom #C(1 -1))` →T

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` $\rightarrow$`T` `(atom 4.0)` $\rightarrow$`T` `(atom #C(1 -1))` $\rightarrow$`T`

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

  `(= 4 4.0)` $\rightarrow$

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` →`T` `(atom 4.0)` →`T` `(atom #C(1 -1))` →`T`

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

  `(= 4 4.0)` → `T`

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` $\rightarrow$`T` `(atom 4.0)` $\rightarrow$`T` `(atom #C(1 -1))` $\rightarrow$`T`

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) →
```

Numbers Are Not Always EQ!

- Numbers are atoms:

```
(atom 5) →T   (atom 4.0) →T   (atom #C(1 -1)) →T
```

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) → nil   ;; mathematically equal
```

Numbers Are Not Always EQ!

- Numbers are atoms:

```
(atom 5) →T   (atom 4.0) →T   (atom #C(1 -1)) →T
```

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) → nil   ;; mathematically equal

(= 1234567890 1234567890 ) →
```

Numbers Are Not Always EQ!

- Numbers are atoms:

```
(atom 5) →T   (atom 4.0) →T   (atom #C(1 -1)) →T
```

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) → nil   ;; mathematically equal

(= 1234567890 1234567890 ) → T
```

Numbers Are Not Always EQ!

- Numbers are atoms:

```
(atom 5) →T   (atom 4.0) →T   (atom #C(1 -1)) →T
```

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) → nil   ;; mathematically equal

(= 1234567890 1234567890 ) → T
(eq 1234567890 1234567890 ) →
```

Numbers Are Not Always EQ!

- Numbers are atoms:

  `(atom 5)` →T `(atom 4.0)` →T `(atom #C(1 -1))` →T

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., equal)
- For efficiency use mathematical equality (e.g. =)
- Numbers are atoms, but are not always eq of each other

```
(= 4 4.0)  →  T
(eq 4 4.0) → nil  ;; mathematically equal

(= 1234567890 1234567890 ) → T
(eq 1234567890 1234567890 ) →
nil ;; distinct bignums
```

Association Lists

- An **Association List** is a list of DOTTED-pairs where:
  `CAR` of each DOTTED-pair is attribute
  `CDR` of each DOTTED-pair is value.

Association Lists

- An **Association List** is a list of DOTTED-pairs where:
  `CAR` of each DOTTED-pair is attribute
  `CDR` of each DOTTED-pair is value.
- Eg:

```
( (name       . (Bart Selman))
  (hair       . black)
  (children   . ( (Mary Louise)
                  (Jean Pierre) ) )
  (habits     . nil) )
```

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` ∈ s-expr

`(cons 'a 'b)` →

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\rightarrow$(a . b)

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\rightarrow$(a . b)

`(cons 4 '(a b c))` $\rightarrow$

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\rightarrow$(a . b)

`(cons 4 '(a b c))` $\rightarrow$`(4 . (a b c))`

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\rightarrow$(a . b)

`(cons 4 '(a b c))` $\rightarrow$`(4 . (a b c))`

`(cons '(t) '(a . b))` $\rightarrow$

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\rightarrow$(a . b)

`(cons 4 '(a b c))` $\rightarrow$`(4 . (a b c))`

`(cons '(t) '(a . b))` $\rightarrow$`((t) . (a . b))`

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)` for any `s1, s2` ∈ s-expr

`(cons 'a 'b)` →(a . b)

`(cons 4 '(a b c))` →`(4 . (a b c))`

`(cons '(t) '(a . b))` →`((t) . (a . b))`

`(cons 4 '(a . b))` →

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` $\in$ s-expr

`(cons 'a 'b)` $\longrightarrow$(a . b)

`(cons 4 '(a b c))` $\longrightarrow$`(4 . (a b c))`

`(cons '(t) '(a . b))` $\longrightarrow$`((t) . (a . b))`

`(cons 4 '(a . b))` $\longrightarrow$`(4 . (a . b))`

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` ∈ s-expr

```
(cons 'a 'b) →(a . b)
(cons 4 '(a b c)) →(4 . (a b c))
(cons '(t) '(a . b)) →((t) . (a . b))
(cons 4 '(a . b)) →(4 . (a . b))
```

- Retreiving components

Dotted-Pair

- CONS can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of (CONS s1 s2) is (s1 . s2)
  for any s1, s2 $\in$ s-expr

```
(cons 'a 'b) → (a . b)
(cons 4 '(a b c)) → (4 . (a b c))
(cons '(t) '(a . b)) → ((t) . (a . b))
(cons 4 '(a . b)) → (4 . (a . b))
```

- Retreiving components

```
(CAR '(a . b)) →
```

Dotted-Pair

- CONS can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of (CONS s1 s2) is (s1 . s2)
  for any s1, s2 $\in$ s-expr

(cons 'a 'b) →(a . b)

(cons 4 '(a b c)) →(4 . (a b c))

(cons '(t) '(a . b)) →((t) . (a . b))

(cons 4 '(a . b)) →(4 . (a . b))

- Retreiving components

(CAR '(a . b)) →a

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` ∈ s-expr

```
(cons 'a 'b) →(a . b)
(cons 4 '(a b c)) →(4 . (a b c))
(cons '(t) '(a . b)) →((t) . (a . b))
(cons 4 '(a . b)) →(4 . (a . b))
```

- Retreiving components

```
(CAR '(a . b)) →a
(CAR (CONS 'a 'b)) →
```

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` ∈ s-expr

`(cons 'a 'b)` →(a . b)

`(cons 4 '(a b c))` →`(4 . (a b c))`

`(cons '(t) '(a . b))` →`((t) . (a . b))`

`(cons 4 '(a . b))` →`(4 . (a . b))`

- Retreiving components

`(CAR '(a . b))` →a

`(CAR (CONS 'a 'b))` →a

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)`
  for any `s1, s2` ∈ s-expr

```
(cons 'a 'b) →(a . b)
(cons 4 '(a b c)) →(4 . (a b c))
(cons '(t) '(a . b)) →((t) . (a . b))
(cons 4 '(a . b)) →(4 . (a . b))
```

- Retreiving components

