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, ...)

Bob Price and Russ Greiner
No Side Effects

(+ 11 23) →
No Side Effects

\[(+ 11 23) \rightarrow 34\]
No Side Effects

\[(+ 11 23) \rightarrow 34\]
\[(\text{CAR } '(A B C)) \rightarrow\]
No Side Effects

\[
(+ 11 23) \rightarrow 34 \\
(CAR '(A B C)) \rightarrow A
\]
No Side Effects

\[(+ 11 23) \rightarrow 34\]
\[(\text{CAR } '(A B C)) \rightarrow A\]

\[(+ 11 23) \rightarrow \]
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) \rightarrow 34
(CAR '(A B C)) \rightarrow A

(+ 11 23) \rightarrow 34

(+ (* X 2) 5) \rightarrow X \text{ undefined}
(+ 11 23) \rightarrow
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

\[ (+ 11 23) \rightarrow 34 \]
\[ (\text{CAR } '(A \ B \ C)) \rightarrow A \]

\[ (+ 11 23) \rightarrow 34 \]

\[ (+ (* X 2) 5) \rightarrow X \text{ undefined} \]
\[ (+ 11 23) \rightarrow 34 \]

\[ (\text{CAR } '(A \ B \ C)) \rightarrow \]
No Side Effects

\[(+ \ 11 \ 23) \rightarrow 34\]
\[(\text{CAR } '(A \ B \ C)) \rightarrow A\]

\[(+ \ 11 \ 23) \rightarrow 34\]

\[(+ (* \ X \ 2) \ 5) \rightarrow X \text{ undefined}\]
\[(+ \ 11 \ 23) \rightarrow 34\]

\[(\text{CAR } '(A \ B \ C)) \rightarrow A\]
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 $\sigma$ has NO side effects if Evaluating $\sigma$ does not affect the value of any other expression $\tau$. 

Examples: Any form using only $+$, CAR, CONS, …
Side-Effect Free — Def’n

- Form $\sigma$ has NO side effects if Evaluating $\sigma$ does not affect the value of any other expression $\tau$.

- Hence: Value of form $\tau$ is the same whether or not $\sigma$ was evaluated.

\[
\begin{align*}
\sigma & \rightarrow \tau \\
\langle v_1 \rangle & \rightarrow \sigma \\
\langle v_2 \rangle & \rightarrow \sigma \\
\langle v_1 \rangle & \rightarrow \tau \\
\langle v_1 \rangle & \rightarrow \tau
\end{align*}
\]
Side-Effect Free — Def’n

- Form $\sigma$ has NO side effects if Evaluating $\sigma$ does not affect the value of any other expression $\tau$.

- Hence: Value of form $\tau$ is the same whether or not $\sigma$ was evaluated.

  $\longrightarrow \tau$
  $\langle v1 \rangle$
  $\longrightarrow \sigma$
  $\langle v2 \rangle$
  $\longrightarrow \tau$
  $\langle v1 \rangle$

- Examples: Any form using only $+$, CAR, CONS, ...
Functions with Side Effects – SETQ

my-const →
Functions with Side Effects – SETQ

my-const → undefined variable
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

\[(\text{SETF } x\ ' (1\ 2\ 3)) \rightarrow\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x \ ' (1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \quad \equiv (\text{SETQ } x \ ' (1 \ 2 \ 3))\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[
\begin{align*}
\text{(SETF } x \ ' (1 \ 2 \ 3) \text{)} & \rightarrow (1 \ 2 \ 3) ; ; \equiv \text{(SETQ } x \ ' (1 \ 2 \ 3)) \\
X & \rightarrow
\end{align*}
\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x \ ' (1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) ; ; \equiv (\text{SETQ } x \ ' (1 \ 2 \ 3)) \]
\[X \rightarrow (1 \ 2 \ 3)\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x \ ' (1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) ;; \equiv (\text{SETQ } x \ ' (1 \ 2 \ 3))\]
\[X \rightarrow (1 \ 2 \ 3)\]

\[(\text{SETF} \ (\text{car } x) \ ' a) \rightarrow \]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[
\text{SETF } x \to (1 \ 2 \ 3) \rightarrow (1 \ 2 \ 3) \equiv \text{SETQ } x \to (1 \ 2 \ 3)
\]

\[
X \rightarrow (1 \ 2 \ 3)
\]

\[
\text{SETF (car } x \to 'a) \rightarrow (a \ 2 \ 3) \equiv (RPLACA x \to 'a)
\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[
\text{(SETF } \text{x } '(1\ 2\ 3)) \rightarrow (1\ 2\ 3) ; \equiv (\text{SETQ } \text{x } '(1\ 2\ 3))
\]
\[
\text{X } \rightarrow (1\ 2\ 3)
\]

\[
\text{(SETF } (\text{car } \text{x})\ 'a) \rightarrow (a\ 2\ 3) ; \equiv (\text{RPLACA } \text{x } 'a)
\]
\[
\text{X } \rightarrow
\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x \ '(1 2 3)) \rightarrow (1 2 3) ;; \equiv (\text{SETQ } x \ '(1 2 3))\]

\[X \rightarrow (1 2 3)\]

\[(\text{SETF} \ (\text{car } x) \ 'a) \rightarrow (a 2 3) ;; \equiv (\text{RPLACA} \ x \ 'a)\]

\[X \rightarrow (a 2 3)\]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[
\text{(SETF x '(1 2 3)) → (1 2 3)} \; ; \equiv (\text{SETQ x '(1 2 3)}) \\
X → (1 2 3)
\]

\[
\text{(SETF (car x) 'a) → (a 2 3)} \; ; \equiv (\text{RPLACA x 'a}) \\
X → (a 2 3)
\]

\[
\text{(SETF A (make-array '(2 2))) → }
\]

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[ (\text{SETF } x \ (1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \equiv (\text{SETQ } x \ (1 \ 2 \ 3)) \]
\[ X \rightarrow (1 \ 2 \ 3) \]

\[ (\text{SETF} \ (\text{car} \ x) \ 'a) \rightarrow (a \ 2 \ 3) \equiv (\text{RPLACA} \ x \ 'a) \]
\[ X \rightarrow (a \ 2 \ 3) \]

\[ (\text{SETF} \ A \ (\text{make-array} \ '(2 \ 2))) \rightarrow \#2A((\text{NIL} \ \text{NIL}) \ (\text{NIL} \ \text{NIL})) \]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x \ (1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) ;\equiv (\text{SETQ } x \ (1 \ 2 \ 3))\]
\[X \rightarrow (1 \ 2 \ 3)\]

\[(\text{SETF } (\text{car } x) \ 'a) \rightarrow (a \ 2 \ 3) ;\equiv (\text{RPLACA } x \ 'a)\]
\[X \rightarrow (a \ 2 \ 3)\]

\[(\text{SETF } A \ (\text{make-array } '(2 \ 2)))
\rightarrow \#\text{2A}((\text{NIL NIL}) \ (\text{NIL NIL}))\]

\[(\text{SETF } (\text{aref } A \ 0 \ 0) \ 'q)\]
\rightarrow \]
The mysterious SETF

- SETF chooses a modifier function according to its first argument

\[(\text{SETF } x ~(1~ 2~ 3)) \rightarrow (1~ 2~ 3)\]
\(\equiv (\text{SETQ } x ~(1~ 2~ 3))\)
\(X \rightarrow (1~ 2~ 3)\)

\[(\text{SETF (car } x~)~'a) \rightarrow (a~ 2~ 3)\]
\(\equiv (\text{RPLACA } x ~'a)\)
\(X \rightarrow (a~ 2~ 3)\)

\[(\text{SETF } A ~(\text{make-array } ~(2~ 2)))\]
\(\rightarrow \#2A((\text{NIL ~NIL}) (\text{NIL ~NIL}))\)

\[(\text{SETF (aref } A~ 0~ 0)~'q)\]
\(\rightarrow \#2A((Q ~\text{NIL}) (\text{NIL ~NIL}))\)
Functions with Side Effects – SETF

\[
\text{( (LAMBDA (X) (CDR X)) \ '(A B C) ) → (B C)}
\]

\[
\text{( (LAMBDA (X) (CDR X)) \ '(Q t) ) → (t)}
\]

(setf (symbol-function \ 'my-fn) (FUNCTION (LAMBDA (X) (CDR X))) ) → (LAMBDA-CLOSURE ...(X) (CDR X)))

\[
\text{(my-fn \ '(A B C) ) → (B C)}
\]

\[
\text{(my-fn \ '(Q t) ) → (t)}
\]

\[
\text{(my-fn (my-fn my-const)) → (C)}
\]

(setf (symbol-function \ 'my-fn) (FUNCTION CAR) ) → #<complied-function car>

\[
\text{(my-fn \ '(A B C) ) → A}
\]

\[
\text{(setf (symbol-function \ 'my-fn) (FUNCTION CAR) ) ) → #<complied-function car>}
\]

\[
\text{(my-fn \ '(A B C) ) → A}
\]
Functions with Side Effects – SETF

\[(\text{LAMBDA} (X) (\text{CDR} X)) \ '(A B C) \rightarrow (B C)\]
Functions with Side Effects – SETF

