Очень краткое введение в язык Лисп

Обращение к читателю.

Честно говоря, автор первоначально не планировал излагать в этом руководстве основы Лиспа. Однако, изучив литературу, изданную по Лиспу на русском языке, автор вынужден признать, что она **весьма немногочисленна**, а последняя книга по Лиспу издана почти 20 лет назад. Получается, что читатель, не знакомый с Лиспом, вынужден либо искать библиографические редкости, либо что-то качать из Интернета.

Хорошая документация должна быть самодостаточна; это обстоятельство и послужило причиной написания раздела, разъясняющего основы Лиспа.

Наиболее просто синтаксис Лиспа можно было бы описать с помощью Бэкусовых Нормальных Форм (БНФ), но такое описание слишком лаконично для новичка. Поэтому пришлось пойти на компромисс: вместо Бэкусовых форм основы Лиспа описываются **словами**. При изучении **начальных разделов, описывающих архитектуру языка**, читателю рекомендуется смотреть на язык Лисп, как на**формальную знаковую систему**. Автор полагает, что это - самый простой способ осознанного понимания правила записи выражений Лиспа. После развернутого изложения правил составления выражений Лиспа приводятся сведения о **внутреннем** представлении выражений. Начиная с этого момента формальная знаковая система наполняется **неформальным содержанием**.

Настоящий раздел руководства был написан последним. Это привело к тому, что многие сведения в документации встречаются дважды - в этом разделе и при описании соответствующих функций. Автор надеется, что подобная избыточность не так уж плоха - читатель, знакомый с языком, может пропустить это введение, а читателю-новичку, не до конца принявшему идеологию Лиспа, в процессе чтения описания встроенных функций классического Лиспа будет даваться **идеологические** разъяснения.

В заключение, автор просит извинения у искушенного читателя (если он сюда забредет!) за навязчивое объяснение элементарных вещей...

О г л а в л е н и е

Лисп-машина.

Любой язык программирования предназначен для кодирования команд, которые выполняет компьютер. Результатом выполнения команд является все то, ради чего человек использует вычислительную технику (обработка текста, графика, звук, расчеты и т.д.). Процессор компьютера, как правило, умеет исполнять только элементарные команды. Поэтому команды, написанные человеком, обычно преобразуются (транслируются) в команды процессора. Возможен и другой подход, при котором программа на языке программирования не преобразуется в команды процессора, а поступает на вход программы-исполнителя (интерпретируется).

Именно так работает Лисп.

Будем далее называть программу, исполняющую команды Лиспа, **Лисп-машиной**. В ранних версиях Лиспа взаимодействие с пользователем было построено на принципе "запрос - ответ". В настоящее время Лисп-машина может быть реализована и как диалоговая, и как **пакетная**. Последнее означает, что программа Лисп-машины стартует, считывает команды из какого-либо источника (например, из файла), выполняет эти команды, и завершается. Для изучения языка Лисп важно то, что программа на языке Лисп состоит из команд, которые исполняются Лисп-машиной.

Алфавит языка Лисп.

Алфавит языка Лисп включает в себя заглавные и строчные латинские буквы, цифры и все специальные знаки, которые есть на клавиатуре. Буквы национальных языков традиционно в алфавит не входят, хотя нет никаких особых запретов на этот счет. В частности, в алфавит **HomeLisp** входят все русские строчные и заглавные буквы.

Среди всех символов алфавита выделяются следующие **шесть** символов, которые используются особым образом: это **пробел**, **точка**,**открывающая и закрывающая КРУГЛЫЕ скобки**, **апостроф** и **двойная кавычка**. Остальные символы, в общем, "равноправны".

Атомы.

Из алфавитных символов Лиспа строятся все его конструкции. Простейшей из этих конструкций является **атом**. Атом - это произвольная строка алфавитных символов, за исключением:

- отдельно стоящей точки

- отдельно стоящей левой или правой скобок или групп левых или правых скобок (за исключением открывающей и закрывающей скобки, стоящих подряд)

- отдельно стоящего пробела или группы пробелов

- отдельно стоящего апострофа или двойной кавычки

Строка символов, изображающая атом, не может содержать пробела и круглых скобок, но может содержать точку. Кроме того, имеется ограничение на использование двойной кавычки внутри строки, изображающей атом.

Среди всех мыслимых атомов Лиспа сразу выделим **четыре** специальные группы атомов:

- **Десятичные числа** - это атомы, которые представляют собой корректное изображение десятичного числа (целого или с дробной частью; в качестве разделителя целой и дробной части используется точка).

- **Шестнадцатеричные (битовые) константы** представляются атомами вида: **&Hnnnn**, где nnnn - от одного до восьми символов из набора:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

- **Строки** - это атомы, первый и последний символ которых - двойная кавычка. Между этими кавычками могут располагаться все символы алфавита (включая пробелы и скобки).

- Атомы **Nil** и **T**. Эти атомы (особенно **Nil**) используются для разнообразных целей.

Ниже приводится таблица, иллюстрирующая правила построения атомов Лиспа.

Строка	Что это
Abc	**Это обычный атом**
1Abc	**И это - тоже атом (хотя имя и начинается с цифры)**
Q$W	**И это - тоже атом (хотя имя и содержит знак доллара)**
123	**Это атом - число**
-12.3	**И это атом - число**
6.02E+23	**И это атом - число**
A.A	**Это атом**
A A	**А это - не атом. Пробел в имени недопустим!**

A(	**И это - не атом. Скобки в имени недопустимы!**
A'B	**И это - не атом. Апостроф в имени недопустим!**
()	**Как ни странно, это - атом. Почему, будет ясно из дальнейшего.**
"Проба пера"	**Это - атом-строка. Внутри строки пробелы вполне допустимы**
"Проба "пера""	**Это - не атом. Кавычки, стоящие внутри строки, при записи должны удваиваться.**
"Проба ""пера"""	**Теперь верно.**
"Проба 'пера'"	**Можно и так. Апостроф внутри строки - обычный символ.**
&HFFFFFF	**Это - атом-битовая шкала.**
&H1122334455667788	**Это - просто атом (а не битовая шкала, как могло бы показаться; слишком много цифр)**

Автор надеется, что читатель вполне уяснил правила составления имен атомов.

Точечные пары - "молекулы" Лиспа.

Точечная пара - это конструкция следующего вида: **левая скобка**, ноль или более пробелов, **атом или точечная пара**, один или более пробелов, **точка**, один или более пробелов, **атом или точечная пара**, ноль или более пробелов, **правая скобка**. Другими словами, точечную пару можно представить следующим образом:

(Нечто . Нечто)

Здесь **Нечто** - это атом или точечная пара. Схема построения точечных пар иллюстрируется таблицей, в которой представлены все типы точечных пар, а также приведены ошибочные построения.

Строка	Что это
(a . b)	**Это правильная точечная пара.**
((1 . 2) . b)	**Это тоже правильная точечная пара.**
((1 . 2) . (3 . 4))	**И это тоже правильная точечная пара.**

(x . (y . z))	И это...
(1 .)	А вот это - не точечная пара. После точки до скобки должен стоять атом или точечная пара.
(. 2)	И это - тоже не точечная пара. После скобки до точки должен стоять атом или точечная пара.
(1 . 2	Не точечная пара. Скобки должны быть сбалансированы.
(name . "Анатолий")	Это снова правильная точечная пара.

Следует обратить внимание на то, что образование точечной пары - это **бинарная** операция. Запись вида:

(A . B . C)

бессмысленна (по крайней мере, в HomeLisp). Однако, две следующие записи представляют собой корректные точечные пары:

(A . (B . C))

((A . B) . C)

В первой из приведенных выше точечных пар, пара **(B . C)** является составной частью пары **(A . (B . C))**. Будем говорить, что пара **(B . C) вложена** в пару **(A . (B . C))**.

Введем важное определение: часть точечной пары, расположенную между левой скобкой и точкой будем называть **A-частью** или **A-компонентой**. Соответственно, часть пары, расположенную между точкой и правой скобкой будем называть **D-частью** или **D-компонентой**.

Что можно сказать о конструкции **(1.2)**? Здесь точка не отделена пробелами от окружения, а является частью атома **1.2** . Несмотря на внешнее сходство с точечной парой, эта конструкция **не соответствует** данному выше формальному определению. Мы еще вернемся к этой конструкции при рассмотрении списков, и вскроем ее истинную природу!

S-выражения.

Атом или точечная пара называются **S-выражением**. В "мире Лиспа" нет ничего, кроме **S-выражений**; S-выражениями являются и программы и данные. В памяти

компьютера все конструкции, кроме атомов, хранятся и обрабатываются в виде точечных пар.

Списки.

Точечная пара - универсальный способ построения агрегатов из атомов. Однако, точечная запись не очень удобна для человека: в ней слишком много скобок и точек. Было предложено правило, позволяющее записывать S-выражения **практически без точек** и с использованием **значительно меньшего количества скобок**.

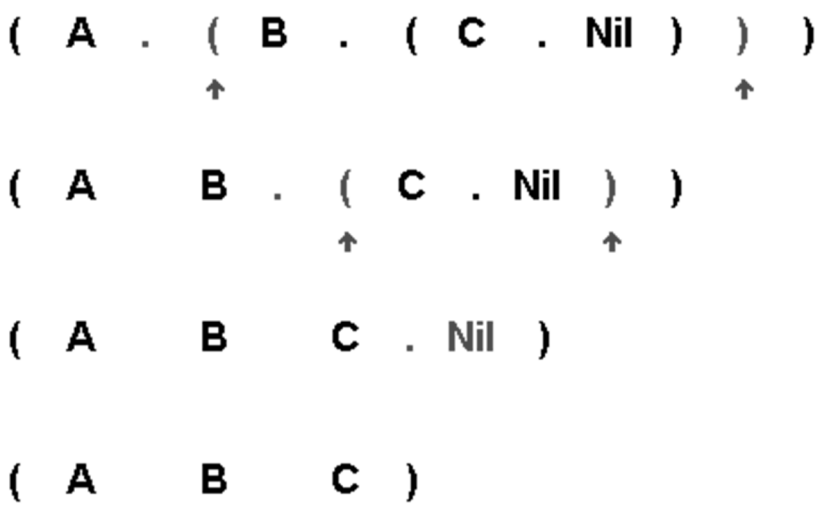
Этих правил всего два:

- Цепочки **. Nil** просто удаляем;
- Цепочки **. (** удаляем вместе с **соответствующей закрывающей скобкой**.

Рассмотрим применение этих правил к записи S-выражения:

(A . (B . (C . Nil)))

На приведенном ниже рисунке показана последовательность упрощений:



Использование описанных правил упрощения привело к тому, что большая часть S-выражений в Лиспе записывается **в чисто скобочной нотации** и называется **списками**.

Можно дать такое определение списка. **Список - это такая точечная пара, в записи которой после применения правил упрощения не остается точек.**

Вот эквивалентное определение. Список - эта точечная пара (состоящая, возможно, из

атомов и других точечных пар), удовлетворяющая условию: **D-частью всех вложенных точечных пар может быть либо точечная пара, либо специальный атом Nil**.

Можно сказать, что **список** - это конструкция следующего вида: **левая скобка**, ноль или более **пробелов**, группа из нуля или более**атомов** или **списков**, разделенная цепочками из одного или более **пробелов**, ноль или более **пробелов** и **правая скобка**.

Это последнее определение обычно и приводится в курсах Лиспа. Оно, разумеется, правильно, но не следует забывать, что все S-выражения хранятся в памяти компьютера в виде точечных пар. Точечная запись "незримо присутствует" при работе Лисп-системы (а иногда и неожиданно проявляется; такой пример будет приведен ниже).

Еще раз следует отметить, что всякий список может быть представлен в точечной записи, но не всякая точечная пара является списком. В ряде случаев правила упрощения, приведенные выше, могут сделать точечную запись даже **менее наглядной**. Вот пример на эту тему: точечная пара **((a . b) . (c . d))** после применения правил упрощения превращается в малонаглядную запись: **((a . b) c . d)**. Исходная запись этой точечной пары нагляднее упрощенной. К счастью, такие конструкции в реальных программах почти не встречаются.

Для представления списка в точечной записи существует достаточно простое правило. В соответствии с последним определением, любой список может быть представлен в виде:

(Нечто-1 Нечто-2 Нечто-3 ...)

где **Нечто** - атом или список, а многоточие означает повторение. Легко убедиться, что эквивалентной точечной формой такого представления будет:

(Нечто-1 . (Нечто-2 . (Нечто-3 . ... . Nil) . Nil) . Nil)

Далее подобному преобразованию следует подвергнуть каждое "Нечто", при условии, что это "Нечто" - список, а не атом. Ниже приводится последовательность преобразований позволяющая получить для списка **((A B) (C D) E F)** эквивалентную точечную форму. При этом красным цветом выделены добавляемые точки и скобки на очередном шаге преобразования.

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

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

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

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

Вот несколько примеров списков в точечной и списковой записи:

Списковая запись	Точечная запись
(A)	(A . Nil)
(A B C)	(A . (B . (C . Nil)))
(A (B C) (E F))	(A . ((B . (C . Nil)) . ((E . (F . Nil)) . Nil)))

Снова вернемся к рассмотрению конструкции **(1.2)**. Она полностью соответствует определению списка из одного элемента - атома 1.2 (и, следовательно, является на самом деле **точечной парой (1.2 . Nil)** !).

В заключение дадим еще **три важных определения.**

- Конструкция **()** соответствует **определению списка**. Такой список называется **пустым списком**. В Лиспе принято считать, что пустой список эквивалентен **атому Nil**.

- Первый элемент списка (он может, в свою очередь, быть атомом или списком) называется **головой** списка.

- Часть списка, за исключением головы называется **хвостом** списка. Если кроме головы список не содержит других элементов, то хвост такого списка есть пустой список.

Внутреннее представление списков.

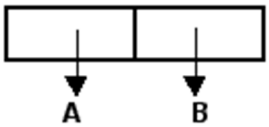
Оперативная память, в которой хранятся S-выражения, обычно делится на две больших области: **список объектов** и **область списочных ячеек**.

В списке **объектов** хранятся **атомы**. Каждый атом занимает блок памяти **переменного** размера. В этом блоке хранится символьное изображение атома и ряд его дополнительных характеристик.

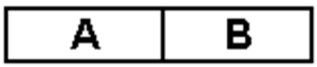
Область списочных ячеек состоит из блоков **фиксированного** размера. Каждая списочная ячейка хранит **два адреса**, которые по историческим причинам называются **A-указатель** и **D-указатель**. Эти адреса могут указывать как на атомы (т.е. хранить адреса областей из списка объектов), так и на другие списочные ячейки.

Для наглядного изображения списков существуют уже устоявшаяся традиция. Списочная ячейка предствляется прямоугольником, разделеным вертикальной линией на две равные части. В левой части хранится A-указатель, в правой - D-указатель. Атомы изображаются символами (буквами или цифрами). Теперь можно сказать, что точечная пара **(A . B)** - это одна списочная ячейка, в A-указателе которой находится адрес атома **A**, а в D-указателе - адрес атома **B**.

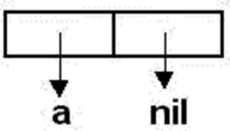
Графически точечную пару изображают так:



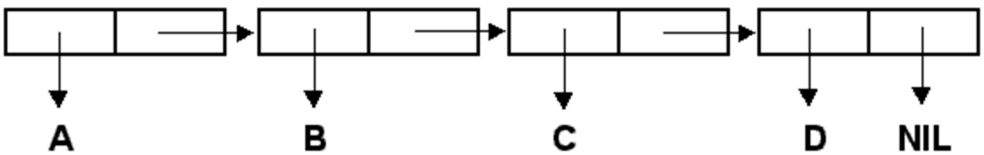
или так:



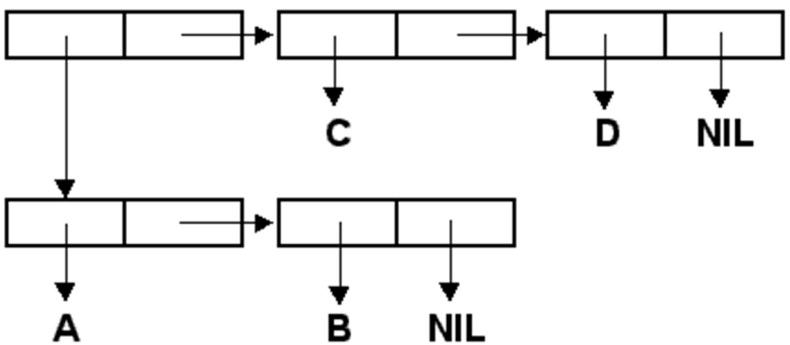
А как можно представить графически список **(A)**? Поскольку список **(A)** есть точечная пара **(A . Nil)**, то графическое изображение строится сразу:



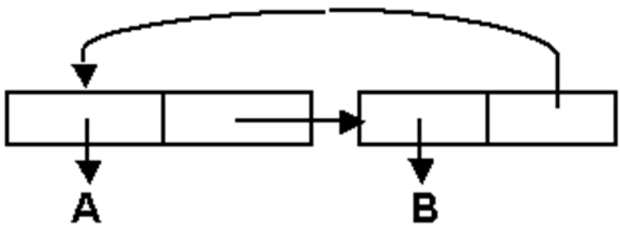
Не представляет труда построить графическое изображение и более сложных списков. Так, например, список **(A B C D)** устроен так:



Полезно соотнести это представление с точечной записью списка **(A B C D)**, которое имеет вид **(A . (B . (C . (D . Nil))))**. Вот еще один пример внутреннего представленния списка более сложной структуры. Список **((A B) (C D))** хранится в памяти в следующем виде:



Автор надеется, что читатель уяснил преимущества графического представления списочных структур. Надо заметить, что с помощью прямоугольников и стрелок можно изобразить S-выражения, которые **невозможно** записать в скобочной нотации. Речь идет о циклических списках. Вот пример такого выражения:



Как видно из рисунка, S-выражение состоит из двух списочных ячеек и двух атомов. В D-указателе **второй** списковой ячейки содержится указатель на **первую** списковую ячейку. Записать это S-выражение в скобочной записи нельзя, но его можно создать (с помощью функций **RPLACA** и **RPLACD**; автор счел нецелесообразным описывать эти функции в элементарном введении в Лисп...)

Говоря о внутреннем представлении S-выражений, следует добавить, что каждый атом **уникален**. Даже если в каком-либо списке атом **А**встречается **многократно**, в списке объектов он представлен **в единственном экземпляре** (т.е. во всех списочных ячейках из которых исходят стрелки, указывающие на атом **А**, будет содержаться один и тот же адрес).

В обращении к читателю было сказано, что синтаксис S-выражений на первых порах проще всего представлять, как формальную знаковую систему. **Начиная с этого момента**, читатель может смотреть на S-выражения **не как на формальные конструкции** из букв и скобок,**а как на реальные структуры данных**, хранящиеся в памяти компьютера.

Взаимодействие с Лисп-машиной - вычисление значений.

Лисп-машина читает входящие команды, имеющие вид S-выражений, **вычисляет значение** каждого из введеных выражений, и выводит результат. Значением S-выражения является, разумеется, тоже S-выражение. (Кроме S-выражений в мире Лиспа ничего нет!) Побочным эффектом вычисления входящих S-выражений является изменение состояния Лисп-машины.

Вычисление значения выполняется по следующим формальным правилам:

- Если входное S-выражение является **атомом** то:

- для атомов **T** и **Nil** их значением является сами атомы **T** и **Nil** соответственно;

- для атомов, представляющих корректное изображение числа, строки или битовой шкалы значением также является сам атом. Такие атомы (**Nil**, **T**, числа, строки и битовые шкалы)

будем далее называть **самоопределенными**.

- все прочие атомы могут иметь значением S-выражение, которое было присвоено атому вызовом функций **SET, SETQ** или **CSETQ**. Если атому не было присвоено значения перечисленными выше функциями, то такой атом **не имеет значения**.

- Если входное S-выражение является **списком**, и **голова списка** представляет собой атом, то этот атом рассматривается как **имя функции**, а оставшаяся часть списка (**хвост списка**) - как список параметров этой функции. Если соответствующая функция существует в системе, а список параметров корректен, то функция вычисляется. **Результат вычисления и есть значение исходного S-выражения.**Будем называть функцию, задаваемую головой S-выражения, **ведущей функцией** S-выражения.

- Если входное S-выражение является **списком**, но **голова списка** представляет собой список, то этот список рассматривается как т.н.**лямбда-выражение**. Лямбда выражение задает безымянную функцию. Хвост исходного S-выражения задает список параметров этой безымянной функции. Разумеется, лямбда-выражение составляется по строгим правилам, которые будут описаны ниже. Если голова списка не является корректным лямбда-выражением, то вычисление завершается ошибкой.

- Все остальные S-выражения значений не имеют. Попытка вычисления таких выражений вызывает ошибку.

В частности, если головой списка является **число**, **строка**, **битовая шкала** или **список** (не являющийся **лямбда-выражением**), то ведущая функция заведомо не существует и не может быть вычислена.

S-выражения, которые имеют значения, называются **формами**.

Вычисление значений функций Лиспа.

Что означает **"вычислить функцию"**? Это означает **по S-выражениям - параметрам получить результирующее S-выражение**. Процесс вычисления зависит от **типа функции**. А функции бывают следующих типов:

- Встроенные в ядро Лиспа функции, реализованные в машинных кодах (или на языке реализации ядра Лиспа). Будем далее называть такие функции функциями типа **SUBR** (от **Subroutine** - подпрограмма);

- Реализованные на языке Лисп. Будем далее называть такие функции функциями типа **EXPR** (от **Expression** - выражение);

Вызовы функций типа **SUBR** и типа **EXPR** не отличаются по форме и выглядят следующим образом:

(Функция Аргумент$_1$ Аргумент$_2$ ... Аргумент$_n$)

Многоточие здесь означает повторение. Каждый из аргументов может быть атомом или списоком. Функция "знает" сколько аргументов ей требуется. Если количество аргументов

оказывается больше или меньше необходимого, возникает ошибка вычисления функции и выдается соответствующее сообщение. Бывают функции с переменным числом аргументов, а бывают и функции без аргументов. Обращение к такой функции выглядит так:

(Функция)

В математике широко применяется конструкция "функция от функции". Например, запись **F(G(x))** означает вычисление **G(x)**, а затем вычисление функции **F**, с аргументом, равным **G(x)**. Как записать эту конструкцию в обозначениях Лиспа? Очевидно, что запись:

(F G X)

абсолютно неверна. Эта запись означает вычисление функции **F** с двумя аргументами: первый - **G** второй **X**! Более логично было бы записать конструкцию **F(G(x))** в виде:

(F (G X))

Эта запись логична хотя бы потому, что здесь у функции **F**, как и положено **один** аргумент - список **(G X)**. Если Лисп-машина, прежде чем вычислять значение функции **F**, сначала вычислит значение функции **G** (т.е. значение выражения **(G X)**) и **заменит** в выражении **(F (G X))**список **(G X)** этим значением, то последующее вычисление функции **F** даст нужный результат.

Именно так Лисп-машина и поступает!

Вычисление выражения общего вида:

(F A_1 A_2 ... A_n)

выполняется следующим образом. Вычисляется **каждый** аргумент. Если аргумент - атом, берется значение атома; если аргумент список - вычисляется соответствующая функция. После чего в исходном списке каждый аргумент заменяется своим значением. На заключительном этапе вычисляется значение функции **F**. Естественно, что в случае, когда какой-либо из аргументов является вызовом функции, то описанный выше процесс применяется к его вычислению точно так же.

Рассмотрим, например, вычисление следующего S-выражения:

(F (G 1 2 (H "q" "p")) (I 12 -1))

Вычисление будет проходить через следующие этапы:

- Вычисляется первый аргумент функции **F** - S-выражение **(G 1 2 (H "q" "P"))**. Это выражение, в свою очередь, является вызовом функции **G** с аргументами **1 2** и **(H "q" "P")**. Значением атома **1** является сам атом **1**, а значением атома **2** является сам атом **2**. А вот значение S-выражения **(H "q" "p")** нужно вычислять.

S-выражение **(H "q" "p")** является вызовом функции **H** с двумя аргументами **"q"** и **"p"**. Значением атома **"q"** является сам атом **"q"**, а значением атома **"p"** является сам атом **"p"**. Далее вычисляется значение функции **H** с двумя аргументами **"p"** и **"q"**. Предположим, что это значение равно S-выражению **Рез_H**. Таким образом, первый аргумент исходного вызова функции **F** принимает вид **(G 1 2 Рез_H)**. Это выражение вычисляется, предположим, что результат вычисления равен **Рез_G**.

Далее исходное выражение приобретает вид **(F Рез_G (I 12 -1))**. Первый аргумент вызова функции **F** вычислен. Вычисляется второй аргумент. Значение второго аргумента равно значению функции **I** с двумя аргументами: **1** и **2**. Предположим, что это значение равно**Рез_I**.

Теперь исходное выражение приобрело вид: **(F Рез_G Рез_I)**. Вычисляется значение функции **F** с заданными аргументами. Это значение и есть значение исходного S-выражения **(F (G 1 2 Рез_H) (I 12 -1))**

Проблема вычисляемых аргументов. Классификация функций Лиспа.

Но как быть в том случае, когда некоторой функции требуется не **значения аргументов**, а сами аргументы? Предположим, есть функция, возвращающая сумму элементов числового списка. Пусть имя этой функции - **SUMLIST**. Тогда вызов **(SUMLIST (1 2 3 4 5))** должен вернуть атом **15**. Однако, в соответствии с написанным выше, Лисп сделает попытку вычислить значение списка **(1 2 3 4 5)**, что, в свою очередь, повлечет за собой попытку вычисления значения функции **1** со списком аргументов **(2 3 4 5)**. Это, естественно, вызовет ошибку.

Получается, что никакой функции Лиспа **принципиально** нельзя передать параметр вида **(1 2 3 4 5)**?

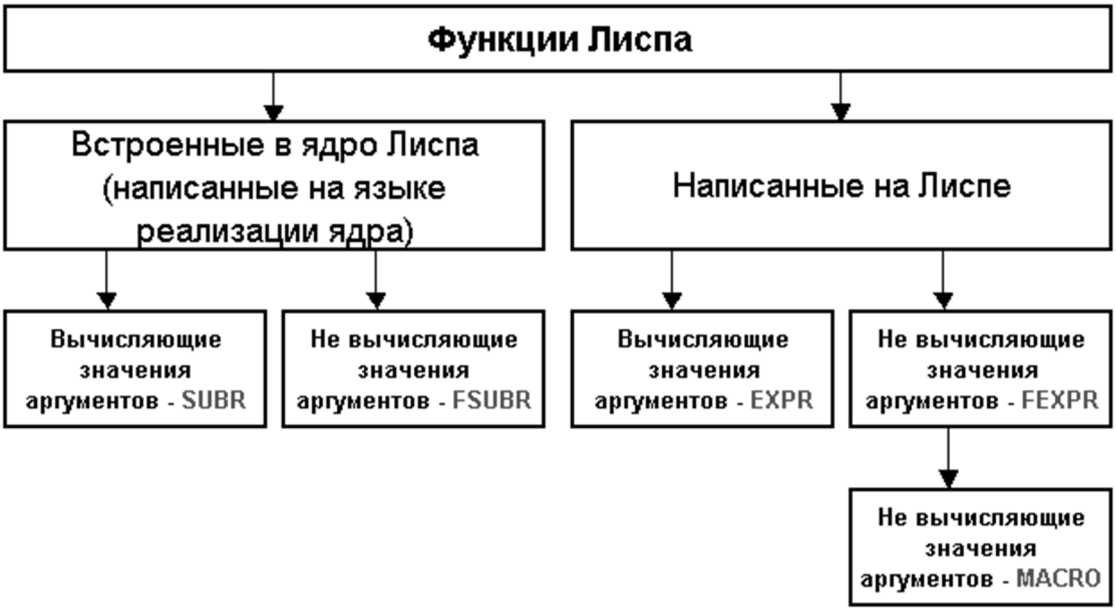
Разумеется, это не так. Отмеченая проблема решается в Лиспе двумя способами: введением специального класса функций, которые **не вычисляют** свои аргументы, а также выборочным использованием функции **QUOTE**, о чем будет сказано ниже.

В Лиспе есть класс встроенных функций, которые не вычислют значения своих аргументов, а используют сами аргументы. Встроенные функции, **вычисляющие значения аргументов**, называются функциями класса **SUBR**, а встроенные функции, **НЕ вычисляющие значения некоторых или всех аргументов**, называются функциями класса **FSUBR**.

Соответственно, функции, написанные на Лиспе, тоже могут не вычислять значения своих аргументов. Функции, написанные на Лиспе и**вычисляющие значения аргументов**, называются функциями класса **EXPR**, а функции, написанные на Лиспе и **НЕ вычисляющие значения некоторых или всех аргументов**, называются функциями класса **FEXPR**.

Чтобы классификация функций Лиспа стала полностью завершенной, следует добавить, что существует еще один класс функций - **MACRO**. Эти функции ближе всего к функциям **FEXPR**, - они тоже не вычисляют значения своих аргументов. Однако вычисление функций типа**MACRO** имеет особенность, которая позволяет выделить эти функции в отдельный класс (он будет рассмотрен ниже).

В целом, классификация функций Лиспа может быть изображена следующим рисунком:



Диалог с Лисп-машиной.

Большинство Лисп-машин работают по "телетайпному" принципу: пользователь вводит S-выражение, Лисп-вычисляет и выводит ответ (тоже, естественно, S-выражение). Простейшим примером такого диалога может служить консольное окно Windows (или UNIX/LINUX). В**HomeLisp** пользователь вводит S-выражение в область ввода и получает ответ в области вывода. Подробности диалога описаны в здесь. Введенное выражение дублируется в области вывода, поэтому область вывода содержит полный протокол диалога с пользователем.

Важно отметить следующее: если вычисление введенного завершено с ошибкой, то **HomeLisp** возвращает в качестве результата**специальный атом ERRSTATE** (ведь результат должен быть тоже S-выражением!). Если вычисления результата заняли слишком много времени, происходит принудительная остановка Лисп-машины и в качестве результата возвращается **специальный атом BRKSTATE**

Кроме того, некоторые функции могут что-то выводить в область вывода. Эта информация выводится **перед** результирующим S-выражением.

Ниже приводится пример взаимодействия с Лисп-машиной. Ответ Лисп-машины предваряется набором символов **"==>"**.

```
_ver

==> "HomeLisp Вер. 1.11.1 (Файфель Б.Л.)"
```

Здесь пользователь ввел имя атома **_ver** (версия ядра) и получил ответ.

Блокировка вычисления - функция QUOTE.

Функция **QUOTE** принимает ровно один аргумент и возвращает S-выражение, совпадающее с аргументом. Основное назначение функции**QUOTE** - выборочная блокировка вычисления аргументов при вызове функций класса **SUBR** и **EXPR**.

Понятно, что сама функция **QUOTE** должна принадлежать к классу **FSUBR**, - она не вычисляет значение своего аргумента, а возвращает сам аргумент.

Выше был приведен пример функции **SUMLIST**, вычисляющей сумму элементов списка, заданного единственным аргументом. Если функция**SUMLIST** принадлежит к классу **EXPR** или **SUBR**, то попытка вычислить **(SUMLIST (1 2 3 4 5))** вызовет ошибку. Ведь сначала будет предпринята попытка вычислить **значение** списка **(1 2 3 4 5)**, а это S-выражение не имеет значения. А вот вызов **(SUMLIST (QUOTE (1 2 3 4 5)))** не вызовет ошибки, ведь сначала будет вычислено значение **(QUOTE (1 2 3 4 5))**, результат вычисления равен **(1 2 3 4 5)**. Этот результат **без повторного вычисления** будет передан на вход функции **SUMLIST**, которая вычислит сумму.

Ниже приводится пример использования функции **QUOTE**. Предполагается, что функция **SUMLIST** имеется в системе, принадлежит к классу**EXPR** или **SUBR** (т.е. вычисляет свои аргументы).

```
(sumlist (1 2 3))

Не найдена функция 1
==> ERRSTATE

(sumlist (quote (1 2 3)))

==> 6
```

По уже укоренившейся традиции вместо того, чтобы писать:

(QUOTE Нечто)

допустимо писать:

'Нечто

Далее в этом разделе будет употребляться именно такая запись. Следует обратить внимание на то, что скобки перед апострофом не ставятся.

На жаргоне лисперов, употребление апострофа называется **квотированием**. Квотировать выражение - значит поставить перед ним апостроф. Понятно, что квотировать аргументы функций класса **SUBR/EXPR** необходимо только в случае, когда требуется блокировать вычисление значения. Когда же аргументом является самоопределенный атом, то квотирование не требуется (хотя и не приведет к ошибке, а только чуть удлинит вычисления).

Присвоение значений атомам. Функции SET, SETQ и CSETQ.

Выше было отмечено, что произвольному атому Лиспа можно присвоить значение. Сейчас будут рассмотрены функции, которые выполняют эти действия.

Функция **SET** класса **SUBR** требует ровно два аргумента. Значением первого аргумента должен быть атом, а значением второго - произвольное S-выражение.
Функция **присваивает** атому, являющемуся значением первого аргумента, значение второго аргумента. Это значение функция одновременно возвращает в качестве результата. Будем называть атом, которому присваивается значение, **целевым атомом**. Если до вызова функции **SET** целевой атом уже имел значение, то это значение будет заменено новым.

Рассмотрим несколько примеров использования функции **SET**:

```
(set a 1)

Символ a не имеет значения (не связан).
==> ERRSTATE

(set 'a 1)

==> 1

a

==> 1

(set 'a a)

==> 1

a

==> 1

(set 'a 'a)

==> a
```

Попытка присвоить атому **a** значение **1** оказалась неудачной, поскольку при вычислении первого аргумента была выполнена попытка вычисления значения атома **a**. Чтобы предотвратить ошибку, нужно квотировать первый аргумент. Вторая попытка оказывается удачной. Теперь атом **a** получил значение **1**, в чем можно убедиться, просто послав на вход Лисп-машины атом **a**.

Следующая команда **(set 'a a)** должна была бы присвоить атому **a** значение **a**. Однако, этого не происходит. Причина заключается в том, что, поскольку втрой аргумент

функции **SET** не квотирован, произойдет его вычисление. Атом **a** имеет значение **1**, поэтому фактически будет выполнена команда **(SET 'a 1)**. Значение атома **a** не изменится.

А вот команда **(set 'a 'a)** присваивает атому **a** значение **a**. ЧуднО, но логично!

Функция **SETQ** (класса **FSUBR**) отличается от **SET** тем, что не вычисляет значение первого аргумента. Поэтому первый аргумент при вызове**SETQ** не нужно квотировать. В остальном функция **SETQ** эквивалентна функции **SET**.

Атом, которому присвоено значение вызовом функций **SET/SETQ** обычно называется **переменной**.

Функция **CSETQ** (класса **FSUBR**) отличается от **SETQ** тем, что целевой атом после возврата получает значение, которое нельзя изменить (превращается в константу). Попытка изменить значение константы с помощью команд **SET/SETQ/CSETQ** вызывает ошибку. Соответственно, если атом является переменой, превратить его в константу уже не удастся. Рассмотрим примеры:

```
(csetq ZZ 111)

==> 111

(setq ZZ 9)

Попытка превратить константу ZZ в переменную
==> ERRSTATE

(csetq ZZ 222)

Csetq - попытка изменить значение константы
==> ERRSTATE

(setq xx 9)

==> 9

(csetq xx -11)

Csetq - попытка превратить переменную в константу
==> ERRSTATE
```

Здесь успешно создана константа **ZZ**. Попытка изменить ее значение или превратить в переменную, вызывает ошибку.

Разбор списков на составные части. Функции CAR, CDR и их комбинации.

Функции **CAR** и **CDR** служат для выделения **головы** и **хвоста** списка соответственно. Обе эти функции принадлежат к классу **SUBR**, т.е. они вычисляют значение своего единственного аргумента. Несколько странные названия этих функций обусловлены историческими причинами (функции названы в честь регистров компьютера IBM-709, на котором была выполнена первая реализация Лиспа). В свременных языках, допускающих обработку списков, эти функции называются **HEAD** и **TAIL**. Лисп, однако, остается верен традициям...

Напомним, что **головой** списка называется его **первый** элемент (голова может быть атомом или списком). **Хвостом** списка называется**остаток списка без первого элемента**. Хвост списка всегда является списком - даже если голова списка является единственным значимым элеменом списка, хвост такого списка **есть пустой список** или, что равнозначно, - атом **NIL**.

Функции **CAR** и **CDR** можно применить и к точечной паре, не являющейся списком. В этом случае, **CAR** вернет **A-компоненту** значения аргумента, а **CDR** - соответственно **D-компоненту**.

Попытка применить функции **CAR** и **CDR** к атому вызовут состояние ошибки.

Вот исчерпывающий набор примеров вызова функций **CAR** и **CDR**:

```
(car '(1 2 3))

==> 1

(cdr '(1 2 3))

==> (2 3)

(cdr (cdr '(1 2 3)))

==> (3)

(cdr (cdr (cdr '(1 2 3))))

==> NIL

(car '(a . b))

==> a

(cdr '(a . b))

==> b

(car 1)

Аргумент CAR - атом (1)
```

```
==> ERRSTATE

(cdr 6)

Аргумент CDR - атом (6)
==> ERRSTATE
```

Чтобы выделить второй, третий и т.д. элементы списка, можно применять комбинации **CAR** и **CDR**. Так например, для того, чтобы получить второй элемент списка **S**, нужно вычислить форму **(CAR (CDR S))**. В связи с тем, что различные комбинации вызовов **CAR** и **CDR**встречаются в реальных программах достаточно часто, в Лисп обычно вноятся их стандартные комбинации, приведенные ниже.

Вызов	Результат
(CDAR '((1 2) (3 4)))	(2)
(CAAR '((1 2) (3 4)))	1
(CADR '((1 2) (3 4)))	(3 4)
(CDDR '((1 2) (3 4)))	Nil
(CDDDR '(1 2 3 4))	(4)
(CADAR '((1 2) (3 4) (5 6)))	2
(CADDR '((1 2) (3 4) (5 6)))	(5 6)
(CADDDR '(1 2 3 4 5 6))	4

Построение списков из составных частей. Функции CONS и LIST.

Функция **CONS**, принадлежащая к классу **SUBR**, объединяет значения двух своих аргументов в точечную пару. Если значением первого аргумента является атом, а второго - список, то результатом функции **CONS** будет **список**, голова которого есть значение первого аргумента, а хвост - значение второго.

Распространенной ошибкой начинающих является ожидание, что результатом вызова **(CONS 'a 'b)** будет список **(a b)**. На самом деле результатом будет точечная пара **(a . b)**. Чтобы в результате вызова получился список, нужно вызвать функцию **CONS** так: **(CONS 'a '(b))**. Ясно также, что вызов **(CONS 'a Nil)** вернет точечную пару **(a . Nil)**, что, в соответствии с правилами упрощения, есть список из одного элемента **(a)**.

Вот реальные примеры вызова **CONS**:

```
(cons 'a 'b)

==> (a . b)

(cons 'a '(b))

==> (a b)
```

```
(cons 'a Nil)

==> (a)

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

==> ((a b) c d)
```

В последнем случае можно было ожидать, что должен был бы получиться список **(a b c d)**, но это не так! Чтобы понять, почему так происходит, достаточно представить оба аргумента **CONS** в виде точечных пар, построить результат (тоже в виде пары), и применить к нему правила упрощения.

Список **(a b)** в точечной нотации имеет вид **(a . (b . nil))**. Соответственно, список **(c d)** в точечной записи представляется как **(c . (d . nil))**. Таким образом, результат **(cons '(a b) '(c d))** будет следующим:

((a . (b . nil)) . (c . (d . nil)))

Цепочка упрощений показана ниже (удаляемые элементы выделены красным):

((a . (b . nil)) . (c . (d . nil)))

((a b . nil) . (c . (d . nil)))

((a b) . (c . (d . nil)))

((a b) c . (d . nil))

((a b) c d . nil)

((a b) c d)

Для получения из двух списков **(a b)** и **(c d)** списка **(a b c d)** служит функция **APPEND**, описываемая ниже.

В отличие от функции **CONS**, функция **LIST** принимает **произвольное число аргументов**. Эта функция принадлежит классу **SUBR** (ее аргументы вычисляются). Функция возвращает список, состоящий из **значений** аргументов.

На первый, поверхностный взгляд, функция **LIST** бесполезна, - ведь чтобы построить, например список из чисел **1**, **2** и **3** достаточно написать **'(1 2 3)** (или, что то-же самое, **(QUOTE (1 2 3))**). Однако, если созданы переменные **z1**, **z2** и **z3** со значениями **1**, **2** и **3**соответственно, то вызов **'(z1 z2 z3)** вернет результат **(z1 z2 z3)** (не произойдет замены атомов значениями). А вот вызов **(LIST z1 z2 z3)** вернет результат **(1 2 3)**. Вот развернутая иллюстрация всему сказанному выше:

```
(setq z1 1)

==> 1

(setq z2 2)

==> 2

(setq z3 3)

==> 3

'(1 2 3)

==> (1 2 3)

(list 1 2 3)

==> (1 2 3)

'(z1 z2 z3)

==> (z1 z2 z3)

(list z1 z2 z3)

==> (1 2 3)
```

Таким образом, **LIST** представляет собой полезнейшую функцию Лиспа: т.к. она позволяет "собирать" из составных частей списки**произвольной** длины. Если какой-либо из аргументов этой функции не требуется вычислять, его можно квотировать.

Проверка на "атомность". Функция ATOM.

Сколь-нибудь содержательная программа на любом языке программирования не может состоять только из "линейных" действий (типа присвоения, композиции сложного из простого и разложения сложного на простые составляющие). Необходимы **средства сравнения и принятия решений**. Лисп не составляет исключения. Ниже будут рассмотрены все необходимые конструкции Лиспа, служащие для сравнения S-выражений.

По-видимому, самый простой вопрос, который можнет относиться к произвольному S-выражению, это вопрос: "Является ли S-выражение**атомом** или нет?". На этот вопрос отвечает функция **ATOM** (класса **SUBR**). Эта функция возвращает атом **T**, если значение ее единственного аргумента есть атом, и **Nil** в противном случае.

Вот примеры вызова функции **ATOM**:

```
(atom 1)
```

```
==> T

(atom '(1 2 3))

==> NIL

(atom (car '(1 2 3)))

==> T

(atom (cdr '(1 2 3)))

==> NIL
```

Первый и второй примеры понятны без комментариев. Некоторое удивление может вызвать третий пример, ведь аргументом функции**ATOM** является **список (car '(1 2 3))**, тем не менее, функция возвращает **T**. Все дело в том, что функции **ATOM** на вход попадает не сам аргумент, а его **значение** (функция принадлежит классу **SUBR**!). Значением же S-выражения **(car '(1 2 3))** является атом **1**. Поэтому функция **ATOM** "увидит" на входе не **(car '(1 2 3))**, а простую единицу. Естественно, результат вычисления будет **T** (единица есть атом).

Сравнение атомов. Функции EQ, NEQ, NOT и NULL.

Функция **EQ**, относящаяся к классу **SUBR** принимает два аргумента. Она работает следующим образом:

- Если значением первого и второго аргумента является **один и тот же** атом, то функция возвращает в качестве результата **атом T**;
- Во всех остальных случаях функция возвращает **атом Nil**.

Все остальные случаи включают ситуации, когда значение одного или обоих аргументов не есть атом. Функция **EQ** вернет **Nil** даже в случае, когда значением обоих аргументов является **одно и то же S-выражение**, но не атом!

Функция **EQ** позволяет корректно сравнивать **только атомы**; для сравнения S-выражений (списков) служит другая функция - **EQUAL**.

Вот примеры вызова функции **EQ**:

```
(eq 'a 'b)

==> NIL

(eq 'a 'a)

==> T
```

```
(eq '(a b) '(a b))

==> NIL

(eq Nil Nil)

==> T

(eq T T)

==> T
```

При вызове функции атомы **Nil** и **T** не квотируются, поскольку, как было отмечено выше, эти атомы являются самоопределенными.

Символы **EQ** при вызове одноименной функции можно заменить знаком равенства (**=**). Таким образом, запись **(EQ a b)** полностью эквивалентна записи **(= a b)**.

Функция **NEQ**, относящаяся к классу **SUBR**, принимает два аргумента. Она выполняет действия, **в точности противоположные функции EQ:**

- Если значением первого и второго аргумента является **один и тот же** атом, то функция возвращает в качестве результата **атом Nil**;
- Во всех остальных случаях функция возвращает **атом T**.

Все остальные случаи, как и для функции **EQ**, включают ситуации, когда значение одного или обоих аргументов не есть атом. Следует обратить внимание на то, что функция вернет **T** даже в случае, когда значением обоих аргументов функции является **одно и то же S-выражение**, но не атом.

Вместо символов **NEQ** можно писать знак неравенства **<>**. Вот несколько примеров вызова функции **NEQ**:

```
(neq 'a 'b)

==> T

(neq 'a 'a)

==> Nil

(eq '(a b) '(a b))

==> T

(neq Nil Nil)

==> Nil
```

```
(neq T T)

==> Nil
```

Из рассмотрения функций **EQ** и **NEQ** следует, что в Лиспе атомы **Nil** и **T** используются как логические индикаторы: атом **T** обозначает логическую **истину**; атом **Nil** - **ложь**.

Функции **NULL** и **NOT** (класса **SUBR**) представляют собой в сущности, одну и ту же функцию. Действие, выполняемое этой функцией, очень простое. Если значение единственного аргумента функции есть **Nil**, то функция возвращает **T**. Во всех остальных случаях (когда значение аргумента НЕ есть **Nil**, функция возвращает **Nil**. Примеры вызова:

```
(not T)

==> NIL

(not Nil)

==> T

(Null Nil)

==> T

(Null 'a)

==> NIL

(Null '(1 2 3))

==> NIL

(Not nil)

==> T
```

Теперь все готово к тому, чтобы рассмотреть главную условную конструкцию Лиспа, чему и посвящен следующий раздел.

Функция COND.

Функция **COND**, принадлежащая к классу **FSUBR**, принимает произвольное количество аргументов. Каждый аргумент функции **COND** должен быть списком ровно из двух элементов. Первый из этих элементов будем называть **условием**, а второй - **результатом**. Таким образом, общий вид вызова **COND** таков:

(COND (Условие$_1$ Результат$_1$) (Условие$_2$ Результат$_2$) ... (Условие$_n$ Результат$_n$))

В свою очередь, каждая из конструкций **Условие$_i$** и **Результат$_i$** могут быть произвольными S-выражениями (обычно - списками или атомами).

Функция вычисляет свое значение следующим образом:

- Вычисляется значение выражения **Условие$_1$**;
- Если это значение есть атом **T**, то вычисляется значение выражения **Результат$_1$** и это значение **возвращается** в качестве результата функции **COND**;
- Если это значение НЕ есть атом **T**, то вычисляется значение выражения **Условие$_2$**;
- Если это значение есть атом **T**, то вычисляется значение выражения **Результат$_2$** и это значение **возвращается** в качестве результата функции **COND**;
- Процесс повторяется до тех пор, пока список аргументов будет исчерпан, либо пока значение одного из условий не окажется атомом **T**.

Если не одно из условий не дает в результате вычисления атом **T**, то функция **COND** вернет атом **Nil**. (Так реализована функция **COND** в**HomeLisp**; в других версиях Лиспа реализация может несколько отличаться. Возможна, например, не сравнения результата условия с атомом **T**, а **несравнение** с атомом **Nil**.)

Легко видеть, что конструкция:

(COND (Условие$_1$ Результат$_1$) (Условие$_2$ Результат$_2$) ... (Условие$_n$ Результат$_n$))

очень похожа на конструкцию оператора **IF** языка **Visual Basic**:

IF Условие$_1$ Then Результат$_1$ ELSEIF Условие$_2$ Результат$_2$ ... ELSEIF Условие$_n$ Результат$_n$ END IF

Впрочем, знакомые с языком **Visual Basic** могут заметить, что в **Visual Basic** перед завершающим **END IF** может стоять конструкция **ELSE Результат**, которая получит управление, если все условия оказались ложными. Для этих целей в функции **COND** обычно в качестве**последнего** условия ставят атом **T**. В этом случае последнее условие оказывается **гарантированно** выполненым.

Рассмотрим примеры вызова **COND** (пока, весьма элементарые):

```
(setq z1 1)

==> 1

(cond ((atom z1) "z1 - атом") (T "z1 - не атом"))

==> "z1 - атом"
```

```
(setq z1 '(1 2))

==> (1 2)

(cond ((atom z1) "z1 - атом") (T "z1 - не атом"))

==> "z1 - не атом"
```

Здесь сначала атому **z1** присваивается значение **1**. При последующем вычислении **COND** первое условие **(atom z1)** оказывается истиным. Первый результат представляет собой строку **"z1 - атом"**. Строка есть **самоопределенный** атом - значение такого атома совпадает с ним самим. Поэтому, строка **"z1 - атом"** выдается в качестве значения функции **COND**. Затем атому **z1** присваивается значение **(1 2)**(список). Условие **(atom z1)** оказывается ложным (значение z1 есть список). Поэтому вычисляется второе условие (атом **T**). Оно всегда истино, - выдается результат **"z1 - не атом"**, что соответствует действительности.

Арифметические функции Лиспа.

Лисп умеет выполнять все привычные действия с числами: сложение, вычитание, умножение, дедение, возведение в степень, сравнения и т.д. Кроме того в **HomeLisp** встроена небольшая библиотека математических функций, подробно описанная здесь. Все арифметические функции принадлежат к классу **SUBR** - они встроены в ядро Лиспа и вычисляют значения всех своих аргументов.

В приведенной ниже таблице представлены элементарные арифметические функции **HomeLisp**.

Имя функции	Сокращение	К-во аргументов	Результат
PLUS	+	переменное	сумма значений
DIFFERENCE	-	переменное	значение первого минус сумма значений остальных
TIMES	*	переменное	произведение значений
QUOTIENT	\	2	целочисленное частное
REMAINDER	%	2	целочисленный остаток
DIVIDE	/	переменное	значение первого аргумента делится на произведение значений оставшихся (с плавающей точкой)
EXPT	^	2	значение первого аргумента, возведенное в степень, равную значению второго.
GREATERP	>	2	T если значение первого аргумента больше значения второго; Nil в противном случае
GREQP	>=	2	T если значение первого аргумента больше или равно значению второго; Nil в противном случае
LESSP	<	2	T если значение первого аргумента меньше значения второго; Nil в противном случае

LEEQP	**<=**	**2**	**T если значение первого аргумента меньше или равно значению второго; Nil в противном случае**
EQ	**=**	**2**	**T если значение первого аргумента равно значению второго; Nil в противном случае**
NEQ	**<>**	**2**	**T если значение первого аргумента не равно значению второго; Nil в противном случае**

Вот примеры использования всех перечисленых функций:

```
(+ 1 2 3 4)

==> 10

(- 1 2)

==> -1

(* 1 2 3)

==> 6

(/ 1 2)

==> 0.5

(\ 1 2)

==> 0

(% 1 2)

==> 1

(> 1 2)

==> NIL

(< 1 2)

==> T

(<= 1 1)

==> T

(>= 1 1)

==> T

(> 1 1)

==> NIL
```

```
(^ 5 3)

==> 125
```

Детали, относящиеся к каждой из арифметических операций, более подробно изложены в разделе, посвященным **встроенным функциям Лиспа**.

Универсальная функция EVAL.

Функция **EVAL**, относящаяся к классу **SUBR**, вычисляет значение своего единственного аргумента и возвращает его в качестве результата. Ядро Лиспа, в сущности, работает следующим образом:

1. Ожидает ввода S-выражения;

2. Передает введенное S-выражение функции **EVAL**;

3. Выводит полученный результат;

4. Переходит к п.1

Функцию **EVAL** можно употреблять, например, при необходимости вычислить **значение** S-выражения, построенного **динамически** (в процессе вычислений). Другое применение функции **EVAL** состоит в вычислении (при необходимости) значений аргументов функции класса**FEXPR** в теле самой функции.

Функция **EVAL** в известном смысле противоположна по своему действию функции **QUOTE** - для любого S-выражения **x** результат вызова**(EVAL (QUOTE x))** совпадает с **x**.

Создание собственных функций. Функция DEFUN.

Язык программирования, который не позволяет программисту создавать собственые функции, бесполезен. В Лиспе пользователь может создавать свои функции трех классов - **EXPR**, **FEXPR** и **MACRO**. В этом разделе будет рассмотрено создание функций класса **EXPR**.

Для создания функции класса **EXPR** служит специальная встроенная функция **DEFUN** (определить функцию). Сама функция **DEFUN**принадлежит классу **FSUBR**. Вызов этой функции требует **трех** аргументов:

- Первый аргумент должен атомом;
- Второй аргумент должен быть списком атомов;
- Третий аргумент может быть произвольной формой (S-выражением, имеющим значение).

Атом, который задан первым аргументом функции **DEFUN**, после возврата "превратится в функцию". Второй аргумент функции **DEFUN**называется **списком формальных параметров**. Третий аргумент функции **DEFUN** называется **телом функции** или **определяющим выражением**.

Если вызов функции **DEFUN** завершился удачно, то в качестве результата возвращается атом, заданный первым параметром, а в системе появляется новая функция, требующая при вызове столько аргументов, сколько элементов содержалось в списке формальных параметров. При вычислении этой новой функции значения аргументов будут вычисляться.

Итак, чтобы создать функцию, необходимо обратиться к порождающей функции **DEFUN**:

(DEFUN имя_функции (список_параметров) (тело_функции))

Построим простейшую арифметическую функцию, которая вычисляет разность квадратов своих аргументов. Для имени функции выберем атом **QDIF** (полагая, что такой функции еще нет). Список параметров нашей функции будет содержать два формальных параметра: **(x y)**. Остается написать тело функции (S-выражение, вычисляющее разность квадратов значений **x** и **y**:

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

Собирая все всместе, получим:

(DEFUN QDIF (x y) (* (- x y) (+ x y)))

Пусть читатель обратит внимание на то, что тело функции содержит атомы, входящие в список формальных параметров. Можно создать функцию, тело которой не обращается к формальным параметрам. Однако, такая функция при любых параметрах давала бы один и тот же результат...

Как работает функция? При вызове:

(QDIF Арг-1 Арг-2)

сначала значение **Арг-1** присваивается атому **x**, значение **Арг-2** - атому **y**. Затем вычисляется тело функции. Результат вычисления возвращается в качестве результата вызова. Важно подчеркнуть, что **значения аргументов вычисляет не наша функция QDIF, а ядро Лиспа**. Поскольку функция **QDIF**, создана вызовом **DEFUN**, то она принадлежит к классу **EXPR**, т.е. ядро Лиспа "знает", что значения аргументов функции **QDIF** перед вызовом нужно вычислять.

Теперь испытаем нашу функцию:

```
(DEFUN QDIF (x y) (* (- x y) (+ x y)))
```

```
==> QDIF

(QDIF 7 8)

==> -15

(QDIF 8 7)

==> 15

(QDIF 11111 22222)

==> -370362963

(QDIF 11111)

Список формальных параметров и список фактических
параметров имеют разную длину
==> ERRSTATE

(QDIF 11111 22222 33333)

Список формальных параметров и список фактических
параметров имеют разную длину
==> ERRSTATE

(setq x 6)

==> 6

(setq y 6)

==> 6

(QDIF 12 11)

==> 23

x

==> 6

y

==> 6
```

Рассмотрим приведенную врезку подробнее. Введенная команда **DEFUN** возвращает в качестве результата атом **QDIF**. Это является признаком успешного создания функции **QDIF**. Следующие затем три вызова "свежесозданной" функции **QDIF** подтверждают ее работспособность.

Последующие попытки вызвать функции **QDIF** с одним или тремя аргументами вызывают ошибку. Признак ошибки - возврат атома**ERRSTATE** в качестве результата. Как правило, ошибка сопровождается диагностическим сообщением (которое не нуждается в

комментариях).

Далее заводятся две переменные **x** и **y** и им присваивается значение **6**. Последующий вызов функции с аргументами **12** и **11** дает правильный результат. А вот проверка значений атомов **x** и **y** может удивить. Ведь, в соответствии со сказанным выше, переменным **x** и **y**при вызове функции должны были присвоиться значения **12** и **11**. Тем не менее, значения переменных **x** и **y** остаются теми же, какими были до вызова функции!

Будем называть атомы, входящие в список **формальных параметров** какой-либо функции **переменными, связанными в теле этой функции**. Переменные, не входящие в список формальных параметров, но использующиеся в теле функции, будем называть**свободными** переменными.

Испытание нашей функции **QDIF** демонстрирует **важный принцип Лиспа**: после возврата из функции значения связанных в теле функции переменных будут теми же самыми, что и до вызова функции.

А вот если функция каким-либо образом изменит значение свободной переменной, то это изменение останется и после выхода из функции. Как же функция может изменить значение свободной переменной? Употреблением функций **SET/SETQ**. Изменим тело нашей функции **QDIF**следующим образом:

(DEFUN QDIF (x y) (setq z (* (- x y) (+ x y))))

Здесь тело функции представляет собой вызов **SETQ**, который присваивает **свободной** переменной **z** значение разности квадратов, которое вычисляется по прежней формуле. Поскольку **SETQ** возвращает значение второго аргумента, функция **QDIF** не теряет работоспособности, однако, если присвоить атому **z** какое-либо значение, то после вызова оно будет заменено разностью квадратов:

```
(DEFUN QDIF (x y) (setq z (* (- x y) (+ x y))))

==> QDIF

(setq z 0)

==> 0

(qdif 5 3)

==> 16

z

==> 16
```

Приемы программирования на Лиспе. Рекурсия.

Сейчас будет рассмотрен первый пример нетривиальной программы на Лиспе, который вполне передает суть этого замечательного языка.

Пусть имеется **произвольный** список, состоящий из чисел. Нужно подсчитать сумму чисел, входящих в список. Как это сделать? Длина списка (количество его элементов) заранее не известна. Читателям, знакомым с традиционными языками программирования, возможно, покажется, что нужно завести переменную для будущей суммы, а затем, перебирая элемент за элементом наш список, копить сумму. Процесс этот повторять до завершения списка. Так поступить действительно можно, но непонятно, как организовать циклическое повторение...

На самом деле, поставленная задача допускает простое и элегантное решение. Обозначим нашу будущую функция **SUMLIST**. Тогда, если входной список **пуст**, то функция должна возвращать **нуль**. Это понятно. А если список **непуст**, то представляется вполне очевидным, что сумма элементов списка равна его первому элементу (**CAR**) плюс сумма элементов остатка (**CDR**). Другими словами, нашу функцию можно представить так:

```
(DEFUN SUMLIST (x) (COND ((NULL x) 0)
                         (T (+ (CAR x) (SUMLIST (CDR x))))))

==> sumlist
```

Поскольку эта функция - первый нетривиальный пример лисповской функции, опишем ее подробнее. Тело функции представляет собой**COND**-конструкцию из двух условий. Первое условие проверяет, не пуст ли список, поданный на вход функции. Если значение выражения**(NULL x)** истино (равно **T**), то функция возвращает нуль. В противном случае происходит переход к проверке второго условия. На месте второго условного выражения стоит **T** (условие всегда истино), поэтому вычисляется сумма **первого элемента** списка **x** и значения функции **SUMLIST**, примененной к списку **x** без первого элемента. Все! Задача решена.

Пусть читатель обратит внимание на **лаконичность** определения функции. Всего две строки, причем **очень понятного кода**! В каком еще языке программирования такое возможно?!

Чтобы убедиться в работоспособности нашей функции, попробуем вычислить сумму списка **(1 2 3 4 5)**:

```
(sumlist '(1 2 3 4 5))

==> 15
```

Функция работает. Можно убедиться, что функция будет работать правильно с любым одноуровневым списком чисел. Следует отметить, что функция **SUMLIST** вызывает сама себя. Такие функции называются **рекурсивными**.
Может возникнуть вполне естственный вопрос: почему функция, вызывающая сама себя, не "зацикливается"? Это происходит потому, что в теле функции **первым условием** стоит проверка значения входного параметра на пустоту. Если значение входного параметра есть пустой список (**Nil**), то происходит выход из функции с возвратом значения **0**. А теперь пусть читатель обратит внимание на то, что повторное обращение к функции **SUMLIST** происходит не к исходному списку, а к

его **остатку** (после отделения головы). Последующее обращение происходит с остатком остатка и т.д. Поскольку список имеет конечное число элементов, то рано или поздно остаток окажется пустым списком. Это гарантирует завершение цепочки вызовов.

Обычно тело рекурсивной функции состоит из проверки различных условий. Условие, гарантирующее выход, называется **терминальным условием**. Обычно, хотя и не обязательно, терминальное условие располагается в списке условий первым (как в функции **SUMLIST**). Что произойдет, если терминальное условие будет опущено? Ответ на этот вопрос дает следующая врезка:

```
(DEFUN BAD_SUMLIST (x) (+ (CAR x) (BAD_SUMLIST (CDR x))))

==> bad_sumlist

(bad_sumlist '(1 2 3))

Переполнение внутреннего стека
==> ERRSTATE
```

Здесь вводится определение неправильной рекурсивной функции (без терминальной ветви). Функция принимается ядром Лиспа, однако, попытка вычислить сумму даже простого списка из трех элементов вызывает ошибку с соответствующей диагностикой.

Рекурсия "изнутри". Трассировка выполнения. Функции TRACE и UNTRACE.

Трудно предположить, какие чувства испытывает читатель при первом знакомстве с рекурсией, но у автора этих строк рекурсия до сих пор вызывает ощущение чуда... Чтобы изучить работу рекурсивной функции, удобно использовать возможность Лиспа, называемую**трассировкой**.

Трассировка заключается в том, что при входе в трассируемую функцию печатаются значения входных параметров, а при выходе - результат вычисления.

Чтобы включить режим трассировки, существует специальная функция **TRACE**, принадлежащая классу **FSUBR** и требующая единственный аргумент - имя функции. Если выполнение **TRACE** завершено удачно, то возвращается имя трассируемой функции. В противном случае возвращается **Nil**. Попробуем включить трассировку нашей функции **SUMLIST** и вновь вычислить сумму элементов списка **(1 2 3 4 5)**:

```
(trace sumlist)

==> sumlist

(sumlist '(1 2 3 4 5))
```

```
Вход в функцию sumlist Аргументы: (1 2 3 4 5)
  Вход в функцию sumlist Аргументы: (2 3 4 5)
    Вход в функцию sumlist Аргументы: (3 4 5)
      Вход в функцию sumlist Аргументы: (4 5)
        Вход в функцию sumlist Аргументы: (5)
          Вход в функцию sumlist Аргументы: NIL
          Возврат из функции sumlist Результат: 0
        Возврат из функции sumlist Результат: 5
      Возврат из функции sumlist Результат: 9
    Возврат из функции sumlist Результат: 12
  Возврат из функции sumlist Результат: 14
Возврат из функции sumlist Результат: 15

==> 15
```

Видно, что рекурсивная функция **SUMLIST**, "точит исходный список, как карандаш". Суммирование элементов происходит от последнего к первому.

Если для функции включена трассировка, то любой вызов функции будет сопровождаться выдачей трассировочной информации. Чтобы прекратить трассировку, следует вызвать функцию **UNTRACE**. Эта функция, как и функция **TRACE**, принадлежит к классу **FSUBR**. Функция**UNTRACE** принимает единственный параметр - имя функции, трассировка которой выключается. При успешном завершении функция возвращает входной параметр. Если запрошенной функции нет в системе, **UNTRACE** вернет атом **Nil**.

Другие примеры рекурсивных функций.

Функция **SUMLIST** верно работает только для **атомных одноуровневых** списков (т.е. списков, состоящих из атомов). Если попытаться вызвать эту функцию для списков другого типа, то она окажется неработоспособной:

```
(sumlist '( 1 2 3 (4 5) 6))

Один из аргументов PLUS - не атом
==> ERRSTATE

(trace sumlist)

==> sumlist

(sumlist '( 1 2 3 (4 5) 6))

Вход в функцию sumlist Аргументы: (1 2 3 (4 5) 6)
  Вход в функцию sumlist Аргументы: (2 3 (4 5) 6)
    Вход в функцию sumlist Аргументы: (3 (4 5) 6)
      Вход в функцию sumlist Аргументы: ((4 5) 6)
        Вход в функцию sumlist Аргументы: (6)
          Вход в функцию sumlist Аргументы: NIL
          Возврат из функции sumlist Результат: 0
        Возврат из функции sumlist Результат: 6
```

```
Один из аргументов PLUS - не атом
==> ERRSTATE
```

В протоколе трассировки видно, что ошибка происходит, когда функция пытается к числу **6** прибавить список **(4 5)**.

Доработаем функцию **SUMLIST** таким образом, чтобы она могла суммировать элементы списков **любой структуры**. Это не так трудно, как может показаться на первый взгляд. Посмотрим еще раз на определение функции **SUMLIST**:

```
(DEFUN SUMLIST (x)
       (COND ((NULL x) 0)
             (T (+ (CAR x) (SUMLIST (CDR x))))))
```

Читатель, вероятно, уже догадывается, что для того, чтобы не возникала ошибка, нужно перед сложением применить к выражению **(CAR x)** функцию **SUMLIST**. Таким образом, наша функция станет "дважды рекурсивной":

```
(DEFUN SUMLIST (x)
      (COND ((NULL x) 0)
            (T (+ (SUMLIST (CAR x)) (SUMLIST (CDR x))))))
```

Это, однако, еще не решает проблему, а, напротив, порождает новые ошибки:

```
(sumlist '(1 2 3 4 5))

Аргумент CAR - атом (1)
==> ERRSTATE
```

Функция "разладилась" - совсем перестала работать. Чтобы понять суть ошибки, включим трассировку, и попытаемся вновь вычислить**(sumlist '(1 2 3 4 5))**:

```
(trace sumlist)

==> sumlist

(sumlist '( 1 2 3 4 5))

            Вход в функцию sumlist Аргументы: (1 2 3 4 5)
              Вход в функцию sumlist Аргументы: 1
Аргумент CAR - атом (1)
==> ERRSTATE
```

Функция правильно выделила голову списка (атом **1**), но сразу же попыталась передать этот **атом** на вход себе при рекурсивном вызове. При повторном входе будет вычисляться выражение**(SUMLIST 1)**, и снова будет сделана попытка вычислить первый элемент значения аргумента. Но беда в том, что теперь значение аргумента - уже не список, а атом! Попытка вычислить функцию **CAR** с атомным аргументом вызывает, естественно, ошибку.

Чтобы предотвратить эту ошибку, давайте будем считать, что если на вход функции **SUMLIST** подано **число**, а не список, то результат вычисления - просто равен этому числу. Это предположение вполне естественно сочетается со смыслом функции **SUMLIST**. Реализовать эту идею несложно:

```
(DEFUN SUMLIST (x)
       (COND ((NULL x) 0)
             ((ATOM x) x)
             (T (+ SUMLIST (CAR x)) (SUMLIST (CDR x))))))
```

Прежде чем активизировать суммирование, проверяем, атом у нас на входе, или список. Если атом - просто возвращаем его в качестве результата.

Можно убедиться, что функция работоспособна:

```
(sumlist '( 1 2 3 4 5))

==> 15

(sumlist '( 1 (2 3 (4)) 5))

==> 15
```

Теперь на вход функции можно подавать сколь угодно сложные списки (лишь бы они состояли **только** из чисел) - результат всегда будет правильным! Если же протрассировать вычисление выражения **(sumlist '(1 2 3 4 5))**, то можно убедиться, что за универсальность пришлось заплатить значительно возросшим объемом вычислений, - теперь каждый аргумент проверяется на атомность:

```
(sumlist '( 1 2 3 4 5))

                Вход в функцию sumlist Аргументы: (1 2 3 4 5)
                  Вход в функцию sumlist Аргументы: 1
                  Возврат из функции sumlist Результат: 1
                  Вход в функцию sumlist Аргументы: (2 3 4 5)
                    Вход в функцию sumlist Аргументы: 2
                    Возврат из функции sumlist Результат: 2
                    Вход в функцию sumlist Аргументы: (3 4 5)
```

```
                    Вход в функцию sumlist Аргументы: 3
                    Возврат из функции sumlist Результат: 3
                    Вход в функцию sumlist Аргументы: (4 5)
                      Вход в функцию sumlist Аргументы: 4
                      Возврат из функции sumlist Результат: 4
                      Вход в функцию sumlist Аргументы: (5)
                        Вход в функцию sumlist Аргументы: 5
                        Возврат из функции sumlist Результат: 5
                        Вход в функцию sumlist Аргументы: NIL
                        Возврат из функции sumlist Результат: 0
                      Возврат из функции sumlist Результат: 5
                    Возврат из функции sumlist Результат: 9
                  Возврат из функции sumlist Результат: 12
                Возврат из функции sumlist Результат: 14
              Возврат из функции sumlist Результат: 15
==> 15

(sumlist '( 1 (2 3 (4)) 5))


              Вход в функцию sumlist Аргументы: (1 (2 3 (4)) 5)
                Вход в функцию sumlist Аргументы: 1
                Возврат из функции sumlist Результат: 1
                Вход в функцию sumlist Аргументы: ((2 3 (4)) 5)
                  Вход в функцию sumlist Аргументы: (2 3 (4))
                    Вход в функцию sumlist Аргументы: 2
                    Возврат из функции sumlist Результат: 2
                    Вход в функцию sumlist Аргументы: (3 (4))
                      Вход в функцию sumlist Аргументы: 3
                      Возврат из функции sumlist Результат: 3
                      Вход в функцию sumlist Аргументы: ((4))
                        Вход в функцию sumlist Аргументы: (4)
                          Вход в функцию sumlist Аргументы: 4
                          Возврат из функции sumlist Результат: 4
                          Вход в функцию sumlist Аргументы: NIL
                          Возврат из функции sumlist Результат: 0
                        Возврат из функции sumlist Результат: 4
                        Вход в функцию sumlist Аргументы: NIL
                        Возврат из функции sumlist Результат: 0
                      Возврат из функции sumlist Результат: 4
                    Возврат из функции sumlist Результат: 7
                  Возврат из функции sumlist Результат: 9
                  Вход в функцию sumlist Аргументы: (5)
                    Вход в функцию sumlist Аргументы: 5
                    Возврат из функции sumlist Результат: 5
                    Вход в функцию sumlist Аргументы: NIL
                    Возврат из функции sumlist Результат: 0
                  Возврат из функции sumlist Результат: 5
                Возврат из функции sumlist Результат: 14
              Возврат из функции sumlist Результат: 15
==> 15
```

Хорошо видно, что теперь функция работает корректно с любыми списками, подаваемыми на ее вход.

Рассмотрим еще одну задачу обработки списков. Пусть заданы два произвольных списка. Требуется их **объединить** (т.е. из списков **(a b c)** и **(d e f)** получить список **(a b c d e f)**.

Сразу обратим внимание читателя на то, что "лобовое" применение функции **CONS** не дает требуемого результата. Значением выражения**(CONS '(a b c) '(d e f))** будет список **((a b c) d e f)**, а отнюдь не **(a b c d e f)**.

Продемонстрируем рекурсивный подход к задаче. Составим функцию **APPEND**, которая будет принимать **два** параметра: список **первый** и список **второй**. Очевидно, что если первый список пуст (равен **Nil**), то результат равен второму списку. Если **первый** спискок непуст, то результат функции **APPEND** должен быть равен **голове первого** списка, присоединенной посредством функции **CONS** к значению функции**APPEND**, с **хвостом первого списка** (первый параметр) и **исходным вторым списком** (второй параметр). И это - все:

```
(DEFUN APPEND (x y)
       (COND ((NULL x) y)
             (T (CONS (CAR x) (APPEND (CDR x) y)))))

==> APPEND

(APPEND '(a b c) '(d e f))

==> (a b c d e f)
```

Решение снова получилось коротким и изящным (что, как правило, всегда характерно для рекурсивных программ).

Безымянные функции. Конструкция LAMBDA.

В Лиспе существует очень интересная конструкция - **безымянные функции**. Суть безымянной функции состоит в том, что задается алгоритм вычисления, но не задается имени функции. Безымянную функцию можно применить к списку аргументов и сразу получить результат. Синтаксис задания безымянной функции таков:

(LAMBDA (список параметров) (тело функции))

Список параметров - это одноуровневый атомный список (т.е. список, состоящий **только из атомов**). **Тело функции** - это S-выражение, зависящее от атомов, входящих в список параметров.

Вот пример задания безымянной функции:

(LAMBDA (x y) (+ (* x x) (* y y)))

Легко видеть, что тело функции обеспечивает вычисление величины x^2+y^2. Как пользоваться безымянными функциями? Если передать приведенное выше выражение Лисп-машине, то мы получим обескураживающий результат:

```
(LAMBDA (x y) (+ (* x x) (* y y)))

Не найдена функция LAMBDA
==> ERRSTATE
```

Впрочем, этот результат удивителен только на первый взгляд. Лисп просто пытается вычислить функцию **LAMBDA** с двумя аргументами:**(x y)** и **(+ (* x x) (* y y))**. Функции **LAMBDA** в системе нет, отсюда и ошибка. Чтобы вычислить значение безымянной функции, нужно составить S-выражение, представляющее собой **список**, головой которого является **вся LAMBDA-конструкция**, а последующими элементами - **фактические параметры**. Ниже показан правильный вызов безымянной функции:

```
( (LAMBDA (x y) (+ (* x x) (* y y) ) ) 3 4 )

==> 25
```

Легко видеть, что приведенная выше **LAMBDA**-конструкция полностью эквивалентна традиционной именованной функции:

```
(defun SQ (x y) (+ (* x x) (* y y)))

==> SQ

(SQ 3 4)

==> 25
```

Разница в том, что при задании именованной функции соответствующий атом (в примере - **SQ**) становится именем функции, при этом, определяющее выражение **(+ (* x x) (* y y))** сохраняется в теле функции. А безымянная функция просто вычисляется и "исчезает".

Впрочем, **HomeLisp** позволяет использовать безымянные функции нетрадиционным образом: если присвоить какому-либо атому в качестве значения корректное **LAMBDA**-выражение, то появляется возможность использовать это **LAMBDA**-выражение без повторного задания:

```
(setq sq2 '(LAMBDA (x y) (+ (* x x) (* y y) )))

==> (LAMBDA (x y) (+ (* x x) (* y y)))

sq2

==> (LAMBDA (x y) (+ (* x x) (* y y)))

(sq2 6 7)

==> 85
```

Последний вызов **(sq2 6 7)** очень похож на вызов функции **sq2**. Сходство, однако, чисто внешнее. Атом **sq2 не является именем функции** (что будет разъяснено ниже, при рассмотрении списков свойств).

Главное применение безымянных функций - рассматриваемое ниже функциональные аргументы.

Буква греческого алфавита **лямбда** взята из т.н. **лямбда-исчисления** разработанного английским математиком **Чёрчем**. Изображение буквы лямбда давно стало негласным символом языка Лисп.

Функциональные аргументы. Функционалы.

Рассмотрим простую задачу: дан произвольный список чисел, требуется построить список квадратов этих чисел. Решение этой задачи довольно просто:

```
(defun p2 (x) (cond ((null x) nil)
                    (T (cons (^ (car x) 2) (p2 (cdr x))))))

==> p2

(p2 '(1 2 3 4 5))

==> (1 4 9 16 25)
```

В теле функции сначала анализируется, не пуст ли входной список. Если это так, функция возвращает пустой список (обязательная терминальная ветвь рекурсии). В противном случае функция отщепляет первый элемент списка, возводит его в квадрат и строит точечную пару из этого квадрата и результата применения функции к остатку исходного списка без первого элемента. Функция успешно работает.

Теперь предположим, что потребовалась аналогичная функция, но возводящая элементы списка не в квадрат, а в куб. Придется составить фукнцию p3:

```
(defun p3 (x) (cond ((null x) nil)
                    (T (cons (^ (car x) 3) (p3 (cdr x))))))

==> p3

(p3 '(1 2 3 4 5))

==> (1 8 27 64 125)
```

Если сравнить определяющие выражения функций **p2** и **p3**, то легко увидеть, что эти функции различаются **только действием, применяющимся к голове списка**, в остальном эти функции идентичны. А что, если **действие**, применяющееся к голове списка, сделать **параметром** функции? Тогда вместо двух (или большего числа) разных функций достаточно будет составить **единственную**функцию:

```
(defun pf (x f) (cond ((null x) nil)
                      (T (cons (f (car x)) (pf (cdr x) f)))))

==> pf
```

Для того, чтобы воспользоваться новой функцией **pf**, нужно составить функцию, возводящую одно число в квадрат (или куб):

```
(defun f^2 (x) (^ x 2))

==> f^2

(defun f^3 (x) (^ x 3))

==> f^3
```

Читатель вероятно уже догадался, что для возведения элементов списка в квадрат, нужно вызвать функцию **pf** следующим образом: **(pf '(1 2 3 4) f^2)**. Однако такой вызов приводит к ошибке:

```
(pf '(1 2 3 4) f^2)

Символ f^2 не имеет значения (не связан).
==> ERRSTATE
```

И понятно, почему это происходит: функция **pf** принадлежит к классу **EXPR**, поэтому, перед вызовом ядро Лиспа делает попытку вычислить значение атома **f^2**. Чтобы предотвратить эту ошибку, атом **f^2** при вызове нужно квотировать:

```
(pf '(1 2 3 4) 'f^2)

==> (1 4 9 16)
```

Второй аргумент вызова функции **pf** представляет собой имя функции. Такой аргумент называется **функциональным**, а функция, имеющая хотя бы один функциональный аргумент, называется **функционалом**.

Для вызова функционалов предназначена специальная "функция" **FUNCTION**. С использованием этой "функции" последний вызов записывается так:

```
(pf '(1 2 3 4) (function f^2))

==> (1 4 9 16)
```

Слово "функция" заключается в кавычки, поскольку S-выражение **(FUNCTION f^2)** не вычисляется в привычном смысле этого слова. Конструкция **(FUNCTION f^2)** сообщает ядру Лиспа, что **f^2** - функциональный аргумент. Такая запись нагляднее квотирования, поскольку функциональный аргумент отмечается явно. Кроме того, использование **FUNCTION** предпочтительнее еще по одной причине (подробно рассмотренной в соответствующем разделе, посвященном описанию встроенных функций Лиспа.

На месте функционального аргумента может стоять и безымянная функция, т.е. лямбда-выражение. Приведенный выше пример возведения элементов списка в квадрат может быть решен без предварительного определения именованной функции следующим образом:

```
(pf '(1 2 3 4 5 6) (function (lambda (x) (^ x 2))))

==> (1 4 9 16 25 36)
```

Применяющие функционалы. Функции FUNCALL и APPLY.

Среди всех мыслимых функционалов можно выделить один специфический вид - функционал, который **применяет** функциональный аргумент к остальным аргументам. Такие функционалы обычно называют **применяющими функционалами**.

В ядро Лиспа встраиваются два стандартных функционала **FUNCALL** и **APPLY**. Рассмотрим их подробнее.

Функция **FUNCALL** принадлежит классу **SUBR** и принимает произвольное количество

аргументов. Эта функция применяет свой **первый**(функциональный) аргумент к оставшемуся списку аргументов. Это выглядит примерно так:

```
(funcall '* 1 2 3 4 5 6 7 8 9 10)

==> 3628800
```

Использование **FUNCALL** позволяет, например, давать именам функций синонимы (примеры взяты из книги Э.Хювенен, Й.Сеппянен **[6]**):

```
(setq сложить '+)

==> +

(funcall сложить 1 2 3)

==> 6
```

Более того, допустимо использовать имя стандартной функции для хранения ссылки на какую-либо другую функцию:

```
(setq cons '*)

==> *

(cons 2 3)

==> (2 . 3)

(funcall cons 2 3)

==> 6

(funcall 'cons 2 3)

==> (2 . 3)
```

Здесь атому **CONS** присвоено значение ***** (что вполне допустимо!). Следующий вызов показывает, что традиционная функция **CONS** не теряет работоспособности (образуется точечная пара). А вот при вызове посредством **FUNCALL** производится замена **CONS** на ***** - образуется произведение. В третьем вызове квотирование блокирует замену **CONS** на *****, и вновь образуется точечная пара.

Любопытно, что если вернуться к предыдущему примеру (с присвоением значения **+** атому **сложить**), то легко убедиться, что квотирование здесь не дает эффекта:

```
(setq сложить '+)

==> +

(funcall сложить 2 3 4)

==> 9

(funcall 'сложить 2 3 4)

==> 9
```

Это связано с особенностью реализации **HomeLisp**, - в других версиях Лиспа вызов **(funcall 'сложить 2 3 4)** завершится ошибкой.

С функцией **FUNCALL** очень схожа функция **APPLY** (также принадлежащая классу **SUBR**). Отличие заключается в том, что у функции**APPLY** ровно два аргумента: первый - функциональный, а второй является списком произвольной длины.
Вызов **APPLY** заключается в том, что вычисляется функция, заданная первым аргументом, со списком параметров, заданным вторым аргументом **APPLY**. Вот пример вызова этой функции:

```
(setq mult '*)

==> *

mult

==> *

(apply mult '(1 2 3 4 5 6 7 8 9 10))

==> 3628800
```

Отображающие функционалы. Функции MAPLIST и MAPCAR.

Еще одним классом функционалов является класс **отображающих** функционалов. Такие функционалы применяют функциональный аргумент к элементам списка, в результате чего строится новый список. Отсюда и название: исходный список **отображается** на результирующий.

Здесь будут рассмотрены два отображающих функционала: **MAPLIST** и **MAPCAR**.

Функционал **MAPLIST** принимает два аргумента. Значение первого аргумента должно быть списком. Второй аргумент - функциональный. Функция, задаваемая вторым аргументом, должна принимать на вход список. Выполнение функционала заключается в том, что функция, заданная вторым аргументом, последовательно применяется к значению первого аргумента, к этому списку без первого элемента, без первых двух

элементов и т.д. до исчерпания списка. Результаты вызова функции объединяются в список, который функционал вернет в качестве значения. Вот примеры вызова **MAPLIST**:

```
(maplist '(1 2 3 4 5 6) (function sumlist))

==> (21 20 18 15 11 6)

(maplist '(1 2 3 4 5 6) (function reverse))

==> ((6 5 4 3 2 1) (6 5 4 3 2) (6 5 4 3) (6 5 4) (6 5) (6))
```

В первом примере используется функция, вычисляющая сумму элементов списка. Первый раз суммируется список **(1 2 3 4 5 6)**; получается**21**. Затем суммированию подвергается список **(2 3 4 5 6)**; получается 20 и т.д. Полученные результаты объединяются в список. Результат равен **(21 20 18 15 11 6)**.

Во втором примере используется функция, обращающая список. Эта функция возвращает не атом, а список. При первом вызове обращается список **(1 2 3 4 5 6)**; получается **(6 5 4 3 2 1)**. Затем обращается список **(2 3 4 5 6)**; получается **(6 5 4 3 2)** и т.д. В результате получается список, состоящий из списков.

В отличие от **MAPLIST**, функционал **MAPCAR** применяет функциональный аргумент не к остаткам списка, а последовательно к каждому элементу. Результаты этих применений объединяются в список, который и возвращает функционал **MAPCAR**. Вот как это выглядит:

```
(mapcar '(1 2 3 4 5 6) (function (lambda (x) (* x x))))

==> (1 4 9 16 25 36)
```

Функциональное выражение, заданное вторым аргументом **MAPCAR**, возводит свой аргумент в квадрат. Вызов **MAPCAR** отображает исходный список на список своих квадратов.

Две парадигмы программирования - функциональная и процедурная.

Читатель, вероятно, обратил внимание на то, что программирование на Лиспе весьма отличается от программирования на традиционных языках (таких, как C/C++, Паскаль, Бэйсик). Вместо того, чтобы разбивать задачу на элементарные шаги, мы составляем одну или более функций, вызов которых приводит к требуемому результату. Часто такие функции оказываются рекурсивными. Подобный подход (или, как сейчас модно говорить - **парадигма**) называется **функциональным** программированием. Чистое функциональное программирование не признает операторов присваивания, не использует ветвления и циклы (вместо последних применяется рекурсия).

Примером функционального подхода может служить язык формул, встроенный в Microsoft

Excel. Для того, чтобы получить результат, пользователь должен написать одну или более формул, аргументы которых охватывают диапазон обрабатываемых ячеек. При этом отсутствует возможность организации циклов для перебора ячеек, - все действия должны быть выражены **только через функции**. Это и есть функциональный подход.

В противоположность функциональной, **процедурная парадигма** соответствует подходам традиционных языков: используются присвоения, явные переходы, циклы.

По изобразительным возможностям функциональный и процедурный подходы примерно равнозначны друг другу. Некоторые задачи легче решаются при процедурном подходе, некоторые - при функциональном. Подобная ситуация приводит к тому, что современные языки программирования (например, **Python**) содержат как процедурные, так и функциональные средства.

Но не следует забывать, что **первым функциональным языком был все-таки Лисп...**.

Процедурное программирование в Лиспе. Функция PROG.

Разумеется, Лисп имеет в своем составе и средства для процедурного программирования. Как известно, краеугольным камнем процедурного подхода является понятие **оператора**. Оператор выполняет различные действия над значениями **переменных**. Для присвоения переменным значений служат функции **SET/SETQ**. А для моделирования блока операторов служит конструкция **PROG**, которая описывается ниже.

Конструкция **PROG** имет следующий общий вид:

```
(PROG
        (Список локальных переменных)

        Атом или вызов функции

        Атом или вызов функции

        Атом или вызов функции

        Атом или вызов функции

        ...
)
```

Опишем составные части этой конструкции подробнее. **Список локальных переменных** представляет собой одноуровневый список**атомов**. Когда конструкция **PROG** начинает вычисляться, каждый атом из списка локальных переменных получает значение **Nil**. Если этот атом уже имел значение, то прежнее значение становится недоступным. Другое название списка локальных переменных - список**связанных** переменных. В противоположность таким переменным, остальные переменные, использованные в теле функции, называются**свободными**.

Как видно из врезки, далее в теле **PROG** подряд располагаются атомы или произвольные вызовы функций. Эти, отдельно стоящие атомы, называются **метками**. Выполнение тела **PROG** заключается в том, что последовательно выполняются вызовы функций, а метки пропускаются. Обычно отдельные вызовы в теле **PROG**-конструкции называются **операторами**. Таким образом, можно сказать, что тело**PROG**-конструкции состоит из операторов и меток.

Могут быть вызваны любые доступные функции, а кроме того, имеются две **специфические** функции, которые могут употреблятся **только в теле PROG**. Это функции **GO** и **RETURN**. Обе эти функции принадлежат к классу **SUBR**.

Значением единственного аргумента функции **GO** должен быть атом. Выполнение **GO** заключается в том, что среди меток тела **PROG**ищется значение аргумента **GO**. Если значение найдено, то в качестве следующего вызова функции выполняется вызов функции, стоящей**после найденной метки**. Если же метка, заданная при вызове **GO**, отсутствует в теле функции, то выполнение **PROG** завершается ошибкой. Легко видеть, что функция **GO** осуществляет **передачу управления**.

Функция **RETURN** осуществляет **выход** из **PROG**-конструкции. Значение аргумента **RETURN** возвращается, как значение всей **PROG**-конструкции. После выполнения функции **RETURN** все атомы, входящие в список локальных переменных, теряют значения, которые могли быть им присвоены в теле **PROG**. Если же атом, входящий в этот список, ранее имел значение, то это значение **восстанавливается**.

Если **последний** вызов функции в теле **PROG**-конструкции (перед закрывающей скобкой) не есть вызов **RETURN**, то происходит принудительный выход из **PROG**-конструкции, а в качестве результата возвращается **Nil**.

В качестве первого примера **PROG**-конструкции рассмотрим уже решенную ранее задачу подсчета суммы элементов одноуровнего списка. Решение получится вполне традиционным: заводим локальную переменную для суммы (**x**) и для остатка списка (**y**). Далее:

1) Присваиваем переменной **x** значение