МИНИСТЕРСТВО ОБРАЗОВАНИЯ РФ
МИНИСТЕРСТВО НАУКИ И ТЕХНОЛОГИИ РФ
МОСКОВСКИЙ КОМИТЕТ ОБРАЗОВАНИЯ
ДЕПАРТАМЕНТ ОБРАЗОВАНИЯ ГОРОДА МОСКВЫ

МОСКОВСКИЙ ГОРОДСКОЙ ДВОРЕЦ ДЕТСКОГО (ЮНОШЕСКОГО) ТВОРЧЕСТВА

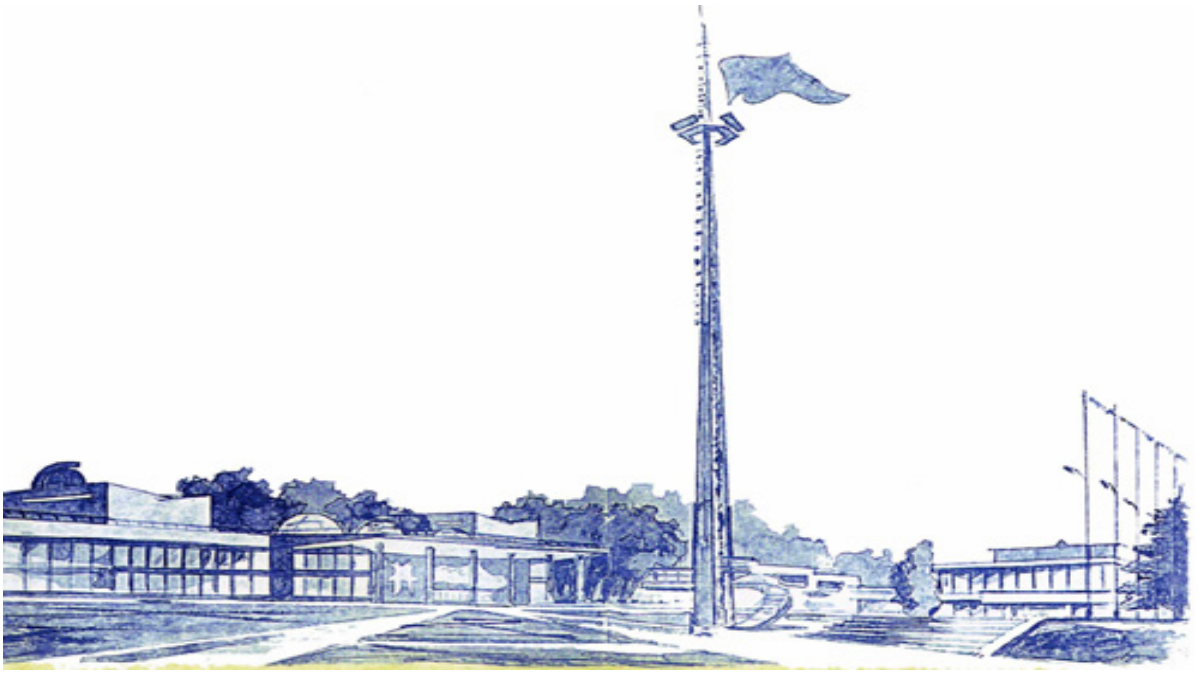
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ (ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)

ГОСУДАРСТВЕННЫЙ НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ТЕЛЕКОММУНИКАЦИЙ “ИНФОРМИКА”

Л.З. Яшин

MICRO LISP. ОСНОВНЫЕ ПОНЯТИЯ, СТРУКТУРЫ И ФУНКЦИИ

под редакцией А.Ю. Паршина



Москва 2004

УДК 9 (2) 240 2

Главный редактор: Первый зам. директора МГДД(Ю)Т В.Е. Соболев
Рук. эксп. техн. комплекса: В.И. Минаков
Литературный редактор: Л.А. Карась
Технологическое обеспечение: С.В. Свечников
Выпускающий редактор: С.В. Свечников

Яшин Л.З. Micro Lisp. Основные понятия, структуры и функции. Под редакцией А.Ю. Паршина – МИРЭА, МГДД(Ю)Т, М., 2004, - 42 с.

В программировании на языке LISP используются символы и символьные структуры, построенные из них. Под символом подразумевается запись или обозначение.

Символ – это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, вещь, действие из реального мира. В LISP символы обозначают числа, другие символы или более сложные структуры, программы, функции и другие объекты языка. Например, символ «+» может обозначать определение операции сложения, «углерод-14» - изотоп урана и т.п.

Методическая разработка внедрена в учебно-творческий процесс МГДД(Ю)Т (сектор Информатики и ВТ) и в МИРЭА.

По специальностям (специализациям):

- 071900 «Информационные системы в образовании»
- 073700 «Информационные технологии в образовании»
- 071903 «Информационные системы в науке и образовании»
- в дополнительном образовании по сектору Информатики и ВТ МГДД(Ю)Т

База данных размещена на сервере Технологической экспериментальной площадки ГНИИ ИТТ “Информика” - МГДД(Ю)Т – МИРЭА (http://www.mgdtd.ru/). Соответствующий автоматизированный глоссарий встроен в ядро информационной системы дополнительного образования московского региона под управлением Lotus Notes.

ББК 3288-421

Лицензия на издательскую деятельность: ЛР №040686 от 27 мая 1999

Адрес в МГДД(Ю)Т: email – cnit@mgdtd.ru 119991, Москва, ул. Косыгина, д.17, комн. 4-21, 4-31.
Адрес в МИРЭА: email – cnit@mirea.ru 117454, Москва, пр-т Вернадского, д. 78.

МГДД(Ю)Т Заказ _____ Тираж 35

ОГЛАВЛЕНИЕ

КРАТКАЯ СПРАВКА ИЗ ТЕОРИИ

Символы

В программировании на языке LISP используются *символы* и *символьные структуры,* построенные из них. Под *символом* подразумевается запись или обозначение.

Символ – это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, вещь, действие из реального мира. В LISP символы обозначают числа, другие символы или более сложные структуры, программы, функции и другие объекты языка. Например, символ «+» может обозначать определение операции сложения, «углерод-14» - изотоп урана и т.п.

Примеры символов:

x
symbol
defun
SteP-1984

Символы могут состоять как из прописных, так и из строчных букв. Однако, в большинстве LISP-систем, все буквы отождествляются с прописными. Поэтому:
symbol ⇔ Symbol ⇔ SYMBOL

Числа

Кроме символов в языке LISP используются также и *числа*, которые, как и символы, записываются при помощи последовательности знаков, ограниченных с двух сторон пробелами. Числа не являются символами, они не могут представлять никаких объектов кроме себя (своего числового значения).

Примеры чисел:

134	целое число
-1.34	десятичное число
1.34E8	число, представленное мантиссой и порядком

Логические значения T и NIL

Символы *T* и *NIIL* имеют в LISP специальное значение. T – логическое значение «истина» (true), *NIL* – логическое значение «ложь» (false). Символом NIL обозначается еще и пустой список. Эти символы всегда имеют только такое значение. Их нельзя переопределить.

Константы и переменные

Числа и логические значения T и NIL являются константами, остальные символы – переменными, которые используются для обозначения других объектов языка.

Атомы

Атомы – это символы и числа. Это простейшие объекты языка LISP, из которых строятся остальные структуры. Их называют «атомарными объектами» или просто атомами.

Списки

Атомы и списки – основные типы данных языка LISP. Список – упорядоченная последовательность, элементами которой являются атомы или списки (подсписки).

Списки заключаются в круглые скобки, элементы списков разделяются пробелами. Список начинается с открывающей скобки и заканчивается закрывающей скобкой.

Например:

(a b (c d) e) - список, состоящий из трех атомов и одного подсписка

То есть, список – многоуровневая (иерархическая) структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии.

Примеры списков:

(+ 2 3) - список из трех элементов

(((((первый) 2) третий) 4) 5) - список из двух элементов

Пустой список (NIL)

Список, в котором нет ни одного элемента, называется «пустым» и обозначается *()* или символом *NIL*. Пустой список – не то же самое, что «ничего». Он похож на нуль в арифметике. NIL может также быть элементом других списков.

Например:

NIL - то же, что и ()

(NIL) - список, состоящий из атома NIL

(()) - то же, что и (NIL)

((())) - то же, что и ((NIL))

(NIL ()) - список из двух пустых списков

Атомы и списки называются *символьными выражениями* (*s-выражениями*).

Префиксная нотация

Как правило, в алгоритмических языках используется такой способ записи вызова функции:

f(x, y)

Однако в LISP используется немного другой, возможно кажущийся непривычным, способ записи:

(f x y)

То есть, как название функции, так и ее аргументы записываются в виде списка, первым элементом которого является название функции.

Арифметические функции

Арифметические операции над числами представлены как функции «+», «-», «/», «*».