```
(CAR '(a . b)) →a
(CAR (CONS 'a 'b)) →a
(CDR '((t) . (a . b))) →
```

Dotted-Pair

- `CONS` can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of `(CONS s1 s2)` is `(s1 . s2)` for any `s1, s2` ∈ s-expr

`(cons 'a 'b)` →(a . b)

`(cons 4 '(a b c))` →`(4 . (a b c))`

`(cons '(t) '(a . b))` →`((t) . (a . b))`

`(cons 4 '(a . b))` →`(4 . (a . b))`

- Retreiving components

`(CAR '(a . b))` →a

`(CAR (CONS 'a 'b))` →a

`(CDR '((t) . (a . b)))` →(a . b)

Notation

- Can write `(`s_1 `. (`s_2 `...))` as `(`s_1 s_2 `...)`
  Hence `(a . (b . c))` $\mapsto$ `(a . (b . c))`

Notation

- Can write `(s`$_1$ `. (s`$_2$ `...))` as `(s`$_1$ `s`$_2$ `...)`
  Hence `(a . (b . c))` $\mapsto$ `(a . (b . c))`
- Can write `(s`$_1$ `s`$_2$ `...s`$_n$ `. nil)` as `(s`$_1$ `s`$_2$ `...s`$_n$ `)`
  Hence `(cons 'a nil)` $\mapsto$ `(a . nil)` $\mapsto$ `(a)`

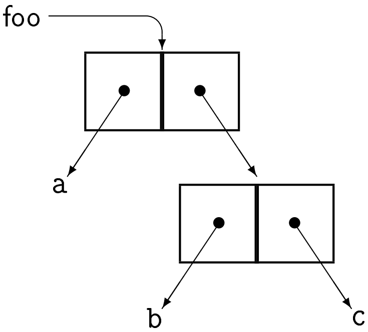
Notation

- Can write `(`s_1 `. (`s_2 `...))` as `(`s_1 s_2 `...)`
  Hence `(a . (b . c))` ↦ `(a . (b . c))`
- Can write `(`s_1 s_2 `...`s_n `. nil)` as `(`s_1 s_2 `...`s_n `)`
  Hence `(cons 'a nil)` ↦ `(a . nil)` ↦ `(a)`
- Notice:
  When `CONS`'s 2nd arg is list, just as before!

Dotted Pair – Internals

```
(SETQ foo (CONS 'a (CONS 'b 'c)))
```

(a . (b . c))



Association Lists

- Can be assigned:

```
(setq bart '( (name Bart Selman)  (hair . black)
              (children (Mary Louise) (Jean Pierre) )
              (habits) )  )
```

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  `(assoc 'name bart)` $\longrightarrow$

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  `(assoc 'name bart)` $\longrightarrow$ `(name Bart Selman)`

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  `(assoc 'name bart)` $\longrightarrow$ `(name Bart Selman)`

  `(assoc 'children bart)`

  $\longrightarrow$

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  ```
  (assoc 'name bart) →(name Bart Selman)
  (assoc 'children bart)
   →(children (Mary Louise) (Jean Pierre)
  ```

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  ```
  (assoc 'name bart) →(name Bart Selman)
  (assoc 'children bart)
   →(children (Mary Louise) (Jean Pierre)
  (assoc 'habit bart) →
  ```

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  ```
  (assoc 'name bart) →(name Bart Selman)
  (assoc 'children bart)
   →(children (Mary Louise) (Jean Pierre)
  (assoc 'habit bart) →(habit)
  ```

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

```
(assoc 'name bart) →(name Bart Selman)
(assoc 'children bart)
 →(children (Mary Louise) (Jean Pierre)
(assoc 'habit bart) →(habit)
(assoc 'mother bart) →
```

The ASSOC Function

- ASSOC takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

```
(assoc 'name bart) →(name Bart Selman)
(assoc 'children bart)
 →(children (Mary Louise) (Jean Pierre)
(assoc 'habit bart) →(habit)
(assoc 'mother bart) →nil
```

The ASSOC Function

- ASSOC takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

```
(assoc 'name bart) →(name Bart Selman)
(assoc 'children bart)
 →(children (Mary Louise) (Jean Pierre)
(assoc 'habit bart) →(habit)
(assoc 'mother bart) →nil
(CDR (assoc 'name bart)) →
```

The ASSOC Function

- `ASSOC` takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

```
(assoc 'name bart) →(name Bart Selman)
(assoc 'children bart)
 →(children (Mary Louise) (Jean Pierre)
(assoc 'habit bart) →(habit)
(assoc 'mother bart) →nil
(CDR (assoc 'name bart)) → (Bart Selman)
```

The ASSOC Function

- ASSOC takes two arguments
  Attribute (an atom)
  Alist (an association list)
  returns *entire* Dotted-Pair if match is found.

  ```
  (assoc 'name bart) →(name Bart Selman)
  (assoc 'children bart)
   →(children (Mary Louise) (Jean Pierre)
  (assoc 'habit bart) →(habit)
  (assoc 'mother bart) →nil
  (CDR (assoc 'name bart)) → (Bart Selman)
  ```

- Requires $2n$ CONS-cells overhead

The ASSOC Function

- The equality test in assoc can be changed with the `:test` parameter

The ASSOC Function

- The equality test in assoc can be changed with the `:test` parameter

  `(assoc '(a) '( ( (a) . 1 ))) →`

The ASSOC Function

- The equality test in assoc can be changed with the `:test` parameter

  `(assoc '(a) '( ( (a) . 1 )))` → `NIL`

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

```
(assoc '(a) '( ( (a) . 1 ))) → NIL
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
→
```

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

```
(assoc '(a) '( ( (a) . 1 ))) → NIL
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
→((A) . 1)
```

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

  ```
  (assoc '(a) '( ( (a) . 1 ))) → NIL
  (assoc '(a) '( ( (a) . 1 )) :test 'equal)
  →((A) . 1)
  ```

- A key pure list data structure:

