Введение в программирование на языке Лисп

Учебное пособие для начинающих

Лидия Городняя
Gorod@iis.nsk.su

Новсибирск, 2005

Содержание

Учебное пособие разработано при поддержке Российского фонда переподготовки кадров и сдано в печать в 2004 году.

Курс разработан на базе Высшего колледжа информатики Новосибирского госуниверситета.

Содержание курса соответствует PF4, PL7, PL5 классификатора CC2001CS.

Предисловие

Целью курса является изучение идей языка Лисп и методов функционального программирования. В курсе будут рассмотрены:

- История языка Лиспа.
- Идеи символьной обработки информации.
- Принципы функционального программирования.
- Методы программирования на Лиспе.

Автор языка Лисп – профессор математики и философии **Джон Мак-Карти**, выдающийся ученый в области искусственного интеллекта. Он предложил проект языка Лисп, идеи которого возбудили не утихающие до наших дней дискуссии о сущности программирования. Сформулированная Джоном Мак-Карти (1958) концепция символьной обработки информации восходит к идеям Чёрча (лямбда-исчисление) и других видных математиков конца 20-ых годов предыдущего века. Джон Мак-Карти предложил **функции** рассматривать как общее понятие, к которому могут быть сведены все другие понятия программирования.

Особый интерес представляют **рекурсивные функции** и методы их реализации в системах программирования. Понятие функции отчасти содержит концепцию времени: сначала вычисляются аргументы в порядке перечисления, затем в соответствии с заданным алгоритмом строится значение функции - ее результат. Лаконизм рекурсии может скрывать нелегкий путь к элегантному решению задачи. А. П. Ершов в предисловии к русскому переводу книги П.Хендерсона привел поучительный пример задачи о рекурсивной формуле, сводящей вычитание единицы из натурального числа к прибавлению единицы:

```
{1 –1 =  0 ;  ( n +1 )  -1 = n  } ,
```

не поддавшейся А.Чёрчу и решенной С.Клини лишь в 1932 году:

```
{ F (x, y, z) = если (x = 1) то 0
                              иначе
                  если  ((y +1) = x) то z
                                        иначе F (x, y +1, z +1) ;

  n –1  = F (n, 0, 0) }


алг F ( цел  x, y, z) арг x, y, z
      нач
          если  (x = 1)
             то знач := 0
```

```
          инес (y +1) /= x
       то знач := F (x, y +1, z +1)
    кон

алг N-1 (цел N) арг N нач  знач := F (N, 0, 0)  кон
```

Решение получилось через введение формально усложненной **вспомогательной функции с накопительными парметрами**, что противоречит интуитивному стремлению к монотонности движения от простого к сложному.

В настоящее время наблюдается устойчивый рост рейтинга интерпретируемых языков программирования и включение в компилируемые языки механизмов символьной обработки и средств динамического анализа, что повышает интерес к Лиспу и функциональному программированию.

В этом курсе мы сконцентрируемся на ключевой идее Лиспа - сведении понятия “программа” к взаимодействию разных категорий функций, а также на основах и методах функционального программирования. Курс может быть адресован студентам, любителям экспериментального программирования, преподавателям информатики и старшеклассникам. Подробно познакомимся с базисом Лиспа, проанализируем конструктивность методов программирования на Лиспе, изучим построение Лисп-системы и узнаем ее архитектурные особенности, рассмотрим методы эффективного и прикладного программирования в функциональном стиле. С математическими основами Лиспа можно ознакомиться подробнее на страницах журнала “Компьютерные инструменты в образовании” [5,6].

1. Рекурсивные функции и структуры

“Вот дом, который построил Джек …”

Примечание [D1]: Примечание. Этот парагарф адресован читателям, не знакомым с математическими оснвоами программирования и мтеодами реализации рекурсивных функций в системах программирования.

Этот парагарф адресован новичкам, не знакомым с математическими основами программирования и методами реализации систем программирования. На игровых, почти шуточных, моделях рассмотрим понятия:

- Рекурсия.
- Иерархия.
- Список.
- Функция.

Модель 1. "Матрешки"
Представим, что перед нами сувенир “матрешки”, вкладывающиеся друг в друга. Эта модель достаточна для показа типовых явлений, вызывающих к жизни **рекурсивные функции**.

Задача 1. Как узнать, сколько матрешек внутри?

Предварительный анализ и описание процесса решения:

1) Глядя на большую матрешку **трудно сказать, сколько** в ней спрятано ее меньших подруг. Но можно ее потрясти и по звуку понять, есть ли что-то внутри или она пуста.

2) **Если матрешка не пуста, то** ее можно разнять и вынуть внутреннюю матрешку, а части внешней матрешки соединить и поставить в сторонку, чтобы не мешали дальнейшему эксперименту.

3) **Возвращаемся к исходной задаче**, но с меньшей матрешкой, что **гарантирует завершение процесса**. (Реальные вещи не могут уменьшаться до бесконечности.)

4) **Если матрешка пуста, то** имеющиеся матрешки можно пересчитать – они все на виду.

Такого рода процесс может быть представлен как **рекурсивная функция, основанная на повторении действий и уменьшении данных** до тех пор, пока уменьшение становится невозможным. В таком случае выполняется завершающее действие. При решении задачи понадобились следующие базовые **функции (действия):**

- Разбираем внешнюю матрешку и убираем ее.
- Вынимаем внутренний комплект матрешек.
- Трясем матрешку, чтобы узнать, есть ли что-то внутри.

Задача 2. Пусть все матрешки разобраны. Надо их собрать в одну.

Предварительный анализ и описание процесса решения:

Для решения такой задачи понадобится еще две **функции**:

- Прячем меньшую матрешку в большую
- Сравниваем матрешки

1) Выбираем самую маленькую матрешку.

2) Сравниваем матрешки, ищем ближайшую к выбранной по размеру.

3) Прячем меньшую матрешку в большую.

4) Если еще остались матрешки, то возвращаемся к исходной задаче с уменьшенным числом матрешек.

5) Если матрешка всего одна, то процесс сборки завершен.

Здесь **рекурсивная функция** решения задачи включает в себя **поиск подходящего элемента из набора, уменьшающегося** на каждом витке процесса. Задача будет решена, когда обнаружится, что не из чего выбирать очередной элемент.

Задача 3. Надо выяснить сколько матрешек, а потом снова собрать их в одну.

Предварительный анализ и описание процесса решения:

При описании решения сразу учитываем, что понадобится не только разбирать матрешек, но и собрать их обратно.

1) Решаем задачу 1, но с небольшой поправкой в пункте 2:

- Если матрешка не пуста, то ее можно разнять и вынуть внутреннюю матрешку, а части внешней матрешки ставить рядом по очереди, чтобы потом их было удобно брать для сборки.

2) Решаем задачу 2, но с учетом, что части разобранных матрешек стоят по порядку. Значит, пункт 2 можно уточнить:

- Берем части очередной матрешки.

Согласованное решений двух задач для более общей задачи здесь выражено в организационных решениях по расположению промежуточных результатов, а также в уточнении ранее существовавшего представления базовых функций.

Модель 2. “Конверты и карточки”.

Другой ряд иллюстраций дает модель из конвертов, вкладываемых друг в друга. Конверты можно надписывать. Пустой конверт можно заменить карточкой с той же надписью – это **атом**. Внутрь коверта можно, как в матрешки, вкладывать конверты или карточки. Подобным образом устроены иерархические структуры данных - **символьные выражения (S-выражения)**. Вкладывание объекта в конверт - простейшая модель **консолидации** двух данных, сцепления их в одно более сложное данное.
Примеры:

Пусть в конверт с надписью «ГОЛОВА» вложен пустой конверт с написью «ХВОСТ». Такую структуру данных можно изобразить в виде S-выражения:

(ГОЛОВА **.** ХВОСТ)

Заключение в скобки символизирует целостность структуры. Если конверт с написью «ХВОСТ» заранее вложить в конверт с написью «СЕРЕДИНА», а его вложить в конверт с надписью «ГОЛОВА», то соотвествующую структуру данных можно записать в виде S-выражения:

(ГОЛОВА **.** (СЕРЕДИНА **.** ХВОСТ))

Пустой конверт или карточку без написи можно изображать парой скобок.

( )

Модель 3. “Кошелек”.
Наклеим на лицевую сторону конверта большой карман. Получится нечто вроде кошелька. Вместо надписи на конверте можно в карман кошелька вкладывать надписанную карточку.

Возьмем карточку с написью «ЭЛЕМЕНТ» и вложим ее в карман пустого кошелька без написи – получится модель одноэлементного списка:

(ЭЛЕМЕНТ)

Многократным применением такого процесса можно строить списки произвольной длины.

(ЧЕМОДАН ПОРТФЕЛЬ РЮКЗАК ПОРТМОНЕ)

Если разрешить в карман кошелька вместо карточек вкладывать другие кошельки, то элементы списка можно конструировать и получать многоуровневые списки вида:

((ЭЛ1-1 ЭЛ1-2) ЭЛ 2 (ЭЛ3-1 ЭЛ3-2 ЭЛ3-3) ЭЛ4)

По традиции в Лиспе пустой список изображается атомом **NIL**. Атомарность пустого списка вполне естественна в мире вещей – это прямой аналог упаковки.

Пустой конверт - обычная вещь, которую можно наполнить, тогда она становится сложной, или опустошить, тогда она становится простой.

Модель 4. “Форма , вид и пометки”.
При организации процессов **функции и их аргументы используются совместно** и каждый комплект аргументов функции надо выделять. Поэтому в Лиспе их располагают в общем списке. Карман кошелька может содержать карточку, на которой написано, что делать с его содержимым - аргументами.
Примеры:

```
(fn arg1 arg2 ... argN)
(Собрать портфель рюкзак)
(+ 1 2 3 4 5 )

(fn (gn arg1) arg2 ... argN)
(Сложить (Найти портфель) (Подготовить Список_предметов))
(+ 5 (*  2  3.14   R) X)
```

При таком подходе список, представляющий вызов функции, внешне мало отличается от списка, представляющего данные, не требующие особой обработки. В таких случаях конверт можно заклеить, а в карман вставить карточку с надписью “QUOTE”, что соотвествует **блокировке вычислений**. Увидев, что конверт заклеен, можно так его и оставить или расклеить. В последнем случае его содержимое становится доступным для обработки в будущем. В записи это можно изобразить символом апострофа "'".

Примеры:

```
(СОБРАТЬ ‘(ШКАТУЛКА КОЛЬЦО СЕРЬГИ) СУМКА)
(fn 'arg1 arg2 ... argN)
(cons '(+ 1 2 3 4 5) arg2 ... argN)
```

При такой записи первые аргументы функций вычисляться не будут. (Расположены вторыми в списках.)

Обычно подразумевают, что функция применяется к заранее вычисленным значениям ее аргументов. Но если в качестве данных допускать не только значения, но и символьные формы - записи программ для вычисления разных значений, то можно рассматривать и реализовывать специальные функции, способные обрабатывать аргументы нестандартно по любой схеме. Такие варинаты нужны для реализации присвоений и управления процессами. Конверты, кошельки и карточки могут быть разного цвета, они могут быть помечены перфорацией, марками, текстами и рисунками. Такие пометки можно понимать как указания на схему обработки конвертов и их содержимого ("Перед прочтением - сжечь!"), что и позволяет моделировать функции и функциональные программы, характерные для программирования на Лиспе. Подобным образом организуют отображения – нечто вроде переноса рисунка вышивки с бумаги на ткань.

2. Идеальный Лисп

- Основы символьной обработки.
- Списочная запись
- Точечная нотация
- Элементарные функции

1. Основы символьной обработки.

Идеальный (чистый, элементарный) Лисп – это свод принципиальных особенностей программирования в функциональном стиле. Начинается он с выбора подходящей структуры данных и минимального набора функций над выбранной структурой. Информационная обработка в языке Лисп отличается от большинства подходов к программированию тремя важными принципами:

1) Природа данных

Все данные представляются в форме **символьных выражений**. В Лиспе Дж. Мак-Карти назвал их S-выражениями. Система программирования над такими структурами обычно использует для их хранения всю доступную память, поэтому программист может быть освобожден от распределения памяти под отдельные блоки данных.

2) Самоописание обработки символьных выражений

Важная особенность программирования на Лиспе - описание способов обработки S-выражений представляется программами, рассматриваемыми как символьные выражения. Программы строятся из **рекурсивных функций** над S-выражениями. Определения и вызовы этих функций, как и любая информация, имеют вид S-выражений, то есть формально они могут обрабатываться как обычные данные, получаться в процессе вычислений и преобразовываться как **значения**.

3) Подобие машинным языкам

Система программирования на Лиспе допускает, что программа может интерпретировать и/или компилировать программы, представленные в виде S-выражений. Это сближает программирование на Лиспе с методами низкоуровневого программирования и отличает от традиционной методики применения языков высокого уровня.

Структуры данных

Любые **структуры данных** начинаются с элементарных значений. В Лиспе такие значения называют **атомами** или **символами**. Внешне атом обычно выглядит как последовательность из букв и цифр, начинающаяся с буквы.

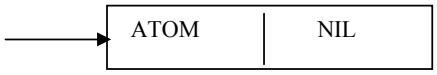
A
Nil
ATOM
LISP
Занятие2
Новый_год
ВотДлинныйАтомНуОченьДлинныйНоЕслиНадоАтомМожетБытьЕщеДлинннее
Ф4длш139к131б

Одинаково выглядящие атомы не различимы по своим свойствам. Термин “атом” выбран по аналогии с химическими атомами, строение которых – предмет другой науки. Согласно этой аналогии атом может иметь достаточно сложное строение, но атом не предназначен для разбора на части базовыми средствами языка.

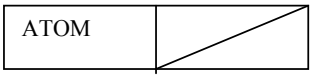
Более сложные данные выстраиваются из одинаково устроенных **блоков памяти**. В Лиспе это бинарные узлы, содержащие пары объектов произвольного вида. Каждый бинарный узел соответствует минимальному блоку памяти, выделяемому системой программирования при организации и обработке структур данных. Выделение блока памяти и размещение в нем пары данных выполняет функция **CONS** (от слова consolidation), а извлечение левой и правой частей из блока выполняют функции **CAR** и **CDR** соответственно (“content of adress part of register” , “content of decrement part of register”).

Функция	Аргументы	Результат
Cons	Атом X	(Атом . X)
Car	(Атом . X)	Атом
Cdr	(Атом . X)	X

По соглашению атом Nil выполняет роль **пустого списка**. Бинарный узел, содержащий пару атомов ATOM и **Nil**, рассматривается как одноэлементный **список** (ATOM) :



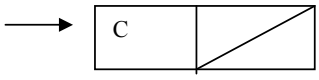
или для наглядности:



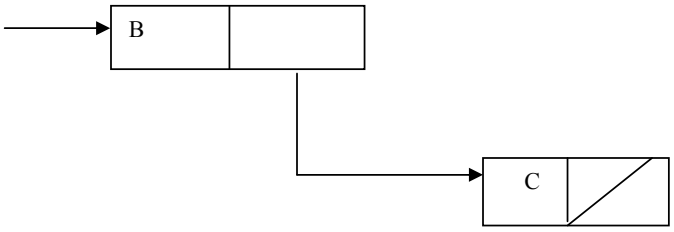
Если вместо атома “ATOM” подставлять произвольные атомы, а вместо “Nil” - произвольные списки, затем вместо атомов - построенные списки и так далее, то мы получим множество всех возможных списков. Можно сказать, что **список - это заключенная в скобки последовательность из разделенных пробелами атомов или списков.**

Примеры:

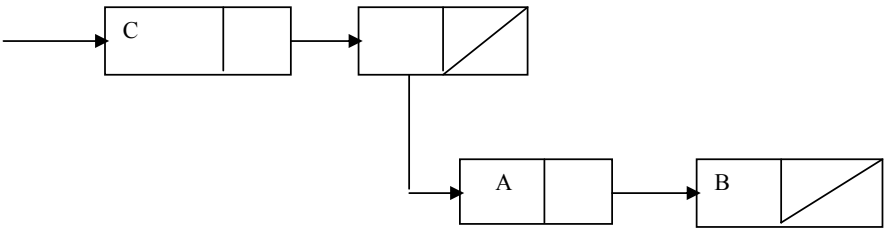
(C)



(B C)



(C (A B))



Такая форма записи S-выражений называется **списочной записью** (list-notation).

Список – это перечень произвольного числа элементов, разделенных пробелами, заключенный в круглые скобки.

Элементы списка могут быть любой природы.

Упражнения: Нарисуйте диаграммы для списков вида:

((A B) C)

((A B) (D C))

((A B)(D(C E)))

Любой список может быть построен из пустого списка и атомов с помощью CONS и любая его часть может быть выделена с помощью подходящей композиции CAR-CDR.

CONS - Функция, которая строит списки из бинарных узлов, заполняя их парами объектов, являющихся значениями пары ее аргументов. Первый аргумент произвольного вида размещается в левой части бинарного узла, а второй, являющийся списком, - в правой.

CAR – Функция, обеспечивающая доступ к первому элементу списка - его "**голове**".

CDR – Функция, укорачивающая список на один элемент. Обеспечивает доступ к "**хвосту**" списка, т.е. к остатку списка после удаления его головы.

ATOM - Функция, различающая **составные** и атомарные **объекты.** На атомах ее значение "истина", а на более сложных структурах данных – "ложь".

EQ – Функция, которая проверяет **атомарные** объекты на **равенство**.

Таблица 2.1 **Элементарные функции над списками**

Примеры соответствия между аргументами и результатами элементарных функций обработки списков .

Функция	Аргументы	Результат
	Конструирование структур данных	
CONS	A и Nil	(A)
CONS	(A B) и Nil	((A B))
CONS CONS	A и (B) (Результат предыдущего CONS) и (C)	(A B) ((A B) C)
CONS	A и (B C)	(A B C)
	Доступ к компонентам структуры данных:	
	Слева	
CAR	(A B C)	A
CAR	(A (B C))	A
CAR	((A B) C)	(A B)
CAR	A	Не определен
	Справа	
CDR	(A)	Nil
CDR	(A B C D)	(B C D)
CDR	(A (B C))	((B C))
CDR	((A B) C)	(C)
CDR	A	Не определен

	Обработка данных:	
CDR CAR	(A B C) Результат предыдущего CDR	(B C) B
CAR CAR	(A C) Результат предыдущего CAR	A Не определен
CONS CAR	A и (B) Результат предыдущего CONS	(A B) A
CONS CDR	A и (B) Результат предыдущего CONS	(A B) (B)
	Предикаты:	
	Атомарность – неделимость	
ATOM	VeryLongStringOfLetters	T
ATOM	(A B)	Nil - выполняет роль ложного значения
CDR ATOM	(A B) Результат предыдущего CDR	(B) Nil
ATOM	Nil	T
ATOM	( )	T
	Равенство	
EQ	A A	T
EQ	A B	Nil
EQ	A (A B)	Nil
EQ	(A B) (A B)	Не определен
EQ	Nil и ()	T

Различие истиностных значений в Лиспе принято отождествлять с разницей между пустым списком и остальными объектами, которым программист может придать в программе некоторый другой смысл. Таким образом, значение “**ложь” – это всегда Nil**.