(+ 5 2) - 5+2=7

(- 10 1) - 10-1=9

(/ 20 2) - 20/2=10

(* 2 3) - 2*3=6

(* (+ 1 2) (* 1 (+ 2 3))) - (1+2)*(1*(2+3))=15

QUOTE

QUOTE блокирует вычисление выражения. В ряде случаев нужно не вычислять значение выражения, а оперировать с ним как с списком. Тогда и используется этот предикат. Имеется сокращенный вариант записи этого предиката – «’». То есть:

выражение ⇔ (QUOTE выражение).

Примеры:

[0] (+ 2 3)

5

[1] (QUOTE (+ 2 3))
(+ 2 3)
[2] '(+ 2 3)
(+ 2 3)

ПРИМИТИВЫ ЯЗЫКА MICRO-LISP

Для построения, разбора и анализа списков существуют очень простые базовые функции, которые в этом языке являются примитивами. В определенном смысле они образуют систему аксиом языка (алгебру обработки списков), к которым в конце концов сводятся символьные выражения. Базовые функции обработки списков можно сравнить с основными действиями в арифметических вычислениях или в теории чисел.

Простота базовых функций и их малое число - это характерные черты Лиспа. С этим связана математическая элегантность языка. Разумно выбранные примитивы образуют, кроме красивого формализма, также и практическую основу программирования.

Базисными функциями обработки символьных выражений (атомов и списков) являются:

CAR, CDR, CONS, ATOM? и EQ?

Функции по принципу их использования можно разделить на функции разбора, создания и проверки:

ИСПОЛЬЗОВАНИЕ ВЫЗОВОВ		РЕЗУЛЬТАТ
Разбор:	(CAR список)	*s-выражение*
	(CDR список)	*список*
Создание:	(CONS *s-выражение список)*	*список*
Проверка:	(ATOM? *s-выражение)*	T или NIL
	(EQ? *символ символ)*	T или NIL

У функций CONS и EQ? имеются два аргумента, у остальных примитивов — по одному. В качестве имен аргументов и результатов функций мы использовали названия типов, описывающих аргументы, на которых определена (т. е. имеет смысл) функция и вид возвращаемого функциями результата. S- выражение обозначает атом или список.

ОСНОВНЫЕ ФУНКЦИИ ОБРАБОТКИ СПИСКОВ

Функция CAR возвращает в качестве значения головную часть списка

Первый элемент списка называется головной, а остаток списка, т. е. список без первого его элемента, называется хвостом списка. Функция CAR имеет смысл только для аргументов, являющихся списками, а следовательно, имеющих голову:

car: список -> s-выражение

Для аргумента атома результат функции CAR не определен, и вследствие этого появляется следующее сообщение об ошибке:

```
[0] (car 'дом)
[VM ERROR encountered!] Invalid operand to VM instruction
(CAR ДОМ)
```

(т. е. ДОМ не является списком).

Головной частью пустого списка считают для удобства NIL:

```
[1] (car nil)          ;голова пустого списка
NIL                    ;пустой список
[2] (car 'nil)         ; знак ' можно опускать
NIL
[3] (car '(nil a))     ; голова списка NIL
NIL
```

Примеры:

```
[0] (car '(6 7 8))
6
[1] (car '(s1 s2))
S1
[2] (car '('first 'second))
(QUOTE FIRST)
[3] (car '(car (a b c)))
CAR
[4] (car (cons 'tail 'bar))
TAIL
[5] (car ())
()
```

Функция CDR возвращает в качестве значения хвостовую часть списка

Функция CDR применима к спискам. Значением ее будет хвостовая часть списка, т. е. список, получаемый из исходного списка после удаления из него головного элемента:

cdr: список -> список

Функция CDR не выделяет второй элемент списка, а берет весь остаток списка, т. е. хвост. Заметим, что хвост списка— тоже список,' если только список не состоял из одного элемента. В последнем случае хвостом будет пустой список (), т. е. NIL:

[1] (cdr '(a))

()

Из соображений удобства значением функции CDR от пустого списка считается NIL:

[2] (cdr nil)

()

Так же как и CAR, функция CDR определена только для списков. Значение для атомов не определено, что может приводить к сообщению об ошибке:

[3] (cdr 'дом)

[VM ERROR encountered!] Invalid operand to VM instruction

(CDR ДОМ)

Примеры:

[0] (cdr '(s1 s2))

(S2)

[1] (cdr '(a b c))

(B C)

[2] (cdr '(a (b c)))

((B C))

[3] (cdr '((a) b c d))

(B C D)

[4] (car (cdr '(ab cd ef)))

CD

Функция CONS включает новый элемент в начало списка

Функция CONS строит новый список из переданных ей в качестве аргументов головы и хвоста:

(CONS *голова хвост)*

Функция добавляет новое выражение в список в качестве первого элемента:

[1] (cons 'a '(b c))

(A B C)

[2] (cons '(a b) '(c d))

((A B) C D)

[3] (cons (+ 1 2) '(+ 4))

(3 + 4)

Для того, чтобы можно было включить первый аргумент функции CONS в качестве первого элемента значения второго аргумента этой функции, второй аргумент должен быть списком. Значением функции CONS всегда будет список:

cons: s-выражение x *список* --> *список*

Примеры

[1] (cons ‘a '())
(A)
[2] (cons "a" '(b c))
("a" B C)
[3] (cons 'a 3)
(A . 3)
[4] (cons 'a '3)
(A . 3)
[5] (cons '(a) '(3))
((A) 3)
[6] (cons 'a '(3))
(A 3)
[7] (cons nil '(b c d))
(() B C D)
[8] (cons '(b c d) nil)
((B C D))
[9] (cons 'nil '(b c d))
(NIL B C D)
[10] (cons nil nil)
(())
[11] (cons 'nil 'nil)
(NIL . NIL)
[12] (cons '(nil) '(nil))
((NIL) NIL)

ПРЕДИКАТЫ ЯЗЫКА MICRO-LISP

Предикат проверяет наличие некоторого свойства

Чтобы осуществлять допустимые действия со списками и избежать ошибочных ситуаций, нам необходимы, кроме селектирующих и конструирующих функций, средства опознавания выражений. Функции, решающие эту задачу, в Лиспе называются предикатами.

Предикат — это функция, которая определяет, обладает ли аргумент определенным свойством и возвращает в качестве значения логическое значение «ложь», т. е. NIL, или «истина», которое может быть представлено символом T или любым выражением, отличным от NIL.

ATOM? и EQ? являются базовыми предикатами Лиспа. С их помощью и используя другие базовые функции, можно задать более сложные предикаты, которые будут проверять наличие более сложных свойств.

Предикат ATOM? проверяет, является ли аргумент атомом

При работе с выражениями необходимо иметь возможность проверить, является ли выражение атомом или списком. Это может потребоваться, например, перед применением функций CAR и CDR, так как эти функции определены лишь для аргументов, являющихся списками. Базовый предикат ATOM? используется для идентифицирования лисповских объектов, являющихся атомами:

(ATOM? *s-выражение)*

Значением вызова ATOM? будет T, если аргументом является атом, и () — в противном случае:

```
[0] (atom? 'x)
#T
[1] (atom? '(a b c))
()
[2] (atom? '(Я программирую - следовательно существую))
()
[3] (atom? (cdr '(a b c)))
()
[4] (atom? (car '(a b c)))
#T
[5] (atom? (+ 5 6))
#T
```

С помощью предиката ATOM? можно убедиться, что пустой список NIL, или (), является атомом:

```
[0] (atom? nil)
#T
[1] (atom? 'nil)
#T
[2] (atom? ())
#T
[3] (atom? '(nil))
()
[4] (atom? (atom? (+ 2 3)))
#T
[5] (atom? (atom? '(+ 2 3)))
#T
[6] (atom? '(atom? (+ 2 3)))
()
```

В Лиспе существует целый набор предикатов, проверяющих тип являющегося аргументом выражения или любого другого лисповского объекта и таким образом идентифицирующих используемый тип данных.

EQ? проверяет тождественность двух символов

Предикат EQ? сравнивает два символа и возвращает значение T, если они идентичны, в противном случае — NIL:

```
[0] (eq? 'x 'кот)
()
[1] (eq? 'x 'x)
#T
[2] (eq? '235 '235)
#T
[3] (eq? 235 235)
#T
[4] (eq? 235.5 235.5)
()
[5] (eq? 'кот (car '(кот пес)))
#T
[6] (eq? () nil)
#T
[7] (eq? t 't)
()
[8] (eq? 't 't)
#T
[9] (eq? t t)
#T
[10] (eq? t (atom? 'мышь))
#T
```

Предикат EQ? накладывает на свои аргументы строго определенные требования. С его помощью можно сравнивать только символы или константы T и NIL, и результатом будет значение T лишь в том случае, когда аргументы совпадают. Для проверки чисел в Лисп-Микро EQ? не используется. Предикатов ATOM? и EQ?, несмотря на простоту EQ?, вполне достаточно для работы со списками.

