CS **352** LECTURE NOTES

M 050103

Syllabus

```
(age 18
 name (sue smith)
 height (5 5)
 father (bill smith)
…
)
```

```
(how are you feeling)
ELIZA
(i feel sad)
(why are you sad)
(i
feel
sad)
```

Lots of
Irritating
Silly
Parentheses

LISTS
variable length
elements are not of uniform type
     (age 18)
recursive
     sublists
order is important
     (a b) (b a)
duplicates are allowed
     (2 2) ((a b)(a b))
access is from the front

W 050105

```
(the dog chased the cat)

(s (np (det the)
       (n dog))
   (vp (v chased)
       (np (det the)
```

```
           (n cat)))
)

Lisp has extensible syntax.
macro

how to get into Lisp
how to get out
how to handle errors

clisp

numbers evaluate to themselves

arity — how many arguments does a function take
nilary — (exit)
unary
binary
ternary

some functions have fixed arity
some functions have flexible arity

eval of a number

eval of a list
1 — the first element must be a function name
2 — is the number of arguments correct?
3 — evaluate the arguments recursively
4 — apply the function to the evaluated args

when EVAL sees a list,
it expects the first element
to be the name of a defined function
```

F 050105

```
normal functions evaluate their arguments

nested function calls

literal atoms — symbols
how long can they be?
what characters are allowed?

how to assign values to variables
```

how to define functions

```
(defun SQ (X) (* X X))

(defun THREE nil 3)

(defun F (X) (print 7) (* X X))
```

Function defs are not checked at define time.

```
(defun BAD1 nil (/ 2 0))
(defun BAD2 nil (* XX YY))

(defun SQ1 (X) (1+ (SQ X)))
```

M 050110

Homework 1
Exercise 1
due F 050114

NIL is a list.

if O is an object & L is a list,
then (cons O L) is also a list.

compute the sum & product of two numbers

(SP 3 4) → (7 12)

```
(defun SP (X Y)
  (cons (+ X Y)
        (cons (* X Y) nil))
)

(defun SP (X Y)
  (list (+ X Y)
        (* X Y))
)

(defun SP (X Y)
  (list (+ X Y) (* X Y))
)
```

Homework 2
Exercise 2

due W 050119

don't worry about errors
sqrt of a negative number
division by 0

W 050112

F 050114

null
atom
listp
numberp
floatp
evenp & oddp
minusp & plusp & zerop

relations
=
/=
>
>=
<
<=

beware of = on floating-point numbers

member — semi-predicate
(member 'a '(a b c)) → (a b c)
(member 'b '(a b c)) → (b c)
(member 'c '(a b c)) → (c)
(member 'd '(a b c)) → nil
(member nil '(a b c)) → nil
(member 'b '(a (b) c)) → nil

(not 'a) → nil
(not nil) → t
looks for nil & returns t if it finds it
otherwise returns nil

x	(not x)
t	nil
nil	t
'a	nil

```
(and t t) → t
(and t nil) → nil
(and nil t) → nil
(and nil nil) → nil

(and) → t
identity element for AND

(and nil FOO) → nil
(and FOO nil) → error

short-circuit
Java &&, &

(or t t) → t
…

(or) → nil

(or t FOO) → t
(or FOO t) → error

(or 3 5) → 3
```

Java multiway branching if (C1) A1; else if (C2) A2; else if (C3) A3; … else if (Cn) An; else A;	Lisp (cond (C1 A1) (C2 A2) (C3 A3) … (Cn An) (t A))

```
> 80 hot
< 70 cold
else ok

(defun TWAIN (TEMP)
  (cond
    ((> TEMP 80) 'HOT)
    ((< TEMP 70) 'COLD)
    (t           'OK)
```

```
  )
)

avoid nested conds

AVOID
(cond
  ((numberp x) … )
  ((not (numberp x)) … )

(cond
  ((numberp x) … )
  (t           … )

AVOID
((eq TEMP t) …)
instead write
(TEMP … )

if ( TEMP == true )
if ( TEMP )

(eq TEMP nil)
(not TEMP)

;
; This is a comment.
;

;
; (DISC A B C) computes the discriminant
; for the quadratic equation
;
(defun DISC (A B C)
  …
)
```