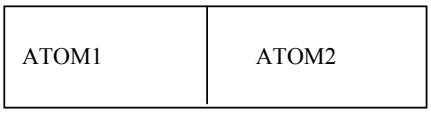
Если требуется явно изобразить значение “истина”, то используется константа – атом **T (true)** как стандартное представление, но роль такого значения может выполнить любой, отличный от пустого списка, объект.

Точечная нотация

Исторически при реализации Лиспа в качестве единой базовой структуры для конструирования S-выражений использовалась так называемая “**точечная**

нотация" (dot-notation), согласно которой левая и правая части бинарного узла равноправны и могут хранить данные любой природы.

Бинарный узел, содержащий пару атомов ATOM1 и ATOM2,



можно представить в виде S-выражения вида:

(ATOM1 **.** ATOM2)

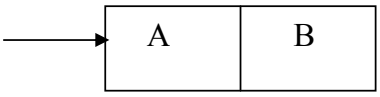
Если вместо атомов "ATOM1", "ATOM2" рекурсивно подставлять произвольные атомы, затем построенные из них пары и так далее, то мы получим множество всех возможных составных S-выражений.

S-выражение - это или атом или заключенная в скобки пара из двух S-выражений, разделенных точкой.

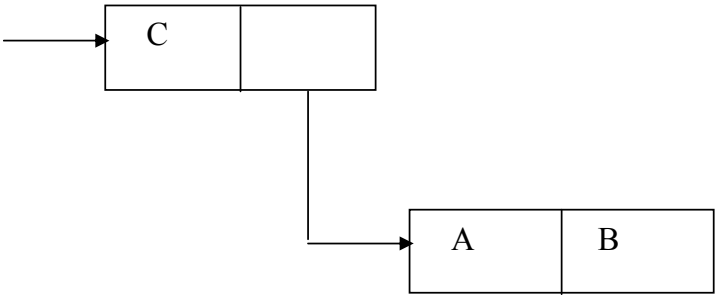
Все сложные данные выстраиваются из одинаково устроенных блоков - бинарных узлов, содержащих пары объектов произвольного вида. Каждый бинарный узел соответствует минимальному блоку памяти.

Примеры:

(A . B)



(C . (A . B))



Любое S-выражение может быть построено из атомов с помощью CONS и любая его часть может быть выделена с помощью CAR-CDR.

Упражнение. Нарисуйте диаграммы для следующих S-выражений:

((A . B) . C)

((A . B) . (D . C))

((A . B) . (D . (C . E)))

Расширение типа данных, допускаемых в качестве второго аргумента “CONS”, ни в малейшей степени не усложняет реализацию этой функции, равно как и реализацию функций “CAR” и “CDR”, зато их описания становятся проще:

CONS - Функция, которая строит бинарный узел и заполняет его парой объектов, являющихся значениями пары ее аргументов. Первый аргумент размещается в левой части бинарного узла, а второй - в правой.

CAR – Функция, обеспечивающая доступ к объектам, расположенным слева от точки в точечной нотации, т.е. в левой части бинарного узла.

CDR - Функция, обеспечивающая доступ к объектам, расположенным справа от точки в точечной нотации, т.е. в правой части бинарного узла.

Таблица Элементарные функции **над произвольными S-выражениями**

Функция	Аргументы	Результат
	Конструирование структур данных	
CONS	A и B	(A . B)
CONS	(A . B) и C	((A . B) . C)
CONS CONS	A B (Результат предыдущего CONS) и C	(A . B) ((A . B) . C)
	Доступ к компонентам структуры данных:	
	Слева	
CAR	(A . B)	A
CAR	((A . B) . C)	(A . B)
	Справа	
CDR	(A . B)	B
CDR	(A . (B . C))	(B . C)
	Обработка данных:	
CDR CAR	(A . (B . C)) Результат предыдущего CDR	(B . C) B
CDR CAR	(A . C) Результат предыдущего CDR	C Не определен
CONS	A и B	(A . B)

CAR	Результат предыдущего CONS	A
CONS CDR	A и B Результат предыдущего CONS	(A . B) B
	Тождества: (на произвольных объектах)	
CONS CAR	Два произвольных объекта x и y Результат предыдущего CONS	Исходный объект x (первый аргумент CONS)
CONS CDR	Два произвольных объекта x и y Результат предыдущего CONS	Исходный объект y (второй аргумент CONS)
CAR CDR CONS	Произвольный составной объект x - не атом. Тот же самый объект x. Результаты предыдущих CAR и CDR	Исходный объект x
	Предикаты:	
	Атомарность – неделимость	
ATOM	(A . B)	Nil - выполняет роль ложного значения
CDR ATOM	(A . B) Результат предыдущего CDR	B T
	Равенство	
EQ	(A . B) (A . B)	Не определен

Точечная нотация точно представляет логику хранения любых структур данных в памяти и доступа к компонентам структур данных. В виде списков можно представить лишь те S-выражения, в которых при движении вправо в конце концов обнаруживается атом Nil, символизирующий завершение списка.

Атом Nil, рассматриваемый как представление пустого списка (), выполняет роль ограничителя в списках. Одноэлементный список (A) идентичен S-выражению (A . Nil). Список (A1 A2 … Ak) может быть представлен как S-выражение вида:

(A1 . (A2 . (… . (Ak . Nil) …))).

В памяти это фактически одна и та же структура данных.

Таблица 2.3. **Соответствие списков и** равнозначных **им S-выражений**

List-notation - списочная запись объекта	Dot-notation - точечная запись того же объекта
(A B C)	(A **.** (B **.** (C **.** Nil)))
((A B) C)	((A **.** (B **.** Nil)) **.** (C **.** Nil))
(A B (C E))	(A **.** (B **.** ((C **.** (E **.** Nil))**.** Nil)))
(A)	(A **.** Nil)
((A))	((A **.** Nil) **.** Nil
(A (B **.** C))	(A **.** ((B **.** C) **.** Nil))
(())	(Nil **.** Nil)
(A B **.** C)	(A **.** (B **.** C))

Для многошагового доступа к отдельным элементам такой структуры удобно пользоваться мнемоничными обзначениями **композиций** из многократных CAR-CDR. Имена таких композиций устроены как цепочки из “a” или “d”, задающие **маршрут движения** из шагов CAR и CDR соотвественно, расположенный между “c” и “r”. Указанные таким способом CAR-CDR исполняются с ближайшего к аргументу шага, т.е. в порядки обратном записи.

Таблица. Примеры многошагового доступа к элементам структуры.

	Композиции CAR-CDR	Вычисляются в порядке, обратном записи:
Caar	((A) B C)	A
Cadr	(A B C)	B - CDR затем CAR
Caddr	(A B C)	C - (дважды CDR) затем CAR
Cadadr	(A (B C) D)	C - два раза (CDR

		затем CAR)

Выводы:

- *Список – это перечень произвольного числа элементов, разделенных пробелами, заключенный в круглые скобки.*
- *Элементы списка могут быть любой природы.*
- *S-выражение - это или атом или заключенная в скобки пара из двух S-выражений, разделенных точкой. Список – частный случай S-выражения.*
- *Любое S-выражение может быть построено из атомов с помощью CONS и любая его часть может быть выделена с помощью CAR-CDR.*
- *Для изображения S-выражений используют различные нотации: графическую, точечную и списочную.*
- *Базис Лиспа содержит элементарные функции CAR, CDR, CONS, EQ,, ATOM .*

3. Запись Лисп-программ

Теперь рассмотрим правила записи программ на Лиспе.

- Правила записи Лисп-программ
- Специальные функции
- Безымянные функции
- Рекурсивные функции

Основные понятия, возникающие при написании программ – это переменные, константы, выражения, ветвления, вызовы функций и определения. Все они представимы с помощью S-выражений.

1) Самая простая **форма** выражения - это **переменная**. Она может быть представлена как атом.

Примеры:

```
X
n
Variable1
Переменная2
LongSong
ДолгаяПесня
```

2) **Имена** функций, как и переменных, лучше всего изображать с помощью атомов, для наглядности можно предпочитать заглавные буквы.

Примеры:

```
CONS
CAR
CDR
ATOM
EQ
```

3) Все более сложные **формы** понимают как применение функции к ее аргументам (**вызов функции**). **Аргументом функции** может быть любая форма. Список, первый элемент которого – представление функции, остальные элементы - аргументы функции, – это основная конструкция в Лисп-программе.

Пример:

```
(функция аргумент1 аргумент2 ... )
```

4) **Композиции функций** естественно строить с помощью вложенных скобок.

Пример:

```
(функция1 (функция2 аргумент21 аргумент22 ... ) аргумент2 ... )
```

Этих правил достаточно, чтобы более ясно выписать **основные тождества** Лиспа, формально характеризующие элементарные функции CAR, CDR, CONS, ATOM, EQ над S-выражениями:

(CAR (CONS x y)) = x
(CDR (CONS x y)) = y
(ATOM (CONS x y)) = Nil
(CONS (CAR x) (CDR x)) = x *для неатомарных x.*
(EQ x x) = T *если x атом*
(EQ x y) = Nil *если x и y различимы*

Любые композиции заданного набора функций над конечным множеством произвольных объектов можно представить таким способом, но класс соответствующих им процессов весьма ограничен и мало интересен. Организация более сложного класса процессов требует более детального представления в программах соответствия между именами и их значениями или определениями, изображения ветвлений и объявления констант.

Специальные функции

5) **Константа** представляется как аргумент специальной функции **QUOTE** в виде списка:

Пример: Список из атомов объявлен константой.

```
(QUOTE  (C O N S T ))
```

Используется и сокращенная запись – апостроф перед произвольным данным.

Пример:

```
‘(C O N S T )
```

В зависимости от контекста одни и те же объекты могут выполнять роль переменных или констант, причем значения и того, и другого могут быть произвольной сложности. Если объект выполняет роль константы, то для объявления константы достаточно заблокировать его вычисление, то есть как бы взять его в кавычки (quotation), отмечающие буквально используемые фразы, не требующие обработки. Для такой **блокировки** вводится специальная функция **QUOTE**, предохраняющая свой единственный аргумент от вычисления.

Примеры:

(QUOTE A)	константа A объявлена.
(QUOTE (A B C))	константа (A B C) объявлена.
(ATOM (QUOTE A)) = T	аргументом предиката является атом “A”

(ATOM (QUOTE (A B C))) = Nil	аргументом предиката является список (A B C)
(ATOM A)	значение не определено, т.к. оно зависит от вхождения переменной A, а ее значение зависит от контекста и должно быть определено или задано до попытки выяснить атом ли это.
(третий (QUOTE (A B C)))	применение новой функции к значению, не требующему вычисления

Упражнение. Запишите выражения, соответствующие применению функций из вышеприведенных таблиц.

6) **Построить функцию** можно с помощью Lambda-конструктора:
Пример:

```
(LAMBDA  (x)       (CAR (CDR (CDR x))) )
              |            |_____________|_____определение функции
              |__________________________ параметр функции
```

При вызове такой безымянной функции заодно происходит задание значений параметров - связанных переменных:

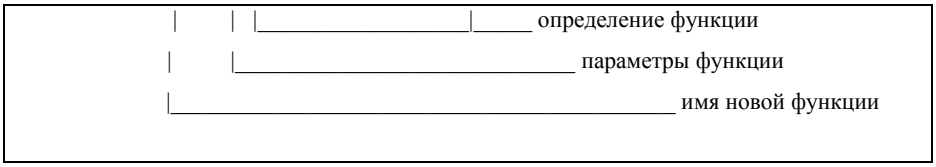
```
((LAMBDA (x) (atom x))  123)  ; = T
```

X получит значение 123 на время применения построенной анонимной функции, действие которой заключается в выяснении, атом ли аргумент функции.

Связанную переменную можно объявить специальной функцией Lambda, а значение она получит при вызове функции.

7) Соответствие между именем функции и ее определением можно задать с помощью специального **конструктора функций DEFUN**, первый аргумент которого - имя функции, второй – собственно именуемое определение функции. Формальным результатом DEFUN является ее первый аргумент, который становится **объектом другой категории**. Он меняет свой статус – теперь это имя новой функции.

```
(DEFUN третий (x) (CAR (CDR (CDR x))) ))
```



Новая функция “третий” действует так же как “Caddr” в таблице 2.4.

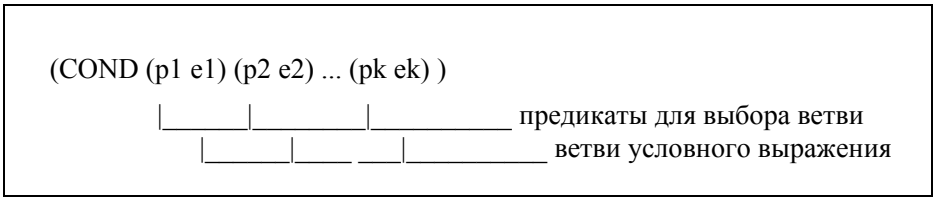
Именование функций работает подобно заданию значений переменным. Идентификатор представляет структуру, символизирующую функциональный объект. В ней содержится список формальных параметров функции и тело ее определения.

Обычно в рассуждениях о переменных и константах молчаливо подразумевается, что речь идет лишь о данных. Разница между константами и переменными заключается лишь в том, что значение переменной может быть в любой момент изменено, а **константа изменяется существенно реже**. Лисп рассматривает представления функций как данные, поэтому **функции могут быть как константными, так и переменными**.

Представления функции могут вычисляться и передаваться как параметры или результаты других функций.

Соответствие между именем функции и ее определением может быть изменено, подобно тому как меняется соответствие между именем переменной и ее значением.

8) Ветвление (**условное выражение**) характеризуется тем, что ход процесса зависит от некоторых условий. Условия следует сгруппировать в общий комплект и соотнести с подходящими ветвями. Такую организацию процесса вычисления обеспечивает специальная
функция **COND** (condition). Ее аргументами являются двухэлементные списки, содержащие **предикаты и соответствующие им выражения**. Аргументов может быть любое число. Обрабатываются они по особой схеме: сначала вычисляются первые элементы аргументов по порядку, **пока не найдется предикат со значением “истина”**. Затем выбирается второй элемент этого аргумента и вычисляется его значение, которое и считается значением всего условного выражения.



Каждый предикат pi или ветвь ei может быть любой формы: переменная, константа, вызов функции, композиция функций, условное выражение.

Обычное условное выражение (if Predicate Then Else) или (если Predicate то Then иначе Else) может быть представлено с помощью функции COND следующим образом:

```
(COND (Predicate Then)(T Else))
```

Или более наглядно:

```
(COND (Predicate Then)
        (T      Else  )
)
```

Вычисление ряда форм в определении может быть обусловлено заранее заданными предикатами.

Пример:

```
(COND ((EQ (CAR x) (QUOTE A)) (CONS (QUOTE B) (CDR x))) (T x) )
```

Атом “T” представляет тождественную истину. Значение всего условного выражения получается заменой первого элемента из значения переменной x на B в том случае, если (CAR x) совпадает с A.

Объявленные здесь **специальные функции** QUOTE, COND LAMBDA и DEFUN существенно отличаются от **элементарных функций** CAR, CDR, CONS, ATOM, EQ правилом обработки аргументов. Обычные функции получают значения аргументов, предварительно вычисленные системой программирования по формулам фактических параметров функции.

Специальные функции не требуют такой предварительной обработки параметров.

Они сами могут выполнять все необходимое, используя представление фактических параметров в виде S-выражений.

Рекурсивные функции: определение и исполнение

9) Определения могут быть рекурсивны.

Как правило рекурсивное применение функций должно быть определено в комплекте с не рекурсивными ветвями процесса. Основное преназначение условных выражений - **рекурсивные определения** функций.

Для примера рассмотрим функцию, выбирающую в списке самый левый атом:

```
алг  Левейший ( список x) арг x
     нач
        если  Atom (x)
           то знач := x
           иначе знач := Левейший (Car (x))
     кон
```

Новая функция "**Левейший**" выбирает первый атом из любого данного.

```
(DEFUN Левейший (x)(COND ((ATOM x) x)
                         (T (Левейший (CAR x))) ))
```

Если x является атомом, то он и является результатом, иначе функцию "**Левейший**" следует применить к первому элементу значения x, которое получается в результате вычисления формулы (CAR x). На составных x будет выполняться вторая ветвь, выбираемая по тождественно истинному значению встроенной константы T.

Определение функции "**Левейший**" рекурсивно. Эта функция действительно работает в терминах самой себя. Важно, что для любого S-выражения существует некоторое число применений функции CAR, после которого из этого S-выражения выделится какой-нибудь атом, следовательно процесс вычисления функции всегда определен, детерминирован, завершится за конечное число шагов. Можно сказать, что для определенности рекурсивной функции следует формулировать **условие ее завершения**.

Введенные обозначения достаточны, чтобы пронаблюдать формирование значений и преобразование форм в процессе исполнения функциональных программ.

Рассмотрим вычисление формы:

```
((DEFUN Левейший  (LAMBDA (x)(COND ((ATOM x) x)
                                   (T (Левейший (CAR x))) )))
  (QUOTE ((A . B) . C) )
```

)

DEFUN дает имена обычным функциям, поэтому фактический параметр функции “**Левейший**” будет вычислен до того как начнет работать ее определение и переменная “x” получит значение “((A . B) . C))”.

x = ((A . B) . C))

Таблица. **Схема вывода результата формы с рекурсивной функцией.**

Вычисляемая форма	Очередной шаг	Результат и комментарии
		Вход в рекурсию
(**Левейший** (QUOTE ((A . B) . C)))	Выбор определения функции и	(COND ((ATOM x) x) (T (**Левейший** (CAR x))))
		Первый шаг рекурсии
	Выделение параметров функции	(QUOTE ((A . B) . C))
(QUOTE ((A . B) . C))	Вычисление аргумента функции	X = ((A . B) . C)
(COND ((ATOM x) x) (T (**Левейший** (CAR x))))	Перебор предикатов: выбор первого	(ATOM x)
(ATOM x)	Вычисление первого предиката	Nil = “ложь”, т.к. X – не атом. Переход ко второму предикату
T	Вычисление второго предиката	T = “истина” – константа. Переход к выделенной ветви
		Второй шаг рекурсии
(**Левейший** (CAR x))	выделение параметров функции	(CAR x)
(CAR x)	Вычисление аргумента функции	X = (A . B) Рекурсивный переход к редуцированному аргументу
(COND ((ATOM x) x)	Перебор предикатов: выбор	(ATOM x)

(T (**Левейший** (CAR x))))	первого	
(ATOM x)	Вычисление первого предиката	Nil = “ложь”, т.к. X – не атом. Переход ко второму предикату
T	Вычисление второго предиката	T = “истина” – константа. Переход ко второй ветви
		Третий шаг рекурсии
(**Левейший** (CAR x))	выделение параметров функции	(CAR x)
(CAR x)	Вычисление аргумента функции	X = A Рекурсивный переход к редуцированному аргументу
(COND ((ATOM x) x) (T (**Левейший** (CAR x))))	Перебор предикатов: выбор первого	(ATOM x)
(ATOM x)	Вычисление первого предиката	T – т.к. X теперь атом Преход к первой ветви
X	Вычисление значений переменной	A Значение функции получено и вычисление завершено
		Выход из рекурсии

Некоторые определения функций могут быть хорошо определены на одних аргументах, но зацикливаться на других, подобно традиционному определению факториала при попытке его применить к отрицательным числам. Результат может выглядеть как исчезновение свободной памяти или слишком долгий счет без видимого прогресса. Такие функции называют **частичными**. Их определения должны включать в себя **ветвления** для проверки аргументов на принадлежность фактической области определения функции - **динамический контроль**. Условные выражения не менее удобны и для численных расчетов.