Предикат EQ? в Лисп - системах обычно таков, что его можно применять к списочным и числовым аргументам, не получая сообщения об ошибке; он не проверяет логического равенства чисел, строк или других объектов, а лишь смотрят, представлены ли лисповские объекты в памяти вычислительной машины физически одной и той же структурой. Одноименные символы представлены в одном и том же месте памяти (не считая нескольких исключений), так что той же проверкой предикатом EQ? символы можно сравнить логически. До сих пор, например, списки могли быть логически (внешне) одинаковы, но они могут состоять из физически различных списочных ячеек.

```
[7] (eq? '(a b c) '(a b c))
()
[8] (eq? 15.0 15)
()
```

Так как EQ? определен лишь для символов, то, сравнивая два выражения, прежде всего надо определить, являются ли они атомами (ATOM?). Если хотя бы один из аргументов является списком, то предикат EQ? нельзя использовать для логического сравнения. При сравнении чисел проблемы возникают с числами различных типов. Например, числа 3.000000, 3 и 0,3E1 логически представляют одно

и то же число, но записываются внешне неодинаково. Для различия видов и степеней равенства в Лиспе наряду с EQ? используются и другие предикаты.

EQV? сравнивает числа разных типов

Более общим по сравнению с EQ? является предикат EQV?, который работает так же, как EO?, но дополнительно позволяет сравнивать числа разных типов (и элементы строк).

```
[0] (eqv? 3 3)
#T
[1] (eqv? 3.14 3.14)
#T
[2] (eqv? 3.104 3.104)
#T
[3] (eqv? 3.00 3)
#T
[4] (eqv? 'a 'a)
#T
[5] (eqv? '(a b c) '(a b c))
()
```

Предикат EQV?, как правило, используется во многих встроенных функциях, осуществляющих более сложные операции сравнения. Его использование для сравнения списков- это часто встречающаяся ошибка.

Предикат = сравнивает числа различных типов

Сложности, возникающие при сравнении чисел, легко преодолимы с помощью предиката =, значением которого является T в случае равенства чисел независимо от их типов и внешнего вида записи:

```
[6] (= 3.000 3)
#T
[7] (= 3.000 3.000000)
#T
[8] (= 3.010 3.000000)
()
```

EQUAL? проверяет идентичность записей

Обобщением EQV? является предикат EQUAL? Он работает как EQV?, но, кроме того, проверяет одинаковость двух списков:

```
[0] (equal? 'x 'x)
#T
[1] (equal? '(x y z) '(x y z))
#T
[5] (equal? '(x y z)  (cons 'x (cdr '(w y z))))
#T
[6] (equal? 3.12 3.12)
#T
[7] (equal? 3.00 3)
()
```

[8] (equal? '(nil) '((nil)))
()

Принцип работы предиката EQUAL? состоит в следующем: если внешняя структура двух лисповских объектов одинакова, то эти объекты между собой равны в смысле EQUAL?. Предикат EQUAL? также применим к числам и к другим типам данных (например, к строкам). Заметим, что в соответствии со своим принципом работы он не подходит для сравнения разнотипных чисел, так как их внешние представления различаются.

ДРУГИЕ ПРИМИТИВЫ

Несмотря на то, что обычную обработку списков всегда можно свести к описанным ранее трем базовым функциям (CONS, CAR, CDR) и двум базовым предикатам (ATOM? и EQ?), программирование лишь с их использованием было бы очень примитивным и похожим на программирование на внутреннем машинном языке. Поэтому в Лисп — систему включено множество встроенных функций для различных действий и ситуаций. Рассмотрим некоторые такие примитивы и их полезные свойства.

NULL? проверяет на пустой список

Встроенная функция NULL? проверяет, является ли аргумент пустым списком:

[0] (null? '())
#T
[1] (null? 'a)
()
[2] (null? (cddr '(a b c)))
()
[3] (null? nil)
#T
[4] (null? 'nil)
()
[5] (null? t)
()

Из примеров видно, что NULL? работает как логическое отрицание, у которого в Лиспе есть и свой, принадлежащий логическим функциям, предикат **(NOT X):**

[0] (not t)
()
[1] (not nil)
#T
[2] (not (not t))
#T
[3] (not (not nil))
()
[4] (not (null? nil))
()

Вложенные вызовы CAM и CDR можно записывать в сокращенном виде

Комбинируя селекторы CAR и CDR, можно выделить произвольный элемент списка. Например:

[5] (cdr (cdr (car '((a b c) (d e) (f)))))

(C)

Комбинируя вызов CAR и CDR образуют уходящие в глубину списка обращения, и в Лиспе используется для этого более короткая запись: желаемую комбинацию вызовов CAR и CDR можно записать в виде одного вызова функции:

(C... R список)

Вместо многоточия записывается нужная комбинация из букв A (для функции CAR) и D(для функции CDR):

(cadr x) —> (car (cdr x))

(cddar x) —> (cdr (cdr (car x)))

Примеры:

[6] (cddar '((a b c) (d e) (f)))

(C)

[7] (cddddr '(Программировать на Лиспе просто ?))

(?)

LIST создает список из элементов

Другой часто используемой встроенной функцией является: (LIST *x1 x2 x3 ...*) которая возвращает в качестве своего значения список из значений аргументов. Количество аргументов функции LIST произвольно:

[0] (list 1 2)

(1 2)

[1] (list 'a 'b (+ 1 2))

(A B 3)

[2] (list 'a '(b c) 'd)

(A (B C) D)

[3] (list (list 'a) 'b nil)

((A) B ())

[4] (list nil)

(())

Построение списков нетрудно свести к вложенным вызовам функции CONS, причем вторым аргументом последнего вызова является NIL, служащий основой для наращивания списка:

[0] (list 'a)

(A)
[1] (cons 'a nil)
(A)
[2] (list 'a 'b)
(A B)
[3] (cons 'a (cons 'b nil))
(A B)
[4] (list 'a 'b 'c)
(A B C)
[5] (cons 'a (cons 'b (cons 'c nil)))
(A B C)

ФУНКЦИИ СВЯЗЫВАНИЯ ПЕРЕМЕННЫХ СО ЗНАЧЕНИЯМИ (SET!)

Если мы хотим связать символ, список с некоторой переменной, то эту переменную в Лисп-Микро необходимо предварительно определить с помощью функции DEFINE, а затем связать со значением SET!

[0] (define x)
X
[1] (set! x '(+ 3 5))
(+ 3 5)
[2] x
(+ 3 5)
[3] (set! x (+ 3 5))
8

Например, мы хотим, чтобы символ ФУНКЦИИ обозначал базовые функции Лисп-Микро:

[4] (define функции)
ФУНКЦИИ
[5] (set! функции '(car cdr cons atom? eq?))
(CAR CDR CONS ATOM? EQ?)

Теперь между символом ФУНКЦИИ и значением (CAR CDR CONS ATOM? EQ?) образована связь, которая действительна до окончания работы, если, конечно, этому имени функцией SET! не будет присвоено новое значение. После присваивания интерпретатор уже может вычислить значение символа ФУНКЦИИ:

[6] функции
(CAR CDR CONS ATOM? EQ?)

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Лямбда - выражение и лямбда - вызов

В основе функциональной парадигмы программирования лежит Лямбда - исчисление Черча, представляющее собой точный и простой формализм вычисления функций. В соответствии с этим формализмом вычисление любого выражения или функции можно определить Лямбда выражением, имеющим в Лисп-Микро следующий формат:

(LAMBDA (V1 V2 ... Vn) S1 S2 ... Sm)

где Vl, V2,..., Vn – формальные параметры

S1, S2, ..., Sm – S-выражения, представляющие тело исходной функции.

Вычисление лямбда – выражения производится с помощью лямбда – вызова, имеющего вид:

(лямбда – выражение al a2 ... an), где al, a2,..., an – фактические параметры.

Полный процесс определения и вычисления некоторой функции в терминах лямбда — исчисления называют лямбда- преобразованием, которое имеет вид:

((LAMBDA (V1 V2 ... Vn) S1 S2 ... Sm) a1 ... an).

Сначала вычисляются фактические параметры al, a2,..., an, значения которых связываются с соответствующими формальными параметрами Vl, V2,..., Vn, а на следующем этапе с учетом новых связей вычисляются S-выражения Sl, S2, ..., Sm, представляющие тело исходной функции. Значение последнего выражения Sm возвращается в качестве результата.

Например, оформление в виде лямбда — выражения вычисления второго элемента в списке и выделение его из списка (A B C):

[1] ((lambda (l) (car (cdr l))) '(a b c))
B

В лямбда – преобразованиях процедуры определения и вызова функции объединены в единой форме, поэтому, если нужно повторить вычисление исходной функции с новыми значениями аргументов, например, выделить второй элемент в списке (1 (B C) 2), нужно вновь записать его полную форму:

[2] ((lambda (l) (car (cdr l))) '(1 (b c) 2))
(B C)

Это вызывает определенные неудобства, особенно, если тело функции представляет собой громоздкую запись.