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

  ```
  (assoc '(a) '( ( (a) . 1 ))) → NIL
  (assoc '(a) '( ( (a) . 1 )) :test 'equal)
  →((A) . 1)
  ```

- A key pure list data structure:
  - New entries can "shadow" old entries (functional modification)

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

```
(assoc '(a) '( ( (a) . 1 ))) → NIL
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
→((A) . 1)
```

- A key pure list data structure:
  - New entries can "shadow" old entries (functional modification)
  - Tails of assoc lists can be shared

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

```
(assoc '(a) '( ( (a) . 1 ))) → NIL
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
→((A) . 1)
```

- A key pure list data structure:
  - New entries can "shadow" old entries (functional modification)
  - Tails of assoc lists can be shared
  - Allows access to values by named key like a structure

The ASSOC Function

- The equality test in assoc can be changed with the :test parameter

```
(assoc '(a) '( ( (a) . 1 ))) → NIL
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
→((A) . 1)
```

- A key pure list data structure:
  - New entries can "shadow" old entries (functional modification)
  - Tails of assoc lists can be shared
  - Allows access to values by named key like a structure
- Convenience functions make it easy to manage

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

  `(rassoc '(Bart Selman) bart)` $\longrightarrow$

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

```
(rassoc '(Bart Selman) bart) →
(name (Bart Selman))
```

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

  `(rassoc '(Bart Selman) bart)` →
  `(name (Bart Selman))`

- The acons functions creates a new entry

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

```
(rassoc '(Bart Selman) bart) →
(name (Bart Selman))
```

- The acons functions creates a new entry

```
(acons 'age 42 bart) ≡(cons (cons 'age 42) bart)
```

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

```
(rassoc '(Bart Selman) bart) →
(name (Bart Selman))
```

- The acons functions creates a new entry

```
(acons 'age 42 bart) ≡(cons (cons 'age 42) bart)
```

- The pairlis function zips two lists together into a association list

Other Association List Functions

- The `rassoc` function (reverse associate) returns an *entry* given a value

  `(rassoc '(Bart Selman) bart)` →
  `(name (Bart Selman))`

- The acons functions creates a new entry

  `(acons 'age 42 bart)` ≡`(cons (cons 'age 42) bart)`

- The pairlis function zips two lists together into a association list

  `(pairlis '(1 2 3) '(a b c))`
  → `(  (1 . a) (2 . b) (3 . c) )`

General List Functions

- The `some` and `every` function take *n* lists and pass the corresponding elements of each list to an *n*-ary predictate

```
(some predicate sequence1 sequence2 ...)
```

General List Functions

- The `some` and `every` function take *n* lists and pass the corresponding elements of each list to an *n*-ary predictate

```
(some predicate sequence1 sequence2 ...)
```

- The `some` function returns true if any element of a list(s) satisfies predicate

```
( (some #'(lambda (x y)   (not (equal x y)))
        '(1 2 3) '(1 2 4)) →
```

General List Functions

- The `some` and `every` function take *n* lists and pass the corresponding elements of each list to an *n*-ary predictate

```
(some predicate sequence1 sequence2 ...)
```

- The `some` function returns true if any element of a list(s) satisfies predicate

```
( (some #'(lambda (x y)   (not (equal x y)))
        '(1 2 3) '(1 2 4)) → T
```

General List Functions

- The `some` and `every` function take *n* lists and pass the corresponding elements of each list to an *n*-ary predictate

```
(some predicate sequence1 sequence2 ...)
```

- The `some` function returns true if any element of a list(s) satisfies predicate

```
( (some #'(lambda (x y)   (not (equal x y)))
       '(1 2 3) '(1 2 4)) → T
```

- The `every` function returns true if every element of a list(s) satisfies pred

```
( (every #'(lambda (x y)   (equal x y))
       '(1 2 3) '(1 2 4)) →
```

General List Functions

- The `some` and `every` function take *n* lists and pass the corresponding elements of each list to an *n*-ary predictate

```
(some predicate sequence1 sequence2 ...)
```

- The `some` function returns true if any element of a list(s) satisfies predicate

```
( (some #'(lambda (x y)   (not (equal x y)))
        '(1 2 3) '(1 2 4)) → T
```

- The `every` function returns true if every element of a list(s) satisfies pred

```
( (every #'(lambda (x y)   (equal x y))
        '(1 2 3) '(1 2 4)) → nil
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
(find 'a '(c b)) →
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
(find 'a '(c b)) →nil
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
(find 'a '(c b)) →nil
```

- The `find-if` function returns the first element that satisfies a predicate

```
(find-if #'oddp '(2 4 7 6 9)) →
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
(find 'a '(c b)) →nil
```

- The `find-if` function returns the first element that satisfies a predicate

```
(find-if #'oddp '(2 4 7 6 9)) → 7
```

General List Functions

- The `find` function returns the first element that matches item, or `nil`

```
(find item sequence)
;; recall any non-nil value is true
(find 'a '(c b a)) →A
(find 'a '(c b)) →nil
```

- The `find-if` function returns the first element that satisfies a predicate

```
(find-if #'oddp '(2 4 7 6 9)) → 7
```

- Many other functions

```
position, mismatch, substitute, remove, sort
```

General List Functions

- There is an overlap with assoc

```
(assoc item list :test fn)
   ≡ (find item list :test fn :key #'car)
```

General List Functions

- More list functions

General List Functions

- More list functions

  `(member 1 '(1 2 3))` $\longrightarrow$

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)   ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →(2 1)
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →(2 1)
(pop x) →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)   ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →(2 1)
(pop x) → 2
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)   ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →(2 1)
(pop x) → 2
x →
```

General List Functions

- More list functions

```
(member 1 '(1 2 3)) →
(1 2 3) ;; i.e., non-nil
(union '(1 2 3) '(2 3 4)) → (1 2 3 4)
(intersection '(1 2 3) '(2 3 4)) →(2 3)
(adjoin 2 '(1 2 3)) →(1 2 3)  ;; add if absent
(adjoin 4 '(1 2 3)) → (4 1 2 3)
```

- *Destructive* List Functions

