Debugging in LISP

✔ `trace` causes a trace to be printed for a function when it is called

```
;;; a function that works like reverse
(defun rev (list)
    (cons (first (last list)) (rev (butlast list))))


USER: (trace rev) ; note trace is a macro
(REV)

USER: (rev '(a b c d e))

0: (REV (A B C D E))
 1: (REV (A B C D))
   2: (REV (A B C))
    3: (REV (A B))
      4: (REV (A))
       5: (REV NIL)
         6: (REV NIL)
          7: (REV NIL)
            8: (REV NIL)
             9: (REV NIL)
               10: (REV NIL)
               ...
               ...
```

Debugging in LISP

✔ tracing

```
;;; a debugged function that works like reverse
(defun rev (list)
 (if (endp list) NIL
    (cons (first (last list)) (rev (butlast list)))))



USER: (rev '(a b c d e))

0: (REV (A B C D E))
 1: (REV (A B C D))
   2: (REV (A B C))
    3: (REV (A B))
      4: (REV (A))
       5: (REV NIL)
       5: returned NIL
      4: returned (A)
    3: returned (B A)
   2: returned (C B A)
 1: returned (D C B A)
0: returned (E D C B A)

(E D C B A)
```

Debugging in LISP

- ✔ several functions can be traced at once

```
(defun rev (list)
    (rev-aux list nil))

(defun rev-aux (list acc)
    (if (endp list) acc
        (rev-aux (rest list) (cons (first list) acc))))


USER: (trace rev-aux)
(REV-AUX)

USER: (trace) ;with no args, trace lists all traced fns
(REV REV-AUX)

USER: (rev '(a b c))
0: (REV (A B C))
 1: (REV-AUX (A B C) NIL)
   2: (REV-AUX (B C) (A))
    3: (REV-AUX (C) (B A))
      4: (REV-AUX NIL (C B A))
      4: returned (C B A)
    3: returned (C B A)
   2: returned (C B A)
 1: returned (C B A)
0: returned (C B A)

(C B A)
```

A note on efficiency

- `rev-aux` is an example of a *tail-recursive* function
- A tail recursive function calls itself and returns the result without doing any additional computation on it
- The trace of `rev-aux` shows that many stack operations could be eliminated because of this
- A clever interpreter or compiler can easily translate a tail-recursive function into an iterative loop for efficiency
- Writing tail-recursive "aux"-type functions with an additional accumulator argument can lead to more efficient (though perhaps less elegant) programs
- Another example:

```
; a function that works like length (not tail recursive)
(defun len (list)
    (if (endp list) 0
        (+ 1 (len (rest list)))))


; a function that works like length
(defun len (list) (len-aux list 0))

; len-aux is tail recursive
(defun len-aux (list acc)
    (if (endp list) acc
        (len-aux (rest list) (+ 1 acc))))
```

Debugging in LISP

- ✔ To stop tracing a particular function, use `untrace`

```
USER: (untrace rev)
(REV)
```

- ✔ To stop tracing all functions, use `untrace` without any arguments

```
USER: (untrace)
NIL

USER: (trace)
NIL
```

Debugging in LISP

- ✔ `step` causes the LISP evaluation process to be stepped through

```
USER: (step (rev '(a b c)))
 1: (REV '(A B C))
[step] USER: <cr>
 2: '(A B C) => (A B C)
 2: (BLOCK REV (REV-AUX LIST NIL))
[step] USER: <cr>
 3: (REV-AUX LIST NIL)
[step] USER: <cr>
 4: LIST => (A B C)
 4: NIL => NIL
 4: T => T
 4: (AND (CONSP EXCL::X)
          (LET ((EXCL::X #)) (MEMBER EXCL:: X `'...)))

[step] USER: :reset

USER:
```

- ✔ `step` may not be very useful

Debugging in LISP

- ✔ `break` causes a computation to be interrupted at a particular point
- ✔ the value of local variables can be inspected during the break
- ✔ the runtime stack of function calls can be inspected during the break
- ✔ insert the expression `(break “message”)` where you want the break to occur
- ✔ the `“message”` will be printed when the break occurs

```
(defun rev-aux (list acc)
    (when (= (length list) 2) (break “length is 2”))
    (if (endp list) acc
     (rev-aux (rest list) (cons (first list) acc))))
```