W 050119

```
(defun QUAD1 (A B C)
  (/ (+ (- B) (sqrt (- (* B B) (* 4 A C))))
     (* 2 A))
)

(defun DIAGNOSE (S1 S2)
  (cond
    ((and S1 S2) 'D1)
```

```
    (S1          'D2)
    (S2          'D3)
    (t           'D4)
  )
)
```

F 050121

```
(defun F nil (F))

(defun G (N) (print N) (G (1+ N)))

(defun FAC (N)
  (if (zerop N) 1
                (* N (FAC (1- N)))
  )
)

(defun FIB (N)
  (cond
    ((zerop N) 1)
    ((= N 1)  1)
    (t         (+ (FIB (1- N)) (FIB (- N 2))))
  )
)

(defun SUM (L)
  (cond
    ((null L) 0)
    (t        (+ (car L) (SUM (cdr L))))
  )
)

(defun LEN (L)
  (cond
    ((null L) 0)
    (t        (+ 1 (LEN (cdr L))))
  )
)

(TOTAL '((sam 5)(sue 6)(sally 4))) → 15

(defun TOTAL (L)
  (cond
    ((null L) 0)
    (t (+ (cadar L) (TOTAL (cdr L))))
)
```

M 050124

```
(defun COUNT-EVEN (L)
  (cond
    ((null L)        0)
    ((evenp (car L)) (+ 1 (COUNT-EVEN (cdr L))))
    (t               (COUNT-EVEN (cdr L)))
  )
)

(defun SSO (L)
  (cond
    ((null L)       0)
    ((oddp (car L)) (+ (* (car L) (car L))
                       (SSO (cdr L))))
    (t              (SSO (cdr L)))
  )
)

(defun SSO (L)
  (cond
    ((null L)        0)
    ((evenp (car L)) (SSO (cdr L)))
    (t               (+ (* (car L) (car L))
                        (SSO (cdr L))))
  )
)
```

So far, we've looked at every element in the list.

```
(ALL-ODD nil) → t
(ALL-ODD '(1 3 5)) → t
(ALL-ODD '(2 4)) → nil
(ALL-ODD '(3 1 4 1)) → nil

(ALL-ODD '(4 1 3 1))
nil

(ALL-ODD '(1 3 5))
(ALL-ODD '(3 5))
(ALL-ODD '(5))
(ALL-ODD nil)
t

(ANY-ODD nil) → nil
```

```
(ANY-ODD '(1 3 5)) → t
(ANY-ODD '(2 4)) → nil
(ANY-ODD '(3 1 4 1)) → t

(ANY-ODD '(3 1 4 1))
t

(ANY-ODD '(2 4 1))
(ANY-ODD '(4 1))
(ANY-ODD '(1))
t

(ANY-ODD '(2 4))
(ANY-ODD '(4))
(ANY-ODD nil)
nil

(defun ALL-ODD (L)
  (cond
    ((null L)       t)
    ((oddp (car L)) (ALL-ODD (cdr L))
    (t              nil)
  )
)

(defun ANY-ODD (L)
  (cond
    ((null L)       nil)
    ((oddp (car L)) t)
    (t              (ANY-ODD (cdr L)))
  )
)
```

```
OPERATE-ALL
(INC-ALL '(3 1 4)) → (4 2 5)
(SQ-ALL '(3 1 4)) → (9 1 16)
(DEC-ALL '(3 1 4)) → (2 0 3)

(INC-ALL nil) → nil

(defun INC-ALL (L)
  (cond
    ((null L) nil)
    (t        (cons (1+ (car L)) (INC-ALL (cdr L))))
  )
)
```

```
(INC-ALL '(2 7))
(cons 3 (INC-ALL '(7))
(cons 3 (cons 8 (INC-ALL nil)))
(cons 3 (cons 8 nil))
(cons 3 '(8))
(3 8)
```

OPERATE-SOME

```
(INC-ODD nil) → nil
(INC-ODD '(3 1 4 1)) → (4 2 4 2)
(INC-ODD '(1 3 5)) → (2 4 6)
(INC-ODD '(2 4 6)) → (2 4 6)

(defun INC-ODD (L)
  (cond
```

W 050126

continue with INC-ODD …

```
(defun INC-ALL (L)
  (cond
    ((null L) nil)
    (t        (cons (1+ (car L)) (INC-ALL (cdr L)))
  )
)
```

