Поиск в глубину на Плэнере:

```
[define SEARCH_DEEP_DOWN (lambda (St)
        [prog (Dst Reslist)
                [set Dst .St]
                [set Reslist (.St)]
                S       [set Dst [among [OPENING .Dst]]]
                        [set Reslist (!.Reslist .Dst)]
                        [cond ( [IS_GOAL .Dst] [return .Reslist])
                        ( t [go S])] ] ) ]
```

Ограниченный поиск в глубину на Плэнере:

```
[define SEARCH_DEEP_DOWN (lambda (St, N)
        [prog (Dst Reslist Dep)
                [set Dst .St]
                [set Reslist (.St)]
                [set Dep 0]
                S       [set Dst [among [OPENING .Dst]]]
                        [set Reslist (!.Reslist .Dst)]
                        [cond ( [IS_GOAL .Dst] [return .Reslist])]
                        [cond ([< Dep N][[set Dep [+ .Dep 1]] [GO S]])
                        (T [fail])] ] ) ]
```

Напишите плэнер-фукнцию, считающую число фишек, стоящих не на своих местах в Игре-8.

```
[define calc1 (lambda (L)
[define diff (lambda ( L1 L2 )
[cond
([empty L1] [length L2])
([empty L2] [length L1])
([eq [elem 1 .L1] [elem 1 .L2]] [diff [rest 1 .L1] [rest 1 .L2]])
(t [+ [diff [rest 1 .L1] [rest 1 .L2]] 1])
])]
[diff L (1 2 3 4 # 5 6 7 8)] ) ]
```

(listp *l*) – функция-предикат, проверяющая **является ли значение ее аргумента списком (на верхнем уровне)**. Если да, то значение функции равно T, иначе – ().

```
(defun  listp  (lambda (x)
        (cond  ((null  x)  T)
                ((atom  x)  ( ))
                ( T  (listp  (cdr  x))) )))
```

(memb *a l*) - функция **ищет атом, являющийся значением первого ее аргумента, в списке (на верхнем его уровне), являющемся значением второго аргумента**. В случае успеха поиска значение функции равно T, иначе – ().

```
(defun  memb  (lambda  (a  l)
        (cond  ((null  l)  nil)
                ((eq  a (car  l))  T)
                (T  (memb  a  (cdr l)) ) ))
```

(out *a l*) - функция **удаляет из списка, являющегося значением ее второго аргумента, все вхождения (на верхнем) атома, являющегося значением первого аргумента**.

```
 (defun  out  (lambda  (a  l)
```

```
(cond ((null l) nil)
      ((eq a (car l)) (out a (cdr l)))
      (T (cons (car l) (out a (cdr l))))))
```

(equal e_1 e_2) – функция, **сравнивающая два произвольных S-выражения – значения своих аргументов**. Если они равны (имеют одинаковую структуру и состоят из одинаковых атомов), то значение функции равно T, иначе – ().

```
(defun equal (lambda (x y)
      (cond ((atom x) (eq x y))
            ((atom y) ( ))
            ((equal (car x) (car y)) (equal (cdr x) (cdr y)))
           ( T ( ) ) )))
```

Пусть задан список L положительных целых чисел. Нужно подобрать набор чисел из L (они могут повторяться), сумма которых равна заданному числу N.

```
[define sum (lambda (L N) [prog ( K (M ( )) (S 0))
    A     [set K [among .L]] [set M ( .K !.M)] [set s [+ .K .S]]
          [cond ([eq .S .N] .M]
                ([lt .S .N] [go A])
                (T [fail])] ])]
```

Если в описанное обращение к функции sum откуда-то извне «придет» неуспех, вычисление может возобновиться (при этом будут выбираться не исследованные ранее альтернативы). Например:

[prog (X) [set X [sum (6 3 2 1) 5]] [cond ([neq [length .X 3]] [fail])] .X] $\rightarrow$ (1 1 3).