В Микро-Лиспе в отличие от Коммон - Лиспа, где такая возможность отсутствует, определение лямбда – выражения можно связывать с определенным именем, а затем вызывать это лямбда – выражение по имени как обычную именованную функцию:

[3] (define second (lambda (l) (car (cdr l))))
SECOND
[4] (second '(a b c))
B

[5] (second '(1 (b c) 2))
(B C)

Форма LET

Другой разновидностью лямбда – исчисления в Лиспе является форма LET, которая имеет вид:

(LET ((V1 A1) (V2 A2)... (Vn An)) S1 S2... Sm),

где Vi, Ai, Si имеют такой же смысл как и в лямбда – преобразовании.

По сути форма LET является синтаксическим видоизменением лямбда–преобразования, в которой пары формальных и фактических параметров помещены в начале формы.

Оформление вычисления второго элемента в списке ‘(A B C) используя форму LET:

[6] (let ((l '(a b c))) (cadr l))
B

Форму LET, подобно лямбда–преобразованию, можно использовать для вычисления именованной функции при ее определении:

[7] (define (second1 l) (let ((l l)) (cadr l)))
SECOND1
[8] (second1 '(1 (b c) 2))
(B C)

Если в форме LET имеется несколько пар формальных и фактических параметров, то связь между соответствующими переменными и их значениями устанавливается одновременно.

Последовательная форма LET*

Если вычислять с помощью формы LET приведенный ниже пример, выйдет ошибка: не связанный атом X:

[9] (let ((x 1) (y (* 2 x))) (list x y))
[VM **ERROR encountered!]**

В то же время так называемая последовательная форма LET*, в которой связи между переменными и их значениями устанавливаются последовательно, примененная в этом примере, даст нормальный результат:

[10] (let* ((x 1) (y (* 2 x))) (list x y))
(1 2)

ВЫЧИСЛЕНИЕ В ЛИСПЕ. ОРГАНИЗАЦИЯ УСЛОВНЫХ И ЦИКЛИЧЕСКИХ ВЫЧИСЛЕНИЙ

Разветвление вычислений. Условное предложение COND

Предложение COND является основным средством разветвления вычислений. Это синтаксическая форма, позволяющая управлять вычислениями на основе определяемых предикатами условий. Структура условного предложения такова:

(COND (pl a1) (p2 a2) ... *(pN* aM))

Предикатами pi и результирующими выражениями ai могут быть произвольные формы. Значение предложения COND определяется следующим образом:

1. Выражение *pi,* выполняющие роль предикатов, вычисляются последовательно слева направо (сверху вниз) до тех пор, пока не встретится выражение, значением которого не является NIL, т.е. логическим значением которого является истина.
2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращение в качестве значения всего предложения COND.
3. Если истинного предиката нет, то значением COND будет NIL.

Рекомендуется в качестве последнего предиката использовать символ T, и соответствующее ему результирующее выражение будет вычисляться всегда в тех случаях, когда ни одно другое условие не выполняется.

В следующем примере с помощью предложения COND определена функция, устанавливающая тип выражения:

```
[1] (define (тип l) (cond ((null? l) 'пусто)((atom? l) 'атом) (t 'список)))
ТИП
[2] (тип '(a b c))
СПИСОК
[3] (тип '())
ПУСТО
[4] (тип 'c)
АТОМ
```

В качестве примера использования условного предложения определим логические действия логики высказываний «и», «или», «не»:

```
[5] (define (и x y) (cond (x y) (t nil)))
И
[6] (и t nil)
()
[7] (define (или x y) (cond (x t) (t y)))
ИЛИ
[8] (или t nil)
#T
[9] (или t t)
#T
[10] (define (не x) (not x))
НЕ
```

[11] (не t)
()
[12] (не nil)
#T

COND условно выбирает и вычисляет одну из серий операндов, например:

[13] (define (сравнить x y) (cond ((> x y) '(x больше y))((< x y) '(x меньше y))(t 'равны)))
СРАВНИТЬ
[14] (сравнить 4 5)
(X МЕНЬШЕ У)
[15] (сравнить 67 20)
(X БОЛЬШЕ У)
[16] (сравнить 6 6)
РАВНЫ

Предикаты AND и OR

AND в случае истинности возвращает значение своего последнего аргумента. Синтаксис:
(AND условие1 условие2 ... условиеN)

[0] (AND ‘a ‘b ‘c NIL)
()
[1] (AND ‘a ‘b ‘c)
C
[2] (AND NIL ‘a ‘b)
()

OR в случае истинности возвращает значение первого аргумента, отличного от NIL. Синтаксис:
(OR условие1 условие2 ... условиеN)

[0] (OR ‘a ‘b nil)
A
[1] (OR nil nil ‘a ‘b)
A

Предложения IF, WHEN, CASE

В ряде случаев использовать COND неудобно. Поэтому введены и другие условные выражения.

(IF условие то-форма иначе-форма) ⇔ (COND (условие то-форма) (T иначе-форма))

(WHEN условие форма1 форма2 ...) ⇔ (COND (условие форма1 форма2 ...)

Условное предложение IF

В простом случае можно воспользоваться вполне естественной и содержащей мало скобок формой IF:

(*IF условие то – форма иначе – форма)*

IF вычисляет по условию одно из двух альтернативных выражений:

```
[1] (if (atom? t) 'атом 'список)
АТОМ
[2] (if (>? 1 9) 'да 'нет)
НЕТ
[3] (if (>? 11 9) 'да 'нет)
ДА
[4] (if (>? 12 8) (- 12 8))
4
[5] (if (>? 5 7) (- 5 7))
()
```

Условное предложение CASE

В форме CASE сначала вычисляется значение ключевой формы *ключ*. Затем проверяется, есть ли это значение в списках *список-ключейN*. Если такой список будет найден, вычисляются альтернативы mNi т возвращается последняя из них в качестве всего значения выражения CASE.

```
(CASE ключ
        (список-ключей1 m11 m12 ...)
        (список-ключей2 m21 m22 ...)
        ...
)
[0] (case 1 ((1 2) 12) ((3 4) 34))
12
[1] (case 2 ((1 2) 12) ((3 4) 34))
12
[2] (case 3 ((1 2) 12) ((3 4) 34))
34
[3] (case 4 ((1 2) 12) ((3 4) 34))
34
```

Предложение DO

Используется для организации повторяющихся вычислений. Синтаксис:

```
(DO ((var1 знач1 шаг1) (var2 знач2 шаг2) ...)
        (условие-окончания форма11 форма12 ...)
        форма21
        форма22
```

...
)

Первый аргумент описывает внутренние переменные *var1, var2,* …, их начальные значения *знач1, знач2, ...* и формы их обновления *шаг1, шаг2, ...* . Вычисление начинается с присвоения начальных значений переменным. Если начальное значение не задано, переменной присваивается NIL. При каждой итерации цикла после этого вычисляется условие окончания. Если оно истинно, вычислются формы *форма1N*, значение последней из которых является значением всего выражения DO. Если ложно, то вычисляются формы *форма2N*, переменным *varN* присваиваются значения форм *шагN*, и опять проверяется условие окончания цикла, и т.д.

[0] (DEFINE a)
A
[1] (do ((a 1 (+ a 1))) ((= a 10) a))
10

ВВОД И ВЫВОД

Ввод и вывод входят в диалог

До сих пор в определенных нами функциях ввод данных (READ) и вывод (PRINT) осуществлялись в процессе диалога с интерпретатором. Интерпретатор читал вводимое пользователем выражение, вычислял его значение и возвращал его пользователю. Сами формы и функции не содержали ничего, связанного с вводом и выводом.

Если не использовать специальную команду ввода, то данные можно передавать лисповской функции только через параметры и свободные переменные. Соответственно, без использования вывода, результат можно получить лишь через конечное значение выражения. Часто все же возникает необходимость вводить исходные данные и выдавать сообщения и тем самым управлять и получать промежуточные результаты во время вычислений, как это делается и в других языках программирования.

READ читает и возвращает выражение

Лисповская функция чтения READ отличается от ввода в других языках программирования тем, что она обрабатывает выражение целиком, а не одиночные элементы данных. Вызов этой функции осуществляется пользователем (немного упрощенно) в виде:

(READ)

Как только интерпретатор встречает предложение READ, вычисления приостанавливаются до тех пор, пока пользователь не введет какой-нибудь символ или целиком выражение:

[0] (read)
(вводимое выражение)
(ВВОДИМОЕ ВЫРАЖЕНИЕ)
[1]

```
()
```

READ лишь читает выражение и возвращает в качестве значения само это выражение, после чего вычисления продолжаются.