```
(setf x '(1))
(push 2 x)
x →(2 1)
(pop x) → 2
x → (1)
```

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
```

-

Property Lists

- Like association lists, but
  - attached to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) →
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
```

-

Property Lists

- Like association lists, but
  - attached to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
(get 'clyde 'age) →
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
(get 'clyde 'age) → 42
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
(get 'clyde 'age) → 42
(remprop 'clyde 'age)
(get 'clyde 'age) →
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
(get 'clyde 'age) → 42
(remprop 'clyde 'age)
(get 'clyde 'age) → nil
```

-

Property Lists

- Like association lists, but
  - attracted to specific symbol and
  - operations are destructive

```
(setf (get 'clyde 'species) 'elephant)
(setf (get 'clyde 'age) 42)
(get 'clyde 'species) → elephant
(get 'clyde 'age) → 42
(remprop 'clyde 'age)
(get 'clyde 'age) → nil
```

- Do not delete plist as some implementations store important information about symbols in their plists
-

Hash Tables

- Hash tables allow efficient storage and retrieval by keys

Hash Tables

- Hash tables allow efficient storage and retrieval by keys
- Have a state and are subject to side effects

```
(setq pops (make-hash-table)) → #S(HASH-TABLE EQL)
(gethash 'calgary pops) → nil; nil
```

Hash Tables

- Hash tables allow efficient storage and retrieval by keys
- Have a state and are subject to side effects

```
(setq pops (make-hash-table)) → #S(HASH-TABLE EQL)
(gethash 'calgary pops) → nil; nil
```

- Note: gethash returns 2 values
  - second value is T or nil if key was found or not
  - allows one to distinguish between not found, and found value nil

Hash Tables

- Hash tables allow efficient storage and retrieval by keys
- Have a state and are subject to side effects

```
(setq pops (make-hash-table)) → #S(HASH-TABLE EQL)
(gethash 'calgary pops) → nil; nil
```

- Note: gethash returns 2 values
  - second value is T or nil if key was found or not
  - allows one to distinguish between not found, and found value nil
- Use multiple value bind to catch both values

```
(multiple-value-bind (value ok) (gethash 'calgary pops)
   (if ok <form> ))
```

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
```

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
(gethash 'calgary pops) →
```

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
(gethash 'calgary pops) → 876519
```

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
(gethash 'calgary pops) → 876519
```

- The test used to match keys can be set with :test

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
(gethash 'calgary pops) → 876519
```

- The test used to match keys can be set with :test
- Other useful hash table functions

```
(remhash 'calgary)  ;; removes entry
```

Hash Table Functions

- Setting entries in a table

```
(setf (gethash 'calgary pops) 876519) → 876519
(gethash 'calgary pops) → 876519
```

- The test used to match keys can be set with :test
- Other useful hash table functions

```
(remhash 'calgary)  ;; removes entry

;; applies fn to each key-value
(maphash fn hash-table)  pair
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v →
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9   ;; index from zero
v →
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9   ;; index from zero
v → #(9 2 3)
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9  ;; index from zero
v → #(9 2 3)
```

- Matricies

```
(setf m (make-array '(2 2)))
→
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9   ;; index from zero
v → #(9 2 3)
```

- Matricies

```
(setf m (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9   ;; index from zero
v → #(9 2 3)
```

- Matricies

```
(setf m (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))
(setf (aref m 0 0) 9) → 9
m →
```

Vectors and Arrays

- Construction and use of a vector

```
(setf u #(2 3 4))
u →
#(2 3 4)    ;; macro constructor is convenient
(setf v (vector 1 2 3))
v → #(1 2 3)
(setf (aref v 0) 9) →9   ;; index from zero
v → #(9 2 3)
```

- Matricies

```
(setf m (make-array '(2 2)))
→ #2A((NIL NIL) (NIL NIL))
(setf (aref m 0 0) 9) → 9
m → #2A((9 NIL) (NIL NIL))
```

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

```
(setq cmput325 (make-course))
```

$\longrightarrow$

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

```
(setq cmput325 (make-course))
→ #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)
```

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

```
(setq cmput325 (make-course))
→ #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)

(setq (course-name cmput325)
     "Non-procedural progamming")

(course-name comput325) →
```

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

```
(setq cmput325 (make-course))
→ #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)

(setq (course-name cmput325)
      "Non-procedural progamming")

(course-name comput325) →"Non-procedural progamming"
```

Structures

- DEFSTRUCT defines methods for creating and accessing elements of a new structure

```
(DEFSTRUCT course name room time)
```

- The methods `make-course`, `course-name`, `course-room` and `course-time` are now defined

```
(setq cmput325 (make-course))
→ #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)

(setq (course-name cmput325)
      "Non-procedural progamming")

(course-name comput325) →"Non-procedural progamming"
```

- Can be compiled to efficient memory accesses

Lisp Objects: CLOS

- CLOS = Common Lisp Object System

Lisp Objects: CLOS

- CLOS = Common Lisp Object System
- Provides functions for defining class data & methods
  - powerful shortcuts for defining initial values, accessors, etc.

Lisp Objects: CLOS

- CLOS = Common Lisp Object System
- Provides functions for defining class data & methods
  - powerful shortcuts for defining initial values, accessors, etc.
- Supports multiple inheritance

Lisp Objects: CLOS

- CLOS = Common Lisp Object System
- Provides functions for defining class data & methods
  - powerful shortcuts for defining initial values, accessors, etc.
- Supports multiple inheritance
- Can define your own method dispatching behaviours through the meta-object protocol

Lisp Objects: CLOS

- CLOS = Common Lisp Object System
- Provides functions for defining class data & methods
  - powerful shortcuts for defining initial values, accessors, etc.
- Supports multiple inheritance
- Can define your own method dispatching behaviours through the meta-object protocol
- Can flexibly call super-class code anywhere is a method

Lisp Objects: Classes

- Class definintion provides a rich language

Lisp Objects: Classes

- Class definintion provides a rich language
- A class with a slot named "color"