( (LAMBDA (X) (CDR X)) '(A B C) ) \rightarrow (B C)
( (LAMBDA (X) (CDR X)) '(Q t) ) \rightarrow
Functions with Side Effects – SETF

\[
\begin{align*}
    \mathbf{( \text{LAMBDA} \ (X) \ (\text{CDR} \ X)) \ '(A \ B \ C)} \ &\rightarrow \ (B \ C) \\
    \mathbf{( \text{LAMBDA} \ (X) \ (\text{CDR} \ X)) \ '(Q \ t)} \ &\rightarrow \ (t)
\end{align*}
\]
Functions with Side Effects – SETF

\[
\begin{align*}
(\text{LAMBDA } (X) (\text{CDR } X)) \ ' (A \ B \ C) & \rightarrow (B \ C) \\
(\text{LAMBDA } (X) (\text{CDR } X)) \ ' (Q \ t) & \rightarrow (t) \\
(\text{setf} \ \text{(symbol-function} \ ' \text{my-fn}) \\
\quad (\text{FUNCTION} \ (\text{LAMBDA} \ (X) (\text{CDR} \ X)))) & \rightarrow
\end{align*}
\]
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

\[
\begin{align*}
(\text{LAMBDA}(X)(\text{CDR}~X))\ '&(A~B~C) &\rightarrow (B~C) \\
(\text{LAMBDA}(X)(\text{CDR}~X))\ '&(Q~t) &\rightarrow (t) \\
\text{setf} \ (\text{symbol-function} \ '\text{my-fn}) &\rightarrow \text{(FUNCTION (LAMBDA} (X) (\text{CDR}~X)))) \\
&\rightarrow \text{(LAMBDA-CLOSURE ...(X) (CDR} X)))) \\
(\text{my-fn} \ '(A~B~C)) &\rightarrow \\
\end{align*}
\]
Functions with Side Effects – SETF

\[(\text{LAMBDA} \ (X) \ (\text{CDR} \ X)) \ ' \ (A \ B \ C) \] \[\rightarrow \ (B \ C)\]
\[(\text{LAMBDA} \ (X) \ (\text{CDR} \ X)) \ ' \ (Q \ t) \] \[\rightarrow \ (t)\]
\[(\text{setf} \ (\text{symbol-function} \ ' \ ' \text{my-fn})\]
\[\quad \text{(FUNCTION} \ (\text{LAMBDA} \ (X) \ (\text{CDR} \ X)))\]
\[\rightarrow \ (\text{LAMBDA-CLOSURE} \ ...(X) \ (\text{CDR} \ X)))\]
\[(\text{my-fn} \ ' \ (A \ B \ C)) \rightarrow \ (B \ C)\]
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) →
( (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

\[
\begin{align*}
(\text{LAMBDA} (X) (\text{CDR} X))'\text{(A B C)} & \rightarrow (\text{B C}) \\
(\text{LAMBDA} (X) (\text{CDR} X))'\text{(Q t)} & \rightarrow (\text{t}) \\
\text{setf (symbol-function 'my-fn)} \\
\quad \text{(FUNCTION (LAMBDA} (X) (\text{CDR} X))) & \rightarrow \text{(LAMBDA-CLOSURE ...(X) (CDR} X))) \\
\text{(my-fn ')(A B C)} & \rightarrow (\text{B C}) \\
\text{(my-fn ')(Q t)} & \rightarrow (\text{t}) \\
\text{(my-fn (my-fn my-const))} & \rightarrow \\
\end{align*}
\]
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 – \texttt{SETF}

\[
\begin{align*}
&\text{\((\text{LAMBDA } (X) (\text{CDR } X)) \ '(A \ B \ C)\) } \rightarrow \ (B \ C) \\
&(\text{LAMBDA } (X) (\text{CDR } X)) \ '(Q \ t)\) } \rightarrow \ (t) \\
&\text{(setf (symbol-function } '\text{my-fn}) \\
&\phantom{=} \text{ FUNCTION (LAMBDA } (X) (\text{CDR } X)))) \\
&\rightarrow \ (\text{LAMBDA-CLOSURE ...(X) (CDR } X)))) \\
&(\text{my-fn } '(A \ B \ C)) \rightarrow \ (B \ C) \\
&(\text{my-fn } '(Q \ t)\rightarrow \ (t) \\
&(\text{my-fn (my-fn my-const))}\rightarrow \ (C) \\
&(\text{setf (symbol-function } '\text{my-fn}) \\
&\phantom{=} \text{ FUNCTION CAR) } ) \\
&\rightarrow
\end{align*}
\]
Functions with Side Effects – SETF

\[
\begin{align*}
(\text{(LAMBDA (X) (CDR X))} \ '(A\ B\ C)) & \rightarrow (B\ C) \\
(\text{(LAMBDA (X) (CDR X))} \ '(Q\ t)) & \rightarrow (t) \\
\text{(setf (symbol-function 'my-fn)} & \\
\quad \text{ (FUNCTION (LAMBDA (X) (CDR X)))} \big) & \rightarrow \text{(LAMBDA-CLOSURE ...)(X) (CDR X)))} \\
(\text{my-fn} \ '(A\ B\ C)) & \rightarrow (B\ C) \\
(\text{my-fn} \ '(Q\ t)) & \rightarrow (t) \\
(\text{my-fn (my-fn my-const)}) & \rightarrow (C) \\
\text{(setf (symbol-function 'my-fn)} & \\
\quad \text{ (FUNCTION CAR) } \big) & \rightarrow \text{#<complied-function car>}
\end{align*}
\]
Functions with Side Effects – SETF

\[
\begin{align*}
\text{( (LAMBDA (X) (CDR X)) '(A B C) ) } & \rightarrow (B C) \\
\text{( (LAMBDA (X) (CDR X)) '(Q t) ) } & \rightarrow (t) \\
\text{(setf (symbol-function 'my-fn)} & \\
\text{ (FUNCTION (LAMBDA (X) (CDR X))) ) } & \rightarrow (LAMBDA-CLOSURE ...(X) (CDR X))) \\
\text{(my-fn '(A B C)) } & \rightarrow (B C) \\
\text{(my-fn '(Q t) } & \rightarrow (t) \\
\text{(my-fn (my-fn my-const))) } & \rightarrow (C) \\
\text{(setf (symbol-function 'my-fn)} & \\
\text{ (FUNCTION CAR) ) } & \rightarrow #<complied-function car> \\
\text{(my-fn '(A B C)) } & \rightarrow 
\end{align*}
\]
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 \rightarrow{} 1  
  b \rightarrow{} 2  
  c \rightarrow{} 3
Functions with Side Effects – SETF

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

▶ Simultaneous assignment

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

- a → 1
- b → 2
- c → 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 → 1
b → 2
c → 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 → 1
b → 2
c → 3
User-Defined Function with Side Effects

(SETQ my-var 5) →

(LAMBDA (X) ...)  
my-var → (A B C)

(fn2 (LIST (+ 3 4))) → (7)
my-var → (7)
User-Defined Function with Side Effects

(SETQ my-var 5) → 5
User-Defined Function with Side Effects

\[(\text{SETQ my-var 5}) \rightarrow 5\]
\[\text{my-var} \rightarrow\]
User-Defined Function with Side Effects

\[(\text{SETQ } \text{my-var } 5) \rightarrow 5\]
\[\text{my-var} \rightarrow 5\]
User-Defined Function with Side Effects

\[
\begin{align*}
(\text{SETQ } \text{my-var } 5) & \rightarrow 5 \\
\text{my-var} & \rightarrow 5 \\
(\text{SETF} \ (\text{symbol-function} \ '\text{fn2}) \ (\text{'(LAMBDA } \ (X) \ (\text{SETQ } \text{my-var} \ X)\ ))) & \rightarrow
\end{align*}
\]
User-Defined Function with Side Effects

\[(\text{SETQ my-var 5}) \rightarrow 5\]
\[\text{my-var} \rightarrow 5\]
\[(\text{SETF (symbol-function 'fn2)}\]
\[\quad '(\text{LAMBDA (X) (SETQ my-var X)})) \rightarrow\]
\[(\text{LAMBDA (X)} \ldots)\]
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

\[
(\text{SETQ } \text{my-var } 5) \rightarrow 5 \\
\text{my-var} \rightarrow 5 \\
(\text{SETF (symbol-function 'fn2)} \\
\hspace{1cm} '(\text{LAMBDA} (X) (\text{SETQ } \text{my-var } X))) \rightarrow \\
(\text{LAMBDA} (X) ...) \\
\text{my-var} \rightarrow 5
\]
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) \rightarrow 5
my-var \rightarrow 5

(SETF (symbol-function 'fn2)
  '(LAMBDA (X) (SETQ my-var X))) \rightarrow 
(LAMBDA (X) ...)
my-var \rightarrow 5
(fn2 '(A B C)) \rightarrow (A B C)
User-Defined Function with Side Effects

\[
\text{(SETQ my-var 5)} \rightarrow 5
\]
\[\text{my-var} \rightarrow 5\]
\[
\text{(SETF (symbol-function 'fn2) }
\text{'(LAMBDA (X) (SETQ my-var X)))} \rightarrow
\]
\[\text{(LAMBDA (X) ...)}\]
\[\text{my-var} \rightarrow 5\]
\[\text{(fn2 ' (A B C))} \rightarrow (\text{A B C})\]
\[\text{my-var} \rightarrow \]
(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

\[
\begin{align*}
(\text{SETQ } \text{my-var } 5) & \rightarrow 5 \\
\text{my-var} & \rightarrow 5 \\
(\text{SETF (symbol-function 'fn2)} & \rightarrow \text{LAMBDA (X) (SETQ my-var X)}) \rightarrow \\
(\text{LAMBDA (X) ...}) & \\
\text{my-var} & \rightarrow 5 \\
(\text{fn2 '(A B C)}) & \rightarrow (A B C) \\
\text{my-var} & \rightarrow (A B C) \\
(\text{fn2 (LIST (+ 3 4)))} & \rightarrow
\end{align*}
\]
(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 (v₁ ... vₙ) ⟨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.