Debugging in LISP

✔ using `break`

```
USER: (rev '(a b c))

Break: length is 2

Restart actions (select using :continue):
 0: return from break.

[1c] USER: :local list
(B C)
[1c] USER: :local acc
(A)
[1c] USER: :zoom
Evaluation stack:
 (BREAK "length is 2")
 ->(PROGN (BREAK "length is 2"))
 (IF (= # 2) (PROGN #) ...)
 (REV-AUX (B C) (A))
 (IF (ENDP LIST) ACC ...)
 (REV-AUX (A B C) NIL)
 (REV (A B C))
 (EVAL (REV '#))
 (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)

[1c] USER: :continue

(C B A)
```

Debugging in LISP

- Many top-level commands are available in Allegro Common LISP
- :reset, :continue, :local, :zoom are the most commonly used
- Use :help to see them all

```
USER: :help

COMMAND ABBR DESCRIPTION
aliases ali print all command aliases
args arg save arguments before calls
arrest arr arrest a process for debugging
boe Mark frame to break when exitted.
bottom bo Zoom at the oldest frame on the stack.
bt Zoom in a very brief manner.
cd change into another directory
cf compile a file
cload cl compile and load a file
continue cont continue from a continuable error
current cur return the expression given by the current stack
frame
dirs di print the Allegro directory stack
dn move down the stack 'n' frames, default 1
edit ed edit the source for the current stack frame
EOF either :pop or :exit
error err print the last error message
evalmode eval examine or set evaluation mode
exit ex exit and return to the shell
find fin find the stack frame calling the function 'func'
focus fo focus the top level on a process
frame fr print info about current frame
...
...
...
```

Debugging in LISP

✔ `time` is useful for checking efficiency of functions

```
USER: (time (rev '(a b c d e f g h i j k l m n o p)))

cpu time (non-gc) 16 msec user, 0 msec system
cpu time (gc) 0 msec user, 0 msec system
cpu time (total) 16 msec user, 0 msec system
real time 20 msec
space allocation:
 238 cons cells, 0 symbols, 608 other bytes,

(P O N M L K J I H G ...)
```

✔ `describe` is useful for finding information about LISP objects

```
USER: (describe 'rev)

REV is a SYMBOL
 It is unbound.
 It is INTERNAL in the COMMON-LISP-USER package.
 Its function binding is #<Interpreted Function REV @#0f2e>
 The function takes arguments (LIST)
```

Iteration in LISP

- ✔ A language with recursion and conditionals is computationally universal, but...
- ✔ Iteration is natural for some problems
- ✔ Iteration may be more efficient than recursion in some implementations
- ✔ Common LISP provides many iteration constructs

Iteration in LISP

- ✔ `dotimes` is used for counting-oriented iteration

```
(dotimes (var count result) expr1 expr2 ...)
```

  - ✗ *var* is a symbol that acts as an iteration variable
  - ✗ *count* is evaluated; it should evaluate to an integer
  - ✗ the expressions *expr1, expr2*, ... in the body of the `dotimes` are each evaluated with *var* bound to integers from 0 (inclusive) to *count* (exclusive)
  - ✗ *result* is evaluated and returned as the value of the `dotimes`
    - *result* is optional; if missing, `dotimes` returns `nil`

```
;;; raise m to the n power
;;; m and n must be integers, n non-negative
(defun dotimes-expt (m n)
    (let ((result 1))
        (dotimes (count n result)
            (setf result (* m result)))))
```

Iteration in LISP

✔ `dolist` is useful for list-oriented iteration

```
(dolist (var list result) expr1 expr2 ...)
```

- ✗ *`var`* is a symbol that acts as an iteration variable
- ✗ *`list`* is evaluated; it should evaluate to a list
- ✗ the expressions *`expr1, expr2`*, ... in the body of the `dolist` are each evaluated with *`var`* bound to the elements of *`list`*, in order
- ✗ *`result`* is evaluated and returned as the value of the `dolist`
    - *`result`* is optional; if missing, `dolist` returns `nil`

```
;;; return the length of a list
(defun dolist-length (list)
    (let ((result 0))
        (dolist (item list result)
            (setf result (+ 1 result)))))
```

Control strategies in LISP: Recursion vs. Mapping vs. Iteration

- In general: write code that is easy to understand, while meeting requirements for efficiency, if any

- When writing a mathematical function, if the *definition* of the function suggests a control strategy, then using that strategy probably makes sense

- If solving a problem requires looking at nested list structure, then recursion probably makes sense

- If solving a problem involves transforming a list into a new list, then mapping probably makes sense

- If solving a problem calls for doing something over and over (and over), then iteration probably makes sense