PRINT переводит строку, выводит значение и пробел

Для вывода выражений можно использовать функцию PRINT. Эта функция с одним аргументом, которая сначала вычисляет значение аргумента, а затем выводит это значение. Функция PRINT перед выводом аргумента пропускает строку, переходит на следующую строку, а после него выводит пробел. Таким образом, значение выводится всегда на новую строку:

```
[0] (print (+ 2 3))

5
[1] (print '(+ 2 3))

(+ 2 3)
[2] (print (eval '(+ 4 5)))

9
```

PRIN1 и PRINC выводят без перевода строк

Если желательно вывести последовательно на одну строку более одного выражения, то можно использовать функции PRIN1 или PRINC. PRIN1 в отличие от PRINT не переходит на новую строку и не выводит пробел:

```
[3] (begin (prin1 1) (prin1 2) (print 3))
12
3
[4] (begin (prin1 1) (prin1 2) (print 3) (print 4))
12
3
4
```

Как функция PRINT, так и PRIN1 можно выводить кроме атомов и списков и другие типы данных, например, строки, представляемые последовательностью знаков, заключенных с обеих сторон в кавычки ("). Таким образом, строка выводится вместе с ограничителями:

```
 [5]  (prin1 "a b c")
"a b c"
```

Более приятный вид с точки зрения пользователя можно получить при помощи функции PRINC. Она выводит лисповские объекты в том же виде, как и PRIN1, но преобразует некоторые типы данных в более простую форму. Функцией PRINC мы можем напечатать строку без ограничивающих ее кавычек и специальные знаки без их выделения:

```
[6]  (princ "a b c")
a b c
```

С помощью функции PRINC можно, естественно, напечатать и скобки:

[7] (begin (princ "(((") (prin1 'луковица) (princ ")))")) (((ЛУКОВИЦА)))
(((ЛУКОВИЦА)))

Пример использования PRINC: определение функции (ЧЕРТА n), печатающей n раз звездочку (*):

[8] (define (черта n) (cond ((= n 0) t) (t (princ "*") (черта (- n 1)))))
ЧЕРТА
[9] (черта 10)
************#T**

WRITE выводит без перевода строки

WRITE является процедурой вывода. WRITE записывает представление его аргумента в выводном порте таким образом, чтобы аргумент мог быть заново прочитан с помощью READ. Поэтому строки заключаются в двойные кавычки, а встроенным литерам обратного слэша и двойных кавычек должны предшествовать обратные слэши.

[10] (begin (write '!) (prin1 '!) (princ '!))
!!!

DISPL4Y записывает строки без кавычек

PRIN1 и PRINC рассматриваются как альтернативные имена для WRITE и DISPLAY, соответственно. DISPLAY записывает свой аргумент в виде, более читабельном для пользователя. Строки не заключаются в кавычки, литеры записываются как обычно, а исключительные литеры в строках и символы не теряются.

[11] (display "Привет!")
Привет!

WRITELN выводит значение, переводит строку

WRITELN записывает каждый из своих аргументов слева направо и затем выпускает новую строку:

[18] (writeln '(a b c))
(A B C)
()
[19] (writeln (+ 4 6))
10
()

PRINT-LENGTH возвращает число выводимых литерных позиций

PRINT-LENGTH возвращает число выводимых литерных позиций, которые будут печататься с помощью WRITE:

[11] (print-length "a b c d e f g h i")
17
[12] (print-length 34)
2

NEWLINE переводит строку

Вывод выражений и знаков часто желательно разбить на несколько строк. Перевод строки можно осуществить функцией PRINT, которая автоматически переводит строку перед выводом, или непосредственно для этого предназначенной функцией NEWLINE. У функции NEWLINE нет аргументов.

[13] (begin (write 'переведи) (newline) (prin1 'строку))
ПЕРЕВЕДИ
СТРОКУ

Используя уже определенную ранее функцию ЧЕРТА, определим функцию (ПРЯМОУГОЛЬНИК n m), заполняющую всю область n X m звездочками:

[13] (define (прямоугольник ширина высота) (cond ((= высота 0) t) (t (черта ширина) (newline) (прямоугольник ширина (- высота 1)))))
[14] (прямоугольник 6 4)
******
******
******
******
#T

LOAD загружает определения

На практике написание программ осуществляется записью в файл определений функций, данных и других объектов с помощью имеющегося в программном окружении редактора. После этого для проверки определений вызывают интерпретатор Лиспа, который может прочитать записанные в файл выражения директивой LOAD:

(LOAD *"файл")*

Читаемые выражения вычисляются так, как будто бы они были введены пользователем. После загрузки можно использовать функции, значения переменных, значения свойств и другие определения.

Пример загрузки файла с помощью функции LOAD:

Текст программы, записанный в файл fl.lsp:

****

```
(define (пересечение x y)
    (cond ((null? x) nil)
```

```
                ((member (car x) y)
                    (cons (car x)
                                (пересечение (cdr x) y)))
                (t (пересечение (cdr x) y))))
```

****

[0] (load "f1.lsp")
OK
[1] (пересечение '(d f g) '(h u g e))
(G)

ОРГАНИЗАЦИЯ ФУНКЦИЙ

Описание функции

Дать функции имя и определить ее можно с помощью функции DEFINE.
(DEFINE (fname param1 param2 …) тело)

Например:

[0] (DEFINE (plus a b) (+ a b))
PLUS
[1] (plus 1 2)
3

В результате fname становится именем функции.

Простая рекурсия

Функция является *рекурсивной*, если в её определении содержится вызов самой этой функции. *Простая* рекурсия – это такой тип рекурсии, в котором вызов функции встречается один раз. В большинстве случаев простую рекурсию можно заменить процедурным циклом, однако, при её использовании программа, как правило, становится более простой для понимания и элегантной.

Пример:

Определим функцию MyCOPY, которая строит копию списка:

```
(define (MyCOPY L)
        (cond
                ((null? L) nil)  ; условие окончания – пустой L
                (t (cons                        ; иначе возвращаем список
                                                ; состоящий из:
                        (car L)                 ; головы исходного
                        (MyCOPY (cdr L))        ; и копии хвоста
                                                ; (рекурсивный вызов)
                        )
                )
        )
)
```

Вот результат работы этой функциии:

[1] (MyCOPY '(a bc def))
(A BC DEF)

Разберем работу этой функции.

(MyCOPY ‘(a bc def))

L = ‘(a bc def)

L не пуст, значит возвращаем список, состоящий из первого элемента L и копии хвоста списка:

(cons (car L) (MyCOPY (cdr L))

Причем, рекурсивный вызов MyCOPY будет сделан уже не для всего списка L, а только для его «хвоста». Действительно, (cdr L) в этом случае будет ‘(bc def).

Нетрудно сообразить, что при каждом следующем вызове MyCOPY передаваемый функции список будет меньше предыдущего на один элемент. Наконец, когда функции будет передан пустой список, сработает условие ((null? L) nil), что приведет к завершению работы этой функции.

Надо отметить, что к разработке рекурсивных функций следует относится крайне тщательно, так как, например, в рассмотренной функции, отсутствие проверки списка на пустоту привело бы к зацикливанию программы (бесконечной рекурсии).

Другие формы рекурсии (параллельная рекурсия)

Рекурсия может принимать различные формы. Так, можно выделить так называемую «параллельную» рекурсию – когда тело функции *f* содержит вызов некоторой функции *g*, несколько аргументов которой являются рекурсивными вызовами функции *f*:

```
(define (f …
        …(g … (f …) … (f …) …)
…)
```

Пример

Рассмотрим копирование списка на всех уровнях.

```
(define (copy-tree1 L)
        (cond ((null? L) nil)    ;условие окончания – пустой L
                ((atom? L) L)            ;атом – возвращаем его же
                (T (cons
                        (copy-tree1 (car L))    ;копия головы
                        (copy-tree1 (cdr L))    ;копия хвоста
                ))
        )
)
```

Результат выполнения этого примера:

```
[0] (COPY-TREE1 '(A (B (C D))))
(A (B (C D)))
```

Как видно из этого примера, функция copy-tree1 «разбивает» список на голову и хвост и вызывает сама себя по отдельности для головы и хвоста исходного списка.

ПРИМЕНЯЮЩИЕ И ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ

Одним из основных типов функционалов являются функции, которые позволяют вызывать другие функции, иными словами, применять функциональный

аргумент к его параметрам. Такие функционалы называют *применяющими* или *аппликативными* функционалами (APPLY и FUNCALL).

Применяющие функционалы родственны универсальной функции Лиспа EVAL. В то время как EVAL вычисляет значение произвольного выражения (формы), применяющий функционал вычисляет значение вызова некоторой функции. Интерпретатор Лиспа EVAL и на самом деле вызывает применяющий функционал APPLY при вычислении вызова, а APPLY в свою очередь вызывает EVAL при вычислении значения других форм.

Применяющие функционалы дают возможность интерпретировать и преобразовывать данные в программу и применять ее в вычислениях. Ниже рассмотрим применяющий функционал интерпретатора Лиспа APPLY.