\[(\text{SETQ } b \ '5)\]
\[B \rightarrow 5\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[(\text{SETQ } b \ '5)\]

\[B \rightarrow 5\]

\[X \rightarrow\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[(\text{SETQ } b \ '5)\]
\[B \rightarrow 5\]
\[X \rightarrow \text{undefined}\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[
\begin{align*}
(\text{SETQ } b \ '5) \\
B & \rightarrow 5 \\
X & \rightarrow \text{undefined} \\
(\text{setq } x \ 'b) & \rightarrow 
\end{align*}
\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[
\begin{align*}
\text{(SETQ b '5)} & \quad \rightarrow \quad 5 \\
B & \rightarrow \quad 5 \\
X & \rightarrow \quad \text{undefined} \\
\text{(setq x 'b)} & \rightarrow \quad B ;; \text{Not an error!}
\end{align*}
\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[(\text{SETQ b '5})\]
\[B \rightarrow 5\]
\[X \rightarrow \text{undefined}\]
\[(\text{setq x 'b}) \rightarrow B\ ;; \text{Not an error!}\]
\[X \rightarrow B\]
The SETQ Function

- SETQ does NOT evaluate its first argument.

\[
\text{\( (\text{SETQ } b \ '5) \)} \\
\text{B } \rightarrow \ 5 \\
\text{X } \rightarrow \ \text{undefined} \\
\text{\( (\text{setq } x \ 'b) \rightarrow B \); Not an error!} \\
\text{X} \rightarrow B \\
\text{B } \rightarrow 5 \quad \text{;; but B’s value unchanged.}
\]
The SET Function

- SET DOES evaluate its first argument.

\[ B \rightarrow \]
The SET Function

- SET DOES evaluate its first argument.

B → undefined
The SET Function

- SET DOES evaluate its first argument.

B → undefined
X →
The SET Function

- SET DOES evaluate its first argument.

  B → undefined
  X → undefined

(set X '(foo bar))
(set 'X '(foo bar))

(setq X 'B)

(set X (+ 100 12))

Note: Changes value of X's value (B)
The SET Function

- SET DOES evaluate its first argument.

  \[ \text{B} \rightarrow \text{undefined} \]
  \[ \text{X} \rightarrow \text{undefined} \]
  \[ (\text{set} \ \text{X} \ '(%\text{foo bar})) \rightarrow \]
The SET Function

- SET DOES evaluate its first argument.

\[ B \rightarrow \text{undefined} \]
\[ X \rightarrow \text{undefined} \]
\[ (\text{set } X '\text{(foo bar)}) \rightarrow x \text{ undefined} \]
The SET Function

- SET DOES evaluate its first argument.
  
  B → undefined
  X → undefined
  (set X '(foo bar)) → x undefined
  (set 'X '(foo bar)) →
The SET Function

- SET DOES evaluate its first argument.

  \[
  \begin{align*}
  B & \rightarrow \text{undefined} \\
  X & \rightarrow \text{undefined} \\
  (\text{set } X \ '(\text{foo bar})) & \rightarrow x \text{ undefined} \\
  (\text{set } 'X \ '(\text{foo bar})) & \rightarrow (\text{foo bar}) ;; \text{Now } X \leftarrow (\text{foo bar})
  \end{align*}
  \]
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.

\[
\begin{align*}
B & \rightarrow \text{undefined} \\
X & \rightarrow \text{undefined} \\
\text{(set X '}(\text{foo bar})) & \rightarrow x \text{ undefined} \\
\text{(set 'X '}(\text{foo bar})) & \rightarrow (\text{foo bar}) \;; \text{ Now X } \leftarrow (\text{foo bar}) \\
\text{(setq X 'B)} & \rightarrow B \;; \text{ Now X } \leftarrow B
\end{align*}
\]
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.

\[
\begin{align*}
B & \rightarrow \text{undefined} \\
X & \rightarrow \text{undefined} \\
\text{(set } X \ ' (\text{foo bar}) & \rightarrow x \text{ undefined} \\
\text{(set } 'X \ ' (\text{foo bar}) & \rightarrow (\text{foo bar}) ;; \text{Now } X \leftarrow (\text{foo bar}) \\
\text{(setq } X \ 'B) & \rightarrow B ;; \text{Now } X \leftarrow B \\
X & \rightarrow B \\
\text{(set } X \ (+ 100 \ 12)) & \rightarrow
\end{align*}
\]
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.

\[
\begin{align*}
B & \rightarrow \text{undefined} \\
X & \rightarrow \text{undefined} \\
(\text{set } X \ '({\text{foo bar}})) & \rightarrow x \text{ undefined} \\
(\text{set } 'X \ '({\text{foo bar}})) & \rightarrow(\text{foo bar}) ;; \text{ Now } X \leftarrow(\text{foo bar}) \\
(\text{setq } X \ 'B) & \rightarrow B ;; \text{ Now } X \leftarrow B \\
X & \rightarrow B \\
(\text{set } X \ (+\ 100\ 12)) & \rightarrow 112 ;;\text{ Note: Changes value of } X\text{'s value (B)} \\
X & \rightarrow
\end{align*}
\]
The SET Function

- SET DOES evaluate its first argument.

\[ \begin{align*}
B & \rightarrow \text{undefined} \\
X & \rightarrow \text{undefined} \\
\text{(set } X \ ')(\text{foo bar}) & \rightarrow x \text{ undefined} \\
\text{(set } 'X \ ')(\text{foo bar}) & \rightarrow(\text{foo bar}) \ ; ; \text{ Now } X \leftarrow(\text{foo bar}) \\
\text{(setq } X \ 'B) & \rightarrow B \ ; ; \text{ Now } X \leftarrow B \\
X & \rightarrow B \\
\text{(set } X \ (+ \ 100 \ 12)) & \rightarrow 112 \ ; ; \text{Note: Changes value of } X\text{'s value (B)} \\
X & \rightarrow B
\end{align*} \]
The SET Function

- SET DOES evaluate its first argument.

\[
\begin{align*}
B & \rightarrow \text{undefined} \\
X & \rightarrow \text{undefined} \\
\text{(set X ‘(foo bar))} & \rightarrow x \text{ undefined} \\
\text{(set ’X ‘(foo bar))} & \rightarrow \text{(foo bar)} \;; \text{ Now } X \leftarrow \text{(foo bar)} \\
\text{(setq X ’B)} & \rightarrow B \;; \text{ Now } X \leftarrow B \\
X & \rightarrow B \\
\text{(set X (+ 100 12))} & \rightarrow 112 \;; \text{Note: Changes value of X’s value (B)} \\
X & \rightarrow B \\
B & \rightarrow
\end{align*}
\]
The SET Function

- SET DOES evaluate its first argument.

