МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

В. С. РОСТОВЦЕВ

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ XLISP

Учебное пособие

Киров
2018

УДК 004.43(07)
Р785

Рекомендовано к изданию кафедрой электронных вычислительных машин факультета автоматики и вычислительной техники ВятГУ

Допущено редакционно-издательской комиссией методического совета ВятГУ в качестве учебного пособия для студентов направления 09.04.01 «Информатика и вычислительная техника» всех профилей подготовки, всех форм обучения

Рецензенты:

канд. техн. наук, доцент,

заведующий кафедрой автоматики и телемеханики ВятГУ

В. И. Семёновых

канд. техн. наук, доцент,

заместитель директора ООО «Литек»

Н. А. Чарушин

Ростовцев, В. С.

Р785 Логическое программирование на языке XLISP: учебное пособие / В. С. Ростовцев. – Киров: ВятГУ, 2018. – 101 с.

Издание предназначено для студентов, изучающих дисциплину «Логическое программирование» и «Функциональное программирование».

УДК 004.43(07)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

LISP представляет собой язык так называемого функционального программирования. Он основан на алгебре списочных структур, лямбда-исчислений и теории рекурсивных функций. Благодаря неразвитости традиционной вычислительной техники, отличающемуся от других языков программирования характеру и из-за наличия элементарных средств обработки списков LISP долгое время являлся основным инструментом исследования искусственного интеллекта и средством теоретического подхода к анализу программирования [1–3].

XLISP – один из диалектов языка LISP. XLISP предназначен для задач обработки символьных данных. Основной структурой в этом языке является список, элементами которого служат атомы или перечни. Характерная особенность языка XLISP в том, что программы в нем тоже представлены в виде списков, т.е. и программы, и данные, которые они обрабатывают, имеют одинаковую структуру. XLISP широко используется для программирования научно-исследовательских задач в области искусственного интеллекта.

LISP (от англ. Lisp – list processing – обработка списков). Кроме функционального программирования в этом языке можно использовать программирование, основанное на обычном последовательном исполнении операторов с присваиваниями, передачами управления и специальными операторами цикла.

С помощью макропрограммирования можно запрограммировать новые структуры языка или реализовать совершенно новые языки. Кроме того, в LISPe можно применять множество методов программирования, известных из традиционных языков. LISP представляет собой язык так называемого функционального программирования. Он основан на алгебре списочных

структур, лямбда-исчислений и теории рекурсивных функций. XLISP используется под управлением операционных систем Windows и UNIX, но может быть легко перенесен на другие платформы. XLISP включает в себя ряд полезных функций Common Lisp.

1. ОРГАНИЗАЦИЯ РАБОТЫ НА ЯЗЫКЕ XLISP

1.1. Общие сведения

При запуске XLISP (xlwin32.exe) пытается загрузить стандартный файл сохраненного рабочего пространства "xlisp.wks" или любой другой файл из текущей директории, указанный после опции “-wfilename". Если такого файла нет или указана опция "-w" в командной строке, то XLISP вновь инициализирует рабочее пространство [1–3].

Далее XLISP пытается загрузить файл "init.lsp" из текущей директории или директорий, путь к которой указан в XLPATH. Этот файл инициализации может быть модифицирован по желанию пользователя.

Если в "init.lsp" встретятся функции которые не имеют аргументов, то они будут немедленно выполнены. Таким образом может быть запущено приложение, сохраненное в рабочем пространстве.

Если переменная *load-file-arguments* не nil (по умолчанию "t"), то загружаются все файлы с расширением «. lsp», имена которых были указаны как параметры в командной строке.

Опция "-tfilename" позволяет открыть файл с именем "filename". После чего XLISP перейдет на уровень ввода команд. Принудительный переход на командный уровень выполняется с помощью функции TOP-LEVEL-LOOP.

Сообщение ">" информирует о том, что XLISP ожидает ввода команды пользователя. Если текущая посылка отличается от USER, имя посылки печатается перед ">". Когда ввод выражения закончен, XLISP пытается вычислить его. Если этот этап завершается успешно, то выводится результат и XLISP переходит к следующему выражению.

Загрузка программы пользователя выполняется в следующем порядке.

1. В текстовом редакторе (можно использовать приложение «Блокнот») создать программу на языке XLISP и сохранить с расширением *.lsp

2. Запустить программу XLISP (xlwin32.exe).

3. В программе XLISP командой OPEN загрузить созданный файл с расширением *.lsp

4. Ввести имя основной функции в круглых скобках из созданного файла с расширением *.lsp. Например, (box)

5. Выполнить отладку программы на языке XLISP.

На рис. 1.1 приведена экранная форма XLISP непосредственно после запуска программы.

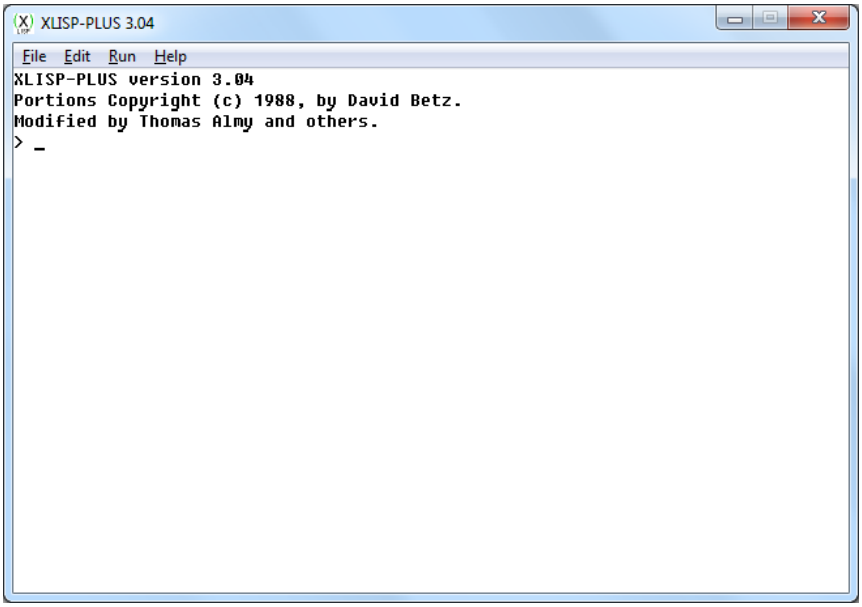

Рис.1.1. Главное меню программы XLISP

При вводе могут применяться следующие управляющие символы:

Backspace – удаляет последний символ

Del – удаляет символ

Tab – вставка символа табуляции (эквивалентно пробелам при чтении)

Ctrl-C – переход в режим ввода команды

Ctrl-G – вернуться на предыдущий уровень

Ctrl-Z – конец файла (возврат на предыдущий уровень или выход из программы)

Ctrl-P – выполнение (продолжение выполнения)

Ctrl-T – вывод информации

Зарезервированные клавиши управления:

Up Arrow – Показать предыдущую команду из буфера

Down Arrow – Показать следующую команду из буфера

Left Arrow – Переместить курсор влево

Right Arrow – Переместить курсор вправо

Home – Переместить курсор на начало строки

End – Переместить курсор в конец строки

Escape – Отменить текущую строку

1.2. Соглашения по лексике

Символьные имена в XLISP могут содержать любые непустые печатные символы за исключением макросимволов:

( ) скобки

Апостроф ‘ `

Запятая ,

Двойные кавычки “

Точка с запятой ;

Решётка #

Управляющие символы | \

Символ истины T

Символ ложь или пустой список NIL

Символы верхнего и нижнего регистров не различаются. Комментарии в программе могут начинаться с любого символа и действуют до конца строки.

Чтобы исключить возможность использования управляющих последовательностей, символьные имена в XLISP могут содержать любые непустые печатные символы за исключением макросимволов:

( ) ' ` , " ;

а также, управляющих символов:

\ |

В частности, первый символ не может быть # (это невыводимый на экран макросимвол) и не может быть символом, который считается числовым литералом.

Строки представляются последовательностью литералов, окруженных двойными кавычками. Строка может включать в себя следующие непечатные символы:

\\ – обратный слэш считается символом;

\n – переход на новую строку

\t – табуляция

\r – символ возврата

\f – символ прокрутки

2. ОСНОВНЫЕ ФУНКЦИИ ЯЗЫКА XLISP

2.1. Функции присваивания

(**set** <sym> <expr>) – Установка глобального значения идентификатора

<sym> идентификатор, который нужно установить

<expr> новое значение

Возвращает новое значение

(**setq** [<sym> <expr>]...) – Присваивание значения идентификатору

<sym> идентификатор, которому нужно присвоить значение

<expr> новое значение

Возвращает последнее новое значение или NIL, если функция используется без аргументов

Примеры

```
(setq T1 100)            ; присвоение идентификатору T1 значения 100
(setq colors (list "Красный" "Синий" "Зеленый"))
; идентификатору colors сопоставляется список строк.
(setq n (1+ n))      ; инкрементирование значения идентификатора n
```

(**psetq** [<sym> <expr>]...)

Параллельная версия SETQ. Все выражения вычисляются, и только затем выполняются присваивания

<sym> идентификатор, которому нужно присвоить значение

<expr> новое значение

Создание новой функции

(**defun** <sym> <fargs> <expr>...) – Определение функции

Примеры

Функция для вычисления факториала числа

```
(defun fucT(num)          ; описание функции
(setq i 0)                ; задание назальных значений переменных
(setq res 1)
(dotimes (i num)          ; цикл от 0 до num-1
(progn
(setq res (* res (1+ i)))
(setq i (1+ i)))
)
(setq num res)
)
(princ (apply 'fuct '(5)))
```

Результат на экране

```
120           ; 5!=120
```

Удалить из исходного списка элементы, значения которых больше 5

```
(defun test1 (x) (if (> x 5) (setq x T) (setq x NIL)))
 ;функция проверки значения элемента
(setq res (delete-if 'test1 '(11 2 8 12 -4 0 6 1 7)))
(princ res)
```

Результат на экране

(2 -4 0 1); удалены элементы, значение которых больше 5

(**defmacro** <sym> <fargs> <expr>...) – Определение макроса

<sym> изменяемый идентификатор

<fargs> список формальных параметров

<expr> выражения "тела" функции

Возвращает идентификатор функции или макроса

(**defconstant** <sym> <val> [<comment>]) – Описание константы

<sym> идентификатор

<val> значение

<comment> строка комментариев

Возвращает значение

Примеры

```
(defconstant n 1.2 Константа)
(princ (setq res (* n 2)))
```

Результат на экране

```
2.4             ; 1.2*2=2.4
(defconstant word "XLISP")
  (princ (concatenate 'STRING "Язык программирования " word))
```

Результат на экране

```
Язык программирования XLISP
; результат объединения строк
```

(**defparameter** <sym> <val> [<comment>]) – Описание параметра

<sym> идентификатор

<val> значение

<comment> строка комментариев

Возвращает значение

(**defvar** <sym> [<val> [<comment>]]) – Описание переменной.

Переменная инициализируется, если только не была описана ранее

<sym> идентификатор

<val> инициализирующее значение или NIL, если не указано

<comment> строка комментариев

Возвращает текущее значение

Примеры

```
(defvar m 12)
(setq list2 (list 11 2 8 12 -4 0 6 1 7))
(dolist (elem list2)
  (if (= elem m) (princ "В списке есть элемент со значением 12")))
```

Результат на экране

В списке есть элемент со значением 12

2.2. Функции работы с массивами

Функции работы с последовательностями также применимы и для работы с массивами.

(**aref** <array> <n>) – Получение одного из элементов массива

<array> массив (или строка)

<n> номер элемента(целое, начиная с нуля)

возвращает значение соответствующего элемента массива

(**make-array** <size> &key :initial-element :initial-contents)

Создание нового массива

<size> размер нового массива (целое число)

:initial-element значение для начальной инициализации всех элементов массива, по умолчанию NIL,

:initial-contents последовательность используется для инициализации всех элементов массива, по одному элементу для каждого элемента массива. Таким образом, длина последовательности должна быть равна длине массива.