```
(defclass shape ()
   ((color :accessor color
           :initarg :color
           :initform 'clear)))
```

Lisp Objects: Classes

- Class definintion provides a rich language
- A class with a slot named "color"

```
(defclass shape ()
   ((color :accessor color
           :initarg :color
           :initform 'clear)))
(setf s1 (make-instance 'shape :color 'red))
```

Lisp Objects: Classes

- Class definintion provides a rich language
- A class with a slot named "color"

```
(defclass shape ()
   ((color :accessor color
           :initarg :color
           :initform 'clear)))
(setf s1 (make-instance 'shape :color 'red))
(color s1) →
```

Lisp Objects: Classes

- Class definintion provides a rich language
- A class with a slot named "color"

```
(defclass shape ()
   ((color :accessor color
           :initarg :color
           :initform 'clear)))
(setf s1 (make-instance 'shape :color 'red))
(color s1) → red
```

Lisp Objects: Inheritance

- Inheritance

```
(defclass circle (shape)
    ((center :accessor center
         :initarg :center
         :initform (list 0 0))
     (radius :accessor radius
             :initarg :r
             :initform 1)))
```

Lisp Objects: Methods

- Methods defined through generic functions with typed arguments

```
(defmethod draw ((c circle))
    (format t "Circle color:~s~% " (color c) ))
```

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
```

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
#\space #\newline #\linefeed #\page
#\return #\backspace #\rubout
```

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
#\space #\newline #\linefeed #\page
#\return #\backspace #\rubout
```

- Can be constructed as constants or dynamically

```
(setf name "bob")
```

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
#\space #\newline #\linefeed #\page
#\return #\backspace #\rubout
```

- Can be constructed as constants or dynamically

```
(setf name "bob")
(setf label (make-string 10 :initial-element #\B))
```

$\longrightarrow$

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
#\space #\newline #\linefeed #\page
#\return #\backspace #\rubout
```

- Can be constructed as constants or dynamically

```
(setf name "bob")
(setf label (make-string 10 :initial-element #\B))
→ "BBBBBBBBBB"
```

Lisp Strings

- Strings are built of characters which are introduced with #\

```
#\g #\G  ;; these are different!
#\space #\newline #\linefeed #\page
#\return #\backspace #\rubout
```

- Can be constructed as constants or dynamically

```
(setf name "bob")
(setf label (make-string 10 :initial-element #\B))
→ "BBBBBBBBBB"
```

- Can be compared by equal or for dictionary ordering with string= string> etc.

Basic IO

- `(read stream)` – reads an s-expr from stream

Basic IO

- `(read stream)` – reads an s-expr from stream
- `(write object stream)` – writes s-expr to stream

Basic IO

- `(read stream)` – reads an s-expr from stream
- `(write object stream)` – writes s-expr to stream
- `(terpri)` – flushes buffer, prints carriage return

Basic IO

- `(read stream)` – reads an s-expr from stream
- `(write object stream)` – writes s-expr to stream
- `(terpri)` – flushs buffer, prints carriage return
- `(load` ⟨`file`⟩`)` – loads file named ⟨`file`⟩.

I/O in Lisp – Input

- Use (read stream) to read from stream

I/O in Lisp – Input

- Use (read stream) to read from stream
- Read one complete s-expr at a time

I/O in Lisp – Input

- Use (read stream) to read from stream
- Read one complete s-expr at a time
- Use t for the console stream

```
(sqrt (read t))
49  ;; user typing
→
```

I/O in Lisp – Input

- Use (read stream) to read from stream
- Read one complete s-expr at a time
- Use t for the console stream

```
(sqrt (read t))
49  ;; user typing
→7
```

I/O in Lisp – Input

- Use (read stream) to read from stream
- Read one complete s-expr at a time
- Use t for the console stream

```
(sqrt (read t))
49  ;; user typing
→7
(car (setq x (read t)))
'(a b c) ;; user typing
→
```

I/O in Lisp – Input

- Use (read stream) to read from stream
- Read one complete s-expr at a time
- Use t for the console stream

```
(sqrt (read t))
49  ;; user typing
→7
(car (setq x (read t)))
'(a b c) ;; user typing
→ A
```

I/O in Lisp – Output

- Use (print object stream) to write object to stream

I/O in Lisp – Output

- Use (print object stream) to write object to stream
- Writes one complete s-expr at a time

I/O in Lisp – Output

- Use (print object stream) to write object to stream
- Writes one complete s-expr at a time
- Use t for the console stream or leave out stream

```
(print 44)
44 ;; output on console
```

I/O in Lisp – Formatted Output

- FORMAT is like printf in C or format in Fortran

I/O in Lisp – Formatted Output

- `FORMAT` is like printf in C or format in Fortran
- The basic form:
  `(FORMAT stream control-string arg1 arg2 ...  argn)`

I/O in Lisp – Formatted Output

- FORMAT is like printf in C or format in Fortran
- The basic form:
  `(FORMAT stream control-string arg1 arg2 ...  argn)`
- The control-string is a template into which arguments are substituted

I/O in Lisp – Formatted Output

- FORMAT is like printf in C or format in Fortran
- The basic form:
  `(FORMAT stream control-string arg1 arg2 ...  argn)`
- The control-string is a template into which arguments are substituted
- Use t to indicate the console stream

I/O in Lisp – Formatted Output

```
(setf name "Fred" age 24)
(setf hobbies '("lambda calculus" "meta-programming"))
(format t
   "Meet ~s, aged ~s who enjoys ~s ~%"
   name age hobbies)
```

I/O in Lisp – Formatted Output

```
(setf name "Fred" age 24)
(setf hobbies '("lambda calculus" "meta-programming"))
(format t
   "Meet ~s, aged ~s who enjoys ~s ~%"
   name age hobbies)
```

Meet "Fred", aged 24 who enjoys ("lambda calculus" "meta-programmi

I/O in Lisp – Control String

Control	Description
~s	print arbitrary s-expr in default form
~a	print s-expr in ASCII form
~%	insert carriage return
~nS	pad output of s-expr to make n-char field
~n,dF	fixed floating point with field width n and decimals d
~n,dE	exponential or scientifc notation
~n,dG	choose most appropriate of F or E