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

```text
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 →

  ;; don’t need quote for numbers
Numbers in Lisp

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

  5 → 5

  ;; don’t need quote for numbers
Numbers in Lisp

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

\[
5 \rightarrow 5 \\
\text{(list 5 'a)} \rightarrow \\
;; \text{don’t need quote for numbers}
\]
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.)
  
  $$5 \rightarrow 5$$
  
  $$(\text{list} \ 5 \ \text{'a}) \rightarrow (5 \ \text{a})$$
  
  ;; 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.)

  \[
  5 \rightarrow 5 \\
  \text{list 5 ’a) } \rightarrow \text{ (5 a) }
  \]

  ;; don’t need quote for numbers

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

  \[
  \text{(numberp 12) } \rightarrow 
  \]
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
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.)
  
  \[ 5 \rightarrow 5 \]
  
  \[(\text{list} \ 5 \ 'a) \rightarrow (5 \ a)\]

  ;; don’t need quote for numbers

- Numberp tests whether an s-expr is a numeric atom.
  
  \[(\text{numberp} \ 12) \rightarrow t\]
  
  \[(\text{numberp} \ 'a) \rightarrow \text{nil}\]
Numbers in Lisp

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

  \[
  5 \rightarrow 5 \\
  (\text{list} \ 5 \ \text{'a}) \rightarrow (5 \ a) \\
  ;; \text{don’t need quote for numbers}
  \]

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

  \[
  (\text{numberp} \ 12) \rightarrow t \\
  (\text{numberp} \ \text{'a}) \rightarrow \text{nil} \\
  (\text{setq} \ n \ 25) \\
  (\text{numberp} \ n) \rightarrow 
  \]
Numbers in Lisp

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

  \[5 \rightarrow 5\]
  \[(\text{list } 5 \ 'a) \rightarrow (5 \ a)\]
  
  ;; don’t need quote for numbers

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

  \[(\text{numberp } 12) \rightarrow \text{t}\]
  \[(\text{numberp } 'a) \rightarrow \text{nil}\]
  \[(\text{setq } n \ 25)\]
  \[(\text{numberp } n) \rightarrow \text{t} \quad \text{;; numberp evaluates its arguments}\]
Numbers in Lisp

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

  \[ 5 \rightarrow 5 \]
  \[ (\text{list } 5 \ 'a) \rightarrow (5 \ a) \]
  \[ ;; \text{ don’t need quote for numbers} \]

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

  (numberp 12) → t
  (numberp 'a) → nil
  (setq n 25)
  (numberp n) →
  t \[ ;; \text{ numberp evaluates its arguments} \]
  (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

\[(\text{expt } 2 \ 5) \rightarrow\]
Integers

- There is no apriori limit on size of an integer

  \((\text{expt } 2 \ 5) \rightarrow 32\)
Integers

- There is no a priori limit on size of an integer

\[(\text{expt } 2 \ 5) \rightarrow 32\]
\[(\text{expt } 2 \ 100)\]
\[\rightarrow\]
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
  
  \[(\text{expt } 2 \, 5) \rightarrow 32\]
  \[(\text{expt } 2 \, 100)\]
  \[\rightarrow 1267650600228229401496703205376\]

- 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

  \[(\text{expt } 2 5) \rightarrow 32\]
  \[(\text{expt } 2 100) \rightarrow 1267650600228229401496703205376\]

- 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

  \[(\text{expt} \ 2 \ 5) \rightarrow 32\]
  \[(\text{expt} \ 2 \ 100) \rightarrow 1267650600228229401496703205376\]

- 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
  
  \((\text{expt } 2 \ 5) \rightarrow 32\)
  
  \((\text{expt } 2 \ 100)\)
  
  \(\rightarrow 1267650600228229401496703205376\)

- 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
  
  \[(\text{expt 2 5}) \rightarrow 32\]
  
  \[(\text{expt 2 100}) \rightarrow 1267650600228229401496703205376\]

- 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
  
  \(#10r15 \rightarrow\)
Integers

- There is no apriori limit on size of an integer
  
  \[(\text{expt } 2 \ 5) \rightarrow 32\]
  
  \[(\text{expt } 2 \ 100) \rightarrow 1267650600228229401496703205376\]

- 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

  \[#10r15 \rightarrow 15\]
Integers

- There is no apriori limit on size of an integer

\[(\text{expt } 2 \text{ 5}) \rightarrow 32\]
\[(\text{expt } 2 \text{ 100}) \rightarrow 1267650600228229401496703205376\]

- 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

\[\#10r15 \rightarrow 15\]
\[\#2r1111 \rightarrow \]
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
  
  \[(\text{expt } 2 \ 5) \rightarrow 32\]
  
  \[(\text{expt } 2 \ 100)\]
  
  \[\rightarrow 1267650600228229401496703205376\]

- 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

  \[#10r15 \rightarrow 15\]
  
  \[#2r1111 \rightarrow 15\]
  
  \[#3r120 \rightarrow\]
Integers

- There is no apriori limit on size of an integer
  \[(\text{expt } 2 \ 5) \rightarrow 32\]
  \[(\text{expt } 2 \ 100) \rightarrow 1267650600228229401496703205376\]

- 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 \rightarrow 15  \ #2r1111 \rightarrow 15  \ #3r120 \rightarrow 15
Ratios, Floats

- Exact ratios can be represented without roundoff error
Ratios, Floats

- Exact ratios can be represented without roundoff error

\[(\text{expt} \ (\text{\texttt{/}} \ 2 \ 3) \ 2) \rightarrow\]
Ratios, Floats

- Exact ratios can be represented without roundoff error

\[ \text{expt} \left( \frac{2}{3} \right)^2 \rightarrow \frac{4}{9} \]
Ratios, Floats

- Exact ratios can be represented without roundoff error
  \[
  (\text{expt} \ (\div 2 3) 2) \rightarrow \frac{4}{9}
  \]

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

- Exact ratios can be represented without roundoff error
  
  \[(\text{expt } (\text{/ } 2 \ 3) \ 2) \rightarrow 4/9\]

- 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

\[(\text{expt} \left(\frac{2}{3}\right) 2) \rightarrow \frac{4}{9}\]

- 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

\[ \text{\#C(real imaginary)} \]
\[ 1-2i = \#C(1 -2) \]

Many Lisp functions will take complex arguments

\[ (* \#c(0 -1) \#c(0 -1)) \rightarrow 1 \]
\[ \pi \rightarrow 3.1415926535897932385L0 \]
\[ (\exp (* \#c(0 -1) \pi)) \rightarrow \#C(-1.0L0 5.0165576136843360246L-20) \approx -1 \]

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

- Complex numbers have their own notation in Lisp

\[ #C( \text{real imaginary} ) \]
\[ 1-2i = #C(1 -2) \]
Complex Numbers

- Complex numbers have their own notation in Lisp
  
  \[
  \#C(\text{real}\ \text{imaginary})
  \]
  
  \[
  1-2i = \#C(1\ -2)
  \]

- Many Lisp functions will take complex arguments

\[
(* \#c(0\ -1)\ \#c(0\ -1))
\rightarrow
1
\]

\[
\pi
\rightarrow
3.1415926535897932385L0
\]

\[
(exp\ (*\ \#c(0\ -1)\ \pi))
\rightarrow
#C(-1.0L0\ 5.0165576136843360246L-20)
\approx
-1
\]

(i.e., the Euler identity
\[
e^{i\pi}=-1
\]
Complex Numbers

- Complex numbers have their own notation in Lisp
  
  \[ \#C( \text{real imaginary} ) \]
  
  \[ 1-2i = \#C(1 -2) \]

- Many Lisp functions will take complex arguments
  
  \[ (* \#c(0 -1) \#c(0 -1)) \rightarrow \]
  
  \[ \pi \rightarrow 3.1415926535897932385L0 \]
  
  \[ (\exp (* \#C(0 -1) \pi)) \rightarrow \#C(-1.0L0 5.0165576136843360246L-20) \]
  
  \[ \approx -1 \]
  
  (i.e., the Euler identity \( e^{i\pi} = -1 \))
Complex Numbers

- Complex numbers have their own notation in Lisp
  
  \[
  \#C( \text{real imaginary} ) \\
  1-2i = \#C(1 -2)
  \]

- Many Lisp functions will take complex arguments
  
  \[
  (* \#c(0 -1) \#c(0 -1)) \rightarrow 1
  \]
Complex Numbers

- Complex numbers have their own notation in Lisp

  \[
  \text{#C( real imaginary )}
  \]

  \[1 - 2i = \text{#C}(1 -2)\]

- Many Lisp functions will take complex arguments

  \[
  (* \text{#c}(0 -1) \text{#c}(0 -1)) \rightarrow 1
  \]

  \[
  \pi \rightarrow 3.1415926535897932385L0
  \]

  \[
  (\exp (* \text{#c}(0 -1) \pi))
  \rightarrow
  \]
Complex Numbers

- Complex numbers have their own notation in Lisp
  \[ \text{#C( real imaginary )} \]
  \[ 1-2i = \text{#C}(1 \ -2) \]

- Many Lisp functions will take complex arguments

\[
(* \ \text{#c}(0 \ -1) \ \text{#c}(0 \ -1)) \rightarrow 1
\]

\[ \pi \rightarrow 3.1415926535897932385L0 \]
\[ (\exp (* \ \text{#c}(0 \ -1) \ \pi)) \rightarrow \text{#C}(-1.0L0 \ 5.0165576136843360246L-20) \approx -1 \]
(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) →
Numerical Operations

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

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

Binary Functions: \texttt{MOD}
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:

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

▶ Binary Predicates:

▶ Unary Predicates:

- ZEROP
Numerical Operations

- Unary Functions:

  \[(1 + 3) \rightarrow 4\]

  \[(1 - 3) \rightarrow -2\]

  \[(\text{ABS} -2) \rightarrow 2\]

  \[\left(\frac{\text{SIN}}{\pi} 2\right) \rightarrow 1.0L0\] (returned a float)

  \[\left(\frac{\text{COS}}{\pi} 2\right) \rightarrow -2.5082788076048218878L^{-20}\]

  \[\approx 0\]

- Binary Predicates:

- Unary Predicates:

  \[\text{ZEROP}\]
Numerical Operations

▶ Unary Functions:

\[(1 + 3) \rightarrow 4\]
Numerical Operations

- **Unary Functions:**

  \[(1+3) \rightarrow 4\]
  \[(1-3) \rightarrow \]
  \[\text{ABS -2} \rightarrow 2\]
  \[\text{SIN (/ pi 2)} \rightarrow 1.0L0\] returned a float
  \[\text{COS (/ pi 2)} \rightarrow -2.5082788076048218878L-20\] \(\approx 0\)
Numerical Operations

- Unary Functions:
  
  $(1+3) \rightarrow 4$
  
  $(1-3) \rightarrow 2$

- Binary Predicates:

- Unary Predicates:

  ZEROP
Numerical Operations

- Unary Functions:

  \[(1+ 3) \rightarrow 4\]
  \[(1- 3) \rightarrow 2\]
  \[(\text{ABS} -2) \rightarrow\]
Numerical Operations

 Unary Functions:

\[(1 + 3) \rightarrow 4\]
\[(1 - 3) \rightarrow 2\]
\[(\text{ABS} -2) \rightarrow 2\]
Numerical Operations

- **Unary Functions:**
  
  \[(1 + 3) \rightarrow 4\]
  \[(1 - 3) \rightarrow 2\]
  \[(\text{ABS} -2) \rightarrow 2\]
  \[(\text{SIN} (/ \text{pi} 2)) \rightarrow \]
Numerical Operations

▶ Unary Functions:

\[(1 + 3) \rightarrow 4\]
\[(1 - 3) \rightarrow 2\]
\[(\text{ABS} -2) \rightarrow 2\]
\[(\text{SIN} (/ \pi 2)) \rightarrow 1.0L0 \;; \text{ returned a float}\]
Numerical Operations

- **Unary Functions:**
  
  \[
  (1 + 3) \rightarrow 4 \\
  (1 - 3) \rightarrow 2 \\
  (ABS -2) \rightarrow 2 \\
  (SIN (/ pi 2)) \rightarrow 1.0L0 ;; returned a float \\
  (COS (/ pi 2)) \rightarrow 
  \]
Numerical Operations

- **Unary Functions:**
  
  - \((1 + 3) \rightarrow 4\)
  - \((1 - 3) \rightarrow 2\)
  - \((\text{ABS} -2) \rightarrow 2\)
  - \((\text{SIN} (/ \pi 2)) \rightarrow 1.0\text{L}0 ;; \text{returned a float}\)
  - \((\text{COS} (/ \pi 2)) \rightarrow -2.5082788076048218878\text{L}-20 \approx 0\)
Numerical Operations

- **Unary Functions:**
  
  \[(1 + 3) \rightarrow 4\]
  
  \[(1 - 3) \rightarrow 2\]
  
  \[(\text{ABS} -2) \rightarrow 2\]
  
  \[(\text{SIN} (/ \text{pi} 2)) \rightarrow 1.0L0 ;; \text{returned a float}\]
  
  \[(\text{COS} (/ \text{pi} 2)) \rightarrow -2.5082788076048218878L-20 \approx 0\]

- **Binary Predicates:** \(<\quad >\quad >=\quad <=\)
Numerical Operations

- **Unary Functions:**
  
  \[
  (1 + 3) \rightarrow 4 \\
  (1 - 3) \rightarrow 2 \\
  (\text{ABS} -2) \rightarrow 2 \\
  (\text{SIN} (/ \text{pi} 2)) \rightarrow 1.0L0 ;; \text{returned a float} \\
  (\text{COS} (/ \text{pi} 2)) \rightarrow -2.5082788076048218878L-20 \approx 0
  \]

- **Binary Predicates:** \(<\>\>\>=\<\=

- **Unary Predicates:** \text{ZEROP}
Numbers Are Not Always EQ!

- Numbers are atoms:
  
  \[
  \text{atom 5} \rightarrow \text{T (atom 4.0)} \rightarrow \text{T (atom \#C(1 -1))} \rightarrow \text{T}
  \]
Numbers Are Not Always EQ!

- Numbers are atoms:

  \(\text{(atom 5)} \rightarrow \text{T} \quad \text{(atom 4.0)} \rightarrow \text{T} \quad \text{(atom \#C(1 -1))} \rightarrow \text{T}\)
Numbers Are Not Always EQ!

- Numbers are atoms:

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

- Recall: equivalent items (e.g., eq) vs. equal items (e.g., 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 Not Always EQ!

- Numbers are atoms:
  
  \[(\text{atom } 5) \rightarrow \text{T} \quad (\text{atom } 4.0) \rightarrow \text{T} \quad (\text{atom } \#C(1 \ -1)) \rightarrow \text{T}\]

- 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:

  \((\text{atom 5}) \rightarrow \text{T} \quad (\text{atom 4.0}) \rightarrow \text{T} \quad (\text{atom \#C(1 -1)}) \rightarrow \text{T}\)

- 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:

  \[(\text{atom 5}) \rightarrow \text{T} \quad (\text{atom 4.0}) \rightarrow \text{T} \quad (\text{atom } \#C(1 \ -1)) \rightarrow \text{T}\]

- 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

\[(= \ 4 \ 4.0) \rightarrow \text{T}\]
Numbers Are Not Always EQ!

- Numbers are atoms:

  \[(\text{atom } 5) \rightarrow \text{T} \quad (\text{atom } 4.0) \rightarrow \text{T} \quad (\text{atom } \#C(1 \ -1)) \rightarrow \text{T}\]

- 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

  \[ (= \ 4 \ 4.0) \rightarrow \text{T} \]

  \[ (\text{eq} \ 4 \ 4.0) \rightarrow \]
Numbers Are Not Always EQ!

- Numbers are atoms:
  
  \[(\text{atom } 5) \rightarrow \text{T} \quad (\text{atom } 4.0) \rightarrow \text{T} \quad (\text{atom } \#\text{C}(1 \ 1)) \rightarrow \text{T}\]

- 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
  
  \[ (= 4 4.0) \rightarrow \text{T} \]
  
  \[ (\text{eq } 4 4.0) \rightarrow \text{nil} \quad ;; \text{mathematically equal}\]
Numbers Are Not Always EQ!

- Numbers are atoms:
  
  \[(\text{atom 5}) \rightarrow T \quad (\text{atom 4.0}) \rightarrow T \quad (\text{atom #C(1 -1)}) \rightarrow T\]

- 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
  
  \[(= 4 4.0) \rightarrow T\]
  
  \[(\text{eq 4 4.0}) \rightarrow \text{nil} \quad ;; \text{mathematically equal}\]

  \[(= 1234567890 1234567890 ) \rightarrow\]
Numbers Are Not Always EQ!

► Numbers are atoms:

\[(\text{atom } 5) \rightarrow T \quad (\text{atom } 4.0) \rightarrow T \quad (\text{atom } \#C(1 \ -1)) \rightarrow T\]

► 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

\[(= 4 4.0) \rightarrow T\]

\[(\text{eq } 4 4.0) \rightarrow \text{nil} \quad ;; \text{mathematically equal}\]

\[(= 1234567890 1234567890) \rightarrow T\]
Numbers Are Not Always EQ!

- Numbers are atoms:

  \[(\text{atom 5}) \rightarrow \text{T} \quad (\text{atom 4.0}) \rightarrow \text{T} \quad (\text{atom } \#\text{C}(1 \ -1)) \rightarrow \text{T}\]

- 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

  \[(= 4 \ 4.0) \rightarrow \text{T} \]
  \[(\text{eq 4 4.0}) \rightarrow \text{nil} \quad ;; \text{mathematically equal}\]

  \[(= 1234567890 \ 1234567890) \rightarrow \text{T} \]
  \[(\text{eq 1234567890 1234567890}) \rightarrow\]
Numbers Are Not Always EQ!

- Numbers are atoms:

  \[(\text{atom } 5) \rightarrow T \quad (\text{atom } 4.0) \rightarrow T \quad (\text{atom } \#C(1 \ -1)) \rightarrow T\]

- 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

  \[(= \ 4 \ 4.0) \rightarrow T\]
  \[(\text{eq } 4 \ 4.0) \rightarrow \text{nil} \quad ;; \text{mathematically equal}\]

  \[(= \ 1234567890 \ 1234567890) \rightarrow T\]
  \[(\text{eq } 1234567890 \ 1234567890) \rightarrow\]
  \[\text{nil} \quad ;; \text{distinct bignums}\]
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

- \textsc{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

Value of (CONS s1 s2) is (s1 . s2) for any s1, s2 ∈ S-exp

(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)
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
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 \((\text{CONS } s_1 \ s_2)\) is \((s_1 \ . \ s_2)\)
  for any \(s_1, s_2 \in \text{s-expr}\)

\((\text{cons } 'a \ 'b) \rightarrow (a \ . \ b)\)
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 ∈ s-expr

(cons 'a 'b) → (a . b)
(cons 4 '(a b c)) →
Dotted-Pair

- CONS can really take ANY pair of S-expressions
- earlier, just dealt with atoms and lists
- Value of \((\text{CONS } s_1 \ s_2)\) is \((s_1 . \ s_2)\)
  for any \(s_1, s_2 \in \text{s-expr}\)

\[(\text{cons } 'a \ 'b) \rightarrow (a . \ b)\]
\[(\text{cons } 4 \ '(a \ b \ c)) \rightarrow (4 . \ (a \ b \ c))\]
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 ∈ s-expr

(cons ’a ’b) → (a . b)
(cons 4 ’(a b c)) → (4 . (a b c))
(cons ’(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))
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 \(s1, \, s2 \in s\text{-expr}\)

\[
\begin{align*}
(cons \ 'a \ 'b) & \rightarrow (a \ . \ b) \\
(cons \ 4 \ '(a \ b \ c)) & \rightarrow (4 \ . \ (a \ b \ c)) \\
(cons \ '(t) \ '(a \ . \ b)) & \rightarrow ((t) \ . \ (a \ . \ b)) \\
(cons \ 4 \ '(a \ . \ b)) & \rightarrow (4 \ . \ (a \ . \ b))
\end{align*}
\]
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 \((\text{CONS } s_1 s_2)\) is \((s_1 . s_2)\) for any \(s_1, s_2 \in \text{s-expr}\)

\[
\begin{align*}
(\text{cons } 'a 'b) & \rightarrow (a . b) \\
(\text{cons } 4 '(a b c)) & \rightarrow (4 . (a b c)) \\
(\text{cons } '(t) '(a . b)) & \rightarrow ((t) . (a . b)) \\
(\text{cons } 4 '(a . b)) & \rightarrow (4 . (a . b))
\end{align*}
\]

- Retrieving components

\[
\begin{align*}
(\text{CAR } '(a . b)) & \rightarrow
\end{align*}
\]
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))

- Retrieving 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

\[
\begin{align*}
(\text{cons } 'a 'b) &\rightarrow (a . b) \\
(\text{cons } 4 '(a b c)) &\rightarrow (4 . (a b c)) \\
(\text{cons } '(t) '(a . b)) &\rightarrow ((t) . (a . b)) \\
(\text{cons } 4 '(a . b)) &\rightarrow (4 . (a . b))
\end{align*}
\]

- Retrieving components

\[
\begin{align*}
(\text{CAR } '(a . b)) &\rightarrow a \\
(\text{CAR } (\text{CONS } 'a 'b)) &\rightarrow a
\end{align*}
\]
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-exp

(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))

- Retrieving 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 \((\text{CONS } s_1 \ s_2)\) is \((s_1 \ . \ s_2)\)
  for any \(s_1, s_2 \in \text{s-expr}\)

\[
\begin{align*}
(\text{cons } 'a \ 'b) & \rightarrow (a \ . \ b) \\
(\text{cons 4 '}(a \ b \ c)) & \rightarrow (4 \ . \ (a \ b \ c)) \\
(\text{cons } '(t) '}(a \ . \ b)) & \rightarrow ((t) \ . \ (a \ . \ b)) \\
(\text{cons 4 '}(a \ . \ b)) & \rightarrow (4 \ . \ (a \ . \ b))
\end{align*}
\]

- Retrieving components

\[
\begin{align*}
(\text{CAR } '(a \ . \ b)) & \rightarrow a \\
(\text{CAR } (\text{CONS } 'a \ 'b)) & \rightarrow a \\
(\text{CDR } '((t) \ . \ (a \ . \ b))) & \rightarrow (a \ . \ b)
\end{align*}
\]
Notation

- Can write \((s_1 \cdot (s_2 \ldots))\) as \((s_1 s_2 \ldots)\)
  
  Hence \((a \cdot (b \cdot c)) \mapsto (a \cdot (b \cdot c))\)
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 \((\text{cons } 'a \ nil)\) \(\mapsto\) \((a . \ nil)\) \(\mapsto\) \((a)\)
Notation

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

- Can write \((s_1 s_2 ...s_n . \text{nil})\) as \((s_1 s_2 ...s_n )\)
  Hence \(\text{cons 'a nil} \mapsto (a . \text{nil}) \mapsto (a)\)

- Notice:
  When \text{CONS’s} 2nd arg is list, just as before!
Dotted Pair – Internals

\[(\text{SETQ } \text{foo } (\text{CONS } \text{'a} (\text{CONS } \text{'b} \text{'c})))\]

\[(\text{a . (b . c)})\]
Association Lists

- Can be assigned:
  ```lisp
  (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) →