возвращает новый массив

(**vector** <expr>...) – Создание инициализирующего вектора

<expr> вектор элементов

возвращает новый вектор.

2.3. Функции для работы с последовательностями

Эти функции работают с последовательностями-списками, массивами и строками.

(**concatenate** <type> <expr> ...)

Конкатанация последовательностей

Если тип выражений – string, последовательности должны включать только символы.

<type> тип результата: CONS, LIST, ARRAY или STRING

<expr> последовательность для конкатанации (количество может быть от 0 и больше)

возвращает последовательность, объединённую с последовательностями, указанными в параметрах функции.

(**elt** <expr> <n>)

Получить элемент последовательности

<expr> последовательность

<n> номер элемента последовательности

возвращает значение элемента под этим номером.

(**map** <type> <fcn> <expr> ...)

Применение функции к последовательным элементам

(**map-into** <target> <fcn> [<expr> ...])

<type> тип результата – это может быть CONS, LIST, ARRAY, STRING или NIL

<target> последовательность, предназначенная для модификации

<fcn> имя функции

<expr> последовательность для каждого параметра функции возвращает новую последовательность <type> для MAP, и <target> для MAP-INTO.

(**every** <fcn> <expr> ...)

Применение функции к элементам последовательности до первого NIL

(**notevery** <fcn> <expr> ...)

<fcn> имя функции

<expr> последовательность для каждого элемента функции

every возвращает результат последней выполненной функции,

notevery возвращает T, если результат функции NIL, иначе NIL.

(**some** <fcn> <expr> ...)

Применение функции к элементам последовательности до

(**notany** <fcn> <expr> ...) первого элемента, не равного TRUE

<fcn> имя функции

<expr> последовательность для каждого параметра функции

some возвращает результат первой не NIL функции, или NIL.

Notany возвращает NIL, если есть результат не NIL функции, иначе T.

(**length** <expr>) – Определение длины последовательности

Необходимо отметить, что круговой список в качестве параметра вызовет ошибку. Для обнаружения кругового списка можно использовать LIST-LENGTH.

<expr> список, вектор или строка

возвращает длину списка, вектора или строки.

(**reverse** <expr>) – Обращение последовательности (последние элементы станут первыми)

<expr> последовательность для обращения

Возвращает новую обращенную последовательность.

Примеры

```
(setq numbers (list 1 2 3 4 5))  ; описание списка numbers
(setq numbers (reverse numbers)) ; обращение списка numbers
(princ numbers)                  ; печать сформированного списка
```

Результат на экране

```
 (5 4 3 2 1)  ; результат – перевернутый список
(setq vec (vector "Утро" "День" "Вечер" "Ночь"))        ; описание вектора
  (princ (reverse vec)) ; печать обращенного вектора
```

Результат на экране

```
#(Ночь Вечер День Утро) ; результат – перевернутый массив
(setq l1 "МИР")
  (princ (reverse l1))
```

Результат на экране

```
РИМ          ; результат – перевернутое слово
```

(**subseq** <seq> <start> [<end>]) – Извлечение последовательности

<seq> последовательность

<start> начальная позиция (первоначально нуль)

<end> конечная позиция плюс 1 (по умолчанию) или NIL – до конца последовательности

Возвращает последовательность между <start> и <end>

Нумерация элементов списка начинается с нуля. В параметре <end> указывается увеличенное на единицу значение номера последнего из выбираемых элементов.

Примеры

```
(setq list1 (list 1 2 3 4 5 6 7 8)) ; описание списка list1
(setq newlist (subseq list1 2 5 ))  ; формирование  нового списка  ; из 3-го, 4-го и 5-го элементов исходного списка
 (princ newlist)                    ; печать сформированного списка
```

Результат на экране

```
(3 4 5)
(setq vec (vector "А" "Б" "В" "Г" "Д" "Е" "Ж")) ; описание вектора
 (setq newvec (subseq vec 0  5))  ; формирование нового вектора
 (princ newvec)                   ; печать сформированного вектора
```

Результат на экране

```
#(А Б В Г Д)
(setq word "ПЕРЕСЕЧЕНИЕ")          ; описание строковой переменной
 (princ (subseq word 4))
```

Результат на экране

```
СЕЧЕНИЕ
; из слова ПЕРЕСЕЧЕНИЕ выделена строка начиная с 5-го
; символа (индекс символа – 4)
```

(**sort** <seq> <test> &key :key) – Сортировка последовательности

<seq> последовательность для сортировки

<test> функция сравнения

:key функция применяемая к аргументам функции сравнения (по умолчанию тождественность)

Возвращает отсортированную последовательность

Примеры

```
(setq list2 (list 11 2 8 12 -4 0 6 1 7))
  (print (sort list2 '>))
```

Результат на экране

(12 11 8 7 6 2 1 0 -4)

; результат – список, отсортированный в порядке возрастания ; значения

; элементов

```
(setq list2 (list "C" "A" "Z" "P"))
   (print (sort list2 'string<))
```

Результат на экране

("A" "C" "P" "Z")

; результат – список символов, отсортированный в порядке

; возрастания

; значения ASCII-кодов элементов

Примеры

Требуется выполнить сортировку списка, элементами которого также являются списки, по третьему (индекс 2) элементу внутреннего списка. Например, граф задается списком, состоящим из трех элементов. Первый и второй элементы – номера вершин, третий элемент списка – вес дуги, соединяющей эти вершины.

```
(setq graf (list (list 1 2 4)    (list 1 4 2) (list 1 6 3) (list 2 3 3) (list 2 4 1) (list 3 4 2)
```

(list 3 5 6) (list 3 9 8) (list 4 5 4) (list 4 6 5) (list 4 7 7) (list 5 7 1) (list 5 8 2) (list 5 9 3) (list 6 7 6) (list 7 8 2) (list 8 9 3))

(defun weight(lst) ;функция возвращает вес ребра

(nth 2 lst) ; (третий элемент списка))

(setq graf (sort graf '< :key 'weight))

; для указания функции, применяемой к каждому элементу списка используется ключевое слово key

Результат на экране

((5 7 1) (2 4 1) (1 4 2) (7 8 2) (3 4 2) (5 8 2) (8 9 3) (1 6 3) (5 9 3) (2 3 3)(1 2 4) (4 5 4) (4 6 5) (6 7 6) (3 5 6) (4 7 7) (3 9 8)) ; отсортированный по возрастанию список по весу дуги графа (третий элемент списка)

(**search** <seq1> <seq2> &key :test :test-not :key :start1 :end1 :start2 :end2) – Поиск последовательности

<seq1> искомая последовательность

<seq2> последовательность, в которой осуществляется поиск

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start1 начальная позиция в последовательности <seq1>

:end1 индекс end плюс1 в последовательности <seq1> или NIL – конец последовательности

:start2 начальная позиция в последовательности <seq2>

:end2 индекс end плюс1 в последовательности <seq1> или NIL – конец последовательности

Возвращает позицию первого соответствия.

> (search "23" "1234512345" :start2 2)

6

> (search "23" "1234512345")

1

(**remove** <expr> <seq> &key :test :test-not :key :start :end) – Удаление элементов из заданной последовательности

<expr> удаляемый элемент

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция применяемая к аргументам функции сравнения последовательности (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс 1, или NIL для (length <seq>)

Возвращает копию последовательности с соответствующими удаленными элементами

Примеры

```
(setq numbers (list 1 0 2 0 3 0 4 0 5 0 6)) ;описание списка
 (setq newlist (remove 0 numbers)) ; удаление нулевых элементов
 (princ newlist) ; печать сформированного списка
```

Результат на экране

```
(1 2 3 4 5 6)        ; удалены нулевые элементы
```

(**remove-if** <test> <seq> &key :key :start :end) – Удалить элементы, которые удовлетворяют условию

(**remove-if-not** <test> <seq> &key :key :start :end) – Удалить элементы, которые не удовлетворяют условию

<test> проверяемое условие

<seq> последовательность

:key функция применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс 1, или NIL для (length <seq>)

Возвращает копию последовательности без соответственно удаленных элементов.

Примеры. *Удалить из исходного списка целые числа*

```
(setq res (remove-if 'integerp '(1 2 3.12 4 4.46 14 0.1)))
(print "Список вещественных чисел")
(princ res)
```

Результат на экране

Список вещественных чисел

(3.12 4.46 0.1)

Удалить из исходного списка вещественные числа

```
(setq res (remove-if-not 'integerp '(1 2 3.12 4 4.46 14 0.1)))
(print "Список целых чисел")
(princ res)
```

Результат на экране

Список целых чисел

(1 2 4 14)

(**count** <expr> <seq> &key :test :test-not :key :start :end) – Подсчитать соответствующие элементы последовательности

<expr> подсчитываемый элемент

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция применяемая к каждому аргументу последовательности (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс 1, или NIL для (length <seq>)

Возвращает число элементов, удовлетворяющих условию

Примеры

```
(setq l1 (list 1 3 5 7 8 0 1 3 5 8 0))
 (setq m 1)
 (princ (count m l1))
```

Результат на экране

```
 2
```

; в исходном списке 2 элемента, значение которых равны 1

(**count-if** <test> <seq> &key :key :start :end) – Подсчитать элементы, которые удовлетворяют условию

(**count-if-not** <test> <seq> &key :key :start :end) – Подсчитать элементы, которые не удовлетворяют условию

<test> проверяемое условие

<seq> последовательность

:key функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс1 или NIL для (length <seq>)

Возвращает число соответствующих или не соответствующих элементов.

Примеры

Подсчет количества нулевых элементов в списке

```
(setq res (count-if 'zerop '(0 2 0 4 19 14 0 9)))
(princ "В списке имеется ")
(princ res)
(princ " нулевых элемента")
```

Результат на экране

В списке имеется 3 нулевых элемента

Подсчет количества четных элементов в списке

```
(setq res (count-if-not 'oddp '(0 2 0 4 19 14 0 9)))
(princ "В списке имеется ")
(princ res)
(princ " четных элементов")
```

Результат на экране

В списке имеется 6 четных элементов

(**find** <expr> <seq> &key :test :test-not :key :start :end) – Найти первый соответствующий элемент в последовательности

<expr> элемент для поиска

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к каждому аргументу последовательности (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс 1, или NIL для (length <seq>)

Возвращает первый соответствующий элемент последовательности или NIL.

Примеры

(setq list2 (list 1 2 3 4 5 6 7 5 6 7))

(princ (find (/ 40 8) list2))

Результат на экране

5

; найден элемент списка, значение которого 5

(**find-if** <test> <seq> &key :key :start :end) – Найти первый элемент, который удовлетворяет условию

(**find-if-not** <test> <seq> &key :key :start :end) – Найти первый элемент, который не удовлетворяет условию

<test> проверяемое условие

<seq> последовательность

:key функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс 1 или NIL для (length <seq>)

Возвращает первый соответствующий элемент или NIL

(**position** <expr> <seq> &key :test :test-not :key :start :end) – Найти позицию первого соответствующего элемента в последовательности

<expr> элемент для поиска

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к каждому аргументу последовательности (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс1 или NIL для (length <seq>)

Возвращает позицию первого соответствующего элемента в последовательности или NIL.

Примеры

```
(setq list2 (list 1 2 3 4 5 6 7))
 (princ (position 5 list2))
```

Результат на экране

4

(**delete** <expr> <seq> &key :key :test :test-not :start :end) – Удаление элементов из последовательности

<expr> удаляемый элемент

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс1 или NIL для (length <seq>)

Возвращает последовательность с соответственно удаленными из нее элементами.