Применяющий функционал APPLY

APPLY применяет функцию, переданную как параметр, к списку аргументов: (APPLY function x1 x2 x3 … xN) ⇔ (function ‘x1 ‘x2 ‘x3 … ‘xN) , *г*де список=(x1 x2 ... xN)

APPLY является (в своей исходной форме) функцией двух аргументов, из которых первый аргумент представляет собой функцию, которая применяется к элементам списка, составляющим второй аргумент функции APPLY:

[1] (apply + '(4 7))
11
[2] (apply cons '(что (пожелаете)))
(ЧТО ПОЖЕЛАЕТЕ)
[3] (define f)
F
[4] (set! f '+)
+
[5] (apply f '(900 77))
977

Использование APPLY дает большую гибкость по сравнению с прямым вызовом функции: с помощью одной и той же функции APPLY можно в зависимости от функционального аргумента осуществлять различные вычисления.

[15] (apply (lambda (x y) (+ x y)) '(6 5))
11
[16] (define s)
S
[17] (set! s '(5 7))
(5 7)
[18] (apply cdr (list s))
(7)
[19] (apply (lambda (n) (* n n)) (cdr s))
49
[20] (apply (lambda (n) (* n n)) (car s))
25

Применяющий функционал FUNCALL

FUNCALL вызывает функцию, переданную как параметр, с остальными аргументами:

(FUNCALL function x1 x2 … xN) ⇔ (function x1 x2 … xN)

Например:

(FUNCALL ‘+ 2 3) ⇔ (+ 2 3)

Основные типы MAP-функций

Есть еще один класс функций, которые некоторым способом отображают список (последовательность) в новую последовательность или порождают какой-либо другой эффект. Имена таких функций начинаются на **MAP** и их семейство получило название «отображающие функционалы». Вызов имеет вид:

(MAPx fn l1 l2 … lN)

l1 … lN – списки, **fn** – функция от N аргументов.

Наиболее часто MAP-функции применяются к одному аргументу:

(MAPx fn l1)

MAPCAR – повторяет вычисление функции на элементах списка

(MAPCAR fn ‘(x1 x2 … xN)) ⇔ (LIST (fn ‘x1) (fn ‘x2) … (fn xN))

MAPLIST – повторяет вычисление на хвостовых частях списка, начиная с самого списка:

(MAPLIST fn ‘(x1 x2 … xN)) ⇔ (LIST (fn ‘(x1 x2 … xN)) (fn ‘(x2 x3 … xN)) … (fn ‘(xN)))

Функции MAPC и MAPL аналогичны MAPCAR и MAPLIST, однако, они не объединяют результаты. Возникающие резхультаты просто теряются.

Задания для самостоятельной работы к разделу «Краткая справка из теории»

1. Сколько элементов самого верхнего уровня в следующих списках:
    a. ((1 2 3))
    b. ((a b) c (d e))
    c. (a ((())) nil nil)
    d. ((((a (b (c d) e) f) g) h ((i (j) k) l) m) n)
2. Замените в следующих списках все пустые списки на символ пустого списка NIL:
    a. (())
    b. (() (()))
    c. ((() (() (())) ()))
3. Чем отличаются:
    a. атомы и символы
    b. переменные и символы
    c. выражения и списки
4. Как можно записать в виде списков выражения логики высказываний, образованные с помощью логических операций NOT, OR, AND, =>, <=>.

Задания для самостоятельной работы к разделу «ПРИМИТИВЫ языка Micro-Lisp»

1. Перечислите базовые функции ипредикаты языка LISP. Каковы типы их агрументов, какие значения они возвращают в качестве результата?
2. Запишите последовательности вызовов CAR и CDR, выделяющие из приведенных ниже списков символ «target». Упростите эти вызовы при помощи C…R.
    a. (1 2 target 3 4)
    b. ((1) (2 target) (3 (4)))
    c. ((1 (2 (3 4 target))))
3. Вычислите и проверьте в интерпретаторе значения следующих выражений:
    a. (cons nil ‘(суть пустой список))
    b. (cons nil nil)
    c. (cons ‘(nil) ‘(nil))
    d. (cons (car ‘(a b)) (cdr ‘(a b)))
    e. (car ‘(car (a b c))
    f. (cdr (car (cdr ‘(a b c))))
    g. (list (list ‘a ‘b) ‘(car (c d)))
4. Какие из следующих вызовов возвращают значение #T?
    a. (atom? ‘(cdr nil))
    b. (equal? ‘(a b) (cons ‘(a) ‘(b)))
    c. (atom? (* 2 (+ 2 3)))
    d. (null? (null? T))
    e. (eq? nil (null? nil))
    f. (equal? 2.0 2)
    g. (= 2.0 2)

Задания для самостоятельной работы к разделу «ВЫЧИСЛЕНИЕ В ЛИСПЕ. ОРГАНИЗАЦИЯ УСЛОВНЫХ И ЦИКЛИЧЕСКИХ ВЫЧИСЛЕНИЙ»

1. С помощью предложений COND или CASE определите функцию, которая возвращает в качестве значения столицу заданного аргументом государства:
   [0] (столица ‘Россия)
   МОСКВА
2. Предикат сравнения (> x y) истинен, если x больше, чем y. Опишите с помощье предиката > и условного предложения функцию, которая возвращает из трех числовых аргументов значение среднего по величине числа:
   [0] (среднее 4 7 6)
   6
3. Можно ли с помощью предложения COND запрограммировать IF как функцию?
4. Запрограммируйте с помощью предложения DO итеративную версию функции факториал.
5. В математике числа Фибоначчи образуют ряд 0, 1, 1, 2, 3, 5, 8... Эту последовательность можно определить с помощью следующей функции:
   fib(n)=0 (если n=0)
   fib(n)=1 (если n=1)
   fib(n)=fib(n-1)*fib(n-2) (если n>1).
   Определите эту функцию.
6. Определите функцию ДОБАВЬ, прибавляющую к элементам списка данное число:
   [0] (добавь ‘(2 7 3) 3)
   (5 10 6)

Задания для самостоятельной работы к разделу «Простая рекурсия»

1. Определите функцию LAST1, возвращающую последний элемент списка.
2. Определите функцию DELLAST1, удалающую из списка последний элемент.
3. Определите предикат (функцию) ATOMLIST, проверяющий, является ли предикат одноуровневым списком.
4. Определите функцию ONION («луковица»), строящую N-уровневый вложенный список.
5. Определите функцию FIRSTATOM, результатом которой будет первый атом списка.
6. Определите функцию, удаляющую из списка первое вхождение заданного элемента.

Задания для самостоятельной работы к разделу «Другие формы рекурсии (параллельная рекурсия)»

1. Определите функцию, вычисляющую общее количество всех атомов в списке.
2. Определите функцию двух аргументов-списков, возвращающую первый элемент, входящий в оба списка и NIL, если такого элемента нет.
3. Определите функцию, преобразующую список в множество, то есть удаляющую повторения элемента в списке.
4. Определите предикат, проверяющий совпадение двух множеств (независимо от порядка следования элементов).

5. Определите функцию, проверяющую, является ли одно множество подмножеством другого.
6. Определите предикат, проверяющий отсутствие общих элементов (отсутствие пересечения) двух множеств.

Задания для самостоятельной работы к разделу «ПРИМЕНЯЮЩИЕ И ОТОБРАЖАЮЩИЕ ФУНКЦИОНАЛЫ»

1. Определите **FUNCALL** через функционал **APPLY**. (Примечание: В Микро-ЛИСПе встроенные функции **FUNCALL** и **&REST** отсутствуют. Из за этого имеется возможность определить **FUNCALL** только для фиксированного количества аргументов)
2. Вычислите значения вызовов:
    a. (apply list ‘(a b))
    b. (funcall ‘list ‘(a b))
    c. (funcall apply list ‘(a b))
    d. (funcall list ‘apply ‘(a b))
3. Вычислите значения следующих функций:
    a. (mapcar list ‘(a b c))
    b. (mapc list ‘(a b c))

Задания для самостоятельной работы к разделу «ОПИСАНИЕ ФУНКЦИИ»

Напишите функцию для вычисления интеграла с указанной точностью:

№	Подинтегральная функция, f(x)	Промежуток, [a;b]	Кол-во частей разбиения	Шаг вычисления, h	Точность вычисления значения первообразной	Точное значение первообразной $\int_a^x f(x)dx$
1	$\frac{\ln x}{x\sqrt{1+\ln x}}$	[1; 3.5]	30	0.25	0.001	$\frac{2}{3}(\ln x+1)^{3/2}-2(\ln x+1)^{1/2}+\frac{4}{3}$
2	tg^2x+ctg^2x	$[\frac{\pi}{6};\frac{\pi}{3}]$	54	$\pi/36$	0.001	$tgx - ctgx - 2x - tg\frac{\pi}{6} + ctg\frac{\pi}{6} + \frac{\pi}{3}$
3	$\frac{1}{x\lg x}$	[2; 3]	36	0.2	0.001	$2.3026(\ln\ln x - \ln\ln 2)$
4	$\frac{\ln^2 x}{x}$	[1; 4]	52	0.5	0.001	$\frac{1}{3}\ln^3 x$
5	$\sqrt{e^x-1}$	[0; ln 2]	104	$\frac{\ln 2}{5}$	0.001	$2\sqrt{e^x-1}-2arctg\sqrt{e^x-1}$
6	$xe^x\sin x$	[0; 1]	48	0.2	0.001	$\frac{xe^x(\sin x-\cos x)+e^x\cos x-1}{2}$
7	$x\sinh x$	[0; 2]	48	0.4	0.001	$\frac{x(e^x+e^{-x})}{2}-\frac{e^x-e^{-x}}{2}$
8	$\frac{1}{\sqrt{9+x^2}}$	[0; 2]	208	0.25	0.001	$\ln(x+\sqrt{x^2+9})-\ln 3$
9	$\frac{1}{x^2}\sin\frac{1}{x}$	[1; 2.5]	44	0.3	0.001	$\cos\frac{1}{x}-\cos 1$
10	$x\arctan x$	$[0;\sqrt{3}]$	48	$\frac{\sqrt{3}}{8}$	0.001	$\frac{x^2}{2}\arctan x-\frac{x}{2}+\frac{1}{2}\arctan x$
11	$\arcsin\sqrt{\frac{x}{1+x}}$	[0; 3]	36	0.5	0.001	$x\arcsin\sqrt{\frac{x}{1+x}}-\sqrt{x}+\arctan\sqrt{x}$
12	$x^x(1+\ln x)$	[1; 3]	40	0.2	0.001	x^x-1
13	$\frac{1}{\sqrt{1+3x+2x^2}}$	[0; 1]	44	0.2	0.001	$\frac{1}{\sqrt{2}}\ln\frac{x+0.75+\sqrt{(x+0.75)^2-0.0625}}{0.75+\sqrt{0.5}}$
14	$\frac{\sqrt{x^2-0.16}}{x}$	[1; 2]	160	1/8	0.001	$\sqrt{x^2-0.16} - 0.4\arccos\frac{0.4}{x} - \sqrt{0.84} + 0.4\arccos 0.4$
15	2^{3x}	[0; 1]	240	0.2	0.001	$\frac{1}{3\ln 2}(2^{3x}-1)$
16	$\frac{x\arctan x}{\sqrt{1+x^2}}$	[0; 1]	22	1/8	0.001	$\sqrt{1+x^2}\arctan x-\ln(x+\sqrt{1+x^2})$
17	$\frac{e^{3x}+1}{e^x+1}$	[0; 2]	48	0.25	0.001	$\frac{e^{2x}}{2}+e^x+x+0.5$

№	Подинтегральная функция, f(x)	Промежуток, [a;b]	Кол-во частей разбиения	Шаг вычисления, h	Точность вычисления значения первообразной	Точное значение первообразной $\int_a^x f(x)dx$
18	$\sin^2 x$	$[0;\frac{\pi}{2}]$	22	$\frac{\pi}{12}$	0.001	$\frac{x}{2}-\frac{1}{4}\sin 2x$
19	$x^2\sqrt{4-x^2}$	[0; 1.9999]	96	0.25	0.001	$2\arcsin\frac{x}{2}-\frac{1}{2}\sin(4\arcsin\frac{x}{2})$
20	$e^x\cos^2 x$	$[0;\pi]$	60	$\frac{\pi}{6}$	0.001	$\frac{e^x}{2}(1+\frac{2\sin 2x+\cos 2x}{5})-0.6$
21	$(x\ln x)^2$	[1; e]	52	$\frac{e-1}{8}$	0.001	$\frac{x^2}{27}(9\ln^2 x-6\ln x+2)-\frac{2}{27}$
22	$\arcsin\sqrt{\frac{x}{1+x}}$	[0; 3]	176	0.6	0.001	$x\arcsin\sqrt{\frac{x}{1+x}}-\sqrt{x}+\arctan\sqrt{x}$
23	$\frac{x^2-1}{(x^2+1)\sqrt{x^4+1}}$	[0; 1]	36	0.25	0.001	$-\frac{\sqrt{2}}{2}\arcsin\left(\frac{\sin(2\arctan x)}{\sqrt{2}}\right)$
24	$\sin x\ln(\tan x)$	[1; 1.5]	52	0.1	0.001	$\ln\left(\tan\frac{x}{2}\right)-(\cos x)(\ln(\tan x))-$ $-\ln\tan 0.5+(\cos 1)\ln\tan 1$
25	$\frac{e^x(1+\sin x)}{1+\cos x}$	[0; 1.5]	132	0.3	0.001	$e^x\tan\frac{x}{2}$
26	$\frac{1}{(x+1)\sqrt{x^2+1}}$	$\left[0;\frac{3}{4}\right]$	40	$\frac{3}{20}$	0.001	$\frac{1}{\sqrt{2}}\left(\ln\frac{1+\sqrt{2}}{2}-\ln\frac{1-x+\sqrt{2(x^2+1)}}{2(x+1)}\right)$
27	$\frac{1}{(3\sin x+2\cos x)^2}$	[0; 1]	78	0.2	0.001	$\frac{3}{26}-\frac{3\cos x-2\sin x}{13(2\cos x+3\sin x)}$
28	$\left(\frac{\ln x}{x}\right)^3$	[1; 2]	40	0.2	0.001	$-\frac{(\ln x)^3+3(\ln x)^2/2+3(\ln x)/2+3/4}{2x^2}+\frac{3}{8}$
29	$\frac{x^3}{3+x}$	[1; 2]	72	0.125	0.001	$9x-\frac{3x^2}{2}+\frac{x^3}{3}-27\ln(3+x)-\frac{47}{6}+27\ln 4$
30	$\frac{x}{x^4+3x^3+2}$	[1; 2]	36	0.25	0.001	$\frac{1}{2}\ln\frac{x^2+1}{x^2+2}-\frac{1}{2}\ln\frac{2}{3}$

Варианты решений

Определите функцию LAST1, возвращающую последний элемент списка.

Функция:

```
(define (LAST1 L)
      (cond
            ((null? (cdr L)) (car L))           ;если только один элемент
                                                ;(нет хвоста) -
                                                ;возвратить его значение
            (t (LAST1 (cdr L)))                 ;иначе - вернуть
                                                ;последний эл-т хвоста
      )
)
```

Результат выполнения:

```
       [1] (last1 '(a bc def))
       DEF
```

Определите функцию DELLAST1, удалающую из списка последний элемент.

Функция:

```
(define (DELLAST L)
      (cond
            ((null? (cdr L)) nil)               ;если в L только один
элемент,
                                                ;то возвращаем nil
            (t (cons (car L) (DELLAST (cdr L))))
                                                ;если больше одного эл-та -
возвращаем
                                                ;список из первого эл-та L
            и
                                                ;"обрезанного" хвоста L
      )
)
```

Результат выполнения:

```
       [4] (DELLAST '(a bc def))
       (A BC)
```

Определите предикат (функцию) ATOMLIST, проверяющий, является ли предикат одноуровневым списком.

Функция:

```
(define (ATOMLIST L)
      (cond
            ((null? L) t)                       ;true - если список пуст
            ((atom? (car L)) (ATOMLIST (cdr L)))
                                                ;если голова списка -
                                                ;атом - продолжаем
                                                ;проверку хвоста
            (t nil)
      )
)
```

Результат выполнения:

```
       [1] (ATOMLIST '(a bc def))
       #T
       [2] (ATOMLIST '(a (bc def)))
       ()
```

Определите функцию ONION («луковица»), строящую N-уровневый вложенный список.

Функция:

```
(define (ONION N)
        (if (= n 0)
```

```
                n
                (cons (ONION (- n 1)) nil)
        )
)
```

Результат выполнения:

```
        [1] (onion 5)
        (((((0)))))
```

Определите функцию FIRSTATOM, результатом которой будет первый атом списка.

Функция:

```
(define (FIRSTATOM L)
        (cond
                ((atom? L) L)
                (t (FIRSTATOM (car L)))
        )
)
```

Результат выполнения:

```
        [1] (firstatom '((a bc) def))
```

A

Определите функцию, DELFIRST удаляющую из списка первое вхождение заданного элемента.

Функция:

```
(define (DELFIRST A L)
        (cond
                ((equal? (car L) A) (cdr L))
                (t (cons (car L) (DELFIRST A (cdr L))))
        )
)
```

Результат выполнения:

```
        [27] (delfirst 'bc '(a bc def bc))
        (A DEF BC)
```

Определите функцию, вычисляющую общее количество всех атомов в списке.

