Функциональное
программирование

Лекция 2-3
Основы языка Лисп

Содержание

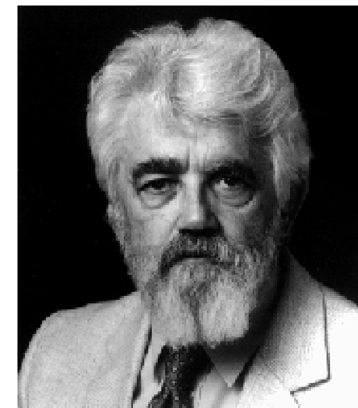
1. Язык Лисп. Особенности.
2. Базовые понятия
3. Функции. Формат записи. Виды.
4. Условные выражения.
5. Рекурсия. Примеры программ.

1

Введение

Язык Лисп

Язык Лисп



- Лисп (LISP) был разработан в1958 году американским ученым

  Джоном Маккарти (MIT) как функциональный язык, предназначенный для обработки списков ~ **LIS**t **P**rocessing
- Лисп - один из наиболее распространенных базовых языков искусственного интеллекта. После его появления различными авторами был предложен ряд других алгоритмических языков, ориентированных на решение задач в области ИИ, среди которых можно отметить Плэнер, Снобол, Рефал, Пролог, Смолтолк. Однако это не помешало Лиспу остаться наиболее популярным языком для решения таких задач. Более того, предполагается, что Лисп наряду с Прологом будет одним из основных языков компьютеров пятого поколения. Наибольшую популярность Лисп получил в США.
- Долгое время язык использовался узким кругом исследователей. Широкое распространение язык получил в конце 70-х - начале 80-х годов с появлением необходимой мощности вычислительных машин и соответствующего круга задач. Сейчас Лисп - одно из главных инструментальных средств систем искусственного интеллекта (вытеснил в свое время язык Ада; система AutoCAD была разработана на Лиспе).

Особенности Лиспа

- Одинаковая форма представления данных и программ – в виде списка
  - Это позволяет программе обрабатывать другие программы и даже саму себя.
- Функциональный образ мышления
- Не требуется явное описание типов данных, используемых в программе
- Лисп как правило является интерпретирующим языком, также как BASIC и др.
- Это безтиповый язык
  - это значит что символы не связываются по умолчанию с каким-либо типом.
- Использует макросы, что делает его эффективным языком для метапрограммирования
- Поддерживает итерационное, объектное, списковое, модульное, процедурное, рефлексивное программирование
- Лисп имеет необычный синтаксис. Из-за большого числа скобок LISP расшифровывают как «Lots of Idiotic Silly Parentheses».
  - Принято считать, что эквивалентные программы, написанные на Лиспе порой во много раз короче, чем написанные на процедурных языках.
- В основу языка положен серьезный математический аппарат:
  - лямбда-исчисление Черча
  - алгебра списочных структур
  - теория рекурсивных функций

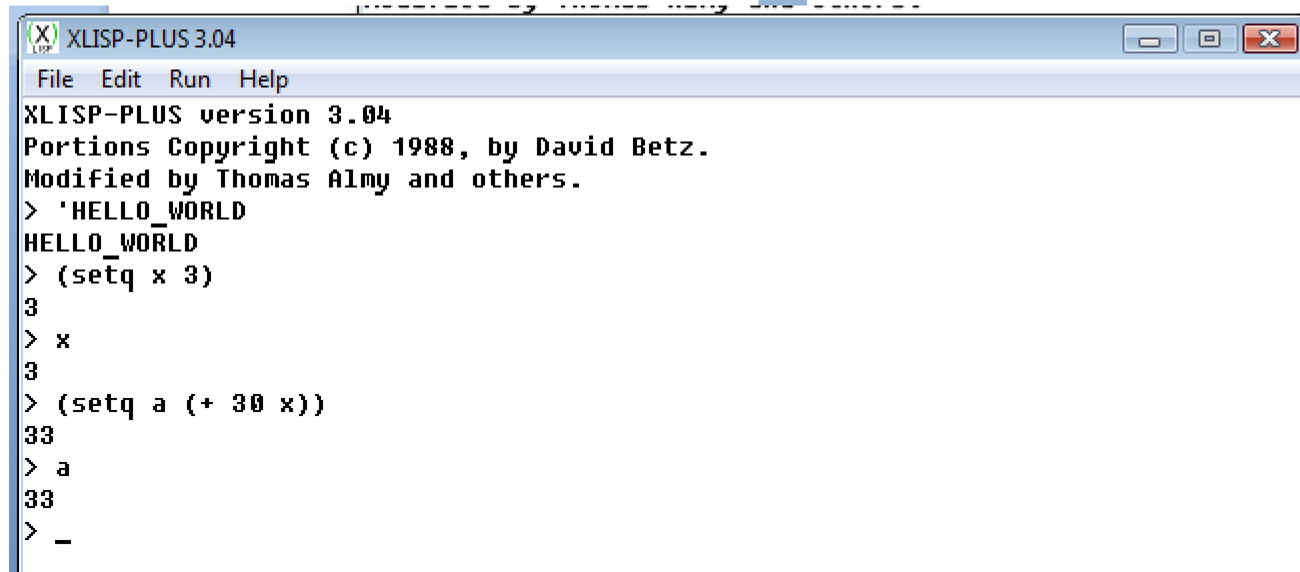
Почему Лисп все еще популярен?

Популярность Лиспа объясняется следующими причинами:

- 1) Лисп ориентирован на работу с символьной информацией, а процесс решения большинства задач искусственного интеллекта сводится к обработке такой информации.
  - Английское название LISP, являющееся аббревиатурой выражения LISt Processing (обработка списков), хорошо подчеркивает основную область его применения
- 2) Лисп представляет собой интерпретирующую систему, а это позволяет значительно облегчить и ускорить процесс создания сложных программных комплексов в интерактивном режиме.
- 3) Идеология Лиспа крайне проста: данные и программы представляются в нем в одной и той же форме. Благодаря такой унификации представления данные могут интерпретироваться как программа, а любая программа может быть использована как данные любой другой программой.
- 4) Язык Лисп является языком функционального программирования. Применение функциональных языков открывает широкие перспективы, позволяя пользователю описывать скорее природу своих задач, чем способ их решения.
- 5) Язык Лисп может служить основой для обучения методам искусственного интеллекта, исследованиям и практическому применению в этой области.

Диалекты (версии) Лиспа

- Лисп имеет множество диалектов, однако многие специалисты склонны отдавать предпочтение диалекту Common Lisp в качестве будущего стандарта
- Common Lisp
- Scheme
- AutoLisp
- **Xlisp**



(*) Ввод в интерпретатор XLisp определений функций и их вызовов может осуществляться как с командной строки интерпретатора, так и из файла. Для последнего случая предусмотрена системная функция LOAD: (LOAD <имя файла>). Читаемые из файла выражения выполняются так, как будто бы они были введены пользователем. В Xlisp <имя файла> - это строчное имя файла или символ (со строчным значением). При этом под именем файла подразумевается путь к файлу, описанный средствами MS DOS, причем символ "/" в пути удваивается. Функция LOAD возвращает T в случае, если файл удачно загружен, и NIL - в противном случае:
> (LOAD "c:\\xlisp\\prog.lsp").

Фундаментальные свойства Лиспа как языка ФП

1. Унификация понятий <функция> и <значение>

- При символьном представлении информации нет принципиальной разницы в природе изображения значений и функций. Следовательно нет и препятствий для обработки представлений функций теми же средствами, какими обрабатываются значения, т.е. представления функций можно строить из их частей и даже вычислять по мере поступления и обработки информации. Так конструируют программы компиляторы.

2. Кроме функций-констант вполне допустимы функции-переменные

- Отсутствие навыков работы с функциональными переменными говорит лишь о том, что надо осваивать такую возможность, потенциал которой растет: программирование становится все более компонентно ориентированным.

3. Самоприменимость

- Первые реализации Лиспа были выполнены методом раскрутки, причем в составе системы сразу были предусмотрены и интерпретатор и компилятор. Оба эти инструмента были весьма точно описаны на самом Лиспе, причем основной объем описаний не превосходил пару страниц.

4. Интегральность ограничений на пространственно-временные характеристики

- Если не хватает памяти, то принципиально на всю задачу, а не на отдельные блоки данных, возможно мало существенных для ее решения. При недостатке памяти специальная программа "мусорщик" пытается найти свободную память. Особенно эффективно при работе на больших объемах памяти.

5. Уточняемость решений

- Реализация Лиспа содержит списки свойств объектов, приспособленные к внешнему доопределению отдельных элементов поведения программируемой системы.

6. Динамическое управление вычислениями и конструированием программ

- В стандартных языках программирования принята императивная организация вычислений по принципу немедленного и обязательного выполнения каждой очередной команды. Это не всегда оправдано и эффективно. Существует много неимперативных моделей управления процессами, позволяющих 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.

2

Основные понятия

Базовые термины…

- **Функцией** называется правило, по которому каждому значению одного или нескольких аргументов ставится в соответствие конкретное значение результата.
  - Функциональный язык обладает двумя мощными механизмами, позволяющими писать более краткие и универсальные программы. (1) аргументами или результатом функции могут быть сложные структуры данных. Однажды построенная структура не может изменяться, но может включаться в другие (2) можно определить функцию высшего порядка, аргументами или результатом которой является функции.
- Функция называется **рекурсивной**, если ее определение прямо или косвенно (через другие функции) содержит обращение к самой себе.
- **Вычисление** – процесс решения задачи, сводимой к обработке чисел, кодов или символов, рассматриваемых как модели реальных объектов.
- Соответствие между моделью и объектом часто называют **интерпретацией**.
- **Список** – основная структура данных в ФП. Список может быть пустым или содержать про-извольное число объектов любой природы. Пустой список используется в качестве значения "ложь" - все, что отлично от пустого списка может выполнять роль "истины".
- **Символ** в ФП аналогичен переменной в традиционном ЯП – это имя, состоящее из букв латиницы, цифр и некоторых специальных литер. Символ, как и перемен-ная, может иметь какое-либо значение, то есть представлять какой-либо объект.
  - Наряду с символами, в ФП используются также: числа (целые и вещественные); специальные константы t и nil, обозначающие логические значения true (истина) и false (ложь); списки.

Атомы

- **Атомы** - это простейшие объекты Лиспа, из которых строятся остальные структуры.

- **Символьные атомы** - последовательность букв и цифр, при этом должен быть по крайней мере один символ отличающий его от числа.
- Примеры: ДЖОН АВ13 В54 10А

- Символьный атом или символ - это не идентификатор переменой в обычном языке программирования. Символ как правило обозначает некий предмет/объект/действие.
  - Он аналогичен переменной в традиционном языке программирования – это имя, состоящее из букв латиницы, цифр и некоторых специальных литер. Символ, как и переменная, может иметь какое-либо значение, то есть представлять какой-либо объект.

- Символьный атом рассматривается как неделимое целое.
- К символьным атомам применяется только одна операция: сравнение.
- В состав символа могут входить:
- + - * / @ $ % ^ _ \ <>

- **Числовые атомы** - обыкновенные числа
- 124 -344 4.5 3.055E8
- Числа это константы.
- Типы чисел зависят от реализации ЛИСПа

Списки

- В ЛИСПЕ список это последовательность элементов (list).
- **Элементами** являются или атомы или списки.
- **Списки** заключаются в скобки, элементы списка разделяются пробелами.
  - (банан) ;1 атом
  - (б а н а н) ; 5 атомов
  - (a b (c d) e) ; 4 элемента

- **Список** - это многоуровневая или иерархическая структура данных, в которой открывающиеся и закрывающиеся скобки находятся в строгом соответствии.
  - (+ 2 3) ; 3 атома
  - (((((первый) 2) второй) 4) 5) ; 2 элемента

- Список, в котором нет ни одного элемента, называется **пустым списком** и обозначается "()" или символом NIL (это и список и атом одновременно).
- Он играет такую же важную роль в работе со списками, что и ноль в арифметике.
- Пустой список может быть элементом других списков.
  - (NIL) ;список состоящий из атома NIL
  - (()) ;то же самое, что и (NIL)
  - ((())) ;- " -((NIL))
  - (NIL ()) ;список из двух других списков
- **Голова** списка – первый элемент списка, **Хвост** – все оставшиеся элементы (в свою очередь являющиеся самостоятельным списком)

Точечная нотация

- Список представляет собой ссылочную структуру.
- Основная ссылочная структура языка - так называемая точечная пара, которая состоит из указателей на первый и второй элементы пары.
- Так, например, точечная пара с указателями на атомы A и B изображается в символьной нотации как (A . B).
- Если точечная пара первым указателем ссылается на атом A, а вторым - на другую точечную пару (B . C), то это изображается в символьной нотации как (A . (B . C)).
- Список (A B C D) в точечной нотации будет иметь следующий вид:

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

Логические константы

- **NIL** обозначает кроме этого, в логических выражениях логическую константу "ложь" (false).
- **T** обозначает логическое "да" (true).

Пример: истина

```
> (= 2 2)
T
```

Пример: ложь

```
> (= 2 3)
NIL
```

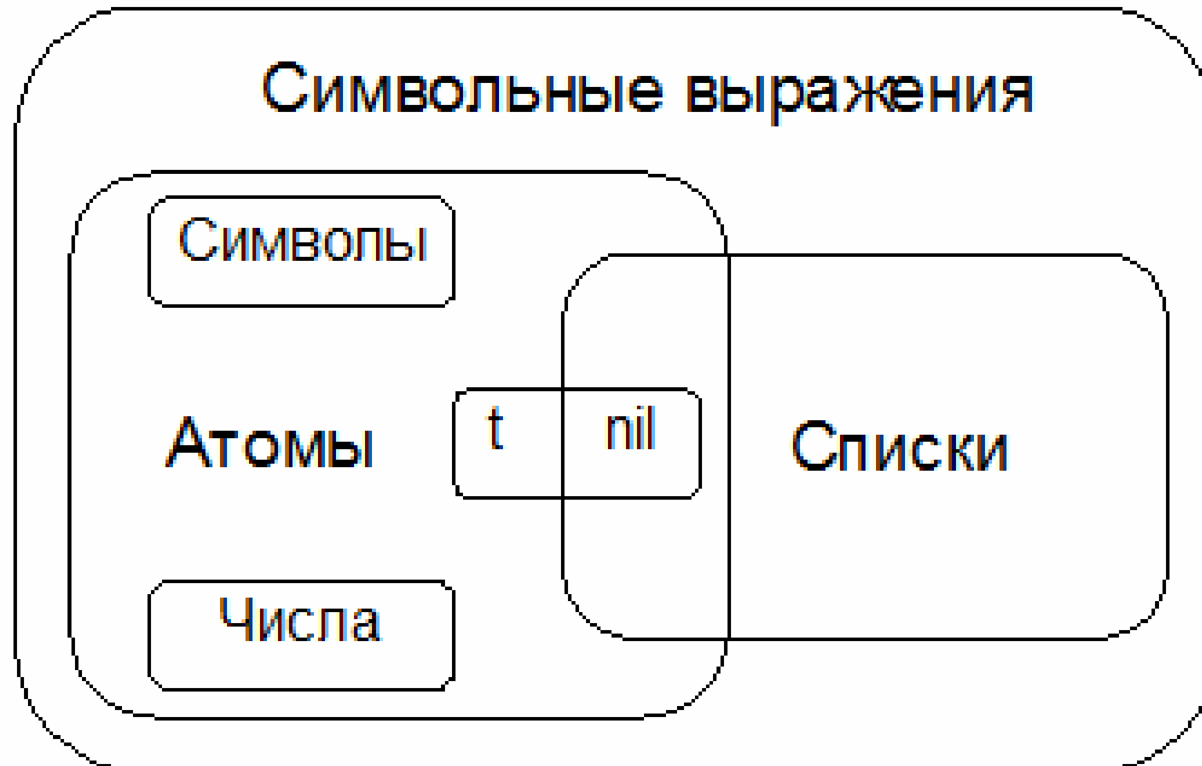
(*) Здесь и далее в примерах знак ">" означает приглашение интерпретатора Xlisp.

После данного знака набирается интерпретируемое выражение.

Ответ интерпретатора представляется на второй строке.

Символьные выражения

Все вышеперечисленные объекты (атомы и списки) называют **символьными выражениями.** Отношения между различными символьными выражениями можно представить следующим образом:



(S)-выражение – это либо атом, либо список; атомы являются простейшими S-выражениями

Основные понятия

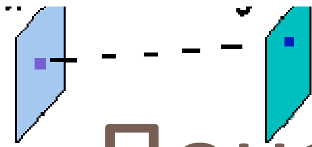
Функции

Функции

- По умолчанию предполагается, что первый элемент в (S)-выражениии (списке) – имя функции или определяющее выражение функции.
Во всех остальных случаях список не имеет значения.

- Примеры.
  - (P Q R) может иметь значение лишь в том случае, когда P – имя функции двух переменных.
  - Выражение ((1 2) 3) не имеет значения.

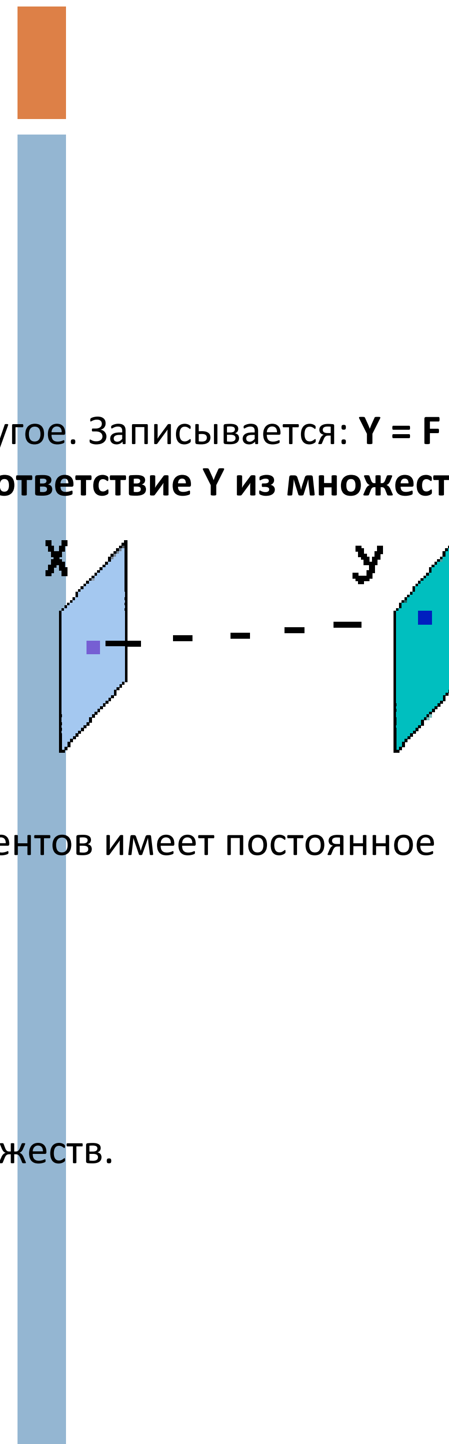
- Значение функции вычисляется при каждом обращении к функции.
- В общем случае значение функции зависит от аргументов, указанных в обращении и от состояния программы. Это значение вычисляется по определенным для этой функции правилам.



Понятие функции

- В математике **функция** отображает одно множество в другое. Записывается: **Y = F (x)**
- **Для x из множества определения (Domain) ставится в соответствие Y из множества значений (range) функции F.**
- Можно записать: $Y = F(x)$ $F(x) \rightarrow Y$

  $F : x \rightarrow Y$



- У функции может быть любое количество аргументов,
  в том числе их может не быть совсем. Функция без аргументов имеет постоянное значение.

- **Примеры функций:**
- **abs(-3) --> 3** ;абсолютная величина.
  + (2 , 3) --> 5 ; сложение
  union ((a , b), (c , b)) --> (a , b , c) ; объединение множеств.
  количество_букв (слово) --> 5

Префиксная нотация

- В математике принята префиксная нотация, в которой имя функции стоит перед аргументами заключенными в скобках.
- в арифметических выражениях используется инфиксная запись :

  f (x) g (x , y) h (x , g (y , z))

  x + y x – y x * (x + z)

- В Лиспе для записи арифметических выражений и функций используется единая префиксная форма записи, в которой имя функции или действия стоит перед аргументами и записывается внутри скобок.

  (f x) (g x y) (h x (g y z))

  (+ x y) (- x y) (* x (+ x z))

Достоинства :

- упрощается синтаксческий анализ выражений, так как по первому символу текущего выражения система уже знает, с какой структурой имеет дело.
- появляется возможность записывать функции в виде списков, т.е. данные (списки) и программа (списки) представляются единым образом.

Диалог с интерпретатором

- Транслятор **Лиспа** работает как правило в режиме интерпретатора.

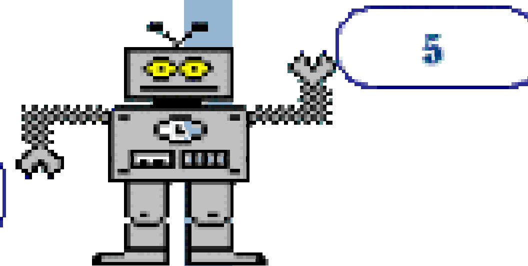
  «Read-eval-print цикл»
- **loop { read evaluate print)**

  В **Лиспе** сразу
  - читается,
  - затем вычисляется (evaluate) значение функции
  - и выдается значение.

Пример :

> (+ 2 3)

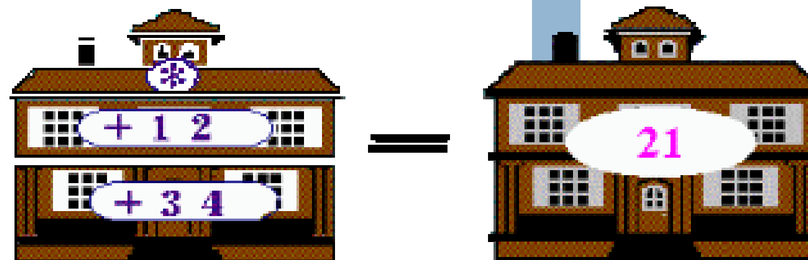
5



- В вводимую функцию могут входить функциональные подвыражения :

> (* (+ 1 2) (+ 3 4))

21



Функции в Лиспе

- Обращение к функции, которое в обычной нотации принято писать ***f(x,y,z),*** имеет вид ***(f x y z).*** Оно имеет вид списка, первый элемент которого атом – имя функции. Следующие элементы – аргументы функции – выражения.

- Каждая функция может накладывать ограничения на число и вид аргументов. Если функция G не требует аргументов, то обращение к ней имеет вид (G).

- **Пример:** Запись (car(quote(a b c))) обозначает обращение к функции car с аргументом (quote(a b c)), который, в свою очередь обозначает обращение к функции quote с одним аргументом – списком (a b c).

Квотирование. Функция quote

- Всякий раз, когда аргументом функции является список, то он рассматривается как (S)-выражение и предполагается, что первый элемент списка – имя функции.
- Исключением из этого правила является функция **quote.** Функция **quote** указывает, что ее аргумент изображает сам себя и его не следует рассматривать как обращение к функции или еще каким-либо способом пытаться оценить его. Сам этот аргумент в том виде, в каком он написан и есть значение функции **quote** при обращении к ней.
- Примеры:
  - (quote a) →a
  - (quote (a b c)) → (a b c)
  - (quote a b c) → ошибка, так как quote функция с одним аргументом.
- Неверно утверждать, что значение функции quote совпадает со значением ее аргумента. Например, значением (quote(car x)) → (car x) является список (car x), а не значение функции car.
- Замечание. В некоторых реализациях вместо quote допускается использование символа ‘ (апостроф).
- Таким образом, обращения к функции

  (car (quote(a b c))) и (car ‘(a b c)) эквивалентны.

Вычисление. Функция eval

- Функция EVAL обеспечивает дополнительный вызов интерпретатора Лиспа.
- При этом вызов может производится внутри вычисляемого S-выражения.
- Функция EVAL позволяет снять блокировку QUOTE.
- Функции quote и eval действуют во взаимно противоположенных направлениях и аннулируют эффект друг друга.
- EVAL - это универсальная функция Лиспа, которая может вычислить любое правильно составленное s-выражение.

- **Примеры:**

```
> ( quote ( + 1 2 ) )
( + 1 2 )
> ( eval ( quote ( + 1 2 ) ) )
3
> ( setq x ' ( a b c ) )
( a b c )
> ' x
x
> x
( a b c )
> ( eval ' x )
( a b c )
```

Можно выделить некоторые общие правила интерпретации S-выражений:

1)Если S-выражение является числом или атомом T или NIL,
то EVAL возвращает это S-выражение без изменений;

2) Если S-выражение является литеральным атомом, то функция
EVAL возвращает последнее значение, которое было присвоено
этому атому, в противном случае возвращается сообщение об ошибке;

3) Если S-выражение представляет собой список вида (f arg1 arg2 … argN),
то функция EVAL пытается интерпретировать его следующим образом:
первый элемент списка интерпретируется как имя функции, которую
необходимо выполнить, взяв в качестве аргументов оставшиеся элементы
списка arg1, arg2,…,argN. В случае удачи функция EVAL возвращает
S-выражение, являющееся результатом выполнения функции f.
Функция f может быть встроенной или определенной пользователем.

Операции присваивания

- Использование символов в качестве переменных
- Изначально символы в Лиспе не имеют значения. Значения имеют только константы.

```
> t
T
> 1.6
1.6
```

- Если попытаться вычислить символ, то система выдает ошибку.
- Значения символов хранятся в ячейках, закрепленных за каждым символом. Если в эту ячейку положить значение, то символ будет связан (bind) со значением. В процедурных языках говорят "будет присвоено значение".
- Для Лиспа есть отличие:
- Не оговаривается, что может хранится в ячейке: целое, атом, список, массив и т.д. В ячейке может хранится что угодно.
- С символом может быть связана не только ячейка со значением, а многие другие ячейки, число которых не ограничено.
- Для связывания символов используется три функции:
  - **SET**
  - **SETQ**
  - **SETF**

Функции SET, SETQ, SETF

- Функция **SET** связывает символ со значением, предварительно вычисляя значения аргументов.
- В качестве значения функция SET возвращает значение второго аргумента.
- Если перед первым аргументом нет апострофа, то значение будет присвоено значению этого аргумента.

- На значение символа можно сослаться записав его без апострофа.

```
> ( set 'd ' ( x y z ) )
( x y z )
```

```
> ( set a ' e )
e
```

```
> ( set ' a ' b )
b
```

```
> a
b
```

- Функция **SETQ** аналогична SET , но не вычисляет значение первого аргумента. Буква q указывает на блокировку.

```
> ( setq m 'k )
k
> m
k
```

- Обобщенная функция **SETF** действует аналогично SETQ , но может использоваться для присвоения символу не только значения.
- В ранних версиях диалектов Lisp функция **SETF** и обобщенные переменные не были доступны. И роль функции присваивания выполняла **SETQ.** Современная реализация использует макрос **SETF** для всех видов присваиваний, и **SETQ** считается атавизмом. Но если внимательно посмотреть на работу отладчика, можно увидеть, что все **SETF** присваивания будут заменены на **SETQ**.

Множественное присваивание

- Предложение let служит для одновременного присваивания значений нескольким переменным. Формат предложения let:

```
(let ((var_1 value_1) (var_2 value_2) … (var_n value_n)) form_1 form_2 … form_m)
```

- Работает предложение следующим образом: переменным var_1, var_2, … var_n присваиваются (параллельно!) значения value_1, value_2, … value_n, а затем вычисляются (последовательно!) значения form_1 form_2 … form_m. В качестве значения всего предложения let возвращается значение последней вычисленной form_m.

```
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
```

- Так как присваивание значений переменным выполняется параллельно, то в следующем примере будет выдано сообщение об ошибке.

```
> (let ((x 3) (y (+ 1 x))) (sqrt (+ (* x x) (* y y))))
```

- После завершения выполнения предложения let переменные var_1, var_2, … var_n получают значения, которые они имели до использования в этом предложении, то есть предложение let выполняет локальные присваивания.

```
> (setq x ‘three)          > (setq y ‘four)
THREE                      FOUR
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
> X                        > y
THREE                      FOUR
```

Определение функции

- Чтобы определить функцию, необходимо:
  - Дать имя функции
  - Определить параметры функции
  - Определить ,что должна делать функция
- Для определения собственной функции можно воспользоваться стандартной функцией **defun** (сокращение от **DE**fine **FUN**ction). Эта стандартная функция позволяет создавать собственные функции, причем не запрещается переопределять стандартные функции, то есть в качестве имени собственной функции использовать имя стандартной функции. Функция defun выглядит следующим образом: (defun name (fp1 fp2 … fpN) (form1 form1 … formN)).
- Name – это имя новой функции, (fp1 fp2 … fpN) – список формальных параметров, а (form1 form1 … formN) – тело функции, то есть последовательность действий, выполняемых при вызове функции.
- **Пример:** функция для вычисления суммы квадратов:

```
> (defun squaresum (x y) (+ (* x x) (* y y)))
```

- Результат работы (пример запуска):

```
> (squaresum 3 4)
25
> (squaresum -2 -4)
20
```

3

Основные понятия

Базовые функции

Базовые функции Лиспа

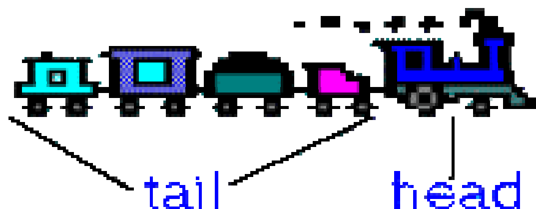
- В классическом Lisp'е существует всего пять основных функций, называемых базовыми:

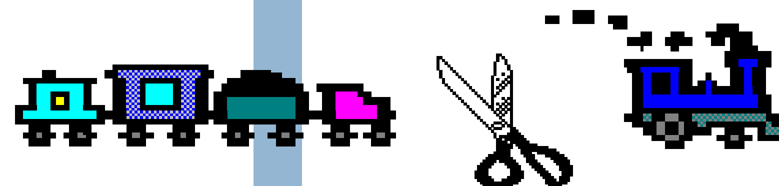
1. **(car list)** – отделяет голову списка (первый элемент списка);
2. **(cdr list)** – отделяет хвост списка (все элементы, кроме первого, представленные в виде списка);
3. **(cons head tail)** – соединяет элемент и список в новый список, где присоединенный элемент становится головой нового списка;
4. **(eq object1 object2)** – проверяет объекты на равенство;
5. **(atom object)** – проверяет, является ли объект атомом.

- Так как функции equal и atom возвращают значения nil или t, их можно назвать базовыми предикатами.

Функция CAR

- Функция (car list) выделяет первый элемент списка list, если список не пустой.
- Car применяется только для списков, т.е. если есть голова списка. Если значение аргумента атом, то значение функции car не определено.
- **Примеры:**





> (car ‘(a b c))

a

> (car ‘((1 2)3))

(1 2)

> (car ((1 2) 3))

ошибка (попытка вычислить выражение)

> (car ‘alpha)

ошибка (аргумент атом, а не список).

Функция CDR

- Функция cdr(list), list -- непустой список, выделяет (возвращает, связывается с) хвост списка (список без первого элемента).
- **Примеры.**

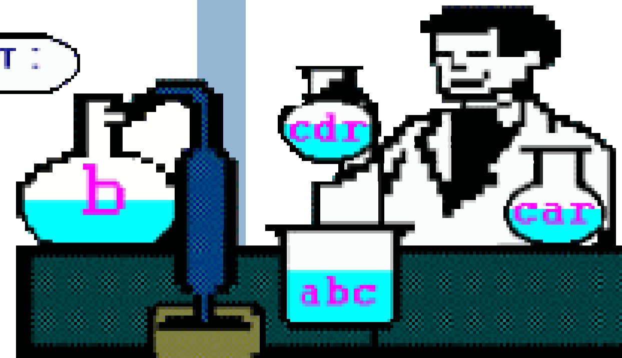
> (cdr '(a b c))	> (cdr '((1 2) 3))	> (cdr '(a))	> (cdr 'alpha)
(b c)	(3)	nil	Ошибка
> (cdr '())	> (cdr nil)		
nil	nil		

- (*) Имена функций CAR и CDR возникли по историческим причинам. Автор Лиспа реализовывал свою первую систему на машине IBM 605. Для хранения адреса головы спискаиспользовался регистр CAR (content of address registr). Для хранения адреса хвоста списка использовался регистр CDR (content of decrement registr)

Комбинация car и cdr

- Рассмотрим (car (cdr ' ((a b) c d)))
- Первым выполняется cdr ,а затем car, т.е. в Лиспе первым выполняются внутренние функции, а затем внешние. Исполнение идет "изнутри наружу".

результат :



- Возможно комбинирование функций car и cdr

> (car (cdr '(a b c))) ⇔ > (cadr '(a b c))

B B

B Для многошагового доступа к отдельным элементам списка удобно пользоваться мне … а таких ком … ия из шаг… таким спос… е, обр…

	Многократные 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)

Функция cons

- Функция CONS соединяет два S-выражения в единое S-выражение, так что первым элементом результата является первый аргумент функции, а хвостом второй аргумент.
  - Она 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.
- **Примеры:**

```
> ( cons ' a ' ( b c ) )
( a b c )

> ( cons ' ( a b ) ' ( c d ) )
( ( a b) c d )
```

- Первый аргумент становится головой второго аргумента, который обязательно является списком.

```
> ( cons ' ( a b ) ' ( ( a b ) ) )
( ( a b ) ( a b ) )

> (cons ( + 1 2 ) ' ( + 3 4 ) )
( 3 + 3 4 )

> (cons ' ( + 1 2 ) ( + 3 4 ) )
Ошибка

> ( cons ' ( + 1 2 ) ' ( + 3 4 ) )
( ( + 1 2 ) + 3 4 )
```

- Примеры манипуляции с пустым списком:

```
> ( cons ' ( a b c ) nil )
( ( a b c ) )

> ( cons nil ' ( a b c ) )
( nil a b c )

> ( cons nil nil )
( nil )
```

Функции eq и equal

- Функция **EQ** выполняет проверку <u>**атомарных**</u> объектов на **равенство**.
- Например, если оба аргумента функция являются списками, то результат не определен, так как анализируется только значение указателя.
- Примеры:

```
> (eq 'a 'a)  ; атомы равны
T
```

```
> (eq '(a) '(a))        ; ложь, т.к. аргументы – не атомы
NIL
```

- Более общим по сравнению с EQ является предикат **EQL**, который дополнительно позволяет сравнивать однотипные большие числа и строки.

```
> (eq 1234567 1234567)

NIL
```

```
> (eql 1234567 1234567)

T
```

- Функция **EQUAL** выполняет проверку <u>**сложных**</u> объектов (списков, строк) на **равенство**.

```
> (equal 'a 'a)           ; два атома
T
> (equal '(a) '(a))       ; два одинаковых списка
T
> (equal 'a '(a))         ; атом и список
NIL
> (equal '"abc de" '"abc de")
T
```

Функция ATOM и др.: проверка типа агумента

- Функция **ATOM** позволяет различать составные и атомарные объекты: на атомах ее зна-чение "истина", а на структурированных объектах — "ложь".
- **Примеры:**

> (atom 'a)	> (atom '(a))	> (atom '())	> (atom '123)
T	NIL	T	T

(*) Аналогично работают функции проверки на соответствие агрумента:

- Символу: (**symbolp** <выражение>)

> (symbolp 3.14)	> (symbolp 'abc)
NIL	T

- Списку: (**listp** <список>)

> (listp '(1 2 3))	> (listp '())
T	T

- Числу: (**numberp** <число>)

> (numberp 3)	> (numberp 3.5)
T	T

- Целому числу: (**integerp** <число>)

> (integerp 3)	> (integerp 3.5)
T	NIL

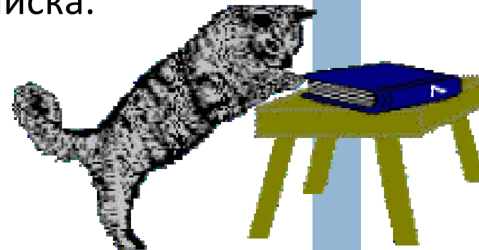
- и др. (полный список см. в справке к Лиспу)

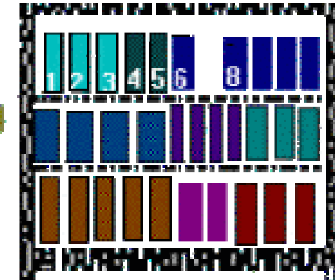
Еще несколько полезных предопределенных функций…

- Функция **NTH** извлекает n-й элемент из списка.
- Форма записи: (NTH < n > <список >)

> (NTH 7 '(1 2 3 4 5 6 7 8 9 10))

7





- Функция **LIST** создает список из s- выражений (списков или атомов).
- Форма записи: (list <выражение 1> … <выражение N>)
  - Число аргументов может быть любое.

> (list 1 2) > (list ' a ' b (+ 1 2)) > (list ' a ' (b c) ' d)

(1 2) (a b 3) (a (b c) d)

- Функция **LENGTH** возвращает в качестве значения длину списка. т.е. число элементов на верхнем уровне

> (length '(1 (2 3) 4)) > (length '((1 (2 3) 4)))

3 1

- Функция **APPEND** объединяет два и более списков в один (аргументами д.б. списки!)

> (append '(a b) nil '(d) '(e))

(A B D E)

Арифметические функции

- Арифметические функции могут быть использованы с целыми или действительными аргументами. Число аргументов для большинства арифметических функций может быть разным.
- **(+ x1 x2 ... xn)** возвращает x1 + x2 + x3 + ... + xn.
- **(- x1 x2 ... xn)** возвращает x1 - x2 - x3 - ... - xn.
- **(* y1 y2 ... yn)** возвращает y1 x y2 * y3 * ... * yn.
- **(/ x1 x2 ... xn)** возвращает x1/x2/... /xn.
- Специальные функции для прибавления и вычитания единицы: **(1+ x)** и **(1- x).**

Математические функции

- Функция логарифм имеет следующий прототип (log arg) и (log arg base)

> (log 2.7)

0.9932518

- Хотя для извлечения корня существует функция SQRT, эту операцию всегда можно выполнить через экспоненту вызывая функции EXP и EXPT.

> (expt 2 1/2)	> (log (expt 2 8) 2)
1.4142135	8

- Вычисление тригонометрических функций:

> (sin 3.14)	> (atan 3.14)
0.00159265	1.26248

Логические операции

- **Сравнение с пустотой** (nil):

> (NULL T)

NIL

- **Отрицание** (по сути то же что null):

> (NOT NIL)

T

- **Логическое "И" (аргументов м.б. 2 и более)**

> (AND T NIL)

NIL

- **Логическое "ИЛИ"**

> (OR T NIL)

T

Арифметические операции сравнения

- Поддерживаются стандартные операции, применимые к числовым вычислениям:

```
=, <, >, <=, >=
> (>= 1 (- 3 2))
T
> (< 1 2)
T
> (= 'a 'a)
error: bad argument type - A
> (= nil '())
error: bad argument type - NIL
```

4

Основные понятия

Условные выражения

Условные выражения

- Функция **cond** (обращение к функции cond наз. условным выражением).
- Общий вид обращения к функции cond:
  - (**cond** (***p1 s1***) (***p2 s2***) ...(***pn sn***)),
  - где ***p1, ..., pn*** – логические выражения,
  - ***s1, ..., sn*** -- произвольные выражения.
- Функция **cond** имеет произвольное число аргументов. Каждый из Аргументов – пара, т.е. список из двух элементов. Первый элемент пары – условие. Второй элемент – выражение.
- Функция **cond** принимает одно из значений ***s1, ... , sn.***
- Выбор происходит следующим образом: просмотр выполняется слева направо до тех пор, пока не встретится аргумент (***pi, si***) со значением ***pi***, отличным от **nil.** Тогда **cond** возвращает значение ***ei.***
- Если все значения ***pi*** равны **nil,** то **cond →nil** (в некоторых реализациях может быть ошибка).
- Последним выражением ***pn*** часто ставят константу **t.** Это условие всегда удовлетв. (аналог default case).

Пример условного выражения

Рассмотрим условное выражение на алголоподобном языке:

```
y :=  if x>50
        then 10
      else if x > 20
        then 30
      else if x > 10
        then 20
      else 0;
```

Его аналогом является условное выражение на Лиспе:

```
(cond   ((> x 50) 10)
        ((> x 20) 30)
        ((> x 10) 20)
        (t 0)
)
```

Пример 2: Абсолютное значение

```
(defun abs(x)
        (cond ((> 0 x) (- 0 x))
              ( t x))
)
```

Значение функции COND определяется следующим образом:
1.Вычисляются последовательно слева направо значения выражений pi до тех пор, пока не встретится выражение, значение которого отлично от NIL, что интерпретируется как ИСТИНА.
2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения функции COND.
3. Если истинного предиката нет, то значением COND будет NIL.
В условном выражении может отсутствовать результирующее выражение pi или на его месте часто может быть последовательность выражений:
(COND (p1 e11)…(pi)…(pk ek2…ekn)…).

Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения COND при истинности предиката выдается само значение предиката. Если же условию соответствует несколько выражений, то при его истинности выражения вычисляются последовательно слева направо и результатом функции COND будет значение последнего выражения последовательности.

Простые циклы

- Для организации циклических действий используется функция **do** след. формата:

```
(do
    ((var_1 value_1) (var_2 value_2) … (var_n value_n))
    (condition form_yes_1 form_yes_2 … form_yes_m)
    form_no_1 form_no_2 … form_yes_k
)
```

- Предложение do работает следующим образом: первоначально переменным var_1, var_2, ..., var_n присваиваются значения value_1, value_2, ..., value_n (параллельно, как в предложении let). Затем проверяется условие выхода из цикла condition.
- Если условие выполняется, последовательно вычисляются формы form_yes_1, form_yes_2, ..., form_yes_m, и значение последней вычисленной формы form_yes_m возвращается в качестве значения всего предложения do. Если же условие condition не выполняется, последовательно вычисляются формы form_no_1, form_no_2, ..., form_yes_k, и вновь выполняется переход в проверке условия выхода из цикла condition.

Пример цикла

Пример использования предложения do:
для возведения x в степень n с помощью умножения определена функция power
с двумя аргументами x и n: x – основание степени, n – показатель степени.

```
> (defun power (x n)
     (do                                 ;присваивание начального значения переменной result
            ((result 1))                 ;условие выхода их цикла
            ((= n 0) result)             ;повторяющиеся действия
            (setq result (* result x)) (setq n (- n 1))
     )
)
POWER

> (power 2 3)
8
```

Основные понятия

Рекурсия

Пример рекурсии

- Функция является рекурсивной, если в ее определении содержится вызов этой же функции. Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.
- Например, задача нахождения значения факториала n! сводится к нахождению значения факториала (n-1)! и умножения найденного значения на n.
- Пример: нахождение значения факториала n!.

```
> (defun factorial (n)
    (cond
        ((= n 0) 1)                          ;факториал 0! равен 1
        (t (* (factorial (- n 1)) n))        ;факториал n! равен (n-1)!*n
    )
)
FACTORIAL
```

Отладка / трассировка

- Для отладки программы можно использовать возможности трассировки. Трассировка позволяет проследить процесс нахождения решения.
- Для того чтобы включить трассировку можно воспользоваться функцией trace. После ввода этой директивы интерпретатор будет распечатывать имя функции и значения аргументов каждого вызова трассируемой функции и полученный результат после окончания вычисления каждого вызова.
- Например, вызовем трассировку определенной выше функции:

```
> (trace factorial)
> (factorial 3)
Entering: FACTORIAL, Argument list: (3)
   Entering: FACTORIAL, Argument list: (2)
         Entering: FACTORIAL, Argument list: (1)
                  Entering: FACTORIAL, Argument list: (0)
                  Exiting: FACTORIAL, Value: 1
         Exiting: FACTORIAL, Value: 1
   Exiting: FACTORIAL, Value: 2
Exiting: FACTORIAL, Value: 6
6
```

- Для отключения трассировки можно воспользоваться функцией untrace:

```
> (untrace factorial)
NIL
```

Виды рекурсии

- Можно говорить о двух видах рекурсии: рекурсии по значению и рекурсии по аргументу. Рекурсия по значению определяется в случае, если рекурсивный вызов является выражением, определяющим результат функции. Рекурсия по аргументу существует в функции, возвращаемое значение которой формирует некоторая нерекурсивная функция, в качестве аргумента которой используется рекурсивный вызов.
- Приведенный выше пример рекурсивной функции вычисления факториала является примером рекурсии по аргументу, так как возвращаемый результат формирует функция умножения, в качестве аргумента которой используется рекурсивный вызов.
  ... (t (* (factorial (- n 1)) n)) ...
- Примером рекурсии по значению м.б. определение принадлежности элемента списку:

```
> (defun member (el list)
    (cond
    ((null list) nil)              ;список просмотрен до конца, элемент не найден
    ((equal el (car list)) t)      ;очередная голова списка равна искомому, элемент найден
    (t (member el (cdr list)))))   ;если элемент не найден, продолжить поиск в хвосте
    списка
MEMBER
> (member 2 '(1 2 3))
T
> (member 22 '(1 2 3))
```

Примеры рекурсий…

Реверс списка (рекурсия по аргументу):

```
> (defun reverse (list)
      (cond
              ((null list) nil)              ;реверс пустого списка дает пустой список
              (t (append (reverse (cdr list)) (cons (car list) nil)))
              ;соединить реверсированный хвост списка и голову списка
      )
)
REVERSE

> (reverse '(one two three))
(THREE TWO ONE)

> (reverse ())
NIL
```

Примеры рекурсий…

Копирование списка (рекурсия по аргументу):

```
> (defun copy_list (list)
      (cond
                  ((null list) nil)                ;копией пустого списка является пустой список
                  (t (cons (car list) (copy_list (cdr list))))
;копией непустого списка является список, полученный из головы и копии
;хвоста исходного списка
      )
)
COPY_LIST

>(copy_list '(1 2 3))
(1 2 3)

>(copy_list ())
NIL
```

Другие выиды рекурсии…

Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

- **параллельная рекурсия** – тело определения функции function_1 содержит вызов некоторой функции function_2, несколько аргументов которой являются рекурсивными вызовами функции function_1.
(defun function_1 … (function_2 … (function_1 …) … (function_1 …) …) …)

- **взаимная рекурсия** – в теле определения функции function_1 вызывается некоторая функции function_2, которая, в свою очередь, содержит вызов функции function_1.
(defun function_1 … (function_2 …) …)
(defun function_2 … (function_1 …) …)

- **рекурсия более высокого порядка** – в теле определения функции аргументом рекурсивного вызова является рекурсивный вызов.
(defun function_1 … (function_1 … (function_1 …) …) …

Параллельная рекурсия

Рассмотрим примеры параллельной рекурсии. В разделе, посвященном простой рекурсии, уже рассматривался пример копирования списка (функция copy_list), но эта функция не выполняет копирования элементов списка в случае, если они являются, в свою очередь также списками. Для записи функции, которая будет копировать список в глубину, придется воспользоваться параллельной рекурсией.

```
> (defun full_copy_list (list)
        (cond
;копией пустого списка является пустой список
                ((null list) nil)
;копией элемента-атома является элемент-атом
                ((atom list) list)
;копией непустого списка является список, полученный из копии головы
;и копии хвоста исходного списка
                (t (cons (full_copy_list (car list)) (full_copy_list (cdr list))))))
FULL_COPY_LIST
> (full_copy_list '(((1) 2) 3))
(((1) 2) 3)
> (full_copy_list ())
NIL
```

Взаимная рекурсия

Пример взаимной рекурсии – реверс списка. Так как рекурсия взаимная, в примере определены две функции: reverse и rearrange. Функция rearrange рекурсивна сама по себе.

```
> (defun reverse (list)
        (cond
                ((atom list) list)
                (t (rearrange list nil)))))
REVERSE

> (defun rearrange (list result)
        (cond
                ((null list) result)
                (t (rearrange (cdr list) (cons (reverse (car list)) result)))))
REARRANGE

> (reverse '(((1 2 3) 4 5) 6 7))
(7 6 (5 4 (3 2 1)))
```

Рекурсия высокого порядка

Пример рекурсии более высокого порядка – второго. Классический пример функции с рекурсией второго порядка – функция Аккермана.
Функция Аккермана определяется следующим образом:

B (0, n) = n+1
B (m, 0) = B (m-1, 0)
B (m, n) = B (m-1, B (m, n-1))
где m>=0 и n>=0.

```
> (defun ackerman
        (cond
                ((= n 0) (+ n 1))
                ((= m 0) (ackerman (- m 1) 1))
                (t (ackerman (- m 1) (ackerman m (- n 1))))))
ACKERMAN

> (ackerman 2 2)
7
> (ackerman 2 3)
9
> (ackerman 3 2)
29
```

Выводы

- Язык Лисп был изначально создан как язык ИИ для символьной обработки данных.
- Базовыми конструкциями в Лиспе являются атомы и списки, в совокупности образующие S-выражения.
- Основной и единственный способ вычислений – вызов функций (в осн. рекурсивный).
- В Лиспе существует множество базовых (встроенных/предопределенных) функций.
  - Есть встроенные функции для организации ветвлений и циклов, а также отладочные функции и др.
- Существуют различные виды рекурсий: вычисляемые по значению, по аргументу, параллельные, взаимные, высших порядков.

Литература

- Л.В. Городняя. Основы функционального программирования. – ИНТУИТ
- М.Н. Морозов.Функциональное программирование. Курс лекций - 1999-2001.
- Ю. В. Новицкая. Основы логического и функционального программирования. Учебное пособие. - Новосибирск, 2006 г.