Примеры

```
(setq list2 (list 1 2 3 4 5 6 7 5 6 7))
 (princ (delete 5 list2))
```

Результат на экране

(1 2 3 4 6 7 6 7)

(**delete-if** <test> <seq> &key :key :start :end) – Удаление элементов, которые удовлетворяют условию

(**delete-if-not** <test> <seq> &key :key :start :end) – Удаление элементов, которые не удовлетворяют условию

<test> проверяемое условие

<seq> последовательность

:key функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end плюс1 или NIL для (length <seq>)

Возвращает последовательность с соответственно удаленными из нее элементами.

Примеры

Удалить из исходного списка элементы, значения которых больше 5.

```
(defun test1 (x) (if (> x 5) (setq x T) (setq x NIL)))
;функция проверки значения элемента
(setq res (delete-if 'test1 '(11 2 8 12 -4 0 6 1 7)))
(princ res)
```

Результат на экране

(2 -4 0 1) ; удалены элементы, значение которых больше 5

Удалить из исходного списка элементы, значения которых по абсолютному значению меньше 3

```
(defun test1 (x) (if (> (abs x) 3) (setq x T) (setq x NIL)))
(setq res (delete-if-not 'test1 '(11 2 8 12 -4 0 6 1 7)))
(princ res)
```

Результат на экране

(11 8 12 -4 6 7) ; удалены элементы, абсолютные значения ;которых меньше 3

Примеры

(defun d2(x)(/ x 2))

(d2 10) ; возвращает d2

Результат на экране

5 ; возвращает 5

(**reduce** <fcn> <seq> &key :initial-value :start :end) – Усечение последовательности до одного значения

<fcn> функция (от двух аргументов) применяемая к результату применения предыдущей функции (или первому элементу) и каждому члену

последовательности

<seq> последовательность

:initial-valuевеличина, используемая как первый аргумент в первой применяемой функции (раньше использования первого элемента последовательности)

:start начальный индекс

:end конечный индекс end плюс1 или NIL для (length <seq>)

Возвращает NIL если последовательность пустая и не имеет начальных значений, иначе возвращается результат применения последней функции.

Примеры

Вычисление значения 5!

(princ (reduce '*' (1 2 3 4 5)))

Результат на экране

120

; 1*2*3*4*5=120=5!

(**remove-duplicates** <seq> &key :test :test-not :key :start :end) – Удаление одинаковых элементов из последовательности

<seq> последовательность

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

:start начальный индекс

:end конечный индекс end+1, или NIL для (length <seq>)

Возвращает копию последовательности без одинаковых элементов.

Примеры

```
(setq numbers (list 1 6 2 5 3 4 4 3 5 2 6 1))
 (setq newlist (remove-duplicates numbers))
 (princ newlist)
```

Результат на экране

(4 3 5 2 6 1)

; из исходной последовательности удалены повторяющиеся элементы

2.4. Функции работы со списками

(**car** <expr>) синоним (**first** <expr>) – Возвращает первый элемент списка

<expr> список узлов

Функции не изменяют значение параметра <expr>

Примеры

(setq days (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

(princ "Первый день недели – ")

(princ (car days))

Результат на экране

Первый день недели – Понедельник

(setq letters (list "А" "Б" "В" "Г" "Д"))

(princ "Первая буква русского алфавита – ")

(princ (first letters))

Результат на экране

Первая буква русского алфавита – А

(**cdr** <expr>) – (**rest** <expr>) – Возвращает оставшиеся кроме первого элементы списка узлов

<expr> список узлов

Функции не изменяют значение параметра <expr>.

Примеры

(setq days (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

(princ "Все дни недели кроме понедельника – ")

(princ (cdr days))

Результат на экране

Все дни недели кроме понедельника – (Вторник Среда Четверг Пятница Суббота Воскресенье))

(**cxxr** <expr>) – Все CxxR комбинации

(**cxxxr** <expr>) – Все CxxxR комбинации

(**cxxxxr** <expr>) – ВсеCxxxxR комбинации

(**second** <expr>) – Синоним CADR

Выполнение указанных функций начинается из самого глубокого вложения

Примеры

(caar sc) = (car(car sc))

(cadr sc) = (car(cdr sc))

(cdar sc) = (cdr(car sc))

(caddr sc) = (car(cdr(cdr sc)))

(cadar sc) = (car(cdr(car sc)))

Примеры

(setq days (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

(princ "Второй день недели – ")

(princ (second days))

Результат на экране

Второй день недели – Вторник

(**third** <expr>) – Синоним CADDR

(setq days (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

(princ "Третий день недели – ")

(princ (Third days))

Результат на экране

Третий день недели – Среда

(**fourth** <expr>) – Синоним CADDDR. (cadddr sc) = (car (cdr (cdr (cdr sc))))

Примеры

(setq sc (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

(princ "Четвертый день недели – ")

(princ (fourth sc))

Выполнение функции начинается из самого глубокого вложения (cdr sc). Результат первого шага список sc, который начинается со вторника. На втором шаге список начинается со среды. На четвертом шаге список sc имеет вид ("Четверг" "Пятница" "Суббота" "Воскресенье"). На пятом шаге функция car выделяет четверг (первый элемент списка).

Результат на экране

Четвертый день недели – Четверг

(**cons** <expr1> <expr2>) – Создает новый список узлов

<expr1> первый элемент нового списка узлов

<expr2> остальные элементы списка узлов

Возвращает новый список узлов

Примеры

```
(setq numbers (list 2 -3 0 8 0 1 7))       ; описание списка
 (setq num ( ))                                    ; описание пустого списка
 (dolist (elem numbers)                    ; цикл по списку
 (cond ((< elem 0) (setq num (cons "NEG" num))) ; формирование нового
((= elem 0) (setq num (cons "ZERO" num)))   ; списка
((> elem 0) (setq num (cons "POS" num)))))
 (setq num (reverse num))                    ; перестановка элементов списка
```

(print num) ; печать элементов списка

Результат на экране

("POS" "NEG" "ZERO" "POS" "ZERO" "POS" "POS")

; сформирован новый список

(**list** <expr>...) – Создает список значений

<expr> выражения, которые будут объединены в список

Возвращает новый список.

Примеры

(setq letters (list «А» «Б» «В» «Г» «Д»))

; идентификатору letters сопоставляется список символов

(setq num (list)) ; описание пустого списка

Пустой список можно задать следующим образом

(setq list1 (list)) ; описание пустого списка

(**append** <expr>...) – Позволяет взять любое число списков и слить их вместе в один список

<expr> список, элементы которого будут добавлены

Возвращает новый список

Примеры

(setq l1 (list 1 2 3 4 5))

(setq l2 (list 6 7 8 9 10))

(setq l3 (list 11 12 13 14 15))

(setq l4 (append l1 l2 l3))

; идентификатору l4 сопоставляется список, представляющий собой результат

; объединения списков l1, l2 и l3

; результирующий список l4 (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

(setq list1 (list 1 2 3 4 5))

(setq newlist (append list1 (list «E» «N» «D»))

; идентификатору newlist сопоставляется список (1 2 3 4 5 «E» «N» «D»)

(**list-length** <list>) – Определяет длину списка

<list> список

Возвращает длину списка или NIL, если список циклический

Примеры

(setq numbers (list 3 6 3 4 7))

(setq num(list-length numbers))

; переменной num присваивается значение 5, равное числу элементов списка numbers

(**last** <list>) – Возвращает последний список узлов из списка

<list> исходный список

Примеры

(setq week (list "Понедельник" "Вторник" "Среда" "Четверг"

"Пятница" "Суббота" "Воскресенье

; описание списка дней недели week

(setq lastday (last week)) ; формирование нового списка lastday

; из последнего элемента списка week

(princ "Последний день недели – ") ; печать сообщения

(princ lastday) ; печать списка lastday

Результат на экране

Последний день недели – (Воскресенье)

(**butlast** <list> [<n>]) Возвращает копию списка, но без последнего элемента

<list> список

<n> число элементов (по умолчанию 1)

Возвращает копию списка, но без последних элементов

Примеры

```
(setq week ( list "Понедельник" "Вторник" "Среда" "Четверг"
                  "Пятница" "Суббота" "Воскресенье" ))
; описание списка дней недели week
(princ " Рабочие дни  – ")          ; печать сообщения
(setq workdays (butlast week 2))  ; формирование нового списка workdays
; из элементов списка week кроме двух последних
(princ workdays)                ; печать сформированного списка
```

Результат на экране

Рабочие дни – (Понедельник Вторник Среда Четверг Пятница)

(**nth** <n> <list>)

Возвращает N-й элемент списка

<n> номер возвращаемого элемента

<list> список

Возвращает N-й элемент списка или NIL, если список не настолько большой.

Нумерация элементов в списке начинается в нуля

Примеры

```
(setq numbers (list 1 2 3 4 5))
  (setq m (nth 0 numbers))
```

; идентификатору m присваивается значение первого элемента списка numbers, т.е. 1

(setq numbers (list 1 2 3 4 5))

(setq m (nth 5 numbers))

; идентификатору m присваивается значение NIL, т.е. в списке numbers отсутствует 6-й элемент

(**nthcdr** <n> <list>) – Возвращает N последних элементов

<n> номер элемента (первоначально нуль)

<list> список

Возвращает элементы списка, начиная с N-го, или NIL, если список не настолько большой.

Нумерация элементов в списке начинается с нуля

Примеры

(setq week (list "Понедельник" "Вторник" "Среда" "Четверг" "Пятница" "Суббота" "Воскресенье"))

; описание списка дней недели week

(princ " Выходные дни – ") ; печать сообщения

(setq weekend (nthcdr 5 week))

; формирование нового списка weekend из двух последних элементов списка week

(princ weekend) ; печать сформированного списка

Результат на экране

Выходные дни – (Суббота Воскресенье)

(**member** <expr> <list> &key :test :test-not :key) – Находит выражение в списке

<expr> искомое выражение

<list> список для поиска

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к аргументам функции сравнения элементов списка (по умолчанию тождественность)

Возвращает остаток от списка, начинающийся с заданного выражения.

Примеры

В исходном списке чисел выбрать элементы, расположенные после элемента со значением 6

```
(setq list1 (list 1 2 3 4 5 6 7 )) ; описание исходного списка list1
(setq newlist (member 6 list1)) ; формирование нового списка  newlist
(princ newlist)                 ; печать сформированного списка
```

Результат на экране

(6 7)

; в исходном списке найден элемент со значением 6 и сформирован новый список из элементов исходного списка, начиная со значения 6.

(**apply** <fcn> <list>) – Последовательное применение функции <fcn> к элементам последовательности <list>.

Примеры

Функция для вычисления факториала числа

```
(defun fuct(num)          ; описание функции
 (setq i 0)               ; задание назальных значений переменных
 (setq res 1)
 (dotimes (i num)         ; цикл от 0 до num-1
  (progn
```

```
  (setq res (* res (1+ i)))
  (setq i (1+ i)))
)
(setq num res)
)
(princ (apply 'fuct '(5)))
```

Результат на экране

```
120             ; 5!=120
```

(**mapc** <fcn> <list1> <list>...) – Последовательное применение функции CARS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает первый список аргументов.

(**mapcar** <fcn> <list1> <list>...) – Последовательное применение функции CARS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает список возвращенных значений.

В качестве параметров данной функции нельзя использовать заранее определенные списки, например, ошибочной является запись.

(mapcar '+' list1' list2)

Списки аргументов функции необходимо разделять символом « ’ » без пробелов, кроме того, между описанием функции и первым списком аргументов пробел также не ставится

Примеры

```
(princ (mapcar '+'(1 2 3 4 5 6 7)'(1 2 3 4 5 6 7)'
  (1 2 3 4 5 6 7)))
```

Результат на экране

```
(3 6 9 12 15 18 21)
; результат сложения соответствующих элементов трех
; последовательностей
```

(**mapl** <fcn> <list1> <list>...) – Последовательное применение функции CDRS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает первый список аргументов.

(**maplist** <fcn> <list1> <list>...) – Последовательное применение функции CDRS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает список возвращенных значений.