OPERATE-SOME

```
(defun INC-ODD (L)
  (cond
    ((null L) nil)
    ((oddp (car L)) (cons (1+ (car L)) (INC-ODD (cdr L))))
    (t              (cons (car L) (INC-ODD (cdr L))))
  )
)
```

DOUBLE-ODD
SQUARE-ODD

KEEP-SOME

Given a list of integers,
return me a new list keeping only the odd integers.

```
(defun KEEP-ODD (L)
  (cond
    ((null L)       nil)
    ((oddp (car L)) (cons (car L) (KEEP-ODD (cdr L))))
    (t              (KEEP-ODD (cdr L)))
  )
)

(defun DELETE-ODD (L)
  (cond
    ((null L) nil)
    ((oddp (car L)) (DELETE-ODD (cdr L)))
    (t              (cons (car L) (DELETE-ODD (cdr L))))
  )
)
```

Homework 4
Exercise 6
due M 1/31

Each group:
write a problem for each of 3 schemata.

F 050128

Examples help to define a problem.

(AL1 '(1) '(2 3 4)) → (3 3 4)

DMAS – Doesn't Match A Schema

```
;
; (AL L1 L2) takes two same-length lists of integers
;            and returns a same-length list where each
;            element is the sum of the two corresponding
;            elements in the input lists.
;
```

(AL nil nil) → nil
(AL '(3 1 4) '(2 7 1)) → (5 8 5)

```
(defun AL (L1 L2)
  (cond
    ((null L1) nil)
```

```
    (t         (cons (+ (car L1) (car L2))
                     (AL (cdr L1) (cdr L2))))
  )
)

(APP nil nil) → nil
(APP '(3 1 4) nil) → (3 1 4)
(APP nil '(2 7 1)) → (2 7 1)

(APP '(3 1) '(2 7 1))
(cons 3 (APP '(1) '(2 7 1)))
(cons 3 (cons 1 (APP nil '(2 7 1))))
(cons 3 (cons 1 '(2 7 1)))
(cons 3 '(1 2 7 1))
(3 1 2 7 1)

(defun APP (L1 L2)
  (cond
    ((null L1) L2)
    ((null L2) L1)
    (t (cons (car L1) (APP (cdr L1) L2)))
  )
)
```

```
(INTERSECT '(4 1 2) '(5 3 1))
(INTERSECT '(1 2) '(5 3 1))
(cons 1 (INTERSECT '(2) '(5 3 1)))
(cons 1 (INTERSECT nil '(5 3 1)))
(cons 1 nil)
(1)

(defun INTERSECT (L1 L2)
  (cond
    ((null L1) nil)
    ((member (car L1) L2) (cons (car L1)
                                (INTERSECT (cdr L1) L2)))
    (t (INTERSECT (cdr L1) L2))
  )
)
```

```
;
; (SETP L) returns T iff L is a set
;
(defun SETP (L)
```

```
  (cond
    ((null L) t)
    ((member (car L) (cdr L)) nil)
    (t (SETP (cdr L)))
  )
)

(defun SETIFY (L)
  (cond
    ((null L) nil)
    ((member (car L) (cdr L)) (SETIFY (cdr L)))
    (t (cons (car L) (SETIFY (cdr L))))
  )
)

(defun SUBSETP (L1 L2)
```

```
(defun SORTEDP (L)
  (cond
    ((null L)             t)
    ((null (cdr L))       t)
    ((> (car L) (cadr L)) nil)
    (t                    (SORTEDP (cdr L)))
  )
)
```

M 050131

Quiz 4
2/10 – Mostly errors.

```
(defun SUM* (E)
  (cond
    ((numberp E) E)
    ((atom E)    0)
    (t           (+ (SUM* (car E))
                    (SUM* (cdr E))))
  )
)
```