I/O in Lisp – Control Strings

```
(format t "~10s ~10s ~10s ~%" 'betty 'sal 'margaret)
(format t "~10s ~10s ~10s ~%" 'june 'sandy 'may)
```

I/O in Lisp – Control Strings

```
(format t "~10s ~10s ~10s ~%" 'betty 'sal 'margaret)
(format t "~10s ~10s ~10s ~%" 'june 'sandy 'may)
BETTY      SAL        MARGARET
JUNE       SANDY      MAY
```

I/O in Lisp – Output to strings

- Turning objects into strings

```
(setf result (make-string-output-stream))
```

I/O in Lisp – Output to strings

- Turning objects into strings

```
(setf result (make-string-output-stream))
(format result "~s calculated PI to 2 decimals: ~3,2F ~%
   'norman pi)
```

I/O in Lisp – Output to strings

- Turning objects into strings

```
(setf result (make-string-output-stream))
(format result "~s calculated PI to 2 decimals: ~3,2F ~%
   'norman pi)
(get-output-stream-string result)
→
```

I/O in Lisp – Output to strings

- Turning objects into strings

```
(setf result (make-string-output-stream))
(format result "~s calculated PI to 2 decimals: ~3,2F ~%
   'norman pi)
(get-output-stream-string result)
→ NORMAN calculated PI to 2 decimals: 3.14
```

- Turning objects into strings

```
(setf result (make-string-output-stream))
(format result "~s calculated PI to 2 decimals: ~3,2F ~%
   'norman pi)
(get-output-stream-string result)
→ NORMAN calculated PI to 2 decimals: 3.14
```

- Turning strings into objects

```
(read-from-string string)
```

Error Handling

- Common Lisp supports a complete error condition signalling system with catch and throw

Error Handling

- Common Lisp supports a complete error condition signalling system with catch and throw
- The simplest error handling is to call "error" which has the same syntax as "format"

```
(error "Object ~s is unknown." an-object)
```

Error Handling

- Common Lisp supports a complete error condition signalling system with catch and throw
- The simplest error handling is to call "error" which has the same syntax as "format"

```
(error "Object ~s is unknown." an-object)
```

- The error invokes the debugger whereopon the user can use :bt to examine the backtrace leading to the bug

Basic IO

- (terpri) – flushs buffer, prints carriage return

Basic IO

- `(terpri)` – flushs buffer, prints carriage return
- `(load` ⟨`file`⟩`)` – loads file named ⟨`file`⟩.

VT100 Console Tricks

```
(defun clear-screen ()
   (let ((string (make-string 7)))
    (setf (aref string 0) #\Escape)
    (setf (aref string 1) #\[)
    (setf (aref string 2) #\2)
    (setf (aref string 3) #\J)
    (setf (aref string 4) #\Escape)
    (setf (aref string 5) #\[)
    (setf (aref string 6) #\H)
    (princ string)))
```

VT100 Console Tricks

```
(defun home-screen ()
  (let ((string (make-string 3)))
   (setf (aref string 0) #\Escape)
   (setf (aref string 1) #\[)
   (setf (aref string 2) #\H)
   (princ string)))
```

PROGN Form

- (progn $\langle$form$_1\rangle$ $\langle$form$_2\rangle$ $\cdots$ $\langle$form$_m\rangle$)
  Evaluates all forms, $\langle \text{form} \rangle_i$ $(i = 1..m)$
  in order.

PROGN Form

- (progn $\langle \texttt{form}_1 \rangle$ $\langle \texttt{form}_2 \rangle$ $\cdots$ $\langle \texttt{form}_m \rangle$)
  Evaluates all forms, $\langle \texttt{form} \rangle_i$ $(i = 1..m)$
  in order.
- Returns value of final form, $\langle \texttt{form} \rangle_m$
  (Ignores other values)

PROGN Form

- `(progn` $\langle$`form`$_1\rangle$ $\langle$`form`$_2\rangle$ $\cdots$ $\langle$`form`$_m\rangle$`)`
  `Evaluates all forms,` $\langle$`form`$\rangle_i$ $(i = 1..m)$
  `in order.`
- Returns value of final form, $\langle$`form`$\rangle_m$
  (Ignores other values)
- Takes ANY number of forms

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

```
(LAMBDA (a1 ... an)  form1 ... form_m)
```

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

```
(LAMBDA (a1 ... an)  form1 ... form_m)
```

- returns value of final form, $\langle\texttt{form}\rangle_m$.

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

```
(LAMBDA (a1 ... an)  form1 ... form_m)
```

- returns value of final form, $\langle \texttt{form} \rangle_m$.
- Only makes sense if forms preceding last have meaningful side-effects

```
(LAMBDA (a) (print a) (setq x a) (+ a 3)) 19)
```

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

```
(LAMBDA (a1 ... an)  form1 ... form_m)
```

- returns value of final form, ⟨`form`⟩$_m$.
- Only makes sense if forms preceding last have meaningful side-effects

```
(LAMBDA (a) (print a) (setq x a) (+ a 3)) 19)
19        ; printed by (print a)
→ 22    ; value of this form
```

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

```
(LAMBDA (a1 ... an)  form1 ... form_m)
```

- returns value of final form, $\langle \texttt{form} \rangle_m$.
- Only makes sense if forms preceding last have meaningful side-effects

```
(LAMBDA (a) (print a) (setq x a) (+ a 3)) 19)
19        ; printed by (print a)
→ 22    ; value of this form
x →
```

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

  `(LAMBDA (a1 ... an)  form1 ... form_m)`

- returns value of final form, ⟨`form`⟩$_m$.
- Only makes sense if forms preceding last have meaningful side-effects

```
(LAMBDA (a) (print a) (setq x a) (+ a 3)) 19)
19        ; printed by (print a)
→ 22    ; value of this form
x →  19 ; side effect causes new x value
```

The Truth about COND

- Earlier, insisted that each COND "clause" take exactly 2 forms. but, ...can take any number, from 1 on.

$$\begin{array}{llllll} \texttt{(COND} & (\langle q^1_1 \rangle & \langle q^1_2 \rangle & \cdots & \langle q^1_{m1} \rangle) & \\ & (\langle q^2_1 \rangle & \langle q^2_2 \rangle & \cdots & \langle q^2_{m2} \rangle) & \\ & & \cdots & & & \\ & (\langle q^n_1 \rangle & \langle q^n_2 \rangle & \cdots & \langle q^n_{mn} \rangle) &) \end{array}$$

where each $\langle q \rangle^i_j$ is a form, $mi \in \mathcal{Z}^+$.