Функция:

```
(define (AtomCount L)
      (cond
            ((null? L) 0)                 ;список пуст – возвращаем 0
            ((atom? (car L))
                  (+ 1 (AtomCount (cdr L))));голова – атом
                                          ;возвращаем 1+число атомов
                        в хвосте
            (t
                  (+ (AtomCount (car L)) (AtomCount (cdr L)))
                                          ;голова – список. Обрабатываем
                        голову и хвост
                                          ;поэлементно.
            )
      )
)
```

Результат выполнения:

```
        [1] (atomcount '(a b c))
        3
        [2] (atomcount '(a (b c (d (e)))))
        5
```

Определите функцию двух аргументов-списков, возвращающую первый элемент,

входящий в оба списка и NIL, если такого элемента нет.

Функция:

```
(define (FirstCoincidence L1 L2)
      (cond
            ((null? L1) ;NIL – если первый список пуст
                  nil)
            ((member (car L1) L2)               ;если первый элемент
первого
                                                ;списка входит во второй
                  (car L1))                     ;возвращаем этот элемент
            (t
                                                ;иначе – применяем функцию
                        к
                                                ;хвосту первого списка
                  (FirstCoincidence (cdr L1) L2)
            )
      )
)
```

Результат выполнения:

```
       [0] (FirstCoincidence '(a b c d) '(f g c d))
       C
       [1] (FirstCoincidence '(a b c d) '(f g h i))
       ()
```

Определите функцию, преобразующую список в множество, то есть удаляющую повторения элемента в списке.

Функция:

```
(define (List2Set L)
      (cond
            ((null? L)
                                    ;если список пуст – возвращаем NIL
                  nil)
            ((member (car L)(cdr L))
                                    ;если голова входит в хвост –
выкидываем голову и
                                    ;применяем List2Set к хвосту
                  (List2Set (cdr L)))
            (t
                                    ;иначе – голова не повторяется в
хвосте.Оставляем
                                    ;ее в множестве и применяем List2Set к
            хвосту
                  (cons (car L) (List2Set (cdr L))))
      )
)
```

Результат выполнения:

```
       [1] (List2Set '(a b c a b c a b c))
       (A B C)
```

Определите предикат, проверяющий совпадение двух множеств (независимо от порядка следования элементов).

Функция:

```
(define (SetEQ X Y)
      (cond
            ((null? X)
                  (null? Y))
            ((member (car X) Y)
                  (SetEQ (cdr X) (delete! (car X) Y)))
            (t nil)
      )
```

```
)
```
Результат выполнения:
```
       [3] (SetEQ '(a b c d) '(a b c d))
       #T
       [4] (SetEQ '(a b c d) '(d c b a))
       #T
       [5] (SetEQ '(a b c d) '(e c b a))
       ()
```

Определите функцию, проверяющую, является ли одно множество подмножеством другого.

Функция:
```
(define (SubSet X Y)
      (cond
            ((null? Y)
                  (null? X))
            ((null? X)
                  t)
            ((member (car X) Y)
                  (SubSet (cdr X) Y))
            (t nil)
      )
)
```
Результат выполнения:
```
       [1] (SubSet '() '(a b c))
       #T
       [2] (SubSet '(c d) '(a b c d))
       #T
       [3] (SubSet '(d e) '(a b c d))
       ()
```

Определите предикат, проверяющий отсутствие общих элементов (отсутствие пересечения) двух множеств.

Функция:
```
(define (NonIntersect X Y)
      (cond
            ((null? X)
                  t)
            ((member (car X) Y)
                  nil)
            (t
                  (NonIntersect (cdr X) Y))
      )
)
```
Результат выполнения:
```
       [1] (NonIntersect '(e f) '(a b c d))
       #T
       [2] (NonIntersect '() '(a b c d))
       #T
       [3] (NonIntersect '(a b c d) '())
       #T
       [4] (NonIntersect '(a b c d) '(d e))
       ()
       [5] (NonIntersect '(a b c d) '(e f))
       #T
```

Определите FUNCALL через функционал APPLY. (Примечание: В Микро-ЛИСПе встроенные функции FUNCALL и &REST отсутствуют. Из за этого имеется возможность определить FUNCALL только для фиксированного количества

аргументов)

```
;Определим следующие функции:
;реализация для 2-х аргументов
(define (funcall1 fn L)
      (apply fn (list L))
)

;реализация для 3-х аргументов
(define (funcall2 fn L1 L2)
      (apply fn (list L1 L2))
)

;реализация для 4-х аргументов
(define (funcall3 fn L1 L2 L3)
      (apply fn (list L1 L2 L3))
)
```

Вычислите значения вызовов:

```
  a. (apply list '(a b))
[1] (apply list '(a b))
(A B)

  b. (funcall list '(a b))
[2] (funcall1 list '(a b))
((A B))

  c. (funcall apply list '(a b))
[3] (funcall2 apply list '(a b))
(A B)

  d. (funcall list 'apply '(a b))
[4] (funcall2 list 'apply '(a b))
(APPLY (A B))
```

Вычислите значения следующих функций:

```
     a. (mapcar list '(a b c))
[5] (mapcar list '(a b c))
((A) (B) (C))

     b. (mapc list '(a b c))
[6] (mapc list '(a b c))
#T
```

Вычислите значения интеграла:

$\frac{\ln x}{x\sqrt{1+\ln x}}$ - подинтегральная функция

```
(begin (newline)
       (display "                                        ln(x)   ")
       (newline)
       (display "Подинтегральная функция f(x) = -------------")
       (newline)
       (display "                                   x [1+ln(x)] ")
       (newline))
```

```
(begin (newline)
       (display "Введите исходные данные:")
       (newline))

(princ "-------------------------------------")
(newline)
(princ "Начало интервала: ")
(define a)
(set! a (read))

(princ "Конец интервала : ")
(define b)
(set! b (read))

(begin (newline)
       (display "Что будем задавать для вычисления:")
       (newline)
       (display "1. Количество частей разбиения")
       (newline)
       (display "2. Шаг интегрирования")
       (newline)
       (display "Ваш выбор (1/2): "))

(define kod)
(set! kod (read))


;(cond ((= kod 1) (princ "Количество частей разбиения: "))
;      ((= kod 2) (princ "Шаг интегрирования: ")))
(if (= kod 1) (princ "Количество частей разбиения: "))
(if (= kod 2) (princ "Шаг интегрирования: "))

(define h1)
(set! h1 (read))

;шаг интегрирования
(define h)
(if (= kod 1) (set! h (/ (- b a) h1)) (set! h h1))


(newline)
(princ "-------------------------------------")
(newline)

;функция
(define (f1 x)  (/
                   (log x)
                    (*
                      x
                        (sqrt (+ 1 (log x) ) )
                    )
                 )
)

;суммирование
(define (integ a b h)
        (cond ((< b a) 0)
                (t (+ (*(f1 b) h) (integ a (- b h) h) ))))

(display "Значение интеграла: ")
(writeln (integ a b h))
```

```
(newline)

;выход или нет
(begin (newline)
       (display "Задать новые исходные данные [Y/N]: ")
)
(set! kod (read))
(if (eq? kod 'Y) (load "integ1.lsp"))
(if (eq? kod 'N) (print '_))
```

Вычислите значения интеграла:

$\frac{\ln^2 x}{x}$ - подинтегральная функция

```
(begin (newline)
     (display "                                        ln²(x) ")
     (newline)
     (display "Подинтегральная функция f(x) = --------")
     (newline)
     (display "                                          x    ")
     (newline))

(begin (newline)
     (display "Введите исходные данные:")
     (newline))

(princ "-----------------------------------")
(newline)
(princ "Начало интервала: ")
(define a)
(set! a (read))

(princ "Конец интервала : ")
(define b)
(set! b (read))

(begin (newline)
     (display "Что будем задавать для вычисления:")
     (newline)
     (display "1. Количество частей разбиения")
     (newline)
     (display "2. Шаг интегрирования")
     (newline)
     (display "Ваш выбор (1/2): "))

(define kod)
(set! kod (read))


;(cond ((= kod 1) (princ "Количество частей разбиения: "))
;     ((= kod 2) (princ "Шаг интегрирования: ")))
(if (= kod 1) (princ "Количество частей разбиения: "))
(if (= kod 2) (princ "Шаг интегрирования: "))

(define h1)
(set! h1 (read))
```

```
;шаг интегрирования
(define h)
(if (= kod 1) (set! h (/ (- b a) h1)) (set! h h1))


(newline)
(princ "-------------------------------------")
(newline)

;функция
(define (f1 x)  (/(* (log x) (log x)) x) )

;суммирование
(define (integ a b h)
      (cond ((< b a) 0)
            (t (+ (*(f1 b) h) (integ a (- b h) h) ))))

(display "Значение интеграла: ")
(writeln (integ a b h))
(newline)

;выход или нет
(begin (newline)
     (display "Задать новые исходные данные [Y/N]: ")
)
(set! kod (read))
(if (eq? kod 'Y) (load "integ4.lsp"))
(if (eq? kod 'N) (print '_))
```