Homework 5
Exercise 10
due W 050202

```
;equality of two tree structures
```

```
(EQ* 3 3) → t
(EQ* 3 4) → nil
(EQ* '(3 (4)) '(3 (4))) → t

(defun EQ* (E1 E2)
  (cond
    ((and (atom E1) (atom E2)) (eq E1 E2))
    ((or (atom E1) (atom E2)) nil)
    (t (and (EQ* (car E1) (car E2))
            (EQ* (cdr E2) (cdr E2))))
  )
)

(INC* nil) → nil
(INC* 5) → 6
(INC* '(3 (1 (4))(1))) → (4 (2 (5)) (2))

(defun INC* (E)
  (cond
    ((numberp E) (1+ E))
    ((atom E)    E)
    (t           (cons (INC* (car E))
                       (INC* (cdr E))))
  )
)

(defun ISORT1 (N L)
  (cond
    ((null L) (list N))
    ((<= N (car L)) (cons N L))
    (t (cons (car L)
             (ISORT1 N (cdr L))))
  )
)

(defun ISORT (L)
  (cond
    ((null L) nil)
    (t (ISORT1 (car L)
               (ISORT (cdr L))))
  )
)

(defun PREFIXP (L1 L2)
  (cond
```

```
    ((null L1) t)
    ((null L2) nil)
    ((eq (car L1) (car L2)) (PREFIXP (cdr L1) (cdr L2)))
    (t nil)
  )
)

SUBLISTP
```

W 050202

respond to the chair's email regarding summer courses

Exam 1
memorize your Student ID
closed book
closed notes
no calculators or computers
special assigned seating
soroka supplies paper
no cell phones or pagers
check additional rules in the syllabus

topics stop with car-cdr recursion — tree recursion

Homework 5

(COUNT# '(3 (1) (A (5)))) → 3

```
(COUNT# 3)          numberp
(COUNT# 'A)         atom
(COUNT# nil)
(COUNT# '(3 (4)))   car-cdr recursion

(defun COUNT# (E)
  (cond
    ((numberp E) 1)
    ((atom E)    0)
    (t           (+ (COUNT# (car E))
                    (COUNT# (cdr E))))
  )
)

(defun SUM (L) (SUM1 L 0))
(defun SUM1 (L N)
  (cond
```

```
    ((null L) N)
    (t (SUM1 (cdr L) (+ N (car L))))
  )
)

(defun FAC (N)
  (cond
    ((zerop N) 1)
    (t         (* N (FAC (1- N))))
  )
)

(defun FAC (N) (FAC1 N 1))
(defun FAC1 (N RESULT)
  (cond
    ((zerop N) RESULT)
    (t (FAC1 (1- N) (* RESULT N)))
  )
)
```

Compute the sum and product of the elements of a list.

```
(defun SP (L) (list (SUM L) (PROD L)))

(defun SUM (L)
  (cond
    ((null L) 0)
    (t (+ (car L) (SUM (cdr L))))
  )
)

(defun SP (L) (SP1 L 0 1))
(defun SP1 (L S P)
  (cond
    ((null L) (list S P))
    (t (SP1 (cdr L)
            (+ S (car L))
            (* P (car L))))
  )
)
```

Homework 6
Exercise 11
due W 2/9

```
introducing LET

(defun QUAD3 (A B C)
  (setf DISC (- (* B B) (* 4 A C)))

    (zerop DISC)
    (minusp DISC)

  …
)
```

DUGV – Don't Use Global Variables

---

writing the expression twice
computing the expression twice

```
(defun DISC (A B C) (- (* B B) (* 4 A C)))
(defun QUAD3 (A B C)
…
   ((zerop (DISC A B C)) …)
   ((minusp (DISC A B C)) …)
```

advantage – write the expression only once
disadvantage – it gets calculated several times

```
(defun QUAD3 (A B C)
  (QUAD3a A B C (- (* B B) (* 4 A C)))
)
(defun QUAD3a (A B C DISC)
…
   (zerop DISC) …
   (minusp DISC) …
```

advantage – write the expression only once
advantage – compute the expression only once

```
(defun QUAD3 (A B C)
  (let ((DISC (- (* B B) (* 4 A C)))

      inside here, DISC has a value

  ) ;end let
) ;end defun
```

```
;
; Prints I and I*I for I = 1,…,N
;
(defun TABLE (N)
```

```
  (do ((I 0 (1+ I)))
      ((> I N) nil)
    (format t "~s~10t~s~%" I (* I I))
  )
)
```

W 050209

Probably useful for Homework 8:

```
[6]> (setf x 678)
678
[7]> (length (format nil "~s" x))
3
[8]>
```

```
>(F '(A B C))
A
B
C
nil

>(F nil)
nil

(defun F (L)
  (do ((L L (cdr L)))
      ((null L) nil)
    (print (car L))
  )
)

(do ((A 3 B)
     (B 5 A))
    (nil nil)
  (format t "~s~t~s~%" A B)
)

(do ((X 0)
     (I 0 (1+ I)))
    ((> I 5) nil)
  (print I)
  (setf X I)
)
```