The Truth about COND

- Earlier, insisted that each COND "clause" take exactly 2 forms. but, ...can take any number, from 1 on.

$$\begin{array}{llllll} \texttt{(COND} & (\langle q_1^1\rangle & \langle q_2^1\rangle & \cdots & \langle q_{m1}^1\rangle) & \\ & (\langle q_1^2\rangle & \langle q_2^2\rangle & \cdots & \langle q_{m2}^2\rangle) & \\ & & \cdots & & & \\ & (\langle q_1^n\rangle & \langle q_2^n\rangle & \cdots & \langle q_{mn}^n\rangle) & \texttt{)} \end{array}$$

  where each $\langle q\rangle_j^i$ is a form, $mi \in \mathcal{Z}^+$.

- If $\langle q\rangle_1^i$ is nonNIL,
  Then evaluate $\langle q\rangle_j^i$ forms, for $j = 2..mi$.

The Truth about COND

- Earlier, insisted that each COND "clause" take exactly 2 forms.
  but, ...can take any number, from 1 on.

```
(COND (⟨q¹₁⟩ ⟨q¹₂⟩ ··· ⟨q¹ₘ₁⟩)
      (⟨q²₁⟩ ⟨q²₂⟩ ··· ⟨q²ₘ₂⟩)
           ···
      (⟨qⁿ₁⟩ ⟨qⁿ₂⟩ ··· ⟨qⁿₘₙ⟩) )
```

$$\texttt{(COND}\ (\langle q_1^1\rangle\ \langle q_2^1\rangle\ \cdots\ \langle q_{m1}^1\rangle)\ (\langle q_1^2\rangle\ \langle q_2^2\rangle\ \cdots\ \langle q_{m2}^2\rangle)\ \ldots\ (\langle q_1^n\rangle\ \langle q_2^n\rangle\ \cdots\ \langle q_{mn}^n\rangle)\ \texttt{)}$$

  where each $\langle q\rangle_j^i$ is a form, $mi \in \mathcal{Z}^+$.

- If $\langle q\rangle_1^i$ is nonNIL,
  Then evaluate $\langle q\rangle_j^i$ forms, for $j = 2..mi$.

- Return value for final form $\langle q\rangle_{mi}^i$.

The Truth about COND

- Earlier, insisted that each COND "clause" take exactly 2 forms. but, ... can take any number, from 1 on.

$$
\begin{array}{llll}
\texttt{(COND} & (\langle q_1^1\rangle & \langle q_2^1\rangle & \cdots & \langle q_{m1}^1\rangle) \\
 & (\langle q_1^2\rangle & \langle q_2^2\rangle & \cdots & \langle q_{m2}^2\rangle) \\
 & & \cdots & & \\
 & (\langle q_1^n\rangle & \langle q_2^n\rangle & \cdots & \langle q_{mn}^n\rangle) \ \texttt{)}
\end{array}
$$

  where each $\langle q\rangle_j^i$ is a form, $mi \in \mathcal{Z}^+$.

- If $\langle q\rangle_1^i$ is nonNIL,
  Then evaluate $\langle q\rangle_j^i$ forms, for $j = 2..mi$.

- Return value for final form $\langle q\rangle_{mi}^i$.

- If $mi = 1$, and if $\langle q\rangle_1^i$ is nonNIL,
  then return $\langle q\rangle_1^i$'s value.

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) →
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
(swp 18)
x was nil, is now 18  ; prints msg, and resets x
x →18                 ; value that form returns.
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
(swp 18)
x was nil, is now 18  ; prints msg, and resets x
x →18                 ; value that form returns.
(setq x nil)→
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
(swp 18)
x was nil, is now 18  ; prints msg, and resets x
x →18                 ; value that form returns.
(setq x nil)→ nil
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
(swp 18)
x was nil, is now 18   ; prints msg, and resets x
x →18                  ; value that form returns.
(setq x nil)→ nil
(swp nil)→
```

Example of Real COND

```
(defun swp (y)
   (COND ( x )
      ( y (print "x was nil, is now")
          (print (setq x y))
          (terpri) 7) ) )
(setq x 'fred)→fred
(swp 18)
fred ; just prints out value of x.
(setq x nil) → nil
(swp 18)
x was nil, is now 18   ; prints msg, and resets x
x →18                  ; value that form returns.
(setq x nil)→ nil
(swp nil)→nil ; COND fails.
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
```

⇝ *1 2 3 4 5 6 7 8 9 10* ; on console

→ nil ; returned as value

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
⇝ 1 2 3 4 5 6 7 8 9 10 ; on console
→ nil  ; returned as value

(LOOP FOR i FROM 1 TO 10
   COLLECT i)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
⇝ 1 2 3 4 5 6 7 8 9 10 ; on console
→ nil  ; returned as value

(LOOP FOR i FROM 1 TO 10
   COLLECT i)
→(1 2 3 4 5 6 7 8 9 10)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
⇝ 1 2 3 4 5 6 7 8 9 10 ; on console
→ nil  ; returned as value

(LOOP FOR i FROM 1 TO 10
   COLLECT i)
→(1 2 3 4 5 6 7 8 9 10)

(LOOP REPEAT 10     ;; constrains max iterations
   DO (format t "*"))
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
⤳ 1 2 3 4 5 6 7 8 9 10 ; on console
→ nil  ; returned as value

(LOOP FOR i FROM 1 TO 10
   COLLECT i)
→(1 2 3 4 5 6 7 8 9 10)

(LOOP REPEAT 10     ;; constrains max iterations
   DO (format t "*"))
⤳ **********
→nil
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10 REPEAT 5
   COLLECT i)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10 REPEAT 5
   COLLECT i)
→(1 2 3 4 5)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10 REPEAT 5
   COLLECT i)
→(1 2 3 4 5)
(LOOP FOR i FROM 10 DOWNTO 1 COLLECT i)
→
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 1 TO 10 REPEAT 5
   COLLECT i)
→(1 2 3 4 5)
(LOOP FOR i FROM 10 DOWNTO 1 COLLECT i)
→(10 9 8 7 6 5 4 3 2 1)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
```

$\longrightarrow$

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
→(10 8 6 4 2)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
→(10 8 6 4 2)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)  collect i)
→
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
→(10 8 6 4 2)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)  collect i)
→(10 9 8 7 6 5 4 3 2 1)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
→(10 8 6 4 2)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)  collect i)
→(10 9 8 7 6 5 4 3 2 1)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)
   BY 'cddr collect i)
→
```

LOOP Construct by Simple Examples

```
(LOOP FOR i FROM 10 DOWNTO 1 BY 2 COLLECT i)
→(10 8 6 4 2)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)  collect i)
→(10 9 8 7 6 5 4 3 2 1)

(LOOP FOR i IN '(10 9 8 7 6 5 4 3 2 1)
   BY 'cddr collect i)
→(10 8 6 4 2)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
```

⟶

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →(1 11 21 31 41)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →(1 11 21 31 41)

(loop for ch across #( 4 3 2) collect ch)
→
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →(1 11 21 31 41)

(loop for ch across #( 4 3 2) collect ch)
→ (4 3 2)
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →(1 11 21 31 41)

(loop for ch across #( 4 3 2) collect ch)
→ (4 3 2)
(loop for ch across "able" collect ch)
→
```

LOOP Construct by Simple Examples

```
(LOOP FOR i from 10 downto 1
      FOR j from 1  to 10
      WHILE (> i j) collect (CONS i j) )
→ ((10 . 1) (9 . 2) (8 . 3) (7 . 4) (6 . 5))

(LOOP FOR item = 1 THEN (+ item 10)
   REPEAT 5 COLLECT ITEM) →(1 11 21 31 41)

(loop for ch across #( 4 3 2) collect ch)
→ (4 3 2)
(loop for ch across "able" collect ch)
→ (#\a #\b #\l #\e)
```

Examples of LOOP

```
(LOOP FOR i from 1 to 10
   when (evenp i) collect i)
```

$\longrightarrow$

Examples of LOOP

```
(LOOP FOR i from 1 to 10
   when (evenp i) collect i)
→ (2 4 6 8 10)
```

Examples of LOOP

```
(LOOP FOR i from 1 to 10
   when (evenp i) collect i)
→ (2 4 6 8 10)

(LOOP FOR i from 1 to 10
   when (evenp i) collect (cons 'even i)
   when (oddp i)  collect (cons 'odd i))
→
```

Examples of LOOP

```
(LOOP FOR i from 1 to 10
   when (evenp i) collect i)
→ (2 4 6 8 10)

(LOOP FOR i from 1 to 10
   when (evenp i) collect (cons 'even i)
   when (oddp i)  collect (cons 'odd i))
→ ((ODD . 1) (EVEN . 2) (ODD . 3) (EVEN . 4)
     (ODD . 5) (EVEN . 6) (ODD . 7) (EVEN . 8)
      (ODD . 9) (EVEN . 10))
```

Comments on LOOP

- LOOP can be used functionally to compute one value from another

```
(LAMBDA (x)
    (LOOP FOR i IN x COLLECT (cons i nil)))
```

Comments on LOOP

- LOOP can be used functionally to compute one value from another

```
(LAMBDA (x)
    (LOOP FOR i IN x COLLECT (cons i nil)))
```

- Many uses of LOOP can be replaced by sequence functions such as FIND or the SERIES package

Documentation

```
(DEFUN add (x y)
   "adds 2 numbers"
   (+ x y))

(documentation 'add 'function)
→"adds 2 numbers"
```

Apropos

- returns all function names containing the given substring

```
(apropos 'add) →
SLOOP::*ADDITIONAL-COLLECTIONS*
⋮
Function COMPILER::ADD-FUNCTION-DECLARATION
⋮
Function CADDDR
⋮
Function ADD
```

Compilation GCL

```
(DEFUN add (x y) (+ x y))
(disassemble 'add)
→
static L1(){
  register object *base=vs_base;
  register object *sup=base+VM1; ...
  {object V1;  object V2;
   V1=(base[0]);
   V2=(base[1]);
   vs_top=sup; ...
   base[2]= number_plus((V1),(V2));
   vs_top=(vs_base=base+2)+1;
 return; }
}
```

Declarations GCL

- Types of arguments and return values can be declared to optimize compilation

```
(defun add (x y)
  (declare
    (fixnum x y)
    (optimize (speed 3) (safety 0) (debug 0)))
   (the fixnum (+ x y)))
```

Compilation with Declarations

```
(disassemble 'add) →
static L1(){ ...

   {int V1; int V2;
    V1=fix(base[0]);
    V2=fix(base[1]);
    ...
    base[2]= CMPmake_fixnum((V1)+(V2));
    ...
}}
```

Compilation: CLISP PPC

```
(DEFUN add (x y) (+ x y))
(DISASSEMBLE 'add)→

Disassembly of function ADD2
required arguments 0
optional arguments No
rest parameter No
keyword parameters
0      (LOAD&PUSH 2)
1      (LOAD&PUSH 2)
2      (CALLSR 2 54)    ; +
5      (SKIP&RET 3)
#<COMPILED-CLOSURE ADD>
```

GUI Development

- Many Lisps provide rich graphical interface libraries

```
(in-package "TK")
(tkconnect)
(button '.hello :text "Hello World"
                :command '(print "hi"))
==>.HELLO
(pack '.hello)
```