(**mapcan** <fcn> <list1> <list>...) – Последовательное применение функции CARS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает список возвращенных значений

(**mapcon** <fcn> <list1> <list>...) – Последовательное применение функции CDRS

<fcn> функция или имя функции

<listn> список для каждого аргумента функции

Возвращает список возвращенных значений.

(**subst** <to> <from> <expr> &key :test :test-not :key)

(**nsubst** <to> <from> <expr> &key :test :test-not :key) – Замена выражений

<to> новое выражение

<from> старое выражение

<expr> выражение, в котором будет производиться замена

:test функция сравнения (по умолчанию равенство)

:test-not функция сравнения (значение инвертируется)

:key функция, применяемая к поддереву функций сравнения выражений (по умолчанию тождественность)

Возвращает выражение с выполненными заменами.

Функция не работает с символьными элементами, а также с массивами. Функция **subst**, в отличие от функции **nsubst**, не изменяет значение параметра <expr>.

Примеры

```
(setq list2 (list 1 2 3 1 4 5 6 1 7))
  (print (subst 100 1 list2))
  (print list2)
```

Результат на экране

(100 2 3 100 4 5 6 100 7) ; элементы со значением 1 заменены на значение 100

(1 2 3 1 4 5 6 1 7) ; исходный список не изменен

(setq list2 (list 1 2 3 1 4 5 6 1 7))

```
(print (subst 100 1 list2))
(print list2)
```

Результат на экране

```
(100 2 3 100 4 5 6 100 7)
; элементы со значением 1 заменены на элементы со значением 100
; исходный список изменен
```

(**adjoin** <expr> <list> :test :test-not :key) – Добавляет уникальный элемент в список

<expr>	новый добавляемый элемент
<list>	список
:test	функция сравнения (по умолчанию равенство)
:test-not	функция сравнения (значение инвертируется)
:key	функция, применяемая к аргументам функции сравнения (по умолчанию тождественность)

Возвращает: если элемента нет в списке, то (**cons** <expr> <list>), иначе <list>.

Примеры

```
(setq list2 (list 1 2 3 4 5 6 7))
(princ (adjoin 200 list2))
```

Результат на экране

```
(200 1 2 3 4 5 6 7)
; элемент 200 добавлен в список, так как такого элемента нет
(setq list2 (list 1 2 3 4 5 6 7))
(princ (adjoin 2 list2))
```

Результат на экране

(1 2 3 4 5 6 7)

; элемент 2 не добавлен в список, так как такой элемент в списке уже есть

2.5. Арифметические функции

(**truncate** <expr> <denom>) - Усечение к нолю

Примеры

— (setq res (/ 10 6))

 (princ res)

 (terpri)

 (princ (truncate res))

Результат на экране

5/3

 1 ; результат округления

(**round** <expr> <denom>) – Округление до ближайшего целого

Примеры

(setq res (/ 10 6))

 (princ res)

 (terpri)

 (princ (round res))

Результат на экране

5/3

 2 ; результат округления

(**floor** <expr> <denom>) – Усечение к отрицательной бесконечности

(**ceiling** <expr> <denom>) – Усечение к бесконечности

<expr> вещественное число

<denom> вещественное число для деления <expr> перед конвертацией

Возвращает целый результат конвертирования и остаток операции, определенный как выражение – resulT*denom, типа числа с плавающей точкой если ее аргумент число с плавающей точкой, иначе рациональное число.

(**float** <expr>) – Конвертирование целого числа в число с плавающей точкой

<expr> вещественное число

Возвращает число с плавающей точкой.

(**rational** <expr>) – Конвертирование вещественного числа в рациональное

<expr> вещественное число

Возвращает вещественное число или целое.

(+ [<expr>...]) – Сложение списка чисел

Если нет аргументов, то используется 0

<expr> числа

Возвращает результат сложения.

(<expr>...) – Вычитание списка чисел или отрицание одиночного числа

<expr> числа

Возвращает результат вычитания.

(***** [<expr>...]) – Перемножение списка чисел

Функция без аргументов возвращает единицу в качестве результата

<expr> числа

Возвращает результат умножения.

(/ <expr>...) – Деление списка чисел или инвертирование одиночного числа

С математическим сопроцессором результат деления целых чисел есть вещественное число. Для выполнения целочисленного деления следует использовать функцию TRUNCATE. При делении комплексного целого на целое число результатом будет вещественное комплексное

<expr> числа

Возвращает результат деления.

(**1+** <expr>) – Инкремент числа

<expr> число

Возвращает увеличенное на единицу число.

В результате применения функции значение параметра <expr> не изменяется

Примеры

```
(setq num 7)
 (setq num (1+ num))
 (princ num)
```

Результат на экране

```
8                    ; 7+1=8
```

(**1-** <expr>) – Декремент числа

<expr> число

Возвращает уменьшенное на единицу число.

В результате применения функции значение параметра <expr> не изменяется.

Примеры

```
(setq num 7)
 (setq num (1- num))
 (princ num)
```

Результат на экране

```
6               ; 7–1=6
```

(**rem** <expr>...) – Остаток от деления чисел

<expr> вещественные числа (без сопроцессора должны быть только целые)

Возвращает результат операции (остаток от целочисленного деления).

(**mod** <expr1> <expr2>) – Остаток от деления вещественных чисел

<expr1> вещественное число – делимое

<expr2> вещественное число – делитель (не может быть нулем)

Возвращает остаток от деления двух чисел, используя в качестве операции деления выполняется деление с плавающей запятой.

(**min** <expr>...) – Возвращает минимальное число из списка чисел

<expr> вещественные числа

Возвращает минимальное число из списка.

Параметр <expr> указывается без скобок.

Примеры

```
(setq minimum (min -6 0 1 8 -5))
 (princ minimum)
```

Результат на экране

-6 ; минимальный элемент в последовательности

(**max** <expr>...) – Возвращает максимальное число из списка чисел

<expr> вещественные числа

Возвращает максимальное число из списка

Параметр <expr> указывается без скобок

Примеры

```
(setq maximum (max -6 0 1 8 -5))
 (princ maximum)
```

Результат на экране

8 ; максимальный элемент в последовательности

(**abs** <expr>) – Модуль числа

<expr> число

Возвращает модуль числа, который является вещественным числом, равным реальной части в случае с комплексными числами.

Примеры

Удалить из исходного списка элементы, значения которых по абсолютному значению меньше 3

```
  (defun test1 (x) (if (> (abs x) 3) (setq x T) (setq x NIL)))
 (setq res (delete-if-not 'test1 '(11 2 8 12 -4 0 6 1 7)))
 (princ res)
```

Результат на экране

(11 8 12 -4 6 7)

; из исходного списка удалены элементы, значение которых по

; абсолютному значению меньше 3

(**signum** <expr>) – Получение знака числа

<expr> число

Возвращает нуль, если число равно нулю, единицу, если число положительное и минус единицу, если число отрицательное. Для комплексных чисел функция возвращает те же значения, но использует мнимую часть в качестве параметра.

(**gcd** [<n>...]) – Вычисление наибольшего общего делителя

Без аргументов функция возвращает нуль, при одном аргументе функция вернет сам аргумент

<n> целое число или числа

Возвращает наибольший общий делитель.

(**lcm** <n>...) – Вычисление наименьшего кратного множителя

<n> целое число или числа

Возвращает наименьший общий множитель.

(**random** <n> [<state>]) – Получение псевдослучайного числа

<n> вещественное число, с которым надо связать псевдослучайное число

<state> произвольное состояние (по умолчанию *random-state*)

Возвращает случайное число из диапазона [0,n).

(**make-random-state** [<state>]) – Создание произвольного состояния

<state> произвольное состояние: T или NIL (по умолчанию NIL). NIL – состояние переменной *random-state*

Если <state> равно T, то функция возвращает произвольное random-state, в другом случае функция вернет копию <state>

(**sin** <expr>) – Вычисление синуса

(**cos** <expr>) – Вычисление косинуса

(t**an** <expr>) – Вычисление тангенса

(**asin** <expr>) – Вычисление арксинуса

(**acos** <expr>) – Вычисление арккосинуса

<expr> вещественное число

Возвращает синус, косинус, тангенс, арксинус или арккосинус числа.

(**atan** <expr> [<expr2>]) – Вычисление арктангенса числа

<expr> вещественное число (числитель)

<expr2> знаменатель, по умолчанию 1

Возвращает арктангенс <expr>/<expr2>

(**sinh** <expr>) – Вычисление гиперболического синуса числа

(**cosh** <expr>) – Вычисление гиперболического косинуса числа

(t**anh** <expr>) – Вычисление гиперболического тангенса числа

(**asinh** <expr>) – Вычисление гиперболического арксинуса числа

(**acosh** <expr>) – Вычисление гиперболического арккосинуса числа

(**atanh** <expr>) – Вычисление гиперболического арктангенса числа

<expr> число

Возвращает гиперболический синус, косинус, тангенс, арксинус, арккосинус или арктангенс числа.

(**expt** <x-expr> <y-expr>) – Возведение x в степень y

<x-expr> число

<y-expr> степень

Возвращает значение x, возведенное в степень y. Если y – целое число, тогда тип результата зависит от типа x, в других случаях результат – вещественное число.

(**exp** <x-expr>) – Возведение "e" в степень x

<x-expr> число

Возвращает "e", в степени x

(**cis** <x-expr>) – Вычисление "e" в степени ix

<x-expr> число

Возвращает "e" в степени x

(**log** <expr> [<base>]) – Вычисление логарифма числа

<expr> число

<base> основание логарифма (по умолчанию "e")

Возвращает логарифм с основанием <base> числа <expr>.

(**sqrt** <expr>) – Вычисление квадратного корня числа

<expr> число

Возвращает квадратный корень числа.

(**numerator** <expr>) – Получение числителя числа

<expr> число, представленное отношением

Возвращает числитель числа (число, если оно целое).

(**denominator** <expr>) – Получение знаменателя числа

expr> число, представленное отношением

Возвращает знаменатель числа (1, число – целое).

(**complex** <real> [<imag>]) – Преобразование к комплексному числу

<real> вещественное число реальной части комплексного числа

<imag> вещественное число мнимой части (по умолчанию нуль)

Возвращает комплексное число

(**realpart** <expr>) – Получение реальной части числа

<expr> число

Возвращает реальную часть комплексного числа или само число, если число не является комплексным.

(**imagpart** <expr>) – Получение мнимой части числа

<expr> число

Возвращает мнимую часть комплексного числа или нуль, если число не является комплексным.

(**conjugate** <expr>) – Получение сопряженного числа

<expr> число

Возвращает сопряженное для комплексного числа или само число, если оно не является комплексным.

(**phase** <expr>) – Получение фазы числа

<expr> число

Возвращает угол-фазу, аналог (**atan** (imagpart <expr>) (realpart <expr>))

(< <n1> <n2>...) Проверка типа «меньше»

(<= <n1> <n2>...) Проверка типа «меньше или равно»

(= <n1> <n2>...) Проверка типа «равенство»

(/= <n1> <n2>...) Проверка типа «неравенство»

(>= <n1> <n2>...) Проверка типа «больше или равно»

(> <n1> <n2>...) Проверка типа «больше»

<n1> первое вещественное число для сравнения

<n2> второе вещественное число для сравнения

Возвращает результат сравнения <n1> с <n2>

2.6. Функции работы с битами

(**logand** [<expr>...]) – Поразрядное "И" над списком аргументов

Функция без аргументов возвращает единицу

<expr> целые числа

Возвращает результат операции "И".

(**logior** [<expr>...]) – Поразрядное "ИЛИ" над списком аргументов

Функция без аргументов возвращает нуль

<expr> целые числа

Возвращает результат операции " ИЛИ".

(**logxor** [<expr>...]) – Поразрядное "исключающее или" над списком аргументов

Функция без аргументов возвращает нуль

<expr> целые числа

Возвращает результат операции "исключающее ИЛИ".

(**lognot** <expr>) – Поразрядная "инверсия" числа

<expr> целое число

Возвращает результат поразрядной операции "инверсия".