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)
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) →
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.

\[
\begin{align*}
\text{(assoc 'name bart)} & \rightarrow \text{(name Bart Selman)} \\
\text{(assoc 'children bart)} & \rightarrow \text{(children (Mary Louise) (Jean Pierre))} \\
\text{(assoc 'habit bart)} & \rightarrow
\end{align*}
\]
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) \rightarrow (name Bart Selman)
(assoc 'children bart) \rightarrow (children (Mary Louise) (Jean Pierre))
(assoc 'habit bart) \rightarrow (habit)
(assoc 'mother bart) \rightarrow 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

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

- A key purified 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
The ASSOC Function

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

```lisp
(assoc '(a) '( ( (a) . 1 )) :test 'equal)
```

- 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
The ASSOC Function

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

```lisp
(assoc '(a) '((a) . 1)) → NIL
```
The ASSOC Function

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

\[(assoc '(a) '( ( (a) . 1 ))) \rightarrow \text{NIL}\]
\[(assoc '(a) '( ( (a) . 1 )) :test 'equal) \rightarrow \]
The ASSOC Function

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

\[
(\text{assoc } '(a) '(((a) . 1))) \rightarrow \text{NIL}
\]
\[
(\text{assoc } '(a) '(((a) . 1)) :\text{test } '\text{equal})
\rightarrow ((A) . 1)
\]
The ASSOC Function

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

\[
(\text{assoc '(a) ')((a) . 1)) \rightarrow \text{NIL}
\]
\[
(\text{assoc '(a) '(((a) . 1)) :test 'equal})
\]
\[
\rightarrow ((\text{A}). \text{1})
\]

- A key pure list data structure:
The ASSOC Function

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

\[
\text{(assoc '(a) '((a) . 1)))} \rightarrow \text{NIL}
\]
\[
\text{(assoc '(a) '((a) . 1)) :test 'equal)} \rightarrow ((A) . 1)
\]

- 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

```lisp
(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

  \[
  \text{(assoc '}(\mathit{a}) '((\mathit{a}) \ 1))) \rightarrow \text{NIL}
  \]
  \[
  \text{(assoc '}(\mathit{a}) '((\mathit{a}) \ 1)) \ :\text{test 'equal)}
  \rightarrow ((\mathit{A}) \ 1)
  \]

- 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

\[
\text{assoc '}(a) \text{'( ( (a) . 1 ))) → NIL}
\]

\[
\text{assoc '}(a) \text{'( ( (a) . 1 ))) :test 'equal) →((A) . 1)
\]

- 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

- The `acons` function creates a new entry

- The `pairlis` function zips two lists together into an association list
Other Association List Functions

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

\[(rassoc '(Bart Selman) bart) →\]
Other Association List Functions

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

  \[(\text{rassoc } '(\text{Bart Selman}) \text{ bart}) \rightarrow (\text{name } (\text{Bart Selman}))\]
Other Association List Functions

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

\[
\text{(rassoc '(Bart Selman) bart)} \rightarrow \text{(name (Bart Selman))}
\]

▶ The `acons` function creates a new entry

\[
\text{(acons 'age 42 bart)} \equiv \text{(cons (cons 'age 42) bart)}
\]

▶ The `pairlis` function zips two lists together into an association list

\[
\text{(pairlis '(1 2 3) '(a b c))} \rightarrow \text{((1 . a) (2 . b) (3 . c))}
\]
Other Association List Functions

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

  \[
  \text{rassoc } '(\text{Bart Selman}) \text{ bart} \rightarrow \text{(name (Bart Selman))}
  \]

- The `acons` function creates a new entry

  \[
  \text{acons } '\text{age 42} \text{ bart} \equiv \text{(cons (cons } '\text{age 42})) \text{ bart)}
  \]
Other Association List Functions

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

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

- The `acons` function creates a new entry

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

- The `pairlis` function zips two lists together into an association list

  `(pairlis '(1 2 3) '(a b c)) → ((1 . a) (2 . b) (3 . c))`
Other Association List Functions

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

\[
\text{(rassoc '}(\text{Bart Selman}) \text{ bart) \rightarrow (name (Bart Selman))}
\]

- The `acons` functions creates a new entry

\[
\text{(acons 'age 42 bart) \equiv (cons (cons 'age 42) bart)}
\]

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

\[
\text{(pairlis '}(1 2 3) '}(a b c)\text{)}
\rightarrow ( (1 . a) (2 . b) (3 . c) )
\]
General List Functions

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

\[
\text{(some predicate sequence1 sequence2 ...)}
\]
General List Functions

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

  \[
  \text{(some predicate sequence1 sequence2 ...)}
  \]

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

  \[
  (\text{(some #'(lambda (x y) (not (equal x y))) (1 2 3) (1 2 4)}) \rightarrow T
  \]
General List Functions

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

\[
\text{(some predicate sequence1 sequence2 ...)}
\]

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

\[
(\text{(some #'(lambda (x y) (not (equal x y))) '(1 2 3) '(1 2 4))} \rightarrow T
\]
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

  \[
  (\text{some} \; \text{#'(lambda (x y) \ (not \ (equal \ x \ y)))}) \\ \to \ \ T
  \]

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

  \[
  (\text{every} \; \text{#'(lambda (x y) \ (equal \ x \ y)))} \\ \to
  \]

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp
General List Functions

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

  \[ \text{(some predicate sequence1 sequence2 ...)} \]

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

  \[
  \left( \text{(some \#'(lambda (x y) \ (not (equal x y)))} \right.
  \\
  \left. (1\ 2\ 3) \ (1\ 2\ 4) \rightarrow T \right)
  \]

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

  \[
  \left( \text{(every \#'(lambda (x y) \ (equal x y)))} \right.
  \\
  \left. (1\ 2\ 3) \ (1\ 2\ 4) \rightarrow \text{nil} \right)
  \]
General List Functions

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

  \[
  \text{find item sequence}
  \]

Many other functions: `position`, `mismatch`, `substitute`, `remove`, `sort`
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)) →

  (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

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

  \[
  \text{(find item sequence)}
  \]

  \[
  ;; \text{recall any non-nil value is true}
  \text{(find 'a '(c b a)) } \rightarrow \text{A}
  \]
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)) →
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`

\[
\text{(find item sequence)}
\]

;; recall any non-nil value is true
\[
\text{(find 'a '(c b a)) → A}
\]
\[
\text{(find 'a '(c b)) → nil}
\]

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

\[
\text{(find-if #'oddp '(2 4 7 6 9)) →}
\]
General List Functions

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

  \[(\text{find item sequence})\]
  \[;;\text{ recall any non-nil value is true}\]
  \[(\text{find } \text{'}a\text{'} (c\ b\ a)) \rightarrow \text{A}\]
  \[(\text{find } \text{'}a\text{'} (c\ b)) \rightarrow \text{nil}\]

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

  \[(\text{find-if #'oddp '(2\ 4\ 7\ 6\ 9)) \rightarrow 7\]
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

\[(\text{assoc item list :test fn})\]
\[\equiv (\text{find item list :test fn :key #’car})\]
General List Functions

- More list functions
General List Functions

- More list functions

(member 1 '(1 2 3)) →
General List Functions

- More list functions

  \[(\text{member } 1 \ '(1 \ 2 \ 3)) \rightarrow \]
  \[(1 \ 2 \ 3) ;; \text{i.e., non-nil} \]
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

- \((\text{member } 1 \ '(1\ 2\ 3)) \rightarrow (1\ 2\ 3)\); i.e., non-nil
- \((\text{union } '(1\ 2\ 3)\ '(2\ 3\ 4)) \rightarrow (1\ 2\ 3\ 4)\)
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

\[ (\text{member} \ 1 \ '(1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \ ;; \ i.e., \ non-nil \]
\[ (\text{union} \ '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow (1 \ 2 \ 3 \ 4) \]
\[ (\text{intersection} \ '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow (2 \ 3) \]
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

\[(\text{member} \ 1 \ '(1 \ 2 \ 3)) \rightarrow \ (1 \ 2 \ 3)\]; i.e., non-nil
\[(\text{union} \ '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow \ (1 \ 2 \ 3 \ 4)\]
\[(\text{intersection} \ '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow \ (2 \ 3)\]
\[(\text{adjoin} \ 2 \ '(1 \ 2 \ 3)) \rightarrow \ (1 \ 2 \ 3)\]; \textit{add if absent}
\[(\text{adjoin} \ 4 \ '(1 \ 2 \ 3)) \rightarrow \ (4 \ 1 \ 2 \ 3)\]
General List Functions

▶ More list functions

\[\text{(member 1 '(1 2 3)) \rightarrow (1 2 3)}\]; i.e., non-nil
\[\text{(union '(1 2 3) '(2 3 4)) \rightarrow (1 2 3 4)}\]
\[\text{(intersection '(1 2 3) '(2 3 4)) \rightarrow (2 3)}\]
\[\text{(adjoin 2 '(1 2 3)) \rightarrow (1 2 3)}\]; add if absent
\[\text{(adjoin 4 '(1 2 3)) \rightarrow (4 1 2 3)}\]

▶ 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

- General List Functions

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

- Destructive List Functions

  - (setf x '(1))

  - (push 2 x)

  - x \rightarrow (2 1)
General List Functions

▶ More list functions

\[(\text{member } 1 \ '(1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \;; \ i.e., \ non-nil\]
\[(\text{union } '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow (1 \ 2 \ 3 \ 4)\]
\[(\text{intersection } '(1 \ 2 \ 3) \ '(2 \ 3 \ 4)) \rightarrow (2 \ 3)\]
\[(\text{adjoin } 2 \ '(1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \;; \ add \ if \ absent\]
\[(\text{adjoin } 4 \ '(1 \ 2 \ 3)) \rightarrow (4 \ 1 \ 2 \ 3)\]

▶ Destructive List Functions

\[(\text{setf } x \ '(1))\]
\[(\text{push } 2 \ x)\]
\[x \rightarrow (2 \ 1)\]
\[(\text{pop } x) \rightarrow\]
General List Functions

- More list functions

\[
\begin{align*}
\text{(member 1 '(1 2 3))} & \rightarrow (1 2 3);; \text{i.e., non-nil} \\
\text{(union '(1 2 3) '(2 3 4))} & \rightarrow (1 2 3 4) \\
\text{(intersection '(1 2 3) '(2 3 4))} & \rightarrow (2 3) \\
\text{(adjoin 2 '(1 2 3))} & \rightarrow (1 2 3);; \text{add if absent} \\
\text{(adjoin 4 '(1 2 3))} & \rightarrow (4 1 2 3)
\end{align*}
\]

- Destructive List Functions

\[
\begin{align*}
\text{(setf x '(1))} \\
\text{(push 2 x)} & \\
x & \rightarrow (2 1) \\
\text{(pop x)} & \rightarrow 2
\end{align*}
\]
General List Functions

▶ More list functions

\[(\text{member } 1 \ (1 2 3)) \rightarrow (1 2 3) ;; \text{i.e., non-nil}\]
\[(\text{union } (1 2 3) \ (2 3 4)) \rightarrow (1 2 3 4)\]
\[(\text{intersection } (1 2 3) \ (2 3 4)) \rightarrow (2 3)\]
\[(\text{adjoin } 2 \ (1 2 3)) \rightarrow (1 2 3) ;; \text{add if absent}\]
\[(\text{adjoin } 4 \ (1 2 3)) \rightarrow (4 1 2 3)\]

▶ Destructive List Functions

\[(\text{setf } x \ (1))\]
\[(\text{push } 2 \ x)\]
\[x \rightarrow (2 1)\]
\[(\text{pop } x) \rightarrow 2\]
\[x \rightarrow \]
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
  - attached to specific symbol and
  - operations are destructive
Property Lists

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

(setf (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)
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) →

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

Do not delete plist as some implementations store important information about symbols in their plists.
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) → nil

- Do not delete plist as some implementations store important information about symbols in their plists
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) → 42
Property Lists

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

