Lisp and POP11

```
(defun add_nums (inlist)
  (let ((total 0) item)
   (dolist (item inlist)
    (setq total (+ item total))
   )
   total
  )
)
== (add_nums '(1 2 3 4 5 6))
21
== (add_nums ())
0
```

```
define add_nums(inlist) -> total;
    lvars total= 0, item;
    for item in inlist do
        item + total -> total
    endfor
enddefine;


: add_nums([1 2 3 4 5 6]) =>
** 21
: add_nums([]) =>
** 0
```

```
(defun my_member (item list)
 (cond ((null list)  nil)
  ((equal item (car list)) t)
  (t (my_member item (cdr list))))
 )
)


== (my_member 2 '(1 2 3 4 5))
T
== (my_member 2 '(1 3 4 5))
NIL
```

```
define my_member(item, list)
                          -> result;
  if list = []
  then false -> result
  elseif item = hd(list)
  then true -> result
  else my_member(item, tl(list))
         -> result
  endif
enddefine;
: my_member(2, [1 2 3 4 5]) =>
** <true>
: my_member(2, []) =>
** <false>
```

The Jugs Problem — in LISP

- Not general purpose search
- Fixed sequence of possible actions on each cycle
- Goal is to get particular amount in either jug
- Procedure constructs a list of actions to achieve goal

(defun jugs)

```
(defun jugs (ina inb capa capb goal)
  (cond
    ((= ina goal) '((Correct amount in A)))
    ((= inb goal) '((Correct amount in B)))
    ((= ina capa) (cons '(Empty A) (jugs 0 inb capa capb goal)))
    ((= inb 0) (cons '(Fill B) (jugs ina capb capa capb goal)))
    ((> (- capa ina) inb)
                (cons '(Empty B into A)
                      (jugs (+ ina inb) 0 capa capb goal)))
    (t (cons '(Fill A from B)
             (jugs capa (- inb (- capa ina)) capa capb goal)))
  )
)
```

output.lsp

```
== (jugs 0 0 3 5 2)
((FILL B) (FILL A FROM B) (CORRECT AMOUNT IN B))
== (jugs 0 0 5 3 2)
((FILL B) (EMPTY B INTO A) (FILL B) (FILL A FROM B) (EMPTY A)
 (EMPTY B INTO A) (FILL B) (EMPTY B INTO A) (FILL B) (FILL A FROM B)
 (CORRECT AMOUNT IN B))
== (jugs 0 0 3 3 2)

; MISHAP - rle: RECURSION LIMIT (pop_callstack_lim) EXCEEDED
; DOING    :  (JUGS 0 0 ...)
```