(**logtest** <expr1> <expr2>) – Тест с помощью операции "и"

<expr1> первое целое число

<expr2> второе целое число

Возвращает T, если результат операции не равен нулю, иначе NIL.

(**ash** <expr1> <expr2>) – Арифметический сдвиг

<expr1> целое число, биты которого требуется передвинуть

<expr2> число битов, на которое требуется произвести сдвиг

Возвращает сдвинутое число.

2.7. Строковые функции

(**string** <expr>) – Преобразование числа в строку

<expr> целое число (оно преобразуется к ASCII символам), строка, символ или идентификатор

Возвращает строковое представление аргумента.

(**string-left-trim** <bag> <str>) – Урезание начала строки

<bag> строка, содержащая символы, которые будут урезаны

<str> строка для урезания

Возвращает урезанную копию строки.

Примеры

(setq str "Экспертная Система")

(princ (string-left-trim "Эксперт" str))

Результат на экране

ная Система

(**string-right-trim** <bag> <str>) – Урезание конца строки

<bag>строка, содержащая символы, которые будут урезаны

<str> строка для урезания

Возвращает урезанную копию строки.

(setq str "Экспертная Система")

(princ (string-left-trim "Система" str))

Результат на экране

Экспертная

(**string-upcase** <str> &key :start :end) – Преобразование к верхнему регистру

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование (NIL – до конца строки)

Возвращает преобразованную копию строки.

См. также функцию **nstring-upcase**

Примеры

(setq str1 "XLISP")

(princ (string-upcase str1))

(terpri)

(princ str1)

Результат на экране

XLISP ; результат применения функции

XLISP ; исходная строка

(**string-downcase** <str> &key :start :end) – Преобразование к нижнему регистру

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование (NIL – до конца строки)

Возвращает преобразованную копию строки.

См. также функцию **nstring-downcase**

Примеры

```
(princ (map 'ARRAY 'string-downcase' ("F" "I" "L" "E")))
```

Результат на экране

```
(f i l e)
(setq str1 "ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ")
 (princ (string-downcase str1))
 (terpri)
 (princ str1)
```

Результат на экране

логическое программирование ; результат применения функции

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ; исходная строка

(**string-capitaize** <str> &key :start :end) – Преобразование к прописной строке

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование(NIL – до конца строки)

Возвращает преобразованную копию строки, каждое слово в строке начинается с большой буквы (см. также функцию **nstring-capitaize).**

Значение параметра <str> функция не изменяет. Некоторые буквы русского алфавита не распознаются (например, буквы «Я» и «Ч»).

Примеры

```
(setq str1 "экспертная система на языке XLISP")
 (princ (string-capitaize str1))
 (terpri)
 (princ str1)
```

Результат на экране

```
Экспертная Система На яЗыке Xlisp   ; результат применения функции
экспертная система на языке XLISP ; исходная строка
```

(**nstring-upcase** <str> &key :start :end) – Преобразование к верхнему регистру

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование (NIL – до конца строки)

Возвращает преобразованную строку (см. также функцию **string-upcase)**

Примеры

```
(setq str1 "XLISP")
(princ (nstring-upcase str1))
(terpri)
```

(princ str1)

Результат на экране

XLISP ; результат применения функции

XLISP ; исходная строка

(**nstring-downcase** <str> &key :start :end) – Преобразование к нижнему регистру

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование (NIL – до конца строки)

Возвращает преобразованную строку (см. также функцию **string-downcase)**

Примеры

```
(setq str1 "ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ")
  (princ (nstring-downcase str1))
  (terpri)
  (princ str1)
```

Результат на экране

логическое программирование ; результат применения функции

(**nstring-capitaize** <str> &key :start :end) – Преобразование к прописной строке

<str> строка

:start номер символа, с которого надо начинать преобразование

:end номер символа плюс 1, на котором надо закончить преобразование (NIL – до конца строки)

Возвращает преобразованную строку (строки, где каждое новое слово начинается с прописной буквы)

Примеры

```
(setq str1 "экспертная система на языке XLISP")
 (princ (nstring-capitaize str1))
 (terpri)
 (princ str1)
```

Результат на экране

Экспертная Система На Языке Xlisp

; результат применения функции

(**string**< <str1> <sTr2> &key :start1 :end1 :start2 :end2)

(**string<=** <str1> <sTr2> &key :start1 :end1 :start2 :end2)

(**string=** <str1> <sTr2> &key :start1 :end1 :start2 :end2)

(**string/=** <str1> <sTr2> &key :start1 :end1 :start2 :end2)

(**string>=** <str1> <sTr2> &key :start1 :end1 :start2 :end2)

(**string>** <str1> <sTr2> &key :start1 :end1 :start2 :end2)

<str1> первая строка для сравнения

<st2> вторая строка для сравнения

:start1 номер символа, с которого надо начинать сравнение в первой строке

:end1 номер символа плюс 1, на котором надо закончить сравнение в первой строке (NIL – до конца строки)

:start2 номер символа, с которого надо начинать сравнение в первой строке

:end2 номер символа плюс 1, на котором надо закончить сравнение в первой строке (NIL – до конца строки)

Возвращают string=: T, если предикат правилен, NIL в другом случае.

Для нижеследующих функций: если сравнение неуспешно, тогда возвращается номер последнего проверенного символа.

(**string-lessp** <str1> <st2> &key :start1 :end1 :start2 :end2)

(**string-not-greatrp** <str1> <st2> &key :start1 :end1 :start2 :end2)

(**string-equal** <str1> <st2> &key :start1 :end1 :start2 :end2)

(**string-not-equal** <str1> <st2> &key :start1 :end1 :start2 :end2)

(**string-not-lessp** <str1> <st2> &key :start1 :end1 :start2 :end2)

(**string-greaterp** <str1> <st2> &key :start1 :end1 :start2 :end2)

<str1> первая строка для сравнения

<st2> вторая строка для сравнения

:start1 номер символа, с которого надо начинать сравнение в первой строке

:end1 номер символа плюс 1, на котором надо закончить сравнение в первой строке (NIL – до конца строки)

:start2 номер символа, с которого надо начинать сравнение в первой строке

:end2 номер символа плюс 1, на котором надо закончить сравнение в первой строке (NIL – до конца строки)

Возвращается T, если строки равны, NIL в других случаях. Если условие не верно, то возвращается номер символа, который не удовлетворяет условию.

Необходимо заметить, что в этих функциях все символы в строках предварительно преобразовываются к нижнему регистру.

2.8. Функции проверки

(**atom** <expr>) – Это атом?

<expr> проверяемое выражение

Возвращает T, если значение – атом, NIL в другом случае.

(**symbolp** <expr>) – Это идентификатор?

<expr> проверяемое выражение

Возвращает T, если значение – идентификатор, NIL в другом случае.

(**numberp** <expr>) – Это число?

<expr> проверяемое выражение

Возвращает T, если значение – число, NIL в другом случае.

(**null** <expr>) – Это пустой список?

<expr> проверяемый список

Возвращает T, если список пуст, NIL в другом случае.

(**not** <expr>) – Это FALSE?

<expr> проверяемое выражение

Возвращает T, если NIL – число, NIL в другом случае.

Примеры

Проверить все ли элементы списка являются целыми числами

```
(setq res (notevery 'integerp '(1 2 3.14 4)))
(if (not res) (princ "Все элементы списка целые числа")
(princ "Не все элементы списка являются целыми числами")).
```

Результат на экране

Не все элементы списка являются целыми числами

(**listp** <expr>) – Это список?

<expr> проверяемое выражение

Возвращает T, если выражение равно NIL или его возможно присоединить к списку, NIL в противном случае.

(**endp** <list>) – Это конец списка?

<list> список

Возвращает T, если выражение равно NIL, NIL в другом случае.

(**consp** <expr>) – Это не пустой список?

<expr> проверяемое выражение

Возвращает T, если выражение его возможно присоединить к списку, NIL в другом случае.

(**constnt** <expr>) – Это константа?

<expr> проверяемое выражение

Возвращает T, если выражение константа, NIL в другом случае.

(**specialp** <expr>) – Это имя специальной переменной?

<expr> проверяемое выражение

Возвращает T, если выражение является именем специальной переменной, NIL в другом случае.

(**integerp** <expr>) – Это целое число?

<expr> проверяемое выражение

Возвращает T, если выражение целое число, NIL в другом случае

Примеры

Удалить из исходного списка вещественные числа

```
 (setq res
 (remove-if-not 'integerp '(1 2 3.12 4 4.46 14 0.1)))
 (print "Список целых чисел")
 (princ res)
```

Результат на экране

(1 2 4 14)

(**float** <expr>) – Это число с плавающей точкой?

<expr> проверяемое выражение

Возвращает T, если выражение число с плавающей точкой, NIL в другом случае.

(**ratonalp** <expr>) – Это рациональное число?

<expr> проверяемое выражение

Возвращает T, если выражение ratonal (integer or rato), NIL в другом случае.

(**stringp** <expr>) – Это строка?

<expr> проверяемое выражение

Возвращает T, если выражение является строкой, NIL в другом случае.

(**charactrp** <expr>) – Это символ?

<expr> проверяемое выражение

Возвращает T, если выражение символ, NIL в другом случае

(**arrayp** <expr>) – Это массив?

<expr> проверяемое выражение

Возвращает T, если выражение является массивом, NIL в другом случае.

(**zerop** <expr>) – Это нуль?

<expr> число для проверки

Возвращает T, если число – нуль, NIL в другом случае

Примеры

Ожидание ввода с клавиатуры числа. Если число положительное, то выдается сообщение «Введено положительное число», если отрицательное – «Введено отрицательно число», если нуль – «Введен нуль».

```
(print "Введите число")        ; печать сообщения
(setq number(read))             ; считывание числа
(if (plusp number) (princ "Введено положительное число")
 (progn
  (if (zerop number) (princ "Введен нуль")
  (princ "Введено отрицательное число"))))
```

— Подсчет количества нулевых элементов в списке

```
(setq res (count-if 'zerop '(0 2 0 4 19 14 0 9)))
(princ "В списке имеется ")
(princ res)
(princ " нулевых элементов")
```

Результат на экране

В списке имеется 3 нулевых элемента

(**plusp** <expr>) – Это положительное число?

<expr> число для проверки

Возвращает T, если число положительное, NIL в другом случае

Примеры

Ожидание ввода с клавиатуры числа. Если число положительное, то выдается сообщение «Введено положительное число», если отрицательное – «Введено отрицательно число», если нуль – «Введен нуль».

```
  (print "Введите число")        ; печать сообщения
 (setq number(read))              ; считывание числа
 (if (plusp number) (princ "Введено положительное число")
  (progn
   (if (zerop number) (princ "Введен нуль")
   (princ "Введено отрицательное число"))))
```

(**oddp** <expr>) – Это нечетное целое?

<expr> число для проверки

Возвращает T, если целое число нечетное, NIL в другом случае

Примеры

Подсчет количества четных элементов в списке.

```
 (setq res (count-if-not 'oddp '(0 2 0 4 19 14 0 9)))
 (princ "В списке имеется ")
 (princ res)
 (princ " четных элементов")
```

Результат на экране

В списке имеется 6 четных элементов.

(**evenp** <expr>) – Это четное целое?

