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, ...)

CMPUT325 Extensions to Pure Lisp

Bob Price and Russ Greiner

5th October 2004
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 $\tau$.

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

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

▷ Examples: Any form using only $+$, CAR, CONS, …
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

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

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

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

(SETF (aref A 0 0) 'q)
→ #2A((Q NIL) (NIL NIL))
Functions with Side Effects – SETF

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

Functions with Side Effects – SETF

\[
\begin{align*}
\text{(setf} & \quad \text{(symbol-function } '+) \\
& \quad (\text{symbol-function } '-)) \quad ;; \text{DON’T DO THIS!!}
\end{align*}
\]

\[
\begin{align*}
\text{(setf} & \quad \text{(symbol-function } '\text{bye}) \\
& \quad (\text{FUNCTION}\ \text{(LAMBDA} () "Not so quick bit brain!")))
\end{align*}
\]

Simultaneous assignment

\[
\begin{align*}
\text{(setf} & \quad a \ 1 \ b \ 2 \ c \ 3) \\
a & \rightarrow 1 \\
b & \rightarrow 2 \\
c & \rightarrow 3
\end{align*}
\]
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.

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

  \[
  \begin{align*}
  \text{(setq } x & \ 'b) \rightarrow B \ ;; \text{Not an error!} \\
  X & \rightarrow B \\
  B & \rightarrow 5 \ ;; \text{but } B\text{'s value unchanged.}
  \end{align*}
  \]

The SET Function

- **SET** DOES evaluate its first argument.

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

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

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

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

  \[
  \text{\texttt{(numberp 12)}} \rightarrow \text{\texttt{t}} \\
  \text{\texttt{(numberp 'a)}} \rightarrow \text{\texttt{nil}} \\
  \text{\texttt{(setq n 25)}} \\
  \text{\texttt{(numberp n)}} \rightarrow \text{\texttt{t}} ;; \text{\texttt{numberp evaluates its arguments}} \\
  \text{\texttt{(numberp '(1 2))}} \rightarrow \text{\texttt{nil}}
  \]

**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

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

▷ 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

\[
\text{#10r15 } \rightarrow 15 \quad \text{#2r1111 } \rightarrow 15 \quad \text{#3r120 } \rightarrow 15
\]

Ratios, Floats

▷ Exact ratios can be represented without roundoff error

\[
\text{(expt (}/ 2 \text{ 3}) 2) \rightarrow 4/9
\]

▷ 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

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

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

▷ Many Lisp functions will take complex arguments

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

\[
\pi \to 3.1415926535897932385L0
\]

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

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

Numerical Operations

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

\[
(+ 1 2 3 4 5 6 7 8 9 10) \to 55
\]

\[
(* 2 2 2) \to 8
\]

\[
(- 10 1 ) \to 9
\]

\[
(- 10 1 3) \to 6
\]

\[
(/ 12 3 4) \to 1
\]

▷ Binary Functions: MOD

\[
(\text{MOD 11 2}) \to 1
\]
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:** \(< > \geq \leq\)

- **Unary Predicates:** \text{ZEROP}

---

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., \text{eq}) vs. equal items (e.g., \text{equal})

- For efficiency use mathematical equality (e.g., \text{=})

- Numbers are atoms, but are not always eq of each other

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

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

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

▶ Eg:

\[(\text{name} . \text{(Bart Selman)})\]
\[(\text{hair} . \text{black})\]
\[(\text{children} . \text{((Mary Louise)}\]
\[
\text{(Jean Pierre) })\]
\[(\text{habits} . \text{nil}) )\]

Dotted-Pair

▶ **CONS** can really take ANY pair of S-expressions

▶ earlier, just dealt with atoms and lists

▶ Value of \((\text{CONS} \ s1 \ s2)\) is \((s1 . \ s2)\)
for any \(s1, s2 \in s\text{-expr}\)

\[(\text{cons} \ 'a \ 'b) \rightarrow (a . \ b)\]
\[(\text{cons} \ 4 \ '(a \ b \ c)) \rightarrow (4 . \ (a \ b \ c))\]
\[(\text{cons} \ '(t) \ '(a . \ b)) \rightarrow ((t) . \ (a . \ b))\]
\[(\text{cons} \ 4 \ '(a . \ b)) \rightarrow (4 . \ (a . \ b))\]

▶ Retrieving components

\[(\text{CAR} \ '(a . \ b)) \rightarrow a\]
\[(\text{CAR} \ (\text{CONS} \ 'a \ 'b)) \rightarrow a\]
\[(\text{CDR} \ '((t) . \ (a . \ b))) \rightarrow (a . \ b)\]
Notation

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

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

- Notice:
  When CONS's 2nd arg is list, just as before!

Dotted Pair – Internals

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

\((a . (b . c))\)

\[\text{foo} \quad \begin{array}{c}
  \bullet \\
  \text{a} \\
  \text{b} \\
  \bullet \\
  \text{c}
\end{array}\]
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.

  ```lisp
  (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

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

- A key pure list data structure:
  - New entries can "shadow" old entries (functional modification)
  - 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 \text{rassoc} function (reverse associate) returns an entry given a value

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

- The \text{acons} functions creates a new entry

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

- The \text{pairlis} function zips two lists together into a association list

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

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

\begin{equation}
\text{(some predicate sequence1 sequence2 ...)}
\end{equation}

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

\begin{equation}
\left(\text{(some #'(lambda (x y) (not (equal x y))) '(1 2 3) '(1 2 4))} \rightarrow T\right)
\end{equation}

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

\begin{equation}
\left(\text{(every #'(lambda (x y) (equal x y)) '(1 2 3) '(1 2 4))} \rightarrow \text{nil}\right)
\end{equation}

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

\begin{equation}
\text{(find item sequence)}
\end{equation}

;; recall any non-nil value is true
\begin{equation}
\text{(find 'a '(c b a))} \rightarrow \text{A}
\end{equation}
\begin{equation}
\text{(find 'a '(c b))} \rightarrow \text{nil}
\end{equation}

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

\begin{equation}
\text{(find-if #'oddp '(2 4 7 6 9))} \rightarrow 7
\end{equation}

▶ Many other functions

position, mismatch, substitute, remove, sort
General List Functions

▶ There is an overlap with assoc

\[
\text{assoc item list :test fn) = (find item list :test fn :key #'car)}\]

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp

31

General List Functions

▶ More list functions

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

▶ Destructive List Functions

\[
\text{(setf x '(1))}
\]
\[
\text{(push 2 x)}
\]
\[
\text{x →(2 1)}
\]
\[
\text{(pop x) → 2}
\]
\[
\text{x → (1)}
\]
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
- Have a state and are subject to side effects

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

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

- Use multiple value bind to catch both values

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

▶ Setting entries in a table

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

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

▶ Other useful hash table functions

(remhash 'calgary) ;; removes entry

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

Vectors and Arrays

▶ Construction and use of a vector

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

▶ Matricies

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

- DEFSTRUCT defines methods for creating and accessing elements of a new structure
  
  (DEFSTRUCT course name room time)

- 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 cmput325) → "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 is a method
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: Inheritance

- Inheritance

(defclass circle (shape)
    ((center :accessor center
           :initarg :center
           :initform (list 0 0))
     (radius :accessor radius
           :initarg :r
           :initform 1)))
Methods defined through generic functions with typed arguments

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

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

Can be constructed as constants or dynamically

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

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

- `(read stream)` — reads an s-expr from stream
- `(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
- 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
- Writes one complete s-expr at a time
- Use `t` for the console stream or leave out stream

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

**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
(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)
BETTY SAL MARGARET
JUNE SANDY MAY

Bob Price and Russ Greiner

CMPUT325 Extensions to Pure Lisp

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 ~\n" '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

- 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 whereupon the user can use :bt to examine the backtrace leading to the bug

Basic IO

- (terpri) – flushes buffer, prints carriage return

- (load ⟨file⟩) – loads file named ⟨file⟩.
VT100 Console Tricks

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

(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, \(\langle \text{form} \rangle_i\) \((i = 1 \ldots m)\)
in order.

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

- Takes ANY number of forms

LAMBDA Form with Side-Effects

- Common Lisp permits multi-form bodies:
  
  \((\text{LAMBDA} (a_1 \ldots a_n) \text{form}_1 \ldots \text{form}_m)\)

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

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

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

19 \quad \text{; printed by} \ \langle \text{print} \ a \rangle

\rightarrow 22 \quad \text{; value of this form}

x \rightarrow 19 \quad \text{; side effect causes new} \ x \ \text{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 \rangle_1^1 \langle q \rangle_2^1 \cdots \langle q \rangle_{m_1}^1)
\quad (\langle q \rangle_1^2 \langle q \rangle_2^2 \cdots \langle q \rangle_{m_2}^2)
\quad \cdots
\quad (\langle q \rangle_1^n \langle q \rangle_2^n \cdots \langle q \rangle_{m_n}^n))\)

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 \(j = 2 .. m_i\).

- 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

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

(fred) → fred
(swp 18)

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

x was nil, is now 18 ; prints msg, and resets x
x → 18 ; value that form returns.
(setq x nil) → nil
(swp nil) → nil ; COND fails.
LOOP Construct by Simple Examples

(LOOP FOR i FROM 1 TO 10
   DO (FORMAT t "~s " i))
→ 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 FOR i FROM 1 TO 10 REPEAT 5
   COLLECT i)
→ (1 2 3 4 5)
(LOOP FOR i FROM 10 DOWNTO 1 COLLECT i)
→ (10 9 8 7 6 5 4 3 2 1)
(LOOP 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  
    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)
→ (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)))

▷ 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
Compilation GCL

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

Declarations GCL

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

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

(disassemble 'add) →
static L1(){ ...
{
  int V1; int V2;
  V1=fix(base[0]);
  V2=fix(base[1]);
  ...
  base[2]= CMPmake_fixnum((V1)+(V2));
  ...
}}

Compilation: CLISP PPC

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

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

(in-package "TK")
(tkconnect)
(button '.hello :text "Hello World"
       :command '(print "hi"))

=> .HELLO

(pack '.hello)