Напечатать (поочередно) все решения рассматриваемой задачи можно с помощью такой конструкции:

```
[prog ( ) [alt ( ) [return T]]
          [print [sum (6 3 2 1) 5]] [fail]]
```

А собрать все решения (в списке Y) и затем выдать этот результат на печать можно так:

```
[prog (X (Y ( )) ) [alt ( ) [return .Y]]
                   [set X [sum (6 3 2 1) 5]]
                   [pset Y (!.Y .X)] [fail]]
```

Замена последнего элемента списка на пустой список.

<u>Лисп (рекурсивно):</u>

```
(defun LAST_NIL (l)
(cond
((eq (length l) 1) (quote (())))
((> (length l) 1) (cons (car l) (LAST_NIL (cdr l)))) ) )
```

<u>Плэнер (нерекурсивно):</u>

```
[define LAST_NIL (lambda (l)
[prog ((num 0) X)
[loop elem .l [set num [+ .num 1]]]
[cond
([eq .num 1] ((()))
([ge .num 2] [set X [head [- .num 1] .l]] (!.X ())) ] ] ]
```

Выделение предпоследнего элемента списка.

<u>Лисп (рекурсивно):</u>

```
(defun PRELAST (l)
(cond
((eq (length l) 2) (car l))
```

```
((> (length l) 2) (PRELAST (cdr l)))
) )
```

Плэнер(нерекурсивно):

```
[define prelast (lambda (lst)
(cond ([lt [length .lst] 2] ())
(T [elem -2 .lst]) ) ]

[define prelast (lambda (lst)
[prog ( (X ()) )
[is (<> *X <list 1>) .lst]
.X ] ]

[define prelast (lambda (lst)
 [prog (required i elem)
   [set required [- [length .lst] 1]]
   [set i 0]
   [set elem [among .lst]]
   [pset i [+ .i 1]]
   [cond ([eq .required .i] [return .elem])
         (T [fail])
   ] ] ) ]

[define prelast (lambda (lst)
 [prog (l i)
   [set required [- [length lst] 1]]
   [set i 0]
   [loop elem lst
     [set i [+ .i 1]]
     [cond ([eq .i .required] [return .elem])]
   ] ] ) ]
```

Удалить из списка все элементы кроме первого и последнего (Лисп):

```
(defun FIRSTLAST (l)
(cond
((> (length l) 1) ((car l) (last l)))
((eq (length l) 1) l) ) )
```

Удаление из списка всех чисел на самом верхнем уровне (Лисп):

```
(defun DELNUM (l)
(cond
((null l) nil)
((numberp (car l)) (DELNUM (cdr l)))
(T (cons (car l) (DELNUM (cdr l) ) ) ) ) )
```

Удаление всех чисел в списке на всех уровнях (Лисп):

```
(defun DELALLNUM (l)
(prog (i X Y)
(setq X (DELNUM l))
(setq i 0)
```

```
beg (cond
( (< i (length X))
(cond
( (listp (nth (+ i 1) X))
(setq Y (append Y ((DELALLNUM (nth (+ i 1) X))))))
(T (setq Y (append Y (nth (+ i 1) X)))) )
(setq i (+ i 1))
(go beg) ) (T Y) ) ) )
```

Beg5 – пятый от начала элемент списка

```
( defun Beg5( lambda( L ) caddddr L  ) )
[ define Beg5 ( lambda( L ) [ elem 5 .L ] ) ]
```

End2 – предпоследний элемент

```
( defun End2( lambda( L ) (cond ( ( eq length L 2 ) car L ) ( T ( End2 cdr L ))  ) )
[ define End2 ( lambda( L ) [ elem -2 .L ] ) ]
```

Напишите плэнер-фукнцию, считающую число фишек, стоящих не на своих местах в Игре-8.

```
[define calc1 (lambda (L)
[define diff (lambda ( L1 L2 )
[cond
([empty L1] [length L2])
([empty L2] [length L1])
([eq [elem 1 .L1] [elem 1 .L2]] [diff [rest 1 .L1] [rest 1 .L2]])
(t [+ [diff [rest 1 .L1] [rest 1 .L2]] 1])
])]
[diff L (1 2 3 4 # 5 6 7 8)]
)]
```

(PROG (CONS) (SETQ CONS (QUOTE (QUOTE (A L))))
(EVAL (LIST (QUOTE CONS) CONS CONS)))

Ответ: ((A L) A L)

1. PROG вводит переменную CONS = ()
2. (quote (A L)) = (A L)
3. (quote (quote (A L))) = '(A L)
4. (SETQ CONS (QUOTE (QUOTE (A L)))) -> CONS = '(A L)
5. (LIST (QUOTE CONS) CONS CONS) = (CONS, '(A L), '(A L))
6. (EVAL (LIST (QUOTE CONS) CONS CONS))) = ((A L) A L)

```
(defun FFF (lambda (x)
 (cond ((null x) T )
     ((atom x) ())
     (T (FFF (cdr x))) )))
```

– проверка список подали на вход или нет

Не рекурсивный вариант:

```
(list x)
```

Найти 4 различные смысловые интерпретации фразы "За безбилетный проезд и провоз одного места багажа взимается штраф 10000 рублей". Найденные интерпретации записать в форме продукций.

А) True: проезд безбилетный & провоз одного места багажа безбилетный -> взять штраф 10000 рублей

Б) True: проезд безбилетный & провоз одного места багажа -> взять штраф 10000 рублей

В) True: проезд безбилетный | провоз одного места багажа безбилетный -> взять штраф 10000 рублей

Г) True: проезд безбилетный | провоз одного места багажа -> взять штраф 10000 рублей

(.X .Y) = list (X Y)
(!.X .Y) = (append X (list Y)) \ (append X quote (Y))
(.X !.Y) = (cons X Y)
(!.X !.Y) = (append X Y)
([.X 1] [.X 5] [.X 3]) = (list (car X) (caddddr X) (caddr X)) \ (list (nth 1 X)(nth 5 X)(nth 3 X))

(cons (car (quote (1))) (cdr (quote (2 3)))) - (1 3) \
(car (quote (cons (car (quote (1))) (cdr (quote (2 3))))))) – cons
[is (1 *X 3) (1 2 3)] - T, побочный эффект X = 2
[is (1 !*X 3) (1 2 3)] - T, побочный эффект X = (2)
[is (1 !*X 2 <>) (1 2 3)] - T, побочный эффект X = ()
[is (1 !*X 2 []) (1 2 3)] - T \ X = ()

cons (quote (12))(list())-> ((12)())
car (quote (cons quote A quote(B)))-> (A B)
[is (* A <list 3> .A)(12321)]-> Результат T. Побочный эффект A=1.
[is (*A [list 3] .A)(12321)]-> Результат ().
[is (*A [] !*B[] .A)(12321)]-> Результат T. Побочный эффект A=1, B=(3)
catch [elem 5 (1234)] 6]-> ошибка. Т.о. результат 6.

Кенга = млекопитающее & имеет_детёныша_Ру & хорошо_прыгает. Написать доказывающую теорему (Кенга?) и описывающую теорему (хорошо_прыгает).

```
[define th
 (conseq (X) (Кенга *X)
   [and [search (mammal .X)]
        [search (hasRooAsAKid .X)]
        [search (jump .X)]])]
 [define th1
 (conseq (X) (хорошо_прыгает *X)
   [achieve (Kenga *X)])]
```

Элиза

```
[define eliza (lambda ()
[cset a [read]]
[cond ([is ([] depressed []) a] [print (why you are depressed)])
     ([is (good bye) a] [exit () ()])
     ([is (pattern) a] [print (answer)])]
[eliza]
)]
[eliza]
```