<expr> число для проверки

Возвращает T, если целое число четное, NIL в другом случае

Примеры

Определить позицию первого четного числа в списке

```
(setq res (position-if 'evenp '(1 4 5.13 0 8 6.7)))
(princ "Позиция первого четного числа в списке ")
(princ res)
```

Результат на экране

Позиция первого четного числа в списке 1

2.9. Функции ввода – вывода

(**read** [<stream> [<eofp> [<eof> [<rflag>]]]]) – Чтение выражения

<stream> входной поток (NIL, если *stardard-input и T, если *terminal-io*)

<eofp> значение T сигнализирует об ошибке конца файла, возвращает NIL, когда найден конец файла (по умолчанию T)

<eof> значение означающее конец файла (по умолчанию NIL)

<rflag> флаг рекурсивного чтения. Значение игнорируется

Возвращает прочитанное выражение

Примеры

```
(princ "Введите целое число от 0 до 9 – ")
(setq c1(read))
(princ "Вы ввели число ")
(princ c1)
```

Результат на экране

Введите целое число от 0 до 9 – 2

Вы ввели число 2

(**print** <expr> [<stream>])

Печать выражения с новой строки

Выражение печатается с помощью prin1, затем делается перевод каретки на следующую строку

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает выражение

Примеры

```
(setq list1 (list "F" "I" "L" "E"))
 (print list1)
 (terpri)
 (princ list1)
```

Результат на экране

```
("F" "I" "L" "E")
(F I L E)
```

(**prin1** <expr> [<stream>]) – Печать выражения

На форматирование влияют глобальные переменные: *print-level* и *print-length* для списков и массивов, *integer-format* для целых чисел, *floatformat* для чисел с плавающей запятой(ПЗ), *ratio-format* для отношений, *print-case* и *readtble-case* для имен переменных и строк.

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает выражение.

(**princ** <expr> [<stream>]) – Печать выражения без кавычек

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает выражение.

Примеры

```
(setq list1 (list "F" "I" "L" "E"))
 (print list1)
 (terpri)
 (princ list1)
```

Результат на экране

```
("F" "I" "L" "E")
(F I L E)
```

(**pprint** <expr> [<stream>]) – Структурная печать выражения

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает выражение.

(t**rpri** [<stream>]) – Переход на следующую строку

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает NIL.

(**fresh-line** <expr> [<stream>]) – Проверка начала строки

<expr> выражение, которое нужно распечатать

<stream> выходной поток (NIL, если *stardard-output* и T, если *terminal-io*)

Возвращает T, если перешли на новую строку, NIL, если курсор уже в начале строки.

2.10. Конструкции управления

(**cond** <pair>...) – Проверка условия

<pair> пара состоящая из (<pred> <expr>...),

где <pred> выражение сравнения

<expr> выражение, вычисляемое, если условие не равно NIL

Возвращает значение первого вычисляемого выражения, если условие не NIL.

Примеры

```
(setq numbers (list 2 -3 0 8 0 1 7))      ; описание списка
(setq num ( ))                            ; описание пустого списка
(dolist (elem numbers)                    ; цикл по списку
(cond ((< elem 0) (setq num (cons "NEG" num)))
 ; формирование нового значения
((= elem 0) (setq num (cons "ZERO" num)))   ; списка
((> elem 0) (setq num (cons "POS" num)))))
  (setq num (reverse num))
  ; перестановка элементов списка
  (print num)                             ; печать элементов списка
```

Результат на экране

```
("POS" "NEG" "ZERO" "POS" "ZERO" "POS" "POS")
```

(**and** <expr>...) – Логическое «И» над списком выражений

<expr> выражение, которое будет складываться по И

Возвращает NIL, если хотя бы одно вычисляемое выражение равно NIL, в другом случае возвращает значение последнего выражения (вычисления прекращаются после первого выражения равного NIL).

Примеры

```
(setq x 13 y NIL z NIL)  ; инициализация переменных
 (princ (and x y z))
```

Результат на экране

```
 NIL          ; так как есть выражения, значение которых равно NIL
(setq x 13 y 15 z 17)  ; инициализация переменных
(princ (and x y z))
```

Результат на экране

```
 17
; так как все выражения не равны NIL. 17 – значение
; последнего выражения
```

(**or** <expr>...) – Логическое «ИЛИ» над списком выражений

<expr> выражение, которое будет вычисляться по ИЛИ

Возвращает NIL, если все вычисляемые выражения равны NIL, в другом случае значение первого не NIL выражения (вычисления прекращаются после первого выражения не равного NIL).

Примеры

```
(setq x NIL y NIL z NIL)  ; инициализация переменных
(princ (or x y z))
```

Результат на экране

```
 NIL
```

```
; так как все выражения равны NIL
(setq x 13 y NIL z NIL)  ; инициализация переменных
 (princ (or x y z))
```

Результат на экране

```
13
; так как не все выражения равны NIL. 13 – значение
; последнего выражения, не равного NIL
```

(**if** <texpr> <expr1> [<expr2>]) – Вычисление условных выражений

<texpr> проверяемое выражение

<expr1> выражение, которое будет вычисляться, если результат сравнения не NIL

<expr2> выражение, которое будет вычисляться, если результат сравнения NIL

Возвращает значение вычисленного выражения.

Примеры

Ожидание ввода с клавиатуры числа. Если число положительное, то выдается сообщение «Введено положительное число», если отрицательное – «Введено отрицательно число», если нуль – «Введен нуль».

```
  (print "Введено число")        ; печать сообщения
 (setq number(read))              ; считывание числа
 (if (plusp number) (princ "Введено положительное число")
  (progn
   (if (zerop number) (princ "Введен нуль")
   (princ "Введено отрицательное число"))))
```

Результат на экране

```
Введено число минус 6
Введено отрицательное число.
```

(**when** <texpr> <expr>...) – Вычисляется, только если условие истинно

<texpr> проверяемое выражение

<expr> выражения, которые будут вычисляться, если результат сравнения не NIL

Возвращает значение последнего выражения или NIL.

Примеры

Подсчет количества элементов в списке, значение которых не превышает единицу

```
(setq num 0)
(setq numbers (list 2 -3 0 8 0 1 7))
(dolist (elem numbers)
(when (<= elem 1) (setq num (1+ num))))
(princ num)
```

Результат на экране

4

(**unless** <texpr> <expr>...) – Вычисляется, только если условие ложно

<texpr> проверяемое выражение

<expr> выражения, которые будут вычисляться, если результат сравнения NIL

Возвращает значение последнего выражения или NIL.

Примеры

Вывод на экран элементов, значение которых не больше единицы

```
(setq numbers (list 2 -3 0 8 0 1 7))
(dolist (elem numbers)
(unless (> elem 1) (print elem)))
```

Результат на экране

-3

0

0

1

(**case** <expr> <case>...[(T <expr>)]) – Выбор с помощью CASE

<expr> выражение выбора

<case> пара, состоящая из:

(<value> <expr>...),

где <value> одно выражение или список выражений

<expr> выражение, вычисляемое, если case-условие истинно

(T <expr>) выражение, вычисляемое по умолчанию (не было предыдущих совпадений)

Возвращает значение последнего вычисленного выражения.

Примеры

Ожидание ввода с клавиатуры числа в диапазоне от 1 до 3. Если введено число 1, выдается сообщение «Введено число 1», если число 2 – «Введено число 2», если число 3 – «Введено число 3». Если число не принадлежит указанному промежутку, то выдается сообщение «Число введено не верно».

```
(print "Введите число от 1 до 3")
(setq number(read))
(case number
      (1  (print "Введено число 1"))
            (2  (print "Введено число 2"))
```

(3 (print "Введено число 3"))

(T "Число введено не верно"))

Результат на экране

Введите число от 1 до 3

Введено число 1

(**let** (<binding>...) <expr>...) – Осуществляет локальное присвоение.

(**let*** (<binding>...) <expr>...) – Осуществляет последовательное присвоение.

<binding> переменная связывания, каждая из которых либо переменная (которая инициализирована в NIL), либо список, элементы которого будут служить выражениями инициализации

<expr> выражение, которое будет вычислено

Возвращает значение последнего выражения.

Область видимости переменных, проинициализированных с помощью let ограничена круглыми скобками, в которые заключена данная функция, т.е. попытка обращения к этим переменным из функции, расположенной за границами области действия let приводит к возникновению ошибки.

Примеры

(let

((number1 10) (flag Nil) (letters "A"))

; Присвоение значений переменным number1, flag и letters

; Эти переменные видимы только в области, ограниченной круглыми скобками, в которые заключена функция let, т.е. эти переменные могут быть использованы только в функциях, которые следуют за блоком инициализации переменных.

(setq list1 (list number1 flag letters))

```
; функции, в которых переменные видимы
    (print number1)
    (print list1))
```

Результат на экране

```
10
(10 NIL «A»)
```

В данном примере функция let* выполняет аналогичные действия

```
(let*
   ((number1 10) (flag Nil) (letters "A"))
    (setq list1 (list number1 flag leters))
    (print number1)
    (print list1))
```

Результат на экране

```
10
(10 NIL «A»)
```

Отличие функций let и let* демонстрируют следующие примеры.

Примеры

```
 (let ((x 2) (y (* 3 x)))
   (setq list1 (list x y))
   (princ list1))
```

Результат на экране

```
(2 39)
; результат верный, так как значения переменным x и
; y присваиваются одновременно (x=2 y=?)
(let* ((x 2) (y (* 3 x)))
   (setq list1 (list x y))
   (princ list1))
```

Результат на экране

(2 6)

; результат неверный, так как значения переменным x и

; y присваиваются последовательно.

(**flet** (<binding>...) <expr>...) – Создает локальную функцию

(**labels** (<binding>...) <expr>...) – Создает рекурсивную функцию

(**macrolet** (<binding>...) <expr>...) – Создает локальный макрос

<binding> функции связывания, каждая из которых:

(<sym> <fargs> <expr>...), где

<sym> имя функции или макроса

<fargs> список формальных аргументов (lambda список)

<expr> выражение, которое будет вычислено

Возвращает значение последнего выражения.

(**catch** <sym> <expr>...) – Вычисление выражений и захват исключений

<sym> имя исключения

<expr> вычисляемое выражение

Возвращает значение последнего вычисляемого выражения или выражения из throw.

(**throw** <sym> [<expr>]) – "Выбрасывание" исключения

<sym> имя исключения

<expr> значение для возврата catch (по умолчанию NIL)

Ничего не возвращает.

(**unwind-protect** <expr> <cexpr>...) – Защита вычисления выражений

<expr> защищаемое выражение

<cexpr> выражение, выполняемое после завершения вычисления <expr> с возвращением значения выражения.

2.11. Конструкции организации циклов

(**loop** <expr>...) – Основная форма цикла

<expr> тело цикла

Ничего не возвращает (должен использоваться нелокальный выход, такой, как RETURN).

Примеры

Ожидание ввода с клавиатуры целого числа в диапазоне от 0 до 9

; организация цикла, сообщение выдается до тех пор, пока не будет введено целое число в диапазоне от 0 до 9

```
(loop (princ " Введите целое число от 0 до 9 – ")
(setq number (read))
; вводимый символ сохраняется в переменной number
(if (and (integerp number)(and (<= number 9)(>= number 0)))
; если введено целое число не больше 9 и не меньше 0, выполняется следующая
; последовательность операторов
(progn
(princ "Введено число "
(princ number)          ; печать введенного числа
(return))               ; выход из цикла
; если введенный символ не удовлетворяет всем требованиям, то выполняются следующие операторы
(progn
    (princ "Число указано не верно") ; печать сообщения
```

```
        (terpri))))     ; переход на начало новой строки
```

Результат на экране

```
 Введите целое число от 0 до 9  –  13
Число указано не верно
Введите целое число от 0 до 9  –  7
Введено число 7
```

(**do** (<binding>...) (<texpr> <rexpr>...) <expr>...) – Общая форма цикла.

(**do*** (<binding>...) (<texpr> <rexpr>...) <expr>...) – Специальная форма функции **do** присваивает одновременно, **do*** присваивает последовательно <binding> значения переменных, каждая из которых:

1) переменная (которая инициализирована в NIL)

2) список следующей формы: (<sym> <iniT> [<step>]), где:

<sym> переменная для присваивания

<init> начальное значение переменной

<step> выражение шага

<texpr> выражение, определяющее условие завершения

<rexpr> выражение результата (по умолчанию NIL)

<expr> тело цикла (обрабатывается подобно неявному prog)

Возвращает значение последнего выражения результата.