Пример 1. Абсолютное значение числа.

(Запись на алгоритмической нотации)

```
алг АБС( цел x) арг x
    нач
        если (x < 0)
            то знач := - x
        иначе знач := x
    кон
```

(Лисп-программа)

```
(DEFUN Абс  (LAMBDA (x)(COND ((< x 0 ) (- x))
                                                (T   x)    )))
```

Пример 2. Факториал неотрицательного числа.

(Запись на алгоритмической нотации)

```
алг ФАКТОРИАЛ ( цел  N) арг N
    нач
        если  (N = 0)
            то знач := 1
        иначе   знач := N * ФАКТОРИАЛ (N - 1)
    кон
```

(Лисп-программа)

```
(DEFUN Факториал  (LAMBDA (N)(COND ((= N 0 ) 1 )
                                    (T  ( *  N  (Факториал (- N 1 )))  )  )))
```

Это определение не завершается на отрицательных аргументах.

Включается и набор наиболее употребимых базовых операций над такими данными. Если введены числа, то введены и традиционные **арифметические операции**, но форма их применения подчинена общим правилам:

```
(+ 1 2 3 4)   = 10
```

Функция, которая определена лишь для некоторых значений аргументов естественной области определения, называется **частичной функцией**.

Пример 3. Алгоритм Евклида для нахождения наибольшего общего делителя двух положительных целых чисел при условии, что определена функция “Остаток”.

(Запись на алгоритмической нотации)

```
алг НОД ( цел  x, y) арг x
    нач
        если  (x < y)
            то знач := НОД ( y, x)
        инес Остаток (y, x) = 0
            то знач := x
```

```
        иначе  знач := НОД (Остаток (y, x), x)
    кон
```

остаток [x, y] - функция, вычисляющая остаток отделения x на y.

(Лисп-программа)

```
(DEFUN НОД (LAMBDA (x y)(COND ((< x   y) (НОД y x))
                                        ((= (остаток y x ) 0 ) x )
                                         (T (НОД (остаток y x) x )) ))
)
```

Как и любое S-выражение, символьные представления функций могут быть значениями аргументов - **функциональные аргументы**.

Базис элементарного Лиспа образуют пять функций над S-выражениями CAR, CDR, CONS, ATOM, EQ и четыре специальных функции, обеспечивающие управление программами и процессами и конструирование функциональных объектов QUOTE, COND, LAMBDA, DEFUN.

Далее мы построим определение универсальной функции **EVAL**, позволяющей вычислять значения выражений, представленных в виде списков, - **правило интерпретации выражений**.

Формально для перехода к практике нужна несколько большая определенность по механизмам исполнения программ, представленных S-выражениями:

- аргументы функций как правило вычисляются в порядке их перечисления,

- композиции функций выполняются в порядке от самой внутренней функции наружу до самой внешней,

- представление функции анализируется до того как начинают вычисляться аргументы, т.к. в случае специальных функций аргументы можно не вычислять,

- при вычислении лямбда-выражений **связи между именами переменных и их значениями**, а также между именами функций и их определениями, накапливаются в так называемом **ассоциативном списке**, пополняемом при вызове функции и освобождаемом при выходе из функции.

Выводы:

- *Основные понятия, возникающие при написании программ – это переменные, константы, выражения, ветвления, вызовов функций и определения. Все они представимы с помощью S-выражений.*

- *Связанную переменную можно объявить специальной функцией Lambda, а значение она получит при вызове функции.*

- *Представления функции могут вычисляться и передаваться как параметры или результаты других функций.*

- *Специальные функции не требуют предварительной обработки параметров.*

- ***Базис элементарного Лиспа** образуют пять функций над S-выражениями CAR, CDR, CONS, ATOM, EQ и четыре специальных функции, обеспечивающие управление программами и процессами и конструирование функциональных объектов QUOTE, COND, LAMBDA.*

- *И данные, и программа в Лиспе представляются с помощью S-выражений. Различать данные от программы можно с помощью функции QUOTE.*

- *Определение и имя новой функции можно задать с помощью специальной функции DEFUN.*

4. **Определение языка**

Теперь можем дать более точное определение Лиспа как языка программирования.

- Синтаксис Лиспа
- Семантика Лиспа
- Накопительные параметры
- Вспомогательные функции

Начнем с синтаксического обобщения.

Определение языка программирования обычно начинают с синтаксических формул, называемых БНФ (формулы Бекуса-Наура). Определение таких формул сводится к следующим положениям:

- Язык характеризуется набором определяемых понятий.
- Каждому понятию соответствует набор вариантов синтаксических формул.
- Каждый вариант – это последовательность элементов.
- Элемент – это или терминальный символ или синтаксическое понятие – нетерминальный символ.

Синтаксис данных в Лиспе сводится к правилам представления атомов и S-выражений.

```
<атом> ::= <БУКВА> <конец_атома>

<конец_атома> ::= <пусто>
            | <БУКВА> <конец_атома>
            | <число> <конец_атома>
```

В Лиспе атомы - это мельчайшие частицы. Их разложение по литерам не имеет смысла.

```
<S-выражение> ::= <атом>
            | (<S-выражение> . <S-выражение>)
            | (<S-выражение> ... )
```

По этому правилу **S-выражения** - это или атомы, или узлы из пары S-выражений, или списки из S-выражений.

/Три точки означают, что допустимо любое число вхождений предшествующего вида объектов, включая ни одного./

Символ «;» - начало комментария до конца строки.

Т.о. “()” есть допустимое S-выражение. Оно в языке Лисп по соглашению эквивалентно атому Nil.

Базовая система представления данных - точечная нотация, хотя на практике запись в виде списков удобнее. Любой список можно представить точечной нотацией:

```
() = Nil
 (a . Nil) = (a)
  - - -
 (a1 . ( ... (aK . Nil) ... )) = (a1 ... aK)
```

Такая **единая структура данных** оказалась вполне достаточной для представления сколь угодно сложных программ. Дальнейшее определение языка Лиспа можно рассматривать как восходящий процесс генерации семантического каркаса, по ключевым позициям которого распределены **семантические действия** по обработке программ.

Другие правила представления данных нужны лишь при расширении и специализации **лексики** языка (числа, строки, имена особого вида и т.п.). Они не влияют ни на общий синтаксис языка, ни на строй его понятий, а лишь характеризуют разнообразие сферы его конкретных приложений.

Синтаксис программ в Лиспе внешне на отличается от синтаксиса данных. Просто выделяем **вычислимые выражения** (формы), т.е. данные, приспособленные к вычислению. Внешне это выглядит как объявление объектов, заранее известных в языке, и представление разных форм, вычисление которых обладает определенной спецификой.

Выполнение программы на Лиспе устроено как **интерпретация данных**, представляющих выражения, имеющие значение. Ниже приведены синтаксические правила для обычных конструкций, к которым относятся идентификаторы, переменные, константы, аргументы, формы и функции.

```
<идентификатор> ::= <атом>
```

Идентификатор - это атомы, используемые при именовании неоднократно используемых объектов программы - функций и переменных. Предполагается, что объекты размещаются в памяти так, что по идентификатору их можно найти.

```
<форма> ::= <константа>
        | <переменная>
        | (<функция> <аргумент> ...  >)
        | (COND (<форма> <форма>) (<форма> <форма>) ... )

<константа> ::= (QOUTE <S-выражение>)
           | '<S-выражение>

<переменная> ::= <идентификатор>
```

Переменная - это идентификатор, имеющий многократно используемое значение, ранее вычисленное в подходящем контексте. Подразумевается, что одна и та же переменная в разных контекстах может иметь разные значения.

Форма - это выражение, которое может быть вычислено. Формами являются переменные и списки, начинающиеся с QUOTE, COND или с представления некоторой функции.

```
<аргумент> ::= <форма>
```

Если форма представляет собой **константу**, то нет необходимости в вычислениях, независимо от вида константы. **Константные значения**, могут быть любой сложности, включая вычислимые выражения. Константы изображаются с помощью специальной функции **QUOTE, блокирующей вычисление**. Представление констант с помощью QOUTE устанавливает границу, далее которой вычисление не идет. Использование апострофа (') - просто сокращенное обозначение для удобства набора внешних форм. Константные значения аргументов характерны при тестировании и демонстрации программ.

Если форма представляет собой переменную, то ее значением должно быть S-выражение, связанное с этой переменной до момента вычисления формы. Следовательно где-то должна храниться некая таблица, по которой, зная имя переменной, можно найти ее значение.

Третья правило гласит, что можно написать **функцию, затем перечислить ее аргументы** и все это как общий список заключить в скобки.

Аргументы представляются формами. Это означает, что допустимы **композиции функций.** Обычно **аргументы вычисляются в порядке вхождения** в список аргументов.

Последнее правило задает формат условного выражения. Согласно этому формату **условное выражение** строится из размещенных в двухэлементном списке синтаксически различимых позиций для условий и обычных форм. Двухэлементные списки из определения условного выражения рассматриваются

как представление предиката и соответствующего ему S-выражения. Значение условного выражения определяется перебором **предикатов по порядку**, пока не найдется форма, значение которой отлично от Nil, что означает логическое значение "истина". Строго говоря, такая форма должна найтись непременно. Тогда вычисляется S-выражение, размещенное вторым элементом этого же двухэлементного списка. Остальные предикаты и формы условного выражения не вычисляют (**логика Мак-Карти**), их формальная корректность или определенность не влияют на существование результата.

Разница между предикатами и обычными формами заключается лишь в трактовке их результатов. Любая форма может выполнить роль предиката.

```
<функция> ::= <название>
        | (LAMBDA <список_переменных> <форма>)
        | (DEFUN <название> <функция>)

<список_переменных> ::= (<переменная> ... )

<название> = <идентификатор>
```

Название - это идентификатор, определение которого хранится в памяти, но оно может не подвергаться влиянию **контекста вычислений**.

Таким образом, **функция** - это или название, или трехэлементный список, начинающийся с LAMBDA или DEFUN.

Функция может быть представлена просто именем. В таком случае ее смысл должен быть заранее известен. Например, встроенные функции CAR, CDR и т.д.

Функция может быть введена с помощью **лямбда-выражения**, станавливающего соответствие между аргументами функции и **связанными переменными**, упоминаемыми в теле ее определения (в определяющей ее форме). Форма из определения функции может включать переменные, не включенные в лямбда-список, - так называемые **свободные переменные**. Их значения должны устанавливаться на более внешнем уровне. Если функция рекурсивна, то следует объявить ее имя с помощью **специальной функции** DEFUN.

Общий механизм вычисления форм будет позднее определен как универсальная функция EVAL, а сейчас запрограммируем ряд **вспомогательных функций**, полезных при обработке S-выражений. Некоторые из них пригодятся при определении интерпретатора.

Начнем с **общих методов обработки S-выражений**.

AMONG – проверка **входит ли заданный атом в данное** S-выражение.

```
(DEFUN among  (x y) (COND
                              ((ATOM y) (EQ x y))
                              ((among x (CAR y)) (QUOTE T))
                              ((QUOTE T) (among x (CDR y) ))
                        )
)
```

EQUAL - предикат, проверяющий **равенство двух S-выражений**. Его значение "истина" для идентичных аргументов и "ложь" для различных. (Элементарный предикат EQ определен только для атомов.) Определение EQUAL иллюстрирует условное выражение внутри условного выражения (двухуровневое условное выражение и двунаправленная рекурсия)

```
(DEFUN equal (x y) (COND
                    ((ATOM x) (COND
                              ((ATOM y) (EQ x y))
                              ((QUOTE T) (QUOTE NIL))
                             )
                    )
                    ((equal (CAR x)(CAR y)) (equal (CDR x)(CDR y)))
                    ((QUOTE T) (QUOTE NIL))
                   )
)
```

При желании можно дать название этой функции по-русски:

```
(DEFUN равно_ли  (x y)  (equal x y))
```

SUBST - функция трех аргументов x, y, z, строящая результат **замены** S-выражением x всех вхождений атома y в S-выражение z.

```
(DEFUN subst (x y z) (COND
                     ((equal y z) x)
                     ((ATOM z) z)
                     ((QUOTE T)(CONS
                              (subst x y (CAR z))
                              (subst x y (CDR z))
                                    )
                      )
                              )
)
```

```
(subst '(x . A) 'B '((A . B) . C))                              ; = ((A . (x . A)) . C)
(subst 'x '(B C D) '((A B C D)(E B C D)(F B C D))) ;= ((A . x)(E . x)(F . x))
```

Символ «;» - начало комментария.

Использование equal в этом определении позволяет подстановку осуществлять и в более сложных случаях. Например, для редукции совпадающих хвостов подсписков:

```
(DEFUN Подстановка (x y z) (subst x y z))
(Подстановка '(x . A) 'B '((A . B) . C))        ;= ((A . (x . A)) . C)
```

NULL - предикат, **отличающий пустой список** от всех остальных S-выражений. Используется, чтобы выяснять, когда список исчерпан. Принимает значение "истина" тогда и только тогда, когда его аргумент - Nil.

```
(DEFUN null (x) (COND
                        ((EQ x (QUOTE Nil)) (QUOTE T))
                      ((QUOTE T)  (QUOTE Nil))
                    )
)
```

При необходимости можно компоненты точечной пары разместить в двухэлементном списке и наоборот, из первых двух элементов списка построить в точечную пару.

```
(DEFUN pair_to_list  (x)  (CONS (CAR x) (CONS (CDR x) Nil)) )
```

```
(DEFUN list_to_pair  (x)  (CONS (CAR x) (CADR x)) )
```

По этим определениям видно, что списочная запись строится большим числом CONS, т.е. на нее расходуется больше памяти.

Основные методы обработки списков

Следующие функции используются, когда **рассматриваются лишь списки**.

APPEND - функция двух аргументов x и y, **сцепляющая два списка** в один.

```
(DEFUN append (x y) (COND
                        ((null x) y)
                        ((QUOTE T) (CONS
                                  (CAR x)
                                  (append (CDR x) y)
                                 )
                        )
)
)

(append '(A B) '(C D E))          ;= (A B C D E)
```

MEMBER - функция двух аргументов x и y, выясняющая **встречается ли S-выражение** x среди элементов списка y.

```
(DEFUN member (x y) (COND
                        ((null x) (QUOTE Nil))
                        ((equal x (CAR y)) (QUOTE T))
                        ((QUOTE T) (member x (CDR y))
                       )
)
```

PAIRLIS - функция трех аргументов x, y, al, **строит список пар** соотвествующих элементов из списков x и y - связывает и присоединяет их к списку al. Полученный список пар, похожий на таблицу с
двумя столбцами, называется ассоциативным списком или ассоциативной таблицей. Такой список может использоваться для связывания имен переменных и функций при организации
вычислений интерпретатором.

```
(DEFUN pairlis (x y al) (COND
                          ((null x) al)
                          ((QUOTE T) (CONS (CONS (CAR x)
                                                 (CAR Y) )
                                           (pairlis (CDR x)
                                                    (CDR y)
                                                    al)
                           ) )              )
)

(pairlis '(A B C) '(u t v) '((D . y)(E . y)))      ;= ((A . u)(B . t)(C . v)(D . y)(E . y))
```

ASSOC - функция двух аргументов x и al. Если al - **ассоциативный список**, подобный тому, что формирует функция pairlis, то assoc выбирает из него первую пару, начинающуюся с x. Таким образом, это функция **поиска определения или значения по таблице**, реализованной в форме ассоциативного списка.

```
(DEFUN assoc (x al) (COND
                    ((equal x (CAAR al)) (CAR al))
                    ((QUOTE T) (assoc x (CDR al))
                    )  )
)

(assoc 'B '((A . (m n)) (B . (CAR x)) (C . w) (B . (QUOTE T))))    ;= (B . (CAR x))
```

Частичная функция - рассчитана на наличие ассоциации.

SUBLIS - функция двух аргументов al и y, предполагается, что первый из аргументов AL устроен как ассоциативный список вида ((u1 . v1) ... (uK . vK)), где u есть атомы, а второй аргумент Y - любое S-выражение. Действие sublis заключается в обработке Y, такой что **вхождения переменных** Ui, связанные в ассоциативном списке со значениями Vi, **заменяются на** эти **значения**. Другими словами в S-выражении Y вхождения переменных U заменяются на соответствующие им V из списка пар AL. **Вводим вспомогательную функцию** SUB2, обрабатывающую атомарные S-выражения, а затем - полное определение SUBLIS:

```
(DEFUN sub2 (al z) (COND
                    ((null al) z)
                    ((equal (CAAR al) z) (CDAR al))
                    ((QUOTE T) (sub2 (CDR al) z))
                   ) )

(DEFUN sublis (al y) (COND
                     ((ATOM y) (sub2 al y))
                     ((QUOTE T)(CONS
                              (sublis al (CAR y))
                              (sublis al (CDR y))
                             ) )))
```

Пример.

```
(sublis '((x . Шекспир)(y . (Ромео и Джульетта))) '(x написал трагедию y))
```

```
;= (Шекспир написал трагедию (Ромео и Джульетта))
```

INSERT – **вставка z перед вхождением ключа** x в список al.

```
(DEFUN insert (al x z) (COND
                        ((null al) Nil)
                        ((equal (CAR al) x) (CONS z al))
                        ((QUOTE T) (CONS (CAR al) (insert (CDR al) x z))
                       )
)

 (insert '(a b c) 'b 's            ; =  (a s b c)
```

ASSIGN – **модель присваивания переменным**, хранящим значения в ассоциативном списке. Замена связанного с данной переменной в первой паре значения на новое заданное значение. Если не было пары вообще, то новую пару из переменной и ее значения размещаем в конец a-списка, чтобы она могла работать как глобальная.

```
(DEFUN assign (x v al) (COND
                         ((Null al) (CONS (CONS x v) Nil ))
                        ((equal x (CAAR al))(CONS (CONS x v) (CDR al)))
                        ((QUOTE T) (CONS (CAR al) (assign x v (CDR al))
                       )
)

(assign 'a 111 '((a . 1)(b . 2)(a . 3)))     ;= ((a . 111)(b . 2)(a . 3))
(assign 'a 111 '((c . 1)(b . 2)(a . 3)))     ;= ((c . 1)(b . 2)(a . 111))
(assign 'a 111 '((c . 1)(d . 3)))            ;= ((c . 1)(d . 3) (a . 111))
```

Упражнение: Введите функции с именами – Пусто, Пара_в_список, Список_в_пару, входит_ли, Соединение, Элемент, Связывание, Ассоциация, Ряд_подстановок, Вставка, Присваивание, как аналоги вышеприведенных функций.