Homework 9
Exercise 13
due M 2/14

```
(defun FAC (N)
  (do ((RESULT 1 (* RESULT I))
       (I 1 (1+ I)))
      ((> I N) RESULT)
  )
)
```

There is no reason to have a LET directly outside a DO.

```
(let ((A 3))
  (do ((I 1 (1+ I)))
    …
  )
)

(do ((A 3)
     (I 1 (1+ I)))
   …
)
```

MACROs

```
COPY macro  DEST,SOURCE
     mov    ax,SOURCE
     mov    DEST,ax
     endm

we write:
     call SETUPX
     COPY Y,X
     add  Y,5

it gets expanded to:
     call SETUPX
     mov  ax,X
     mov  Y,ax
     add  Y,5
```

Macro lets you write something one way, and it gets converted into something else.

We might want to write:

```
(case N
  (2 …)
  (3 …)
  ((4 5 6) …)
  ((range 7 20) …)
  (t …)
)
```

We want this converted to:

```
(cond
  ((= N 2) …)
  ((= N 3) …)
  ((member N '(4 5 6)) …)
  ((and (>= N 7) (<= N 20)) …)
  (t …)
)
```

we write

```
(ZERO X)
```

Lisp converts it to:

```
(setf X 0)
```

```
(ZERO* X Y Z)
```

```
(progn (setf X 0)
       (setf Y 0)
       (setf Z 0))
```

```
(for I 1 10 (print I))
```

```
(do ((I 1 (1+ I)))
    ((> I 10) nil)
  (print I)
)
```

Macros allow us to extend the syntax of Lisp.

quasiquote

'(A B (list 3 5) C) → (A B (list 3 5) C)

`(A B (list 3 5) C) → (A B (list 3 5) C)

```
`(A B ,(list 3 5) C) → (A B (3 5) C)

`(A B ,@(list 3 5) C) → (A B 3 5 C)

, means "unquote"
,@ means "unquote-splice"

(defun LOVES (X Y)
  (list X 'LOVES Y)
)

(defun LOVES (X Y)
  `(,X LOVES ,Y)
)
```

F 050211

```
&rest
permits a function to have variable number of arguments

(defun FOO (A B &rest C)
  (print A)
  (print B)
  (print C)
  nil
)

(for I 1 10 (print I) (print (* I I)))
(for I 3 N (print I))
(for I 1 1000000)
```

How macros work:

1. takes unevaluated arguments & generates a result
2. that result is substituted for the invoking form

```
A macro doesn't do the work.
It generates the code which will do the work.

(INC X)
(setf X (1+ X))

(INC* X Y Z)
(progn (setf X (1+ X))
       (setf Y (1+ Y))
       (setf Z (1+ Z)))
```

Steps in writing a Macro:

1. What is the input syntax?
2. What is the desired output?
3. Write code which produces the output from the input.

M 050214

```
(defun SUM (LO HI)
  (do ((I LO (1+ I))
       (RESULT 0))
      ((> I HI) RESULT)
    (setf RESULT (+ RESULT I))
  )
)

(defun SUM (LO HI)
  (do ((I LO (1+ I))
       (RESULT 0 (+ RESULT I))
      ((> I HI) RESULT)
  )
)
```

TA — You compute this, but you throw it away.

```
(+ RESULT I)
```

2/10 — mostly errors

ANNOUNCE

```
(announce X)

(format t "X = ~s~%" X)

(defmacro ANNOUNCE (VAR)
  `(format t
           ,(concatenate 'string
                         (string VAR)
                         " =  ~s~%")
           ,VAR)
)

(INC X)
(setf X (1+ X))

(defmacro INC (S)
  `(setf ,S (1+ ,S))
```

```
)
```

A more complicated INC macro

in	out
(INC X)	(setf X (+ X 1))
(INC X 2.4)	(setf X (+ X 2.4))
(INC X Y)	(setf X (+ X Y))
(INC X (* Z Z))	(setf X (+ X (* Z Z)))

```
(defmacro INC (VAR &rest ARGS)
  (let ((AMT (if ARGS (car ARGS) 1)))
    `(setf ,VAR (+ ,VAR ,AMT))
  )
)
```