Примеры

Формирование таблицы квадратных корней из целых чисел в диапазоне от 0 до 9

```
 (do ((i 0) (= i 10))
        (princ "X=")         ; печать сообщения
    (princ i)
(princ " SQRT(X)=")      ; печать сообщения
```

```
        (princ (sqrt i))
(terpri)
      (setq i(+ i 1)))
 ; изменение значения переменной i
```

Результат на экране

X=0 SQRT(X)=0

X=1 SQRT(X)=1

X=2 SQRT(X)=1.41421

X=3 SQRT(X)=1.73205

X=4 SQRT(X)=2

X=5 SQRT(X)=2.23607

X=6 SQRT(X)=2.44949

X=7 SQRT(X)=2.64575

X=8 SQRT(X)=2.82843

X=9 SQRT(X)=3

(**dolist** (<sym> <expr> [<rexpr>]) <expr>...) – Цикл по списку

<sym> переменная, принимающая значение каждого элемента списка

<expr> список выражений

<rexpr> выражение результата (по умолчанию NIL)

<expr> тело цикла (обрабатывается подобно неявному **prog**)

Возвращает значение выражения результата.

Примеры

Вывод на экран всех элементов списка строк

```
(setq seasons (list "Зима" "Весна" "Лето" "Осень"))
 ; описание элементов списка seasons
```

```
(dolist (elem seasons)      ; цикл по списку seasons
  ;  переменная elem принимает значение очередного элемента списка
  (princ elem)   ; печать очередного элемента
  (terpri))        ; переход на начало следующей строки
```

Результат на экране

```
  Зима
  Весна
  Лето
  Осень
```

Поиск максимального значения в списке чисел

```
(setq list1 (list 5 18 9 22 40 13))
  ;описание элементов списка list1
(setq maximum (nth 0 list1))
  ; идентификатору maximum присваивается значение первого элемента списка list1
(dolist (elem list1)          ; цикл по списку list1
 ; переменная  elem принимает значение очередного элемента
(if (< maximum elem) (setq maximum elem)))
```

; если очередной элемент elem списка больше значения переменной maximum, то ее значение заменяется на значение элемента elem.

```
 (princ maximum)
 ; вывод на экран полученного значения
```

Результат на экране

```
   40
```

(**dotimes** (<sym> <expr> [<rexpr>]) <expr>...) – Цикл от ноля до N-1

<sym> переменная, от 0 до N-1

<expr> число итераций (N)

<rexpr> выражение результата (по умолчанию NIL)

<expr> тело цикла (обрабатывается подобно неявному prog)

Возвращает значение выражения результата.

Примеры

Вычисление значения 2^4

```
(setq num 2)          ; переменной num присваивается значение 2
(setq res 1)          ; переменной res присваивается значение 1
(dotimes ( i 4 ) ; цикл от 0 до 3
; тело цикла
  (setq res (* res num)))   ; вычисление нового значения переменной res
  (princ res)               ; печать результата
```

Результат на экране

16

Выборка из исходного списка элементов, значение которых не меньше 5

```
 (setq list1 (list 8 2 0 5 7 1))              ; описание списка list1
 (setq list2 ( ))                             ; описание пустого списка list2
 (dotimes (i (list-length list1))
; цикл от 0 до (число элементов минус 1)
 ; тело цикла
 (setq elem (nth i list1))  ; переменной elem  присваивается значение
 ; очередного элемента списка list1
  (if (>= elem 5) (setq list2 (cons elem list2))))
```

; если значение очередного элемента больше 5, элемент добавляется в конец списка list2

```
 (princ list2); вывод элементов сформированного списка на экран
```

Результат на экране

(7 5 8)

3. РАЗРАБОТКА ЛОГИЧЕСКИХ ПРОГРАММ

Примеры разработки логических программ на функциональном языке XLISP приведены в приложении 1-3.

В начале необходимо составить логическую схему решения задачи. Затем с помощью функции defun создать макрофункции проверки условий решения логической задачи, обработки и вывода результатов.

Напрмер, требуется найти такую последовательность ходов конём, при которой он становится на каждую клетку только один раз. Размер шахматной доски 4*5. Задать исходной состояние коня. Необходимо сформировать логические правила поиска решения и в отдельном окне вывести допустимые ходы. Исходный код программы приведен в приложении 2.

Для того, чтобы решить данную задачу, необходимо хранить информацию о состоянии всех клеток (была посещена или нет), а также необходимо задать список правил, которые описывают все возможные ходы. Так как LISP обеспечивает ряд функций для работы со списком, поле можно представить в виде списка, состоящего из списка элементов. Для доступа к элементам необходимо будет реализовать функции записи и чтения из нужной ячейки. Для проверки, доступна ли для посещения клетка, достаточно посмотреть на значение соответствующей ячейки, каждый ход сопровождается записью в нужную ячейку. Каждая клетка задаётся двумя координатами, что позволяет избежать описания всех возможных ходов для каждой клетки. Вместо этого достаточно описать возможные направления и правила для каждого направления. Вначале необходимо проверить, не приведёт ли переход к выходу за пределы поля. После этого проверяется, доступна ли клетка. Если да, то функция запускается для новой клетки с

увеличенным числом сделанных ходов и обновлённым списком. В случае, если число ходов достигает 20, алгоритм выводит ответ и завершает работу.

Для реализации алгоритма были написаны правила перехода, а также правила определения границы, правила проверки доступности клетки, условие выхода из алгоритма и функция вывода информации из списка. Экранные формы представлены на рис. 3.1 и рис. 3.2.

В данной вариации программы запись посещённых клеток производилась не в список, а в массив, поэтому доступ к элементам получился более удобным. Язык XLISP обладает большим функционалом, чем язык PROLOG, однако непривычен из-за обилия скобок и указания названия функции внутри скобок. Отсутствие или наличие в ненужном месте скобок может привести к ошибке (например, неправильный тип данных), которую затем сложно выявить.

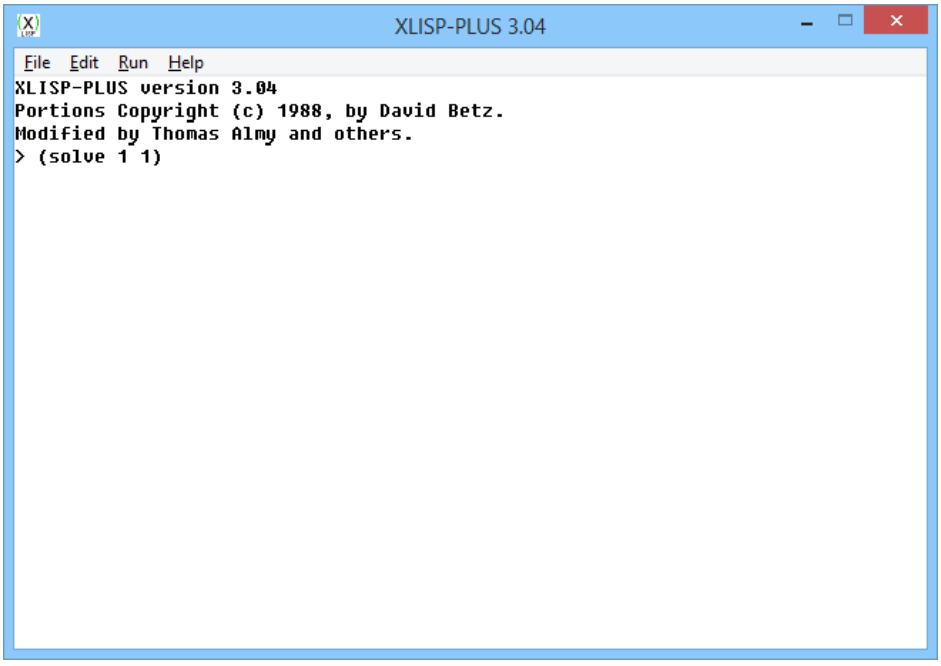

Рис. 3.1. Команда для запуска алгоритма поиска

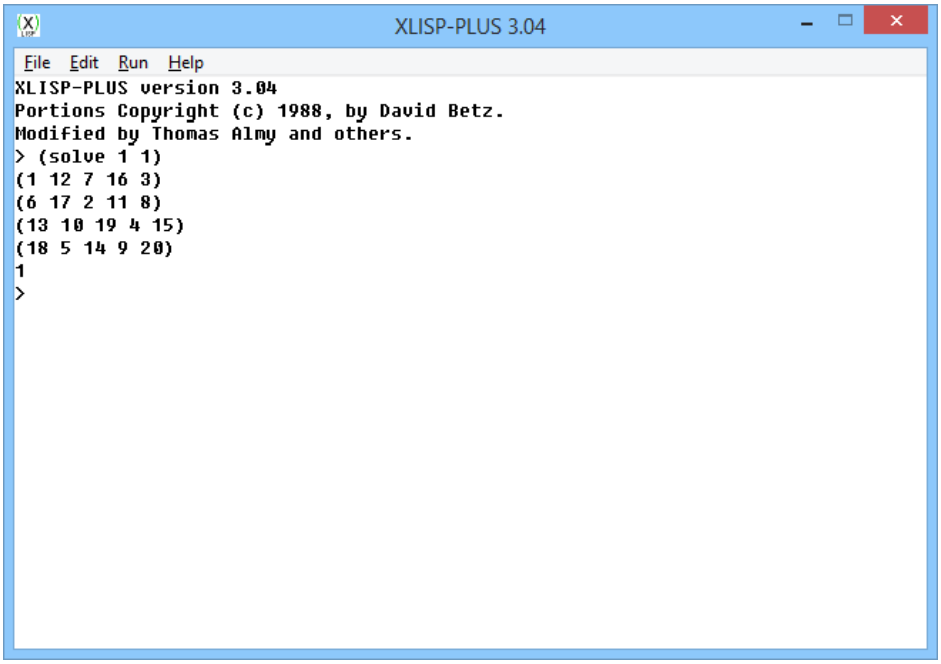

Рис. 3.2. Результат работы программы

4. ОРГАНИЗАЦИЯ СВЯЗИ XLISP СО СРЕДОЙ DELPHI

В связи с тем, что язык XLISP не предоставляет достаточных средств для создания полноценного интерфейса пользователя и возможность визуальной разработки отсутствует, возникает проблема “связывания” программы XLISP с другими средами программирования, на которые можно переложить интерфейсные и графические функции.

Данная методика годится для программ, которые условно можно разделить на три части:

– ввод исходных данных;

– расчет;

– вывод результатов.

Далее предполагается, что такая программа написана и отлажена.

Изменяем эту программу так, чтобы входные данные можно взять из жестко заданных констант

Пример

```
(setq p1 0.00)
        (setq p2 0.00)
```

Вывод результатов осуществляется не на экран, а в файл.

Пример

```
(setq fp (open "output.res" :direction :output :if-exists :supersede))
    (print hyp1 fp)
    (print hyp2 fp)
    (close fp)
```

В самые последние строки программы помещаются строки, запускающие процедуру расчета (главную процедуру), а так же команду выхода.

Пример

```
(main)
(exit)
```

Создаем программу в среде Delphi с необходимым нам интерфейсом пользователя, позволяющую:

– вводить исходные данные в некоторые внутренние переменные;

– интерпретировать результирующие данные, которые находятся в файле результата, созданного программой на XLISP;

– выводить результат работы в нужной форме.