Задание: Напишите определение функции REVERSE – **обращение списка**

Ответ:

```
(defun reverse (m) (cond ((null m) NIL)(T (append(reverse(cdr m))
     (list(car m)) )) ))
```

Теперь посмотрим ее вариант как пример использования накапливающих параметров и вспомогательных функций:.

```
(defun rev (m n) (cond ((null m) N)(T  (rev(cdr m) (cons (car m) n)))))

(defun reverse (m) (rev m Nil))
```

Выводы:

- *Синтаксис Лиспа очень прост. В некотором смысле это сосредотачивает разработчиков на семантике, что породило большое количество диалектов.*

- ***Форма*** *- это выражение, которое может быть вычислено. Формами являются переменные и списки, начинающиеся с QUOTE, COND или с представления некоторой функции.*

- *Выполнение программы на Лиспе устроено как* ***интерпретация данных****, представляющих выражения, имеющие значение.*

5. Интерпретатор

Теперь рассмотрим определение операционной семантики Лиспа в виде универсальной функции, задающей правила вычисления форм и применения функций к аргументам.

- Требования к определению.
- Синтаксическая сводка.
- Определение универсальной функции
- EVAL-APPLY.
- Предикаты и истинность.

Интерпретация или универсальная функция - это функция, которая умеет вычислять значение любой формы, включая формы, сводимые к вычислению произвольной заданной функции, применяемой к представленным в этой же форме аргументам, по доступному описанию данной функции. (Конечно, если функция, которой предстоит интерпретироваться, имеет бесконечную рекурсию, то интерпретация будет повторяться бесконечно.)

Определим универсальную функцию eval от аргумента expr - выражения, являющегося произвольной вычислимой формой языка Лисп.

Универсальная функция должна предусматривать основные виды вычисляемых форм, задающих значения аргументов, а также представления функций, в соответствии со сводом вышеприведенных правил языка. При интерпретации выражений учитывается следующее:

- **Атомарное выражение** обычно понимается как переменная. Для него следует найти связанное с ним значение. Например, могут быть переменные вида “x”, “elem”, смысл которых зависит от контекста, в котором они вычисляются.

- **Константы**, представленные как аргументы функции QUOTE, можно просто извлечь из списка ее аргументов. Например, значением константы (QUOTE T) является атом T, обычно символизирующий значение "истина".

- **Условное выражение требует специального алгоритма** для поиска истинных предикатов и выбора нужной ветви. Например, интерпретация условного выражения

```
(COND  ((ATOM x)   x)
                        ((QUOTE T) (first (CAR x) )) )
```

должна обеспечивать выбор ветви в зависимости от атомарности значения аргумента. Семантика чистого Лиспа не определяет значение условного выражения при отсутствии предиката со значением "истина".

- Остальные формы выражений рассматриваются по общей схеме как **список из функции и ее аргументов**. Обычно аргументы вычисляются и затем вычисленные значения передаются функции для интерпретации ее определения. Так обеспечивается возможность писать композиции функций. Например, в выражении (first (CAR x)) внутренняя функция CAR сначала получит в качестве своего аргумента значение переменной x, а потом свой результат передаст как аргумент более внешней функции first.

- Если функция представлена своим названием , то среди названий различаются имена **встроенных функций**, такие как CAR, CDR, CONS и т.п., и имена функций, введенных в программе, например, first. Для встроенных функций интерпретация сама знает как найти их значение по заданным аргументам, а для введенных в программе функций - использует их **определение**, которое находит по имени или по контексту.

- Если функция использует **лямбда-конструктор**, то прежде, чем ее применять, **понадобится связывать переменные из лямбда-списка со значениями аргументов**. Функция, использующая лямбда-выражение,

```
(LAMBDA (x)  (COND   ((ATOM x)   x)
                ((QUOTE T) (first (CAR x) ))   ))
```

зависит от одного аргумента, значение которого должно быть связано с переменной x. В определении используется **свободная функциональная переменная** first, которая должна быть определена в более **внешнем контексте**.

- Если представление функции начинается с DEFUN, то **понадобится сохранить имя функции с соответствующим ее определением** так, чтобы корректно выполнялись рекурсивные вызовы функции. Например, предыдущее LAMBDA-определение безымянной функции **становится рекурсивным**, если его сделать вторым аргументом специальной функции DEFUN, первый аргумент которой – fisrt, имя новой функции.

```
(DEFUN first (x)   (COND   ((ATOM x)    x)
                                  ((QUOTE T) (first (CAR x) ))   ))
```

Можно сказать, что DEFUN замыкает выражение, содержащее функциональную переменную.

Таким образом, интерпретация функций осуществляется как **взаимодействие** четырех подсистем:

- обработка структур данных (cons, car, cdr, atom, eq),

- конструирование функциональных объектов (lambda, defun),
- идентификация объектов (имена переменных и названия функций),
- управление порядком вычислений (композиции, quote, cond, **eval**).

Определение универсальной функции

Универсальная функция **eval**, которую предстоит определить, должна удовлетворять следующему условию: **если представленная аргументом форма сводится к функции, имеющей значение на списке аргументов этой же формы, то это значение и является результатом функции eval.**

```
(eval '(fn arg1 ... argK))
```

Результат применения "fn" к аргументам "arg1, ..., argK".

Вычисление

Явное определение такой функции позволяет достичь четкости механизмов обработки Лисп-программ.

```
(eval '((LAMBDA (x y) (CONS (CAR x) y)) '(A B) '(C D) ))
```

= (A C D)

Вводим **две основные функции eval и apply** для обработки форм и обращения к функциям соответственно. Каждая из этих функций **использует ассоциативный список для хранения связанных имен - значений переменных и определений функций**. Сначала этот список пуст.
Вернемся к синтаксической сводке вычислимых форм.

Таблица. **Синтаксическая сводка программ на языке Лисп.**

```
<форма> ::= <переменная>
        | (QOUTE <S-выражение>)
        | (COND (<форма> <форма>) ... (<форма> <форма>))
        | (<функция> <аргумент> ...  <аргумент>)

<аргумент> ::= <форма>

<переменная> ::= <идентификатор>


<функция> ::= <название>
          | (LAMBDA <список_переменных> <форма>)
          | (DEFUN <название> <функция>)

<список_переменных> ::= (<переменная> ... )

<название> = <идентификатор>

<идентификатор> ::= <атом>
```

Каждой ветви этой сводки соответствует ветвь универсальной функции:

```
(DEFUN eval0 (e)   (eval e '((Nil . Nil))))
```

Вспомогательная функция eval0 понадобилась, чтобы в eval ввести **накапливающий параметр** – ассоциативный список, в котором будут храниться связи между переменными и их значениями и названиями функций и их определениями.

```
(defun eval(e a) (COND
                        ( (atom e)  (cdr(assoc e a)) )
                        ( (eq (car e) 'QUOTE)  (cadr e))
                        ((eq(car e) 'COND)    (evcon(cdr e) a))
                        ( T  (apply (car e) (evlis(cdr e) a) a) ))    )

(defun apply (fn x a) (COND
            ((atom fn)(cond
                        ((eq fn 'CAR)(caar x))
                        ((eq fn 'CDR)(cdar x))
                        ((eq fn 'CONS) (cons (car x)(cadr x)) )
                        ((eq fn 'ATOM)(atom (car x)) )
                        ((eq fn 'EQ) (eq (car x)(cadr x)) )
                        ((QUOTE  T) (apply (eval fn a) x a)) ) )
            )
((eq(car fn)'LAMBDA) (eval (caddr fn) (pairlis (cadr fn) x a) ))
((eq (car fn) 'DEFUN) (apply (caddr fn) x (cons (cons (cadr fn)(caddr fn)) a)))))
```

Упражнение: Это определение можно немого уточнить, если а-список при связывании названий функций поплнять не в начале, а в конце.

assoc и pairlis уже определены ранее.

```
(DEFUN evcon (c a) (COND
                        ((eval (caar  c) a) (eval (cadar c) a) )
                        ( T                 (evcon (cdr c) a)     ) ))
```

**) Примечание.* Не допускается отсутствие истиного предиката, т.е. пустого С.

```
(DEFUN evlis (m a) (COND
                        ((null m) Nil )
                        ( T         (cons(eval (car m) a)
                                         (evlis(cdr m) a)    ))   )
```

При

```
(DEFUN eval0 (e)  (eval e ObList ))
```

определения функций могут накапливаться в системной переменной ObList, то есть работать как глобальные определения. ObList обязательно должна содержать глобальное определение встроенной константы Nil, можно и сразу разместить в ней T.

Поясним ряд пунктов этих определений.

Первый аргумент eval - форма. Если она - атом, то этот атом может быть только именем переменной, а **значение переменной должно бы уже находиться в ассоциативном списке**.

Если CAR от формы - QUOTE, то она представляет собой **константу**, значение которой **вычисляется как CADR от нее самой**.

Если CAR от формы - COND, то форма - условное выражение. **Вводим вспомогательную функцию EVCON**, (определение ее будет дано ниже), которая обеспечивает вычисление предикатов (пропозициональных термов) по порядку и выбор формы, соответствующей первому предикату, принимающему значение "истина". Эта форма передается EVAL для дальнейших вычислений.

Все остальные случаи рассматриваются как **список из функции с последующими аргументами**.

Вспомогательная функция EVLIS обеспечивает вычисление аргументов, затем представление функции и список вычисленных значений аргументов передаются функции APPLY.

Первый аргумент apply - функция. Если она - атом, то существует две возможности: атом представляет **одну из элементарных функций (car cdr cons atom eq)**. В таком случае соответствующая ветвь вычисляет значение этой функции на заданных аргументах. В противном случае, этот атом - **имя ранее заданного определения**, которое можно найти в ассоциативном списке.

Если функция начинается с LAMBDA, то ее **аргументы попарно соединяются со связанными переменными**, а тело определения (форма из лямбда-выражения) передается как аргумент функции eval для дальнейшей обработки.

Если функция начинается с DEFUN, то ее **название и определение соединяются в пару и полученная пара размещается в ассоциативном списке**, чтобы имя функции стало определенным при дальнейших вычислениях. Они произойдут как рекурсивный вызов apply, которая вместо имени функции теперь работает с ее определением при более полном ассоциативном списке - в нем теперь размещено определение имени функции. Поскольку определение размещается на "верху" стека, оно становится доступным для всех последующих переопределений, то есть работает как **локальный объект**. Глобальные объекты, такие как обеспечивает псевдо-функция DEFUN, устроены немного иначе, что будет рассмотрено в следующей лекции.

Определение универсальной функции является важным шагом, показывающим **одновременно и механизмы реализации языков программирования, и технику функционального программирования на любом языке**. Пока еще не описаны многие другие особенности языка Лисп, которые будут рассмотрены позднее. Но все они будут увязаны в единую картину, основа которой согласуется с этим определением.

1) В **строгой теории элементарного Лиспа** все функции следует определять всякий раз, как они используются. На практике это неудобно. Реальные системы имеют большой запас встроенных функций (более тысячи в Clisp-е), известных языку, и возможность присоединения такого количества новых функций, какое понадобится.

2) В чистом языке Лисп базисные функции CAR и CDR не определены для атомарных аргументов. Такие функции, имеющие осмысленный результат не на всех значениях естественной области определения, называют частичными. Отладка и применение частичных функций требует большего контроля, чем работа с тотальными, всюду определенными функциями.

Во многих Лисп-системах все элементарные функции вырабатывают результат и на списках, и на атомах, но его смысл зависит от системных решений, что может создавать трудности при переносе программ на другие системы. Базисный предикат EQ всегда имеет значение, но смысл его на неатомарных аргументах

будет более понятен после знакомства со структурами данных, используемыми для представления списков в машине.

3) Для расширений и диалектов языка Лисп характерно большое разнообразие условных форм, конструкций выбора, ветвлений и циклов, практически без ограничений на их комбинирование. Форма COND выбрана для начального знакомства как наиболее общая. За редким исключением в Лисп-системе нет необходимости писать в условных выражениях (QUOTE T) или (QUOTE NIL). Вместо них используются **встроенные константы** T и Nil соответственно.

4) В реальных системах функционального программирования **обычно поддерживается работа с целыми, дробными и вещественными числами в предельно широком диапазоне**, а также работа с кодами и строками. Такие данные, как и атомы, являются минимальными объектами при организации обработки нформации, но отличаются от атомов тем, что их смысл задан их собственным представлением непосредственно. Их понимание не требует ассоциаций или связывания. Поэтому и константы такого вида не требуют префикса в виде апострофа.

Предикаты и истинность в Лиспе

В Лиспе есть два атомных символа, которые представляют **истину и ложь** соответственно. Эти два атома - **T и NIL**. Эти символы - действительные значения всех предикатов в системе. Главная причина в удобстве кодирования. Во многих случаях **достаточно отличать произвольное значение от пустого списка**.

Не существует формального различия между функцией и предикатом в Лиспе. Предикат может быть определен как функция со значениями либо T либо NIL. Можно использовать форму, не являющуюся предикатом там, где требуется предикат: предикатная позиция условного выражения или аргумент логического предиката. Семантически любое S-выражение, только не NIL, будет рассматриватсья как истинное в таком случае. Предикат EQ ведет себя следующим образом:

1) Если его аргументы различны, значением EQ является NIL.

2) Если оба его аргументы являются одним и тем же атомом, то значение - T.

3) Если значения одинаковы, но не атомы, то его значение T или NIL в зависимости от того, **идентично ли представление аргументов в памяти**.

4) Значение EQ всегда T или NIL. Оно никогда не бывает неопределено, даже если аргументы плохие.

Выполнено достаточно строгое построение совершенно **формальной математической системы**, называемой “**Элементарный ЛИСП**”. Составляющие этой формальной системы:

1) Множество символов, называемых S-выражениями.

2) Система функциональных обозначений для основных понятий, трубемых при программировании обрботки S-выражений.

3) Формальное представление функциональных обозначений в виде S-выражений.

4) Универсальная функция (записанная в виде S-выражения), интерпретирующая обращение произвольной функции, записанной как S-выражение, к ее аргументам.

5) Система базовых функций, обеспечивающих техническую поддержку обработки S-выражений, и специальных функций, обеспечивающих управление вычислениями.

Выполненное определение универсальной функции – макетный образец Лисп-системы, основные черты которой унаследованы многими системами программирования.

Выводы:

- *Универсальная функция – это функция, которая может вычислять значение любой формы и представляет собой инетрпретатор в миниатюре.*

- *При реализации универсальной функции необходимо учитывать способ представления контекста и виды вычисляемых форм.*

- *Контекст удобно представлять с помощью ассоциативных списков.*

- *Универсальная функция должна предусматривать основные виды вычисляемых форм, задающих значения аргументов, а также представления функций, в соответствии со сводом вышеприведенных правил языка.*

6. **Отображения и функционалы**

В этом параграфе мы рассмотрим следующие механизмы:
- отображения,
- числа и строки,
- безымянные определения функций,
- фильтры и свертки (редукции).

Отображения обычно используются при анализе и обработке данных, представляющих информацию разной природы. Нумерация, кодирование, идентификация, шифрование - каждый их этих процессов использует исходное множество номеров, текстов, идентификаторов, сообщений, по которым конкретная отображающая функция находит занумерованный объект, строит закодированный текст, выделяет идентифицированный фрагмент, получает зашифрованное сообщение. Таким же образом работает любое введение обозначений - от знака происходит переход к его смыслу. Перевод с одного языка на другой, фотография, киносъемка, спортивный репортаж - все это можно рассматривать как примеры отображений.

Отображения - ключевой механизм информатики. Построение любой информационной системы сопровождается определением и реализацией большого числа отображений. Во-первых, выбираются данные, с помощью которых представляется информация. В результате, по данным можно восстановить представленную ими информацию - извлечь информацию из данных. Например, по символам записи числа можно восстановить его величину. Во-вторых, конструируется набор структур, достаточный для размещения и обработки данных в памяти компьютера. Данные отображены в наборы специально организованных кодов, по которым можно преобразовывать и строить новые коды. Например, по коду команды можно выбрать хранимую в памяти подпрограмму, которая построит новые коды чисел или структур данных. В-третьих, коды из нулей и единиц отображаются в наборы точек, образующие буквы или рисунки, которые можно вывести на печать или высветить на дисплее. Каждый такой шаг достаточно очевиден, но композиции отображений могут обладать весьма сложным поведением.

Говорят, что отображение существует, если задана пара множеств и отображающая функция, для которой первое множество - область определения, а второе - область значения. При определении отображений прежде всего должны быть ясны следующие вопросы:

- что представляет собой отображающая функция;
- как организовано данное, представляющее отображаемое множество;
- каким способом выделяются элементы отображаемого множества.

Это позволяет задать порядок перебора множества и метод передачи фактических аргументов для вычисления отображающей функции. При обходе структуры, представляющей множество, отображающая функция будет применена к каждому элементу множества, следовательно может быть выработана подобная структура множества результатов. Возможно, не все полученные результаты нужны, поэтому целесообразно прояснить заранее еще ряд вопросов:

- где размещается множество всех полученных результатов;

- чем отличаются нужные результаты от полученных попутно;
- как строится итоговое данное из отобранных результатов.

При функциональном стиле программирования ответ на каждый из таких вопросов может быть дан в виде отдельной функции, причем роль каждой функции в схеме реализации отображения достаточно четко фиксирована.

Схема реализации отображения может быть представлена в виде определения, формальными параметрами которого являются обозначения этих ролей. Такое определение называется "функционал". Более точно, функционал может оперировать функциями в качестве аргументов или результатов. Чтобы определить конкретное отображение с помощью функционала, надо в качестве фактических параметров задать функции, выполняющие конкретные роли, т.е. дающие ответы на вышеприведенные вопросы. Такие функции могут быть достаточно общими, универсальными, полезными при определении разных отображений, - они получают имена для многократного использования в разных системах определений. Но могут быть и частными, разовыми, нужными лишь в данном конкретном случае – тогда можно обойтись без имен, использовать тело определения непосредственно в точке вызова функции как значение аргумента.

Таким образом, определение отображения может быть декомпозировано на части (функции и функционалы) разного назначения, типичного для многих схем информационной обработки. Это позволяет упрощать отладку систем определений, повышать коэффициент повторного использования отлаженных функций. Не будет преувеличением утверждение, что отображения - эффективный механизм абстрагирования, моделирования, проектирования и формализации крупномасштабной обработки информации. Возможности отображений в информатике значительно шире, чем освоено практическим программированием, но их применение требует дополнительных пояснений, которые и являются целью этой главы.

Числа и строки

Любую информацию можно представить символьными выражениями. В качестве основных видов символьных выражений выбраны списки и атомы. Атом - неделимое данное, представляющее информацию произвольной природы:
Но во многих случаях знание природы информации дает более четкое понимание особенностей изучаемых механизмов. Программирование работы с числами и строками – привычная, хорошо освоенная область информационной обработки, удобная для оценки преимуществ использования функционалов. Опуская технические подробности, просто отметим, что числа и строки рассматриваются как **самоопределимые атомы**, смысл которых не требует никакого ассоциирования, он понятен просто по виду записи.

Например, **натуральные числа** записываются без особенностей и могут быть почти произвольной длины:

```
1
-123
98765432100000000000000123456789
```

Можно работать с **дробными и вещественными** числами:

```
8/12            ;=  2/3
3.1415926
```

Строки заключаются в обычные двойные кавычки:

```
"строка любой длины, из произвольных символов, включая все что угодно"
```

Список - составное данное, первый элемент которого может рассматриваться как функция, применяемая к остальным элементам, также представленным как символьные выражения. Это относится и к операциям над числами и строками:

```
(+ 1 2 3 4 5 6)       ;= 21
(- 12  6   3)         ;= 3
(/  3 5)              ;= 3/5
```

Большинство операций над числами при префиксной записи естественно рассматривать как **мультиоперации** от произвольного числа аргументов.

```
(string-equal  "строка  1"  " строка1")     ;= Nil
(ATOM "a+b-c")                              ;= T
(char  "стр1" 4 )                           ;= "1"
```

Со строками можно при необходимости работать **посимвольно**, хотя они рассматриваются как атомы.

Любой список можно превратить в константу, поставив перед ним "'" апостроф. Это эквивалентно записи со специальной функцией "QUOTE". Для чисел и строк в этом нет необходимости, но это не запрещено

```
‘1            ;= 1
‘"abc"        ;=  "abc"
```

Можно строить композиции функций. Ветвления представлены как результат специальной функции COND, использующей отличие от Nil в качестве значения “истина”. Числа и строки таким образом оказываются допустимыми представлениями значения “истина”.

```
 (cond ((= 1 2) 1)
         (  1       2)
```

```
)                    ; = 2
```

Символ «;» - начало комментария.

Отказ от барьера между представлениями функций и значений дает возможность символьные выражения использовать как для изображения заданных значений, включая любые структуры над числами и строками, так и для определения функций, обрабатывающих любые данные. (Напоминаем, что определение функции - данное.)

Функционалы - это функции, которые используют в качестве аргументов или результатов другие функции. При построении функционалов **переменные могут выполнять роль имен функций**, определения которых находятся во внешних формулах, использующих функционалы.

Сводка ранее пройденного

Любую информацию можно представить символьными выражениями. В качестве основных видов символьных выражений выбраны списки и атомы. Атом - неделимое данное, представляющее информацию произвольной природы:

Список - составное данное, первый элемент которого при записи программ выполняет роль функции, применяемой к остальным элементам, также представленным как символьные выражения:

Любой список можно превратить в константу, поставив перед ним "'" апостроф. Это эквивалентно записи со специальной функцией "QUOTE" :

Можно строить композиции функций:

Ветвления представлены как результат специальной функции COND:

Отказ от барьера между представлениями функций и значений дает
возможность символьные выражения использовать как для изображения заданных значений, так и для определения функций, обрабатывающих любые данные. (Напоминаем, что определение функции - данное.)

Функционалы - это функции, которые используют в качестве аргументов или результатов другие функции. При построении функционалов переменные могут выполнять роль имен функций, определения которых находятся во внешних формулах, использующих функционалы.

Функционалы - общее понятие

Рассмотрим технику использования функционалов и наметим, как от простых задач перейти к более сложным.

Пример 1. Для каждого числа из заданного списка получить следующее за ним число и все результаты собрать в список.

```
(defun next (xl)                    ; Следующие числа:
     (cond                              ; пока список не пуст
        (xl (cons (1+ (car xl))         ; прибавляем 1 к его голове
                  (next (cdr xl)))      ; и переходим к остальным,
)    )  )   )                          ; собирая результаты в список
```

```
(next '(1 2 5 ))    ; = (2 3 6 )
```

Пример 2. Построить список из "голов" элементов списка

```
(defun 1st (xl)                    ; "головы" элементов = CAR
     (cond                           ; пока список не пуст
        (xl (cons (caar xl)          ; выбираем CAR от его головы
                  (1st (cdr xl))     ; и переходим к остальным,
)    )  )   )                       ; собирая результаты в список

(1st '((один два )(one two )(1 2 )) )            ; = (один one 1)
```

Пример 3. Выяснить длины элементов списка

```
(defun lens (xl)                   ; Длины элементов
     (cond                           ; Пока список не пуст
        (xl (cons (length (car xl))  ; вычисляем длину его головы
                  (lens (cdr xl))    ; и переходим к остальным,
)    )  )   )                       ; собирая результаты в список

(lens '((1 2 ) () (a b c d ) (1 (a b c d ) 3 )) )  ; = (2 0 4 3 )
```

Внешние отличия в записи этих трех функций малосущественны, что позволяет ввести более общую функцию `map-el`, в определении которой имена "car", "1+" и "lenth" могут быть заданы как значения параметра fn:

```
 (defun map-el (fn xl) ; Поэлементное преобразование XL
                       ; с помощью функции FN
     (cond                                ; Пока XL не пуст
        (xl (cons (fn (car xl) )
              ; применяем FN как функцию голове XL[1]


                  (map-el fn (cdr xl))   ; и переходим к остальным,
)    )  )   )                           ; собирая результаты в список
```

[1] Примечание: На Clisp это определение выглядит не столь лаконично, оно требует встроенной функции funcall:

Эффект функций next, 1st и lens можно получить выражениями:

```
(map-el #'1+ xl)              ; Следующие числа:
(map-el #'car xl)             ; "головы" элементов = CAR
```

#’ x ;= (FUNCTION x) - сокращенное обозначение функции-значения

```
(map-el #'length xl)          ; Длины элементов

 (map-el #'1+   '(1 2 5 ))    ; = (2 3 6 )
 (map-el #'car  '((один два )(one two )(1 2 )) )  ; = (один one 1)
 (map-el #'length '((1 2 ) () (a b c d ) (1 (a b c d ) 3 )) )
                               ; = (2 0 4 3 )
```

соответственно.

Эти определения функций формально эквивалентны ранее приведенным – они сохраняют отношение между аргументами и результатами.

Все три примера можно решить с помощью таких определяющих выражений:

```
(defun next (xl) (map-el #'1+ xl ))      ; Очередные числа:
(defun 1st  (xl) (map-el #'car xl ))     ; "головы"
элементов = CAR
(defun lens (xl) (map-el #'length xl )) ; Длины элементов
```

Параметром функционала может быть любая вспомогательная функция.

Пример 4. Пусть дана вспомогательная функция sqw, возводящая числа в квадрат

```
(defun sqw (x) (* x x))          ; Возведение числа в квадрат
(sqw 3)                                   ; = 9
```

Построить список квадратов чисел, используя функцию sqw:

```
(defun map-el (fn xl)
   (cond
     (xl (cons (funcall fn (car xl) )
                    ; применяем первый аргумент как функцию
                         ; к первому элементу второго аргумента
           (map-el fn (cdr xl) )
)  ) ) )[1]
```

```
 (defun sqware (xl) ;           ; Возведение списка чисел в квадрат
    (cond                                ; Пока аргумент не пуст,
       (xl (cons (sqw (car xl))      ; применяем sqw к его голове
                 (sqware (cdr xl))  ; и переходим к остальным,
)    )  )   )                          ; собирая результаты в список

(sqware '(1 2 5 7 ))                  ; = (1 4 25 49 )
```

Можно использовать map-el:

```
 (defun sqware (xl) (map-el #'sqw xl))
```

Ниже приведено определение функции sqware- без вспомогательной функции, выполняющее умножение непосредственно. Оно влечет двойное вычисление (CAR xl), т.е. такая техника не вполне эффективна:

```
(defun sqware- (xl)
     (cond
        (xl (cons (* (car xl) (car xl) )          ; квадрат головы
                                          ; вычислять приходится дважды
                  (sqware- (cdr xl))
)    )  )   )
```

Пример 5. Пусть дана вспомогательная функция tuple,
превращающая любое данное в пару:

```
(defun tuple (x) (cons x x))
(tuple 3)                                        ; = (3 . 3)
(tuple 'a)                                       ; = (a . a)
(tuple '(Xa))    ; = ((Xa) . (Xa)) = ((Xa) Xa) – это одно и то же!
```

Чтобы преобразовать элементы списка с помощью такой функции, пишем сразу:

```
(defun duble (xl) (map-el #'tuple xl))   ; дублирование элементов
(duble '(1 (a) ()))                    ; = ((1 . 1) ((a) a) (()))
```

Немногим сложнее организовать покомпонентную обработку двух списков.

Пример 6. Построить ассоциативный список, т.е. список пар из имен и соответствующих им значений, по заданным спискам имен и их значений:

```
 (defun pairl (al vl)                       ; Ассоциативный список
    (cond                                      ; Пока AL не пуст,
       (al (cons (cons (car al) (car vl))      ; пары из "голов".
                 (pairl (cdr al) (cdr vl)) ; Если VL исчерпается,
```

```
                                          ; то CDR будет давать NIL
)    )  )   )

(pair '(один два two three) '(1 2 два три))
  ; = ((один . 1)(два . 2)(two два )(three три ))
```

Пример 7. Определить функцию покомпонентной обработки двух списков с помощью заданной функции fn:

```
 (defun map-comp (fn al vl)            ; fn покомпонентно применить
                              ; к соотвественным элементам AL и VL
     (cond
        (xl (cons (fn (car al) (car vl))
           ; Вызов данного FN как функции
                 (map-comp (cdr al) (cdr vl))
)    )  )   )
```

Теперь покомпонентные действия над векторами, представленными с помощью списков, полностью в наших руках. Вот списки и сумм, и произведений, и пар, и результатов проверки на совпадение:

```
 (map-comp #'+ '(1 2 3) '(4 6 9))        ; = (5 8 12 ) Суммы
(map-comp #'* '(1 2 3) '(4 6 9))        ; = (4 12 27 ) Произведения
(map-comp #'cons '(1 2 3) '(4 6 9))
                                  ; = ((1 . 4)(2 . 6)(3 . 9)) Пары
(map-comp #'eq '(4 2 3 ) '(4 6 9))      ; = (T NIL NIL ) Сравнения
```

Достаточно уяснить, что надо делать с элементами списка, остальное делает отображающий функционал map-comp.

Пример 8. Для заданного списка вычислим ряд его атрибутов, а именно - длина, первый элемент, остальные элементы списка без первого.

```
 (defun mapf (fl el)
     (cond                        ; Пока первый аргумент не пуст,
        (fl (cons ((car fl) el )
                     ; применяем очередную функцию
                                          ; ко второму аргументу
                  (mapf (cdr fl) el)
                     ; и переходим к остальным функциям,
)    )  )   )         ; собирая их результаты в общий список

(mapf '(length car cdr) '(a b c d ))             ; = (4 a (b c d ))
```

Композициями таких функционалов можно применять серии функций к списку общих аргументов или к параллельно заданной последовательности списков их аргументов. Естественно, и серии, и последовательности представляются списками.

Безымянные функции

Определения в примерах 4 и 5 не вполне удобны по следующим причинам:

- ✓ В определениях целевых функций duble и sqwure встречаются имена специально определенных вспомогательных функций.
- ✓ Формально эти функции независимы, значит программист должен отвечать за их наличие при использовании целевых функций на протяжении всего жизненного цикла программы, что трудно гарантировать.
- ✓ Вероятно, имя вспомогательной функции будет использоваться ровно один раз - в определении целевой функции.

С одной стороны последнее противоречит пониманию смысла именования как техники, обеспечивающей неоднократность применения поименованного объекта. С другой стороны придумывать хорошие, долго сохраняющие понятность и соответствие цели, имена - задача нетривиальная.

Учитывая это, было бы удобнее вспомогательные определения вкладывать непосредственно в определения целевых функций и обходиться при этом вообще без имен. Конструктор функций
lambda обеспечивает такой стиль построения определений. Этот конструктор любое выражение expr превращает в функцию с заданным списком аргументов (x1 ... xK) в форме так называемых lambda-выражений:

```
( lambda (x1 ... xK) expr )
```

Имени такая функций не имеет, поэтому может быть применена лишь непосредственно. Defun использует этот конструктор, но, кроме того, дает функциям имена.

Пример 9. Определение функций duble и sqwure из примеров 4 и 5
без использования имен и вспомогательных функций:

```
(defun sqware (xl) (map-el  (lambda (x) (* x x)) xl))

(defun duble (xl) (map-el  (lambda (x) (cons x x)) xl))
```

Любую систему взаимосвязанных функций можно преобразовать к одной функции, используя вызовы безымянных функций.

Композиции функционалов, фильтры, редукции

Вызовы функционалов можно объединять в более сложные структуры таким же образом, как и вызовы обычных функций, а их композиции можно использовать в определениях новых функций.

Композиции функционалов позволяют порождать и более мощные построения, достаточно ясные, но требующие некоторого внимания.

Пример 10. Декартово произведение хочется получить определением вида:

```
(defun decart (x y)
  (map-el #'(lambda (i)
                          (map-el #'(lambda (j) (list i j))
                          y)
                    )  x)  )
```

Но результат вызова

```
(decart '(a s d) '( e r t))
```

дает

```
(((A E) (A R) (A T)) ((S E) (S R) (S T)) ((D E) (D R) (D T)))
```

вместо ожидаемого

```
((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Дело в том, что функционал map-el, как и map-comp (пример 7), собирает результаты отображающей функции в общий список с помощью операции cons так, что каждый результат функции образует отдельный элемент.
А по смыслу задачи требуется, чтобы список был одноуровневым.
Посмотрим, что получится, если вместо cons при сборе результатов воспользоваться функцией append.

Пример 11. Пусть дан список списков. Нужно их все сцепить в один общий список.

```
(defun list-ap (ll)
     (cond
        (ll (append (car ll)
                    (list-ap (cdr ll) )
)    )  )   )

(list-ap '((1 2)(3 (4))))                ; = (1 2 3 (4))
```

Тогда по аналогии можно построить определение фукнционала map-ap:

```
(defun map-ap (fn ll)
     (cond
        (ll (append (fn (car ll) )
                    (map-ap fn (cdr ll) )
)    )  )   )

(map-ap 'cdr  '((1 2 3 4) (2 4 6 8) ( 3 6 9 12)))
              ; = (2 3 4     4 6 8      6 9 12)
```

Следовательно, интересующая нас форма результата может быть получена:

```
(defun decart (x y)
   (map-ap #'(lambda (i)
                                  (map-el #'(lambda (j) (list i j))
                           y)     )   x)    )
 (decart '(a s d) '( e r t))
       ; = ((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Соединение результатов отображения с помощью append обладает еще одним полезным свойством: при таком сцеплении исчезают вхождения пустых списков в результат. А в Лиспе пустой список используется как ложное значение, следовательно такая схема отображения пригодна для организации фильтров. **Фильтр** отличается от обычного отображения тем, что окончательно собирает не все результаты, а лишь удовлетворяющие заданному предикату.

Пример 12. Построить список голов непустых списков можно следующим образом:

```
(defun heads (xl) (map-ap
       #'(lambda (x)(cond (x (cons (car x) NIL))))
                             ; временно голова размещается в список,
                             ; чтобы потом списки сцепить
                          xl
)                              )
(heads '((1 2) () (3 4) () (5 6)) )     ; = (1 3 5)
```

Рассмотрим еще один типичный вариант применения функционалов. Представим, что нас интересуют некие интегральные характеристики результатов, полученных при отображении, например, сумма полученных чисел, наименьшее или наибольшее из них и т.п. В таком случае говорят о свертке результата или его редукции. Редукция заключается в сведении множества элементов к одному элементу, в вычислении которого задействованы все элементы множества.

Пример 13. Подсчитать сумму элементов заданного списка.

```
(defun sum-el ( xl)
     (cond ((null xl) 0)
        (xl (+ (car xl)
```

```
                    (sum-el (cdr xl) )
)     )   )    )

(sum-el '(1 2 3 4 ) )                    ; = 10
```

Перестроим определение, чтобы вместо "+" можно было использовать произвольную бинарную функцию:

```
 (defun red-el (fn xl)
     (cond ((null xl) 0)
        (xl (fn (car xl)
                    (red-el fn (cdr xl) )    ))))
 (red-el '+ '(1 2 3 4 ) )                          ; = 10
```

В какой-то мере map-ар ведет себя как свертка - она сцепляет результаты без сохранения границ между ними.

Такие формулы удобны при моделировании множеств, графов и металингвистических формул, а к их обработке сводится широкий класс задач не только в информатике.

Выводы:

- Отображающие функционалы позволяют строить программы из крупных действий.
- Функционалы обеспечивают гибкость отображений.
- Определение функции может совсем не зависить от конкретных имен.
- С помощью функционалов можно управлять выбором формы результатов.
- Параметром функционала может быть любая функция, преобразующая элементы структуры.
- Функционалы позволяют формировать серии функций от общих данных.
- Встроенные в Clisp функционалы приспособлены к покомпонентной обработке произвольного числа параметров.
- Любую систему взаимосвязанных функций можно преобразовать к одной функции, используя вызовы безымянных функций.

7. Имена и контексты

- Определения функций. Псевдо-функции.
- Именование значений и подвыражений
- Переменные и константы.
- Функции и данные.

Реализация языка программирования всегда сопровождается некоторым уточнением границ, в которых применяются общеизвестные понятия. Цель уточнения - удобство программирования и повышение эффективности программ. Рассмотрим отдельные решения, уточненные при реализации ряда Лисп-систем, на небольшом примере моделирования работы с множествами.

Задача: Пусть множества представлены с помощью списков. Для начала рассмотрим простые множества, элементами которых могут быть только атомы. Надо реализовать объединение (UNION) и пересечение (INTERSECTION) множеств.

Предварительный анализ задачи:
Функции UNION и INTERSECTION применяют к множествам, каждое множество представлено в виде списка атомов. Заметим, что обе функции рекурсивны и используют вспомогательную функцию, выясняющую входит ли атом в список (MEMBER).

Работу этих функций можно выразить следующим образом:

MEMBER – это функция двух аргументов, первый аргумент “A” - атом, а второй аргумент – список “X”. Функция вырабатывает значение “T”, если “A” входит в список “X”.

Алгоритм:
Определение тела функции состоит из трех ветвей:
- **Если** второй аргумент – пустой список,
  то значение функции Nil, т.е. атом в списке не найден.
- **Иначе если** атом “A” совпадает с “головой” второго аргумента,
  то значение функции T, т.е. атом имеется в списке.
- **Иначе** продолжаем поиск в “хвосте” списке, т.е. рекурсивно применяем исходную функцию к редуцированному второму аргументу.

```
алг member ( атом  a, список x) арг a, x
      нач
          если  пусто (a)
              то знач := Nil
           инес равно (a, голова (x) )
              то знач := T
          иначе   знач := member (a, хвост (x))
      кон
```

UNION – это функция двух аргументов, оба аргумента “X” и “Y” - списки, представляющие множества. Функция вырабатывает новый список, в который входят все атомы из списков “X” и “Y”.

Алгоритм:
Определение тела функции состоит из трех ветвей:

- **Если** первый аргумент – пустой список,
  то значением является второй аргумент, т.е. можно ничего не строить.
- **Иначе если** “голова” первого аргумента входит во второй аргумент,
  то достаточно объединить хвост первого аргумента со вторым аргументом, т.е. рекурсивно применяем исходную функцию, редуцируя первый аргумент.
- **Иначе** “голову” первого аргумента присоединяем к результату объединения редуцированного первого аргумента со вторым аргументом.

```
алг UNION (список x,y) арг  x, y
     нач
         если  пусто (x)
             то знач := y
          инес member ( голова (x), y )
             то знач := UNION (хвост (x), y)
         иначе   знач := cons (голова (x), UNION (хвост (x), y))
     кон
```

INTERSECTION – это функция двух аргументов, оба аргумента “X” и “Y” - списки, представляющие множества. Функция вырабатывает новый список, в который входят атомы списка “X”, входящие в список “Y”.

Алгоритм:
Определение тела функции состоит из трех ветвей:

- **Если** первый аргумент – пустой список,
  то и пересечение - пустой список.
- **Иначе если** “голова” первого аргумента входит во второй аргумент,
  то “голову” первого аргумента присоединяем к результату пересечения редуцированного первого аргумента со вторым аргументом.
- **Иначе** применяем пересечение к редуцированному первому аргументу со вторым аргументом.

```
алг INTERSECTION (список x,y) арг  x, y
     нач
         если  пусто (x)
             то знач := Nil
          инес member ( голова (x), y )
             то знач := cons (голова (x), INTERSECTION (хвост (x),
y))
         иначе   знач := INTERSECTION (хвост (x), y)
     кон
```

Определяя эти функции на Лиспе, мы используем специальную псевдо-функцию DEFUN. Программа выглядит так:

```
 (DEFUN  MEMBER (A X)
;определение проверки входит ли атом в список
        (COND
          ((NULL X)       Nil)
          ((EQ A (CAR X)) T)
          (T              (MEMBER A (CDR X)) )
)       )

(DEFUN UNION (X Y)
;определение объединения двух множеств
       (COND
         ((NULL X) Y)
         ((MEMBER (CAR X) Y) (UNION (CDR X) Y) )
         (T (CONS (CAR X) (UNION (CDR X) Y))) )) )
))

(DEFUN INTERSECTION (X Y)
;определение пересечения двух множеств
             (COND
    ((NULL X) NIL)
    ((MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECTION (CDR
X) Y)) )
    (T (INTERSECTION (CDR X) Y))
))

(INTERSECTION '(A1 A2 A3) '(Al A3 A5))
;тест на пересечение двух множеств
(UNION '(X Y Z) '(U V W X))
;тест на объединение двух множеств
```

Эта программа предлагает Лисп-системе вычислить пять различных форм. Первые три формы сводятся к применению псевдо-функции DEFUN. Значение четвертой формы - (A1 A3). Значение пятой формы - (Y Z C B D X). Анализ пути, по которому выполняется рекурсия, показывает, почему элементы множества появляются именно в таком порядке.

Псевдо-функция - это функция, которая выполняется ради ее воздействия на систему, тогда как обычная функция - ради ее значения. DEFUN заставляет функции стать определенными и допустимыми в системе равноправно со встроенными функциями. Ее значение - имя определяемой функции, в данном случае - MEMBER, UNION, INTERSECTION. Можно сказать более точно, что полная область значения псевдо-функции DEFUN включает в себя некоторые доступные ей части **системы**, обеспечивающие хранение информации о

функциональных объектах, а формальное ее значение – атом, символизирующий определение функции.

В этом примере продемонстрировано несколько элементарных правил написания функциональных программ, выбранных при реализации интерпретатора Лисп 1.5 в дополнение к идеализированным правилам, сформулированным в строгой теории Лиспа, которая описана в предыдущем разделе.

1) **Программа состоит из последовательности вычисляемых форм**. Если форма список, то ее первый элемент интерпретируется как функция. Остальные элементы списка – аргументы для этой функции. Они вычисляются с помощью EVAL, а функция применяется к ним с помощью APPLY и полученное значение выводится как результат программы.

2) **Нет особого формата** для записи программ. Границы строк игнорируются. Формат программы, включая идентификацию, выбран просто для удобства чтения.

3) **Любое число пробелов и концов строк** можно разместить в любой точке программы, но не внутри атома.

4) Не используются (QUOTE T), (QUOTE NIL). Вместо них используется T, NIL, что влечет соответствующее изменение определения EVAL.

5) **Атомы должны** начинаться с букв, чтобы **легко отличаться от чисел**.

6) Точечная нотация может быть привлечена наряду со списочной записью. Любое число пробелов перед или после точки, кроме одного, будет игнорироваться (один пробел обязательно нужен).

7) Точечные пары могут появляться как элементы списка, и списки могут быть элементами точечных пар.

Например:

```
((A . B) X (C . (E F D))) - есть допустимое S-выражение.
```

Оно может быть записано как

```
((A . B) . ( X . ((C . (E . ( F . (D . Nil))) ) . Nil)))
```

или

```
((A . B) X (C E F D))
```

8) Форма типа (A B C **.** D) есть сокращение для (A **.** (B **.** (C **.** D))). Любая другая расстановка запятых или точек на одном уровне есть ошибка, например, (A**.** B C) или (A B **.** C D) не соответствуют никакой структуре данных. (Реализационное расширение списочной записи. "**.** D" здесь означает, что вместо Nil, по умолчанию завершающего список, в данной структуре размещен атом "D")

9) Набор основных функций обеспечен системой. Другие функции могут быть введены программистом. Любая функция может использоваться в определении другой функции с **учетом иерархии** построений.

Вывод S-выражений на печать и в файлы выполняет псевдо-функция **PRINT**, чтение данных обеспечивает псевдо-функция **READ**. Программа из файла может быть загружена псевдо-функцией **LOAD**.

Пример:

```
(LOAD 'TEST.LSP)
```

При наборе форм в диалоге интерпретатор сам напечатает результаты, а при загрузке программы их файла надо позаботиться о выводе результатов программы с помощью псевдо-функции PRINT.

Пример:

```
(PRINT (INTERSECTION '(A1 A2 A3) '(Al A3 A5))  )
(PRINT (UNION '(X Y Z) '(U V W X)) )
(PRINT (UNION (READ) '(1 2 3 4)) )
; объединение вводимого списка со списком '(1 2 3 4)
```

Именование значений и подвыражений

Переменная - это символ, который используется для представления аргумента функции.

Атом может быть как переменной, так и фактическим аргументом. Некоторые сложности вызывает то обстоятельство, что иногда аргументы могут быть переменными, вычисляемыми внутри вызова другой функции.

Часть интерпретатора, которая при вычислении функций связывает переменные, называется APPLY. Когда APPLY встречает функцию, начинающуюся с LAMBDA, список переменных попарно связывается со списком аргументов и добавляется к началу ассоциативного списка.

Часть интерпретатора, которая потом вычислит переменные, называется EVAL. При вычислении тела функции EVAL может обнаружить переменные. Она их ищет в ассоциативном списке. Если переменная встречается там несколько раз, то используется последнее или самое новое значение.

Проиллюстрируем это рассуждение на примере. Предположим, интерпретатор получает следующее S-выражение:

```
((LAMBDA (X Y) (CONS X Y)) 'A 'B)
```

Функция:

```
(LAMBDA (X Y) (CONS X Y))
```

Аргументы: (A B)

EVAL передает эти аргументы функции APPLY. (См. 6).

```
(apply #'(LAMBDA (X Y) (CONS X Y)) '(A B) Nil )
```

APPLY свяжет переменные и передаст функцию и удлинненный a-список EVAL для вычисления.

```
(eval '(CONS X Y) ' ((X . A) (Y B) Nil))
```

EVAL вычисляет переменные и сразу передает их функции CONS, строящий из них бинарный узел.

```
(Cons 'A 'B) = (A . B)
```

Можно добиться большей прозрачности сложных определений функций, используя **иерархию контекстов** и средства **именования выражений**. Специальная функция **Let** сопоставляет локальным переменным **независимые выражения**. С ее помощью можно вынести из сложного определения любые совпадающие подвыражения.

Пример:

```
(defun UNION- (x y)  (let  ( (a-x (CAR x))
                             (d-x (CDR x))  )

      (COND  ((NULL x)            y)
             ((MEMBER a-x y) (UNION d-x y)  )
             (T                      (CONS a-x (UNION d-x y)) )  )
))
```

Обычно переменная считается связанной в области действия лямбда-конструктора функции, который связывает переменную внутри тела определения функции методом размещения пары из имени и значения в начале ассоциативного списка (**а-списка)**. В том случае, когда переменная всегда имеет определенное значение независимо от текущего состояния а-списка, она будет называться константой. Такую неизменяемую связь можно установить, размещая пару (а . v) в конец а-списка. Но в реальных системах это организовано с помощью так называемого **списка свойств атома**, являющегося представлением переменной. Каждый атом имеет свой список свойств (property list **- р-список**), доступный через **хэш-таблицу идентификаторов**, что работает эффективнее, чем а-список. С каждым атомом связана специальная структура данных, в которой размещается имя атома, его значение, определение функции, представляемой этим же атомом, и список произвольных свойств, помеченных **индикаторами**. При вычислении переменных EVAL исследует эту структуру до поиска в а-списке. Такое устройство констант не позволяет им служить переменными в а-списке.

Глобальные переменные реализованы аналогично, но их значение можно изменять с помощью специальной функции **SET**.

Особый интерес представляет тип констант, которые всегда означают себя – **Nil** и **T**, примеры таких констант. Такие константы как T, Nil и другие самоопределимые константы (числа, строки) не могут использоваться в качестве переменных. Числа и строки не имеют списка свойств.

Ситуация, когда атом обозначает **функцию**, реализационно подобна той, в которой атом обозначает аргумент. Если функция рекурсивна, то ей надо дать имя. Это делается связыванием названия с определением функции в ассоциативном списке**.** Название связано с определением функции точно также, как переменная связана со своим значением.

На практике связывание в ассоциативном списке для функций используется редко. Удобнее связывать название с определением другим способом - размещением определения функции в **списке свойств атома**, символизирующего ее название. Выполняет это псевдо-функция DEFUN, описанная в начале этого параграфа. Когда APPLY интерпретирует функцию, представленную атомом, она исследует р-список до исследования текущего состояния а-списка.

Тот факт, что большинство функций - константы, определенные программистом, а не переменные, изменяемые программой, происходит отнюдь не вследствие какого-либо недостатка понятий Лиспа. Напротив, этот резерв указывает на потенциал подхода, который мы не научились использовать лучшим образом.

Некоторые функции вместо определений с помощью S-выражений закодированы как замкнутые машинные подпрограммы. Такая функция будет иметь особый индикатор в списке свойств с указателем, который позволяет интерпретатору связаться с подпрограммой. Существует три случая, в которых **низкоуровневая подпрограмма** может быть включена в систему:

1) Подпрограмма **закодирована внутри** Лисп-системы.

2) Функция **кодируется пользователем вручную** на языке типа ассемблера.

3) Функция сначала определяется с помощью S-выражения, затем **транслируется компилятором**. Компилированные функции могут выполняться гораздо быстрее, чем интерпретироваться.

Обычно EVAL вычисляет аргументы функций до применения к ним функций с помощью APPLY. Таким образом, если EVAL задано (CONS X Y), то сначала вычисляются X и Y, а потом работает CONS над полученными значениями. Но если EVAL задано (QOUTE X), то X не будет вычисляться. QUOTE - специальная форма, которая препятствует вычислению своих аргументов.

Специальная форма отличается от других функций двумя чертами. Ее аргументы не вычисляются до того, как **специальная форма сама просмотрит свои аргументы**. COND, например, имеет свой особый способ вычисления аргументов с использованием EVCON. Второе отличие от функций заключается в том, что **специальные формы могут иметь неограниченное число аргументов**.

Определение рекурсивной функции можно преобразовать к безымянной форме. Техника **эквивалентных преобразований** позволяет поддерживать целостность системы функций втягиванием безымянных вспомогательных функций внутрь тела основного определения. Верно и обратное: любую конструкцию из лямбда-выражений можно преобразовать в систему отдельных функций. Техника функциональных определений и их преобразований позволяет рассматривать решение задачи с естественной степенью подробности, гибкости и мобильности.

Специальная функция **FUNCTION** обеспечивает доступ к функциональному объекту, а функция **FUNCALL** обеспечивает применение функции к произвольному числу ее аргументов.

```
(funcall f a1 a2 ... ) = (apply f (list a1 a2 ...))
```

Разрастание числа функций, манипулирующих функциями, связано с реализационным различием структурного представления данных и представляемых ими функций.

Программы для Лисп-интерпретатора.

Цель этой части - помочь на первых шагах избежать некоторых общих ошибок.

Пример 1.

```
(CAR '(A B)) = (CAR (QUOTE(A B)))
```

Функция: CAR

Аргументы: ((A B))

Значение есть A. Заметим, что интерпретатор ожидает список аргументов. Единственным аргументом для CAR является (A B). Добавочная пара скобок возникает т.к. APPLY подается список аргументов.

Можно написать (LAMBDA(X)(CAR X)) вместо просто CAR. Это корректно, но не является необходимым.

Пример 2.

```
(CONS 'A '(B . C))
```

Функция: CONS

Аргументы: (A (B . C))

Результат (A . (B . C)) программа печати выведет как (A B . C)

Пример 3.

```
(CONS '(CAR (QUOTE (A . B))) '(CDR (QUOTE (C . D))) )
```

Функция: CONS

Аргументы:

```
((CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))))
```

Значением такого вычисления будет

```
((CAR (QUOTE (A . B))) . (CDR (QUOTE (C . D))))
```

```
```

Скорее всего это отнюдь не то, что ожидал новичок. Он рассчитывал вместо (CAR (QUOTE (A . B)) получить A и ожидал (A . D) в качестве итогового значения CONS. Кроме очевидного стирания апострофов:

```
(CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))) )
```

ниже приведены еще три правильных способа записи нужной формы. Первый состоит в том, что CAR и CDR части функции задаются с помощью LAMBDA в определении функции. Второй - в переносе CONS в аргументы и вычислении их с помощью EVAL при пустом a-списке. Третий - в принудительном выполнении константных действий в представлении аргументов,

```
((LAMBDA (X Y) (CONS (CAR X) (CDR Y))) '(A . B) '(C . D))
```

Функция:

```
(LAMBDA (X Y) (CONS (CAR X) (CDR Y)))
```

Аргументы:

```
((A . B)(C . D))
```

```
(EVAL '(CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D)))) Nil)
```

Функция: EVAL

Аргументы:

```
((CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D)))) Nil)
```

Значением того и другого является (A . D)

```
```

```
((LAMBDA (X Y) (CONS (EVAL X) (EVAL Y))) '(CAR (QUOTE (A . B)))
                                          '(CDR (QUOTE (C . D))) )
```

Функция:

```
(LAMBDA (X Y) (CONS (EVAL X) (EVAL Y)))
```

Аргументы:

```
((CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))))
```

Решения этого примера показывают, что **грань между функциями и данными достаточно условна** - одни и те же вычисления можно осуществить при разном распределении промежуточных вычислений внутри выражения, передвигая эту грань.

Выводы:

- *Чтобы выполнить вычисление на реальном Лисп-интерпретаторе необходимо уточнить ряд правил по оформлению текста Лисп-программы.*

- *Определения функций можно сделать обозримее именованием выражений. Это дает и экономию на времени вычисления совпадающих форм.*

- *При реализации системы программирования возникают дополнительные механизмы, такие как списки свойств атома, повышающие скорость доступа к необходимой информации.*

- *Некоторые функции системы необходимо реализовывать в виде машинных подпрограмм. Они образуют ядро системы.*

- *При подготовке программ для Лисп-интерпретатора грань между программой и данными может устанавливаться в зависимости от текущих обстоятельств.*

8. Управление процессами

Теперь рассмотрим более тонкие методы повышения эффективности и результативности вычислений:

- замедленные вычисления
- рецепты вычисления

Замедленные вычисления (lazy evaluation)

Интуитивное представление о вычислении выражений, согласно которому функция применяется к заранее вычисленным аргументам, не всегда гарантирует получение результата. Ради полноты вычислений, гибкости программ и результативности процессов такое представление можно расширить и ввести категорию специальных функций, которые "сами знают", когда и что из их аргументов следует вычислить. Примеры таких функций - специальные функции QUOTE и COND, которые могут анализировать и варьировать условия, при которых вычисление необходимо. Такие функции манипулируют данными, представляющими выражения, и явно определяют в программах позиции обращения к интерпретатору.

Источником неэффективности стандартных “прямолинейных” вычислений может быть целостность больших сложных данных, избыточное вычисление бесполезных выражений, синхронизация формально независимых действий. Такую неэффективность можно смягчить простыми методами **замедленных вычислений** (lazy evaluation). Необходимо лишь вести учет дополнительных условий для их фактического выполнения, таких как востребованность результата вычислений. Такие соображения по обработке параметров функций называют “**вызов по необходимости**”.

Любое большое сложное данное можно вычислять "по частям". Вместо вычисления списка

```
(x1  x2  x3  ... )
```

можно вычислить x1 и построить структуру

```
(x1  (рецепт вычисления остальных элементов))
```

Получается принципиальная экономия памяти ценой незначительного расхода времени на вспомогательное построение. Рецепт - это ссылка на уже существующую программу, связанную с контекстом ее исполнения, т.е. с состоянием ассоциативного списка в момент построения рецепта.

Пример: Построение ряда целых от M до N с последующим их суммированием. Обычная формула:

```
(defun ряд_цел (M N) (cond ((> M N) Nil)
                           (T(cons M (ряд_цел (1+ M) N)))))

(defun сумма (X) (cond ((= X 0) 0)
                       (T (+ (car X)( сумма (cdr X))))) )
```

Введем специальные операции || - приостановка вычислений и @ - возобновление ранее отложенных вычислений. Приостановка сводится к запоминанию символьного представления программы, которая временно не вычисляется, но можно вернуться к ее вычислению с помощью операции возобновления. Отложенная программа преобразуется в так называемый **рецепт вычисления,** который можно хранить как данное и выполнять в любой момент.
В рецепте хранится не только вычисляемая форма, но и контекст, в котором первоначально было предусмотрено ее вычисление. Таким образом, гарантируется, что переход от обычной программы к программе с отложенными действиями не нарушает общий результат.

Избежать целостного представления большого и возможно бесконечного ряда чисел можно небольшим изменением формулы, отложив вычисление второго аргумента CONS "до лучших времен" – когда его значение действительно понадобится более внешней функции:

```
(defun ряд_цел (M N) (cond ((> M N) Nil)
                           (T(cons M ( || (ряд_цел (1+ M) N))))))

(defun сумма (X) (cond ((= X 0) 0)
                       (T (+ (car X)( @ ( сумма (cdr X))))) ))
```

Можно **исключить повторное вычисление совпадающих рецептов**. Для этого во внутренне представление рецепта вводится флаг, имеющий значение T - истина для уже выполненных рецептов, Nil - ложь для невыполненных.

Таким образом рецепт имеет вид (Nil e AL) или (T X), где X = (eval e AL) для произвольного e, в зависимости от того, понадобилось ли уже значение рецепта или еще не было в нем необходимости.

Это заодно дает возможность **понятие данных распространить на бесконечные** множества. Например, можно манипулировать рядом целых, превосходящих M с помощью функции:

```
(defun цел (M) (cons M ( || (цел (1+ M) ))))
```

Можно из таким образом организованного списка выбирать нужное количество элементов, например, первые K элементов можно получить по формуле:

```
(defun первые (K Int) (cond ((= Int Nil) Nil)
                            ((= K 0) Nil)
                      (T (cons (car Int)( первые ( @ (cdr Int))) )) ))
```

Формально эффект таких **приостанавливаемых и возобновляемых вычислений** получается следующей реализацией операций || и @:

```
 ||e   = > (lambda () e )
@e   = >  (e ),
```

что при интерпретации дает связывание функционального аргумента с ассоциативным списком для операции || - приостановка и вызов функции EVAL для операции @ - возобновление вычислений.

Обычно в языках программирования различают вызовы по значению, по имени, по ссылке. Техника приостановки и возобновления функций может быть названа вызовом по необходимости.

При небольшом числе значений заданного типа, например, для истиностных значений, возможны вычисления с вариантами значений с последующим выбором реальной альтернативы пользователем, но это удобнее обсудить позднее, при изучении вариантов и недетерминированных вычислений.

Техника замедленных вычислений позволяет выполнять декомпозицию программы на обязательно выполнимые и возможно невыполнимые действия. Выполнимые действия в тексте определения замещаются на их результат, а невыполнимые преобразуются в остаточные, которые будут выполнены по мере появления дополнительной информации.

Многие выражения по смыслу используемых в них операций иногда определены при частичной определенности их операндов, что часто используется при оптимизации кода программ (умножение на 0, разность или частное от деления совпадающих форм и т.п.)

Отложенные действия - естественный механизм программирования асинхронных и параллельных процессов.

Выводы:

- *Замедленные вычисления могут быть результативнее обычных.*
- *Хранимые вместе с данными рецепты их вычислений позволяют работать с бесконечными множествами.*

- *В ряде случаев возможно получение полного результата при неопределенных значениях частей.*

9.**Традиционное (стандартное) программирование**

- Императивное программирование
- Списки свойств атома
- Структуры данных и памяти
- Деструктивные операции
- Свободная память и мусорщик

Форма “Prog” и циклы

Противопоставление функционального и **императивного (операторно-процедурного) стилей программирования** порой напоминает свифтовские бои остроконечников с тупоконечниками. Впрочем, переписать функциональную программу в императивную проще, чем наоборот.

С практической точки зрения любые конструкции стандартных языков программирования могут быть введены как функции. Это делает их вполне легальными средствами в рамках функционального подхода. Надо лишь четко уяснить цену такого дополнения и его преимущества, обычно связанные с наследованием решений или с привлечением пользователей. В первых реализациях Лиспа были сразу предложены специальные формы и структуры данных, служащие мостом между разными стилями программирования. Они заодно смягчали на практике недостатки **упрощенной схемы** интерпретации С-выражений, выстроенной для учебных и исследовательских целей. Важнейшие средства такого рода, выдержавшее испытание временем, - **prog-форма**, списки свойств атома и деструктивные операции. В результате язык программирования расширяется так, что становятся возможными оптимизирующие преобразования структур данных, программ и процессов и раскрутка систем программирования.

Рассмотрим предложенный МакКарти пример, показывающий возможности **prog**-формы при императивном стиле определения функции **Length**. Эта функция сканирует список и вычисляет число элементов на верхнем уровне списка. Значение функции Length - целое число. Алгоритм можно описать следующими словами:

"Это функция одного аргумента L.
Она реализуется программой с двумя рабочими переменными z и v.
Записать число 0 в v.
Записать аргумент L в z.
A: Если z содержит NIL, то программа выполнена и значением является то,
что сейчас записано в v.
Записать в z cdr от того, что сейчас в z.
Записать в v на единицу больше того, что сейчас записано в v.
Перейти к A"

Эту программу можно записать в виде Паскаль-программы с несколькими подходящими типами данных и функциями. Строкам вышеописанной программы соответствуют строки определения функции **LENGTH**, в предположении, что существует библиотека Лисп-функций на Паскале:

```
function LENGTH (L: list) : integer;

          var Z: list;
             V: integer;
begin
     V := 0;
     Z := l;
A:    if null (Z) then LENGTH := V;
     Z := cdr (Z);
     V := V+1;
          goto A;
end;
```

Переписывая в виде С-выражения, получаем программу:

```
(defun
LENGTH (lambda (L)
               (prog (Z V)
     (setq V 0)
     (setq Z L)
A       (cond  ((null Z)(return V)))
     (setq Z (cdr Z))
     (setq V (+ 1 V))
(go  A)  )))  ))

;;=======================ТЕСТЫ============
(LENGTH '(A B C D))
(LENGTH ‘((X . Y) A CAR (N B) (X Y Z)))
```

Последние две строки содержат тесты. Их значения 4 и 5 соответственно.

Форма Prog имеет структуру, подобную определениям функций и процедур в **Паскале**: (PROG, список рабочих переменных, последовательность операторов и атомов ...) Атом в последовательности выполняет роль **метки**, локализующей оператор, расположенный вслед за ним. В вышеприведенном примере метка A локализует оператор, начинающийся с "COND".

Первый список после символа PROG называется **списком рабочих переменных**. При отсутствии таковых должно быть написано NIL или (). С рабочими переменными

обращаются примерно как со связанными переменными, но они не могут быть связаны ни с какими значениями через lambda. Значение каждой рабочей переменной есть NIL, до тех пор, пока ей не будет присвоено что-нибудь другое.

Для **присваивания** переменной применяется форма **SET**. Чтобы присвоить переменной **pi** значение 3.14 пишется:

```
(SET (QUOTE PI)3.14)
```

SETQ подобна SET, но она еще и **блокирует вычисление первого аргумента**. Поэтому

```
(SETQ PI 3.14)
```

запись того же присваивания. SETQ обычно удобнее. **SET и SETQ могут изменять значения любых переменных** из ассоциативного списка более внешних функций. Значением SET и SETQ является значение их второго аргумента.

GO-форма, используемая для указания перехода (GO A) указывает, что программа продолжается оператором, помеченным атомом A, причем это A может быть и из более внешнего prog.

Условные выражения в качестве операторов программы обладают полезными особенностями. Если ни один из предикатов не истинен, то программа продолжается оператором, следующим за условным выражением.

RETURN - нормальное завершение программы. Аргумент return вычисляется, что и является значением программы. Никакие последующие операторы не вычисляются.

Если программа прошла все свои операторы, не встретив Return, она завершается со значением NIL.

Prog-форма может быть рекурсивной.

Пример: Функция REV, обращающая список и все подсписки, столь же естественно пишется с помощью рекурсивной Prog-формы.

Лисп	Паскаль

(DEFUN rev (x)
(prog (y z)
 	
A (COND ((null x)(return y)))
(setq z (CDR x))
 	
(COND ((ATOM z)(goto B)))
	function rev (x: list) :List
	var y, z: list;
	begin
	A: if null (x) Then rev := y;
	z := cdr (x);
	
	if atom (z) then goto B;

(setq z (rev z))
 	
B (setq y (CONS z y))
(setq x (CDR x))
(goto A)
))
	z := rev (z);
	
	B: y := cons (z, y);
	x := cdr (x);
	goto A
	end;

Функция rev обращает все уровни списка, так что rev от (A ((B C) D)) даст ((D (C B))A) .

Для того, чтобы форма prog была полностью законна, необходима возможность дополнять ассоциативный список рабочими переменными. Кроме того операторы этой формы требуют специального расширения языка - в него включаются формы go, set и return, не известные вне prog. (Формы Go, Set, Return возникли как операторы лишь на верхнем уровне PROG или внутри COND, находящегося на верхнем уровне PROG. Но в современных версиях Лиспа их можно встретить и в других позициях.)

Атомы, выполняющие роль меток, работают как **указатели помеченного блока**.

Кроме того произошло уточнение механизма условных выражений, - отсутствие истинного предиката не препятствует формированию значения **cond-оператора**, т.к. все операторы игнорируют выработанное значение. Это позволяет считать, что значением является Nil. Такое доопределение условного выражения давно перекочевало и в области обычных функций, где часто дает компактные формулы для рекурсии по списку. Исчезает необходимость в ветви вида “(T NIL)” .

В принципе SET и SETQ могут быть реализованы с помощью а-списка примерно также как и доступ к значению аргумента, только с копированием связей, расположенных ранее изменяемой переменной (см. функцию assign из параграфа 4). Более эффективная реализация, на основе списков свойств, будет описана ниже.

```
(DEFUN set (x y) (assign  x  y  Alist))
```

Обратите внимание, что введенное таким образом присваивание работает разнообразнее, чем традиционное присваивание: допущена **вычисляемость левой части присваивания**, т.е. можно в программе **вычислять имена переменных**, значение которых предстоит поменять.

Пример:

```
(setq x 'y)
(set x 'NEW)
(print x)
(print y)
```

Напечатается Y и NEW.

Списки свойств атомов

До сих пор атом рассматривался только как уникальный указатель, обеспечивающий быстрое выяснение различимости имен, названий или символов. В настоящем разделе описываются **списки свойств**, начинающиеся в **указанных ячейках**. (Образно говоря, переходим от химии к физике.)

Каждый атом имеет список свойств. Когда атом читается (вводится) впервые, тогда и создается для него список свойств. Список свойств характеризуется специальной структурой, подобной записям в Паскале, но поля в такой записи сопровождаются индикаторами, символизирующими смысл или назначение хранимой информации. Первый элемент этой структуры расположен по адресу, который задан в указателе атома. Остальные элементы доступны по этому же указателю с помощью ряда специальных функций. Элементы структуры содержат различные свойства атома. Каждое свойство помечается атомом, называемым **индикатором**, или расположено в фиксированном поле структуры.

Согласно стандарту Common Lisp глобальные значения переменных и определения функций хранятся в фиксированных полях структуры атома. Они доступны с помощью специальных функций symbol-function и symbol-value соответственно. Список произвольных свойств можно получить функцией symbol-plist. Функция remprop в Clisp удаляет лишь первое вхождение заданного свойства. Новое свойство можно ввести формой вида:

```
(setf (get Индикатор ) Свойство)
```

Числа представляются в Лиспе как специальный тип атома. Атом такого типа состоит из указателя с тэгом, специфицирующим слово как число, тип числа (целые, дробные, вещественные), и адрес собственно числа, код которого может быть произвольной длины. В отличие от обычного атома одинаковые числа не совмещаются при хранении

Структура списков и памяти

До этого момента списки рассматривались на уровне текстового ввода-вывода. В настоящем разделе рассматривается кодовое представление списков внутри машины и механизм сборки мусора, обеспечивающий повторное использование памяти.

В машине списки хранятся не как последовательности символов, а в виде структурных форм, построенных из машинных слов как частей деревьев, подобно записям в Паскале при реализации односвязных списков. Адреса в таких записях сопровождаются так называемыми тегами, специфицирующими тип данного, расположенного по указателю. При схематичном изображении структуры списка в виде диаграммы машинное слово рисуется как прямоугольник, разделенный на две части: **адрес и декремент.**

Теперь можно дать правило представления С-выражений в машине. Представление атомов будет пояснено ниже.

Преимущества структур списков для хранения С-выражений в памяти:

1) Размер и даже число выражений, с которыми программа будет иметь дело, можно не предсказывать. Кроме того, трудно организовать для размещения выражений блоки памяти фиксированной длины.

2) Ячейки можно переносить в список свободной памяти, как только исчезнет необходимость в них. Даже возврат одной ячейки в список имеет значение, но если выражения хранятся линейно, то трудно организовать использование лишнего или освободившегося пространства из блоков ячеек.

3) Выражения, являющиеся продолжением нескольких выражений, могут быть представлены только в одном экземпляре.

Для примера рассмотрим типичную двухуровневую структуру (A (B C)).
Она может быть построена из A, B и C с помощью

```
(cons ‘A (cons (cons ‘B (cons ‘C NIL)) NIL))
```

или, используя функцию list, можно то же самое записать как

```
(list ‘A (list ‘B ‘C))
```

Если дан список x из трех атомов x = (A B C), то аргументы A, B и C, используемые в предыдущем построении, можно найти как

```
A = (car x)
B = (cadr x)
C = (caddr x)
```

Исходную структуру из такого списка можно получить функцией **grp**, строящей (X (Y Z)) из списка вида (X Y Z).

```
(grp x) = (list (car x) (list (cadr x) (caddr x)))
```

Здесь grp применяется к списку X в предположении, что он заданного вида.

Деструктивные (разрушающие) операции

Элементарный Лисп универсален в смысле теории вычислительных функций от символьных выражений. Но для практичности как система программирования требуется дополнительный инструмент, увеличивающий выразительную силу и эффективность языка.

В частности, элементарный Лисп не имеет возможности модифицировать структуру списка. Единственная базисная функция, влияющая на структуру списка - это cons, а она не изменяет существующие списки, а создает все новые и новые. Функции, описанные в чистом Лиспе, такие как subst, в действительности не модифицируют свои аргументы, но делают модифицированную копию оригинала.

Элементарный Лисп работает как расширяемая система, в которой информация как бы никогда не пропадает. Set внутри Prog лишь формально смягчает это свойство, сохраняя ассоциативный список и моделируя присваивания теми же CONS.

Теперь же Лисп обобщается с точки зрения структуры списка добавлением разрушающих средств - **деструктивных базисных операций** над списками **rplaca** и **rplacd**. Эти операции могут применяться для замены адреса или декремента любого узла в списке подобно стандартным присваиваниям. Они используются ради их воздействия на память и относятся к категории псевдо-функций.

(rplaca x y) заменяет адресную часть x на y. Ее значение - x, но x, отличное от того, что было раньше. На языке значений rplaca можно описать равенством

(rplaca x y) = (cons y (cdr x))

Но действие совершенно различно: никакие cons не вызываются и новые слова не создаются.

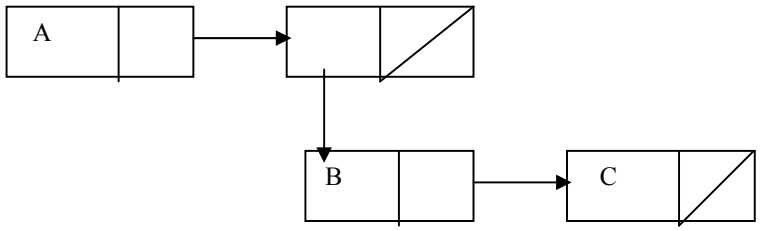
(rplacd x y) заменяет декремент x на y.

Деструктивные операции должно применять с осторожностью! Они могут совершенно преобразить существующие определения и основную память. Их применение **может породить циклические списки**, возможно, влекущие **бесконечную печать** или **выглядящие бесконечными** для таких функций как equal и subst.

Такие функции используются при реализации списков свойств атома и ряда эффективных, но **небезопасных, функций** Clisp-а, таких как nconc, mapc и т.п.

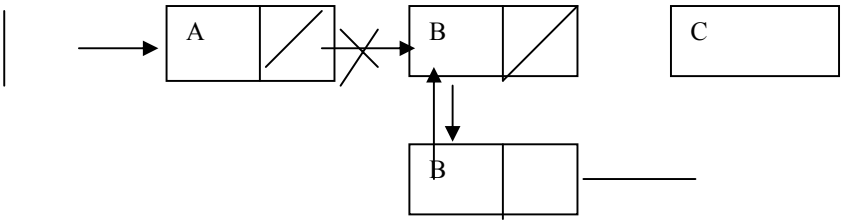
Для примера вернемся к функции grp. Это **преобразующая список функция**, которая преобразует копию своего аргумента, реорганизуя подструктуру

в структуру из тех же атомов:



Вышеприведнное определение функции делает это созданием новой структуры и использует для этого четыре cons. Из-за того, что в оригинале только три слова, по крайней мере один cons необходим, а grp можно переписать с помощью rplaca и rplacd.

Изменение состоит в следующем:



Пусть новое машинное слово строится как (cons (cadr x) (cddr x)) . Тогда указатель на него заготавливает форма:

```
(rplaca (cdr x) (cons (cadr x) (cddr x)))
```

Другое изменение состоит из удаления указателя из второго слова на третье. Оно делается как (rplaca (cdr x) NIL).

Новое опреление функции pgrp можно определить как соотношение:

```
(pgrp x) = (rplacd (rplaca (cdr x) (cons (cadr x) (cddr )x))) NIL)
```

Функция pgrp используется в сущности ради ее действия. Ее значением, неиспользуемым, является подструктура ((B C)). Поэтому необходимо, чтобы pgrp выполнялось, а ее значение игнорировалось.

Расширенный деструктивными функциями Лисп хорошо приспособлен к **оптимизации программ**. Любые совпадающие подвыражения можно локализовать и вынести за скобки.

Выводы:

- *При реализации языка программирования происходит его расширение с целью*
- *повышения практичности программирования.*
- *Стандартные, операторно-процедурные построения моделируются с помощью функций.*

- *Списки свойств атомов обеспечивают прямой доступ к значениям и определениям, а также к произвольным свойствам, как встроенным, так и программируемым.*
- *Деструктивные операции могут дать большую эффективность вычислений ценой надежности программирования.*
- *Автоматическое перераспределение памяти позволяет программисту не отвлекаться от решения своих задач на технические проблемы.*

10. Парадигмы программирования

Средства и методы программирования на Лиспе образуют два слоя. Глубинный слой - **локальное программирование,** нацеленное на определение:
- строгих функций,
- безотходных структур данных,
- регулярных отображений,
- средств управления вычислениями.

Внешний слой - функциональное моделирование практичных **парадигм программирования**:

- прототипы и макеты программ,
- интеграция разных стилей и методов программирования,
- экспериментальное программирование,
- проверка новых идей и подходов к организации информационных систем.

Стиль разработки программ на Лиспе, получивший название "функциональное программирование" (ФП), воспринят многими языками программирования с **общей логикой уточнения решаемых задач** и обобщения решений на основе выбранных специально базовых конструкций:

1) **Базовые конструкции** определяются как строгие функции.

2) При необходимости выполняются **преобразования программ**, (компиляция, оптимизация и т.п.) для улучшения эксплуатационных характеристик, связанных с процессами исполнения программ.

3) Важный критерий качества ФП - **полнота системы функций и универсальность определений** для синтаксически управляемой обработки данных функциями высоких порядков (компилятор и т.п.), что существенно повышает надежность программирования.

4) Разработка ИС средствами ФП успешно выполняет **роль прототипа** для реализации другими, более привычными, средствами.

Отправляясь от **однозначных функций**, мы разрешили предельно широкое толкование понятия “значение”, включающее понятие “структура данных” и “функция”.

1) Ориентируясь на **рекурсивные определения функций**, мы ввели несложную схему, достаточно удобную для построения формул, задающих функциональные определения. В качестве примера рассмотрен **элементарный Лисп**.

2) Представления функций отображены на множество списков и атомов и определена **универсальная функция**, по списочному представлению функции и ее аргументов строящая ее результат.

3) Изучено расширение функционального языка, достаточное для **стандартного программирования**, естественного для привычных задач.

Конструирование функций средствами чистого Лиспа доставляет интеллектуальное удовольствие, оно сродни решению математических головоломок. В этом исключительно мощном языке не только реализованы основные средства, обеспечившие практичность и результативность функционального программирования, но и впервые опробован целый ряд поразительно точных построений, ценных как концептуально, так и методически и конструктивно, понимание и осмысление которых слишком отстает от практики применения. Понятийно-функциональный потенциал языка Lisp 1.5 в значительной мере унаследован стандартом Common Lisp, но не все идеи получили достойное развитие. Возможно это дело будущего - для нового поколения системных программистов, но это уже другая история.

По мере накопления опыта реализации Лиспа и других языков сформированы обширные библиотеки функций, весьма эффективно поддерживающих обработку основных структур данных - списков, векторов, множеств, хэш-таблиц, а также строк, файлов, директорий, гипертекстов, изображений. Существенно повысилась результативность системных решений в области работы с памятью, компиляцией, манипулирования пакетами функций и классами объектов. Все это доступно в современных системах, таких как GNU Clisp, Python, CMUCL и др., основная проблема при изучении которых – слишком много всего, разбегаются глаза, трудно выбрать первоочередное. Все это превращает любой диалект Лиспа в практичный инструментарий, обладающий интересными перспективами.

Результативность идей Лиспа подтверждена самой историей развития его диалектов и родственных ему языков программирования. (Pure Lisp, Lisp 1.5, Lisp 2, Interlisp, CommonLisp, MicroLisp, MuLisp, Sail, Hope, Miranda, Scheem, ML, GNU Clisp, CLOS, Emacs, Elisp, xLisp, Vlisp, AutoLisp, Haskell, Python, CMUCL). Стандарт **Common Lisp** в сравнении с Лиспом от МакКарти имеет ряд отличий, несколько влияющих на программотехнику. GNU Clisp, xLisp, CMUCL соответствуют стандарту Common Lisp.

Функциональное программирование сформировалось при исследовании и развитии искусственного интеллекта и освоении новых горизонтов в информатике. Продуманность и методическая обоснованность первых реализаций Лиспа позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют сотни языков функционального программирования, ориентированных на разные классы задач и виды технических средств.

Идеи Лиспа выдержали многолетнюю шлифовку и заслужили достойную оценку специалистов и любителей. Универсальность Лиспа достаточна для моделирования любого стиля программирования. Выразительная сила Лиспа обретает новое дыхание на каждом эволюционном витке развития информационных технологий. Потенциал Лиспа нам еще предстоит раскрыть. Стилистика Лиспа несколько противоречат традиционным подходам к представлению программ. Но это противоречие отступает перед обаянием строгой логики языка. Определение Лисп-систем средствами самого Лиспа дает гибкую основу для развития языка и реализующей его системы программирования. На Лиспе решение задачи выражается в терминах постановки задачи без привлечения реализационных сущностей и интерфейсных эффектов.

Базис Лиспа идеально лаконичен - атомы и простые структуры данных –девять функций и функционалов - обычные функции, которые анализируют, строят и разбирают любые структурные значения (atom, eq, cons, car, cdr,), и специальные функционалы, которые управляют обработкой структур, представляющих вычисляемые выражения (quote, cond, lambda, eval).

Синтаксис Лиспа изысканно прост. Разделитель - пробел, ограничители - круглые скобки. В скобки заключается представление функции с ее аргументами. Все остальное - вариации в зависимости от категории функций, определенности атомов и вычислимости выражений, типов значений и структур данных. **Функционалы** - это одна из категорий функций, используемая при организации управления вычислениями.

Лисп - **язык символьной обработки информации**. Методы программирования на Лиспе часто наывают “функциональное программирование”. Лисп прочно утвердился как эсперанто для задач искусственного интеллекта. К середине семидесятых годов XX века на Лиспе решались **наиболее сложные в практике программирования** задачи из области дискретной и вычислительной математики, экспериментального программирования, лингвистики, химии, биологии, медицины и инженерного проектирования. На Лиспе реализована AutoCAD - система автоматизации инженерных расчетов, дизайна и комплектации изделий из доступного конструктива, и Emacs – весьма популярный текстовый редактор в мире UNIX/Linux. Многие созревшие на базе Лиспа системные решения постепенно обрели самостоятельность и выделились в отдельные направления и технологии.

Реализационные находки Лиспа, такие как ссылочная организация памяти, “сборка мусора” - автоматизация повторного использования памяти, частичная компиляция программ с интерпретацией промежуточного кода, длительное хранение атрибутов объектов в период их использования и др., перекочевали из области исследований и экспериментов на базе Лиспа в практику реализации операционных систем и систем программирования.

Приверженцы Лиспа ценят его не только за **элегантность, гибкость,** но и за **способность к точному представлению программистских идей и удобной отладке.** В стандартных языках программирования принята императивная организация вычислений по принципу немедленного выполнения каждой очередной команды. Это не всегда обосновано и эффективно. Неимперативные модели управления процессами позволяют прерывать и откладывать процессы, а потом их восстанавливать и запускать или отменять, что обеспечено в Лиспе средствами конструирования функций, блокировки вычислений и их явного выполнения.

История и выводы (вместо заключения)

История Лиспа пронизана жаркими спорами, притиворечивыми суждениями, яркими достижениями и смелыми изобретениями:

1958 - Первые публикации Джона Мак-Карти о замысле языка символьной обработки.
1962-1964- Авторские проекты первых Лисп-систем .
1964- Демонстрация принципиальной решаемости проблем искусственного интеллекта. (Написанная Дж.Вейценбаумом на Лиспе программа-собеседник “Элиза”, имитирующая речевое поведение психоаналитика, дала положительный ответ на вопрос о возможности искусственного разума.)
1972-1974 - Разрешение теоретических парадоксов, связанных с бестиповым лямбда-исчислением.
1972-1980 - Стандартизация языка Лисп.
1978 – Появление Лисп-компьютеров.
1965-1990 - Построение специализированных диалектов Лиспа и создание практичных реализаций для широкого спектра весьма различных применений, включая инженерное проектирование и системы математической обработки информации
1992-2002 - Разработка визуальных и сверхэффективных Лисп-систем, таких как CMUCL.

В нашей стране программирование знакомство с языком Лисп состоялось из первых рук. В конце 1968 года Джон Мак-Карти лично познакомил программистов Москвы и Новосибирска с Лиспом, что побудило к реализации отечественных версий языка.

Литература и сайты

1. McCarthy J. LISP 1.5 Programming Mannual.- The MIT Press., Cambridge, 1963, 106p.

2. Хендерсон П. Функциональное программирование.- М.: Мир, 1983

3. Хьювенен Э., Сеппанен Й. Мир Лиспа., т.1,2, М.: Наука, 1994

4. Сергеев Л.О. Удивительный мир языка Лисп. Введение в язык и задачи, задачи, задачи. // Информатика – 2000, N 29.

5. Лавров С.С. Функциональное программирование. // Компьютерные инструменты в образовании. – 2002, N 2-4.

6. Городняя Л.В., Лавров С.С. Функциональное программирование. Принципы реализации языка Лисп. // Компьютерные инструменты в образовании. – 2002, N5, с. 49-58

7. http://www-formal.stanford.edu/jmc/ - Личный сайт Дж. МакКарти с текстами его публикаций

8. http://www.marstu.mari.ru/mmlab/home/lisp/title.htm – Дистанционный учебник М.Н.Морозова

9. http://grimpeur.tamu.edu/~colin/lp/ - Небольшой учебник для начального знакомства с Лиспом

10. http://www.paulgraham.com/onlisptext.html - Интересные материалы по Clisp Паула Грэхема, автора книги по стандарту ANSI Clisp

11. Городняя Л.В. Основы функционального программирования. Курс лекций. Учебное пособие. Серия «Основы информационных технологий» /М.: ИНТУИТ.РУ «Интернет-университет Информационных Технологий», 2004. – 280 с.

Термины

Термин	Term	Определение
Алгоритм	algorithm	Абстрактное представление процесса, который может быть механизирован, т.к. ему предстоит быть запрограммированным для выполнения машиной.
А-список	a-list	**Ассоциативный список**
Ассоциативный список	Association list	Список пар, эквивалентный таблице с двумя столбцами. Используется для связывания **переменных** с их значениями и **функций** с их опрелениями. Например, ((VAR1 . VAR2) (B . (U V (W)))(C . Z))
Атом	Atom	**Атомный символ**
Атомный символ	Atomic symbol	Минимальный элемент **S-выражений**. Допустимые атомные символы – это определенные последовательности букв, цифр и специальных литер. Примеры: AI ИИ NIL База_2-1
Базисные функции	Basic functions: CAR, CDR, CONS, EQ, ATOM	Эти функции называются базисными, потому что их достаточно для построения полного класса вычислимых функций от S-выражений использованием композиции, условных выражений и рекурсии.
Всюду-определенные функции	total function	Функция, которая возвращает результат на любом аргументе - **тотальная функция**
Деструктивные функции	Destructive functions	Функции, изменяющие структуру входных данных. Например, nconc
Замедленные вычисления	Lazy evaluation	Модель вычислений, при которой задерживается вычисление всех фактических аргументов всех функций, а также задерживается вычисление самих функций. Для получения результата необходимо возобновлять вычисления.
Замыкание функции	Lexical closure	Определение функции + контекст в момент определения этой функции. В Лиспе замыкание строится с помощью функции FUNCTION. Построение замыканий позволяет более надежно оперировать с функциями, у которых есть свободные переменные
Идеальная функция	Pure function	**Чистая функция**, без побочных эффектов с однозначным результатом
Индикатор свойства	Indicator	Имя **свойства атома** (атрибут), расположенное непосредственно перед значением этого свойства в списке свойств атома.

Интерпретатор	Interpreter	Интерпретатор выполняет программу, используя ее исходный язык и формально исполняя представленные программой алгоритмы. В этом отличие от компилятора, который переводит программу с исходного языка на язык машины для многократного исполнения алгоритмов потом.
Истинностные значения	True values	**Истина** и **Ложь**. В Лиспе этим понятиям соответствуют атомы **T** и **NIL**.
Компилятор	compiler	Программа, переводящая с исходного языка на машинный (объектный) язык. В отличие от большинства компиляторов, LISP – компилятор не пытается компилировать полную программу до ее исполнения. Более эффективна компиляция отдельных функций, описанных как **S-выражения**, на машинный язык во время вычисления, по мере их отладки.
Композиция	composition	Композиция функций – это использование значения одной функции в качестве аргумента другой. Обычно композиция записывается с помощью вложенных скобок. Пример: **(CONS (CAR Y) X)**
Константа	Constant	Символ, значение которого не меняется в течение вычислений. Примеры: **T NIL 1 2 3 «строка» …**
Контекст	context	Состояние среды, в которой происходит вычисление
Лисп	Lisp	Универсальный язык программирования, созданный Джоном Маккарти в конце 60-е годов.
Лисп-процессоры	Lisp processor	Аппаратные реализации подмножеств языка Лисп, концептуально близких SECD и языку Scheme - активно используется статический контекст. Были выпущены рядом ведущих фирм в конце 70-х годов XX века.
Логическая форма	Logical form	**Специальная форма**, включающая **AND, OR** или **NOT**, которая может использоваться для построения значений истинности и конструирования **предикатов**.
Лямбда-выражение	Lambda notation	Обозначение, впервые предложеное Черчем для обобщения выражений (форм) и функций, использующее греческую букву «лямбда» для четкого выделения в формулах аргументов функций.
Мемо-функция	Memo-function	Функция, которая накапливает ранее вычисленные соответствия между аргументом и результатом
Отображение	Imaging, map	Подмножество декартового произведения области определения и области значения
Переменная	variable	Переменная - символ, значение которого может меняться в процессе выполнения программы.
Предикат	Predicate	Функция со значениями истина (true) или ложь (false)
Псевдо-функция	Pseudo-function	Функция с побочным эффектом
Пустой список	Empty list	Список, в котором нет элементов. В Лиспе обозначается как **NIL**

Рекурсия	Recursion	Определение является рекурсивным, если оно ссылается на себя. Это может быть явная ссылка из определения или косвенная, через серию определений
Сборка «мусора»	Garbage collector, reclaimer	Проверка доступных структур с пометкой, чтобы выделить ненужные, недоступные слова – «мусор» . Затем весь «мусор» собирается в **список свободной памяти**, с тем чтобы эти слова использовать повторно.
Символ	Symbol	Атом, обладающий списком свойств для хранения имени, значения, определения функции, идентификации принадлежности пакету и пр.
Список свойств	Property list, p-list	Набор индикаторов и значений свойств, объединенных в одноуровневый список и связанный с некоторым атомом в таблице атомов.
Специальная форма	Special form	Встроенные функции, которые сами вычисляют свои аргументы. Такая техника обеспечивает связывание переменных, блочную структуру, циклы и другие процессы управления вычислениями.
Свободная переменная	Free variable	Переменная, не объявленная ни как рабочая в блоке, ни как связанная в функции. Переменная может рассматриваться как связанная или свободная только внутри контекста, в котором она появляется. Для вычисления интерпретатором переменная должна быть связана на некотором уровне или иметь постоянное значение (константа).
Свойство атома	Propert y	Данное, размещенное в **списке свойств атома** непосредственно вслед за индикатором этого свойства.
Связанные переменные	Bound variable	Переменная, входящая в список связанных переменных, расположенный за LAMBDA. Значением связанной переменной является аргумент, расположенный в позиции, соответствующей вхождению переменной в LAMBDA-список. Например, в выражении вида: ((LAMBDA (X Y) E)(QUOTE (A B)) (QUOTE C)) **Y** имеет значение **C**, для любых его вхождений в **E, а X – (A B).**
Список свободной памяти	Free-storage list	Список доступных слов памяти, используемых при вычислении. Каждое обращение к **CONS** преобразует первое слово списка свободной памяти, которое из этого списка исключается. Когда список свободной памяти исчерпан, тогда проводится **сборка мусора** и строится новый список свободной памяти.
Список	List	**S-выражение**, для которого движение вправо ограничено символом **NIL**. Предикат LISTP(x) = NULL(x) OR (NOT (ATOM (x)) AND LISTP (CDR (x)))
Списочная запись	List notation	Метод изображения **S-выражений** в виде **(m1 m2 … mn)** вместо вида **(m1 . (m2 . (…. (mn . NIL)))**

Таблица атомов	Atom table	Форма хранения информации об атомах в реализациях Лиспа.
Тег	Tag	Код **типа значения** в структуре данных. Сопровождает адрес в указателе.
Тотальная функция	Total functio n	**Всюду определенная функция**
Точечная нотация	Dot notation	Метод записи **S-выражений** составлением заключенных в скобки пар S-выражений, разделенных точкой. Точечная нотация – базовая структура данных в Lisp-е. **Списочная запись** определяется с помощью точечной нотации. Примеры: (A . B) (A . NIL)
Указатель	Pointer	Обычно - **адрес**, по которому, можно найти продолжение информации. В Лиспе такой адрес сопровождается тегом, задающим тип значения, расположенного по адресу.
Универсальна я функция	Univers al functio n	Функция, аргументы которой – S-выражения, представляющие любую вычислимую форму или функцию и ее аргументы. Значение универсальной функции – это значение формы или результат вычислимой функции, примененной к ее аргументам. Примеры: **EVAL, APPLY**
Условное выражение	Conditional expression	Запись **ветвления** в виде выражения, состоящего из списка предикатов и соответствующих им форм. Значением условного выражения является значение формы, соответствующей первому истинному предикату. Пример: **(COND ((< A 0) B) (T C))**
Форма	Form	**Выражение**, которое может быть вычислено, если установлено некоторое соответствие между входящими в него переменными и набором фактических аргументов. Функции в отличие от формы следует явно задать аргументы.
Фуннкциональный аргумент	Functional argument	**Функция**, являющаяся аргументом функционала. В Лиспе функциональные аргументы обозначают формой “#” (**FUNCTION).**
Функция	Function	Функция преобразует входные данные в выходные. Функции в Лиспе описываются с помощью специальной функции **DEFUN**. Программа на Лиспе - это коллекция функций.
Чистый Лисп **APPLY**	Pure lisp APPLY	Абстрактный язык программирования на базе минимального набора функций: CAR, CDR, CONS, EQ, ATOM, COND, LAMBDA,QUOTE. Метод применения функции к списку ее аргументов. EVAL и APPLY - функции с помощью которых можно сконструировать интерпретатор Лиспа. Все функции высших порядков в той или иной степени используют APPLY или FUNCALL

CAR	CAR	Голова неатомарного **S-выражения**. Имя образовано от **Contents of Addres of Register**. Голова списка или точечной пары. Левая часть
CDR	CDR	Хвост неатомарного S-выражения. Имя образовано от **Contents of Decrement of Register**
EVAL	EVAL	Функция, которая вычисляет вражения в Лиспе в соответсвии со списком фактических параметров и семантикой вычисляемого выржения
NIL	NIL	**Пустой список**. Ложь.
S - выражение	S - expression	Универсальная структура данных для символьных вычислений – **символьное выражение**.