Homework 10
due M 2/21
Exercise 15

```
(for J M (* N N) (print (* 2 J)))(

(defun FAC (N)
  (let ((RESULT 1))
    (for I 1 N (setf RESULT (* I RESULT)))
  )
)

(for I 1 10 (print I) (print (* I I)))

(do ((I 1 (1+ I)))
    ((> I 10) nil)
  (print I)
  (print (* I I))
)

(defmacro FOR (VAR LO HI &rest BODY)
  `(do ((,VAR ,LO (1+ ,VAR)))
       ((> ,VAR ,HI) nil)
     ,@BODY
   )
)
```

functions as arguments

car → error
#'car → gets us a pointer to the function itself

```
(apply #'cons '(a b)) → (a . b)

(funcall #'cons 'a 'b) → (a . b)
```

W 050216

```
(defun NONE (L)
  (do ((L L (cdr L))
       (NO 0)
       (NE 0))
      ((null L) (list NO NE))
    (if (oddp (car L)) (setf NO (1+ NO))
                       (setf NE (1+ NE)))
  )
)

ARITH

(arith)

(i0) 9
(o0) 9

(i1) 8
(o1) 8

(i2) (* i0 i1)
(o2) 72

(i3) i0
(o3) 9

(o4) exit

history
```

```
every
some

(every #'odd '(1 3 5 7)) → t

; (PERFSQ N) returns t iff N is a perfect square
;
(defun PERFSQ (N)
```

```
  (let ((ROOT (round (sqrt N))))
    (= N (* ROOT ROOT))
  )
)

(defun PERFSQ (N) (not (floatp (sqrt N))))

(defun ALL (PRED L)
  (cond
    ((null L) t)
    ((funcall PRED (car L)) (ALL PRED (cdr L)))
    (t nil)
  )
)

(ALL #'perfsq '(1 4 9 16))

ACCUMULATE-ALL

LENGTH
SUM
SUMSQ
PROD
```

function	BASE-VAL	ACC-OP	CAR-FN
LEN	0	#'+	#'ONE
SUM	0	#'+	#'ID
PROD	1	#'*	#'ID
SUMSQ	0	#'+	#'SQ

```
(defun ONE (X) 1)
(defun ID (X) X)
(defun SQ (X) (* X X))

super-function

(defun LEN (L) (ACC-ALL 0 #'+ #'ONE L))
(defun SUM (L) (ACC-ALL 0 #'+ #'ID L))
(defun PROD (L) (ACC-ALL 1 #'* #'ID L))
(defun SUMSQ (L) (ACC-ALL 0 #'+ #'SQ L))

(defun ACC-ALL (BASE-VAL ACC-OP CAR-FN L)
  (cond
    ((null L) BASE-VAL)
    (t (funcall ACC-OP
                (funcall CAR-FN (car L))
```

```
                (ACC-ALL BASE-VAL ACC-OP CAR-FN (cdr L))))
  )
)
```

Homework 12
Exercise 17
due M 2/28

M 050221

no class on W 2/23 and F 2/25

office hour today
2-3 pm

homework 8

M 050228

Homework 13
Exercise 15a
due F 3/4

head
tail

W 050302

allbits([]) → Yes
allbits([1,0,1,1]) → Yes
allbits([1,0,2,1]) → No

```
bit(0).
bit(1).

allbits([]).
allbits([Head|Tail]) :- bit(Head),allbits(Tail).


bit(B) :- mem(B,[0,1]).

digit(0).
…
digit(9).

digit(D) :- mem(D,[0,1,2,3,4,5,6,7,8,9]).
```

precedes(a,b,[c,a,b,d]) → Yes
precedes(a,b,[x,y,z]) → No
precedes(a,b,[c,b,a,d]) → No
precedes(a,b,[c,a,d,b]) → No
precedes(a,b,[]) → No
precedes(a,b,[a]) → No
precedes([a],b,[b,[a],b,y]) → Yes

F 050304

```
precedes(X,Y,[X,Y|_]).
precedes(X,Y,[_|Tail]) :- precedes(X,Y,Tail).

precedes(X,Y,[X,Y]).  ← unnecessary
```

Find X and Y in order but separated by an arbitrary number of elements:

```
f(X,Y,[X|T]) :- g(Y,T).
f(X,Y,[_|T]) :- f(X,Y,T).
g(Y,[Y|_]).
g(Y,[_|T]) :- g(Y,T).
```

Homework 14
Exercise 22
due W 3/9

Homework 15
Exercise 23
due W 3/9

M 050307