Приложение 1

Листинг программы на языке XLISP решения логической задачи перевоза волка, козы и капусты с левого берега на правый. В программе запрещается оставлять на берегу волка с козой и козу с капустой

```
 (defun make_state (f w g c)
  (list f w g c)
)
(defun farmer_side (state)
  (nth 0 state)
)
(defun wolf_side (state)
  (nth 1 state)
)
 (defun goat_side (state)
  (nth 2 state)
)
(defun cabbage_side (state)
  (nth 3 state)
)

(defun opposite (side)
  (cond
    (
      (equal side 'e) 'w
    )
```

```
    (
      (equal side 'w) 'e
    )
  )
)
(defun safe (state)
  (cond
    ((and
      (equal (goat_side state) (wolf_side state))
      (not (equal (farmer_side state) (wolf_side state)))
    ) nil)
    ((and
      (equal (goat_side state) (cabbage_side state))
      (not (equal (farmer_side state) (goat_side state)))
    ) nil)
    (t state)
  )
)
(defun farmer_takes_self (state)
  (print "farmer_takes_self")
  (safe
    (make_state
      (opposite
        (farmer_side state)
      )
      (wolf_side state)
      (goat_side state)
```

```
      (cabbage_side state)
    )
  )
)
(defun farmer_takes_wolf (state)
  (print "farmer_takes_wolf")
  (cond
    (
      (equal (farmer_side state) (wolf_side state))
      (safe
        (make_state
          (opposite
            (farmer_side state)
          )
          (opposite
            (wolf_side state)
          )
          (goat_side state)
          (cabbage_side state)
        )
      )
    )
    (t nil)
  )
)
(defun farmer_takes_goat (state)
  (print "farmer_takes_goat")
```

```
  (cond
    (
      (equal (farmer_side state) (goat_side state))
      (safe
        (make_state
          (opposite
            (farmer_side state)
          )
          (wolf_side state)
          (opposite
            (goat_side state)
          )
          (cabbage_side state)
        )
      )
    )
    (t nil)
  )
)
(defun farmer_takes_cabbage (state)
  (print "farmer_takes_cabbage")
  (cond
    (
      (equal (farmer_side state) (cabbage_side state))
      (safe
        (make_state
          (opposite
```

```
          (farmer_side state)
         )
         (wolf_side state)
         (goat_side state)
         (opposite
          (cabbage_side state)
         )
        )
       )
     )
     (t nil)
   )
)
(defun path (state goal been_list)
  (print state)
  (cond
    ((null state) nil)
    ((equal state goal) (reverse (cons state been_list)))
    ((not (member_lis state been_list))
      (or
        (path (farmer_takes_self state) goal (cons state been_list))
        (path (farmer_takes_wolf state) goal (cons state been_list))
        (path (farmer_takes_goat state) goal (cons state been_list))
        (path (farmer_takes_cabbage state) goal (cons state been_list))
      )
    )
  )
```

```
)
(defun member_lis (x lis)
  (cond
    ((null lis) nil)
    ((equal x (car lis)) t)
    (t (member_lis x (cdr lis)))
  )
)
(defun solve_fwgc (state goal)
  (path state goal nil)
)
(solve_fwgc '(w w w w) '(e e e e))
```

Приложение 2

Листинг программы на языке XLISP решения логической задачи нахождения последовательности ходов конём на шахматной доске размером 4x5

```
;функция проверки выхода за пределы поля
(defun check(a b)
	(
		cond
		(
			(and	(and (< 0 a) (> 5 a))
				(and (< 0 b) (> 6 b)))
			1
		)
		(
			T 0
		)
	)
)
;функция чтения из указанной ячейки матрицы поля
(defun getvar(mas i j)
	(
		cond
		(
			(= 1 (check (1+ i) (1+ j)))
			(nth j (nth i mas))
		)
		(
			T -1
		)
	)
)
;функция записи в указанную ячейку матрицы поля
```

```
(defun setvar(mas i j var)
	(
		progn
		(setq mass NIL)
		(dotimes (K 4)
			(setq A NIL)
			(dotimes (L 5)
				(if
					(and (= i (1+ K)) (= j (1+ L)))
					(progn
						(setq A (append A (list var)))
					)
					(progn
						(setq A (append A (list (getvar mas K L))))
					)
				)
			)
			(setq mass (append mass (list A)))
		)
		mass
	)
)

;инициализация исходной матрицы
(defun initmas()
	(
		progn
		(setq mas NIL)
		(dotimes (K 4)
			(setq A NIL)
			(dotimes (L 5)
				(setq A (append A '(0)))
			)
			(setq mas (cons A mas))
		)
```

```
            mas
        )
)
;основная функция решения задачи
(defun state(tt mas a b)
        (
            progn
            (setq q 0)
;Проверка успешного обхода – если было совершено 20 ходов, то выводим ответ и выходим
            (if (= tt 20)
                (
                    progn
                    (setq mas3 (setvar mas a b tt))
                    (print (nth 0 mas3))
                    (print (nth 1 mas3))
                    (print (nth 2 mas3))
                    (print (nth 3 mas3))
                    (setq q 1)
                )
                (
                    let
                    ((mas2 (setvar mas a b tt)))
                    (
                        progn
                        (setq dx (list -2 -2 -1 -1 1 1 2 2))
                        (setq dy (list 1 -1 2 -2 2 -2 1 -1))
                        (dotimes (K 8)
                            ; Проверка на выход за пределы поля и возможность посещения клетки
                            (if (and (= q 0) (and (= 1 (check (+ a (nth K dx)) (+ b (nth K dy)))) (= 0 (getvar mas2 (1- (+ a (nth K dx))) (1- (+ b (nth K dy)))))))
                                (
```

```
                                        ; рекурсивный запуск с увеличенным значением tt
                                        progn
                                        (setq q (state (1+ tt) mas2 (+ a (nth K dx)) (+ b (nth K dy))))
                                    )
                                )
                            )
                        )
                    )
                )
                q
        )
)

(defun solve(x y)
        (
                state 1 (initmas) x y
        )
)

(defun main()
        (
                state 1 (initmas) 1 1
        )
)
```

Приложение 3

Листинг программы на языке XLisp решения логической задачи в виде криптарифма.

Криптарифм (cryptarithm) – это математический ребус, в котором зашифрован пример на выполнение одного из арифметических действий. При этом одинаковые цифры шифруются одной и той же буквой, а разным

цифрам соответствуют различные буквы. Считается, что никакое число не должно начинаться с нуля. Криптарифм можно считать хорошим, если в результате шифрования получилась какая-то осмысленная фраза. Например, классическим криптарифмом является пример на сложение, придуманный Генри Э. Дьюдени еще в начале нашего века: SEND + MORE = MONEY. Кроме того, еще одно требование к правильному криптарифму: он должен иметь единственную возможную расшифровку. Например, единственным решением криптарифма Дьюдени является 9567 + 1085 = 10652.

Алгоритм решения криптарифма, содержащего N различных букв, будет следующим:

1) выписать все уникальные буквы в фиксированной последовательности;

2) сгенерировать новую перестановку цифр от 0 до 9, такую, чтобы комбинация первых N цифр ни разу ранее не встречалась;

3) провести соответствие между последовательностью букв и первыми N цифрами перестановки;

4) проверить, выполняется ли заданное в криптарифме равенство; если нет – перейти на шаг 2;

5) ответ получен.

Для генерации уникальных перестановок забирается по очереди каждый из K элементов и добавляя к нему все возможные перестановки K-1 элементов.

Листинг программы

```
; Переводит строку s в список символов
(defun str-to-list(s) (
progn
(setq res (list))
```

```
(do ((i 0)) ((= i (length s))) (
        progn
        (setq res (append res (list (char s i))))
        (setq i (+ i 1))
))
res
))
; Делает из списка строк одну длинную строку
(defun concat-string-list (list) (
progn
(setq res "")
(dolist (item list)    (setq res (concatenate 'string res item)))
res
))
; Получает список уникальных символов из списка строк
(defun get-unique-letters(strList) (
progn
(setq tmp (concat-string-list strList))
(setq res (list))
(do ((i 0)) ((= i (length tmp))) (
        progn
        (setq cur (char tmp i))
        (if (not (find cur res)) (setq res (append res (list cur))))
        (setq i (+ i 1))
))
res
))
```

```
; Получает цифру, соотвествующую букве с согласно переданным perm и letters (текущая перестановка и массив уникальных букв)
(defun find-number(c perm letters) (nth (position c letters) perm))
; Переводит буквенный код в число, согласно переданным perm и letters
(defun convert(code perm letters) (
progn
(setq codeLetterList (str-to-list code))
(setq res 0)
(do ((i 0)) ((= i (list-length codeLetterList))) (
        progn
        (setq res (+ (* res 10) (find-number (nth i codeLetterList) perm letters)))
        (setq i (+ i 1))
))
res
))
; Возвращает новый список, в котором элемент n равен val
(defun change-nth(list n val) (
progn
(setq res (list))
(setq i 0)
(dolist (cur list) (
        progn
        (if (= i n) (setq res (append res (list val))) (setq res (append res (list cur))))
        (setq i (+ i 1))
))
res
))
```

```
; Проверяет, что при переданных perm и letters старшие цифры в числах не равны 0
(defun bounds(s1 s2 r perm letters) (
progn
(setq s1l0 (char s1 0))
(setq s2l0 (char s2 0))
(setq rl0 (char r 0))
(setq res (and (> (find-number s1l0 perm letters) 0) (> (find-number s2l0 perm letters) 0) (> (find-number rl0 perm letters) 0)))
res
))
; Переводит коды s1, s2 и r в числа, согласно переданным perm и letters, и проверяет сумму r == s1 + s2.
; Если проверка верна, то печатает ответ в консоль
(defun check(s1 s2 r perm letters) (
progn
(setq s1n (convert s1 perm letters))
(setq s2n (convert s2 perm letters))
(setq rn (convert r perm letters))
(setq res (= rn (+ s1n s2n)))
	(if res (
	progn
	(princ s1n)(princ " + ")(princ s2n)(princ " = ")(princ rn)
))
	res
))
```

```
; Функция для генерации перестановок из начального списка perm и их проверки.
(defun permute-and-check (s1 s2 r perm letters depth) (
progn
(setq len (list-length perm))
(if (= depth (- len (list-length letters)))
    (           ; Проверка перестановки
        return-from permute-and-check (and (bounds s1 s2 r perm letters) (check s1 s2 r perm letters))
    ) (     ; Генерация "дочерних" перестановок
        progn
        (if (permute-and-check s1 s2 r perm letters (- depth 1))
            (return-from permute-and-check T)
            (                                   progn
                (do ((i (+ (- len depth) 1))) ((= i len)) (
                    progn       (setq tmp (nth i perm))
                (setq perm (change-nth perm i (nth (- len depth) perm)))
                (setq perm (change-nth perm (- len depth) tmp))
                    (if (permute-and-check s1 s2 r perm letters (- depth 1)) (return-from permute-and-check T))
                        (setq i (+ i 1))
                ))
            )
        )
    )
))
```

; Фунция для решения криптарифм вида А + Б = В. Использование: чтобы решить

; SEND + MORE = MONEY, нужно вызвать (solve "SEND" "MORE" "MONEY").

```
(defun solve(s1 s2 r) (
progn
(setq letters (get-unique-letters (list s1 s2 r)))
(setq perm (list 0 1 2 3 4 5 6 7 8 9))
(permute-and-check s1 s2 r perm letters 10)
 ))
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Хювёнен, Э. Мир Лиспа [Текст] : пер. с финск. : в 2-х т. / Э. Хювёнен, Й. Сеппянен. – Москва : Мир, 1990. – 2 т.

2. Очень краткое введение в язык Лисп [Электронный ресурс]. - Режим доступа : http://homelisp.ru/help/lisp.html. – 21.03.2018.

3. Логическое и функциональное программирование [Электронный ресурс]. – Режим доступа : http://mirznanii.com/a/309803/logicheskoe-i-funktsionalnoe-programmirovanie. – 21.03.2018.

Учебное издание

Ростовцев Владимир Сергеевич

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ XLISP

Учебное пособие

Авторская редакция

Технический редактор А. Е. Свинина

Подписано в печать 20.03.2018. Печать цифровая. Бумага для офисной техники.
Усл. печ. л. 6,21. Тираж 5 экз. Заказ № 4979.

Федеральное государственное бюджетное образовательное учреждение высшего образования «Вятский государственный университет».

610000, г. Киров, ул. Московская, 36, тел.: (8332) 74-25-63, http://vyatsu.ru