Создание меню в ЛИСПе

Конструкции, которые понадобятся

- **loop** - бесконечный цикл без связанных с ним переменных.
- **let** - объявление локальной переменной
- **dolist -** DOLIST проходит по всем элементам списка, выполняя тело цикла с переменной, содержащей последовательно элементы списка
- **print - ф**ункции вывода с переходом на новую строку
- **princ -** не переходят на новую строку и не выводит пробел
- **setq** - форма применяется для присваивания переменной и блокирует вычисление первого аргумента
- **Read** - Функция ввода
- **cond** - ветвление, специальная функция **COND** (condition). Ее аргументами являются двухэлементные списки, содержащие предикаты и соответствующие им выражения
- **return** 'ok - RETURN - нормальное завершение программы. Аргумент return вычисляется, что и является значением программы. Никакие последующие операторы не вычисляются. Если программа прошла все свои операторы, не встретив Return, она завершается со значением NIL.
- **nth** - функция NTH, выделяющая n-й элемент списка
- **Funcal**l – функционвл, функция, которая позволяет вызывать другие функции,

Универсальное меню

```
;; Универсальное меню:

(defun uni-menu (lst-name lst-act)
 (loop
   (let ((p 0) (c 0))
    (dolist (m lst-name nil)
      (print p)(princ ".")(princ m)
      (setq p (1+ p)))
    (print p)(princ ". END")
    (print "Ваш выбор: ")
    (setq c (read))
    (cond ((= c p) (return 'ok))
          ((or (< c 0) (> c p)) (princ "Ошибка! Повторите."))
          (t (let ((f (nth c lst-act)))
                (funcall f)))))) )
```

Составим две функции-исполнителя:

```
;; сложить два числа

(defun f+ ()
  (let ((x (progn (print 'x=) (read)))
        (y (progn (print 'y=) (read))))
       (print (+ x y))))

;; Умножить два числа

(defun f* ()
  (let ((x (progn (print 'x=) (read)))
        (y (progn (print 'y=) (read))))
       (princ (* x y))))
```

Форма Prog имеет структуру, подобную определениям функций и процедур в **Паскале**:
(PROG, список рабочих переменных, последовательность операторов и атомов ...) Первый список после символа PROG называется **списком рабочих переменных**. При отсутствии таковых должно быть написано NIL или (). **Форма Prog** имеет структуру, подобную определениям функций и процедур в **Паскале**:

Испытаем все вместе:

- (uni-menu '("ADD 2 numbers" "MULTIPLY 2 numbers") '(f+ f*))

Функциональный подход
Определим функцию **SUM1** суммирования элементов списка:

- (defun sum1 (l)
- (cond ((null l) 0)
- (t (+ (car l)
- (sum1 (cdr l))))))
- **SUM1**
- * (sum1 '(10 20 30 40 50))
- **150**
- * (sum1 '(1))
- **1**
- * (sum1 '())
- **0**

Императивный подход

- Приведенные выше примеры можно переписать, используя цикл вместо рекурсии:
- (defun sum3 (l)
- (let ((sum 0))
- (dolist (x l)
- (setq sum (+ sum x))
-)
- sum))

- Трассировка — это отслеживание информации о ходе выполнения программы (например, о вызовах и аргументах функций) в целях ее отладки. В Lisp для этого можно воспользоваться макросом trace.
  Пример:

  (defun fact (n)
- (if (zerop n) 1
- (* n (fact (1- n)))))
- CL-USER>
- (trace fact)
- (FACT)
- CL-USER> (fact 5)

Предложения LET и LET*.

Предложение LET создает локальную связь внутри формы:

```
(LET ((m1 знач1) (m2 знач2)...)
          форма1 форма2 ...)
```

Вначале статические переменные m1, m2, ... связываются (одновременно) с соответствующими значениями знач1, знач2, ... Затем слева на право вычисляются значения формы1, формы2, ... Значение последней формы возвращается в качестве значения всей формы. После вычисления связи статических переменных ликвидируются.

Предложения LET можно делать вложенными одно в другое.

```
_(LET ((x 'a) (y 'b))
            (LET ((z 'c)) (LIST x y z))) ( (a b c)
_(LET ((x (LET ((z 'a)) z)) (y 'b))
            (LIST x y)) ( (a b)
_(LET ((x 1) (y (+ x 1)))
            (LIST x y)) ( ERROR
```

При вычислении y У и X еще нет связи. Значения переменным присваиваются одновременно. Это означает, что значения всех переменных mi вычисляются до того, как осуществляется связывание с формальными параметрами.

Подобной ошибки можно избежать с помощью формы LET*:

```
_(LET* ((x 1) (y (+ x 1)))
             (LIST x y)) ( (1 2)
```