```lisp
(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
  - 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) → 42
(remprop 'clyde 'age)
(get 'clyde 'age) → nil
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) → 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

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

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
```

Bob Price and Russ Greiner
CMPUT325 Extensions to Pure Lisp
Hash Tables

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

(example)

```lisp
(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

\[
\text{(setq } \text{pops} \text{ (make-hash-table)}\text{)} \rightarrow \text{#S(HASH-TABLE EQL)}
\]
\[
\text{(gethash } \text{’calgary pops}\text{)} \rightarrow \text{nil; nil}
\]

- 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

\[
\text{(multiple-value-bind (value ok) (gethash } \text{’calgary pops}\text{) (if ok <form> ))}
\]
Hash Table Functions

Setting entries in a table

\[
\text{(setf (gethash 'calgary \textit{pops}) 876519)} \rightarrow 876519
\]
Setting entries in a table

(setf (gethash 'calgary pops) 876519) \rightarrow 876519
(gethash 'calgary pops) \rightarrow
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

\[
\text{(setf (gethash 'calgary pops) 876519)} \rightarrow 876519
\]

\[
\text{(gethash 'calgary pops) } \rightarrow 876519
\]

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

- Setting entries in a table

  \[(\text{setf (gethash ‘calgary pops) 876519}) \rightarrow 876519\]
  \[(\text{gethash ‘calgary pops}) \rightarrow 876519\]

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

- Other useful hash table functions

  \[(\text{remhash ‘calgary}) ;; \text{removes entry}\]
Hash Table Functions

- Setting entries in a table

\[(\text{setf} (\text{gethash} \ '\text{calgary} \ \text{pops}) \ 876519) \rightarrow \ 876519\]
\[(\text{gethash} \ '\text{calgary} \ \text{pops}) \rightarrow \ 876519\]

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

- Other useful hash table functions

\[(\text{remhash} \ '\text{calgary}) \ ;; \text{removes entry}\]
\[; ; \text{applies fn to each key-value}\]
\[(\text{maphash} \ \text{fn} \ \text{hash-table}) \ \text{pair}\]
Vectors and Arrays

- Construction and use of a vector

\[
\text{(setf } u \text{ } #(2 \ 3 \ 4))
\]

\[u \rightarrow\]
Vectors and Arrays

- Construction and use of a vector

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

- 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))
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 \rightarrow
#(2 3 4) ;; macro constructor is convenient
(setf v (vector 1 2 3))
v \rightarrow #(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)

▶ Matrices

(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)

▶ Matrices

(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))
  →

  #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)

  (setq (course-name cmput325) "Non-procedural programming")
  →
  "Non-procedural programming"

- Can be compiled to efficient memory accesses
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 programing")

(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))
  \rightarrow #S(COURSE :NAME NIL :ROOM NIL :TIME NIL)

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

  (course-name comput325) →"Non-procedural programming"
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 programming")

(course-name comput325) →"Non-procedural programming"

- Can be compiled to efficient memory accesses
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 in a method
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 definition provides a rich language
Lisp Objects: Classes

- Class definition 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: Classes

- Class definition 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 definition 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 definition provides a rich language

- A class with a slot named "color"

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

- Inheritance

```lisp
(defun class 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

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 dictionary ordering with

string= string> etc.
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))
  →
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))
  \rightarrow "BBBBBBBBBBB"
Lisp Strings

- Strings are built of characters which are introduced with `#\`
  ```lisp
  #\g #\G ;; these are different!
  #\space #\newline #\linefeed #\page
  #\return #\backspace #\rubout
  ```

- Can be constructed as constants or dynamically
  ```lisp
  (setf name "bob")
  (setf label (make-string 10 :initial-element #\B))
  → "BBBBBBBBBBB"
  ```

- 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) – flushes 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

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

\[
\text{(sqrt (read t))}
\]
\[
49 \quad ;; \text{user typing}
\]
\[
\rightarrow 7
\]
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

```lisp
(print 44)
44
```

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp
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
```
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)
(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-programming")
# I/O in Lisp – Control String

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~s</td>
<td>print arbitrary s-expr in default form</td>
</tr>
<tr>
<td>~a</td>
<td>print s-expr in ASCII form</td>
</tr>
<tr>
<td>~%</td>
<td>insert carriage return</td>
</tr>
<tr>
<td>~nS</td>
<td>pad output of s-expr to make n-char field</td>
</tr>
<tr>
<td>~n,dF</td>
<td>fixed floating point with field width n and decimals d</td>
</tr>
<tr>
<td>~n,dE</td>
<td>exponential or scientific notation</td>
</tr>
<tr>
<td>~n,dG</td>
<td>choose most appropriate of F or E</td>
</tr>
</tbody>
</table>
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))

(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)
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

```lisp
(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
I/O in Lisp – Output to strings

- Turning objects into strings

```lisp
(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
```
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 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 whereon the user can use :bt to examine the backtrace leading to the bug
Basic IO

▶ (terpri) – flushs buffer, prints carriage return
Basic IO

- `(terpri)` – flushes buffer, prints carriage return
- `(load '<file>`) – loads file named `<file>`.
(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)))
(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

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

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

- Returns value of final form, \(\langle \text{form} \rangle_m\)
  (Ignores other values)
PROGN Form

- (progn ⟨form\(_1\)⟩ ⟨form\(_2\)⟩ \ldots ⟨form\(_m\)⟩)
  Evaluates all forms, ⟨form\(_i\)⟩; \((i = 1..m)\)
in order.

- Returns value of final form, ⟨form\(_m\)⟩
  (Ignores other values)

- Takes ANY number of forms
LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

  \[(\text{LAMBDA} \ (a_1 \ldots \ a_n) \ \text{form}_1 \ldots \ \text{form}_m)\]
LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

  \[
  (\text{LAMBDA} \ (a_1 \ldots \ a_n) \ \text{form}_1 \ldots \ \text{form}_m)
  \]

- returns value of final form, \(\langle \text{form}\rangle_m\).
LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

  \[(\text{LAMBDA} \ (a_1 \ldots \ a_n) \ \text{form}_1 \ \ldots \ \text{form}_m)\]

- returns value of final form, \(\langle \text{form} \rangle_m\).

- Only makes sense if forms preceding last have meaningful side-effects

\[(\text{LAMBDA} \ (a) \ (\text{print} \ a) \ (\text{setq} \ x \ a) \ (+ \ a \ 3)) \ 19)\]
LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

\[
\text{(LAMBDA (a1 \ldots \.an) \ form1 \ldots \ form_m)}
\]

- returns value of final form, \(⟨\text{form}⟩_m\).

- Only makes sense if forms preceding last have meaningful side-effects

\[
\text{(LAMBDA (a) (print a) (setq x a) (+ a 3)) 19)
\]

19 ; printed by (print a)
→ 22 ; value of this form
LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:

\[
(LAMBDA (a_1 \ldots a_n) \text{\textlt{form\textcolor{red}{1}} \ldots \text{\textlt{form\textcolor{red}{m}}}})
\]

- returns value of final form, \(\text{\textlt{form\textcolor{red}{m}}}\).

- Only makes sense if forms preceding last have meaningful side-effects

\[
(LAMBDA (a) (\text{print\textcolor{red}{a}}) (\text{setq\textcolor{red}{x\ a}}) (+ a 3)) 19
\]

19 ; printed by (print\textcolor{red}{a})

\[\rightarrow\] 22 ; value of this form

x \rightarrow
Common Lisp permits multi-form bodies:

\[(\text{LAMBDA} \ (a_1 \ldots \ a_n) \ \text{form}_1 \ \ldots \ \text{form}_m)\]

returns value of final form, \(\langle\text{form}\rangle_m\).

Only makes sense if forms preceding last have meaningful side-effects

\[(\text{LAMBDA} \ (a) \ (\text{print} \ a) \ (\text{setq} \ x \ a) \ (+ \ a \ 3)) \ 19)\]

19 ; printed by \((\text{print} \ a)\)
→ 22 ; value of this form
\(x \rightarrow 19\) ; 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.

\[(\text{COND } (\langle q_1^1 \rangle \langle q_2^1 \rangle \cdots \langle q_{m_1}^1 \rangle )
\quad (\langle q_1^2 \rangle \langle q_2^2 \rangle \cdots \langle q_{m_2}^2 \rangle )
\quad \cdots
\quad (\langle q_1^n \rangle \langle q_2^n \rangle \cdots \langle q_{m_n}^n \rangle ))\]

where each \( \langle q \rangle^j \) is a form, \( m_i \in \mathbb{Z}^+ \).
The Truth about COND

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

\[
\text{(COND } (\langle q_1^1 \rangle \langle q_2^1 \rangle \ldots \langle q_{m1}^1 \rangle) \\
(\langle q_1^2 \rangle \langle q_2^2 \rangle \ldots \langle q_{m2}^2 \rangle) \\
\ldots \\
(\langle q_1^n \rangle \langle q_2^n \rangle \ldots \langle q_{mn}^n \rangle) )
\]

where each \( \langle q \rangle^i_j \) is a form, \( m_i \in \mathbb{Z}^+ \).

- If \( \langle q \rangle^i_1 \) is nonNIL,
  Then evaluate \( \langle q \rangle^i_j \) forms, for \( j = 2 .. m_i \).
The Truth about `COND`

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

\[
\text{(COND } \langle q^1_1 \rangle \langle q^1_2 \rangle \ldots \langle q^1_{m_1} \rangle \\
\langle q^2_1 \rangle \langle q^2_2 \rangle \ldots \langle q^2_{m_2} \rangle \\
\ldots \\
\langle q^n_1 \rangle \langle q^n_2 \rangle \ldots \langle q^n_{m_n} \rangle \text{ )}
\]

where each \( \langle q \rangle^i_j \) is a form, \( mi \in \mathbb{Z}^+ \).

- If \( \langle q \rangle^i_1 \) is nonNIL,
  Then evaluate \( \langle q \rangle^i_j \) forms, for \( j = 2 \ldots mi \).

- Return value for final form \( \langle q \rangle^i_{mi} \).
The Truth about COND

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

\[
\text{(COND } \quad \langle q_1^1 \rangle \langle q_2^1 \rangle \ldots \langle q_{m_1}^1 \rangle \\
\langle q_1^2 \rangle \langle q_2^2 \rangle \ldots \langle q_{m_2}^2 \rangle \\
\ldots \\
\langle q_1^n \rangle \langle q_2^n \rangle \ldots \langle q_{m_n}^n \rangle \text{)}
\]

where each \( \langle q \rangle^j \) is a form, \( m_i \in \mathbb{Z}^+ \).

- If \( \langle q \rangle^j_1 \) is nonNIL,
  Then evaluate \( \langle q \rangle^j \) forms, for \( j = 2 \ldots m_i \).

- Return value for final form \( \langle q \rangle^i_{m_i} \).

- If \( m_i = 1 \), and if \( \langle q \rangle^i_1 \) is nonNIL,
  then return \( \langle q \rangle^i_1 \)'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) →

(swp 18)→

(setq x nil)→

(swp 18) x was nil, is now 18; prints msg, and resets x

(x)→ 18; value that form returns.

(setq x nil)→

(swp nil)→ nil; COND fails.
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

(setq x 'fred)→fred

(swp 18)→fred

(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.
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)
  (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

```lisp
(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
  (swp 18) → fred

(fred) ; just prints out value of x.
  (setq x nil) → nil
  (swp 18) → fred
  (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.
(swp nil) → nil

(x was nil, is now 18) ; prints msg, and resets x
x → 18 ; value that form returns.
(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) \(\rightarrow\) fred

(swp 18)
\textit{fred}; just prints out value of x.

(setq x nil) \(\rightarrow\) nil

(swp 18)
\textit{x was nil, is now 18}; prints msg, and resets x
\(x \rightarrow 18\); value that form returns.

(setq x nil) \(\rightarrow\) nil

(swp nil) \(\rightarrow\) 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 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

\[
(\text{LOOP FOR } i \text{ FROM } 1 \text{ TO } 10 \text{ REPEAT } 5 \\
\quad \text{COLLECT } i)
\]
LOOP Construct by Simple Examples

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

Bob Price and Russ Greiner
CMPUT325 Extensions to Pure Lisp
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

\[
\text{(LOOP FOR} \ i \ \text{FROM} \ 1 \ \text{TO} \ 10 \ \text{REPEAT} \ 5 \\
\quad \text{COLLECT} \ i) \\
\rightarrow (1 \ 2 \ 3 \ 4 \ 5) \\
\text{(LOOP FOR} \ i \ \text{FROM} \ 10 \ \text{DOWNTO} \ 1 \ \text{COLLECT} \ i) \\
\rightarrow (10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1)
\]
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)
→

(10 8 6 4 2)
(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

\[
(\text{LOOP FOR } i \text{ FROM 10 DOWNTO 1 BY 2 COLLECT } i) \\
\rightarrow (10 \ 8 \ 6 \ 4 \ 2)
\]

\[
(\text{LOOP FOR } i \text{ IN } '(10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \text{ collect } i) \\
\rightarrow (10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1)
\]
LOOP Construct by Simple Examples

\[
(\text{LOOP FOR } i \text{ FROM 10 DOWNTO 1 BY 2 COLLECT } i) \\
\rightarrow (10 \ 8 \ 6 \ 4 \ 2)
\]

\[
(\text{LOOP FOR } i \text{ IN } '(10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \text{ collect } i) \\
\rightarrow (10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1)
\]

\[
(\text{LOOP FOR } i \text{ IN } '(10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \\
\text{ BY 'cddr collect } i) \\
\rightarrow 
\]
LOOP Construct by Simple Examples

\[(\text{LOOP FOR } i \text{ FROM 10 DOWNTO 1 BY 2 COLLECT } i) \rightarrow (10 \ 8 \ 6 \ 4 \ 2)\]

\[(\text{LOOP FOR } i \text{ IN } '(10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \text{ collect } i) \rightarrow (10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1)\]

\[(\text{LOOP FOR } i \text{ IN } '(10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1) \text{ BY 'cddr collect } i) \rightarrow (10 \ 8 \ 6 \ 4 \ 2)\]
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))

→

(10 1) (9 2) (8 3) (7 4) (6 5)
(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 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 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 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)
→

(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))
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
(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
(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; }
}
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));
 ...
}

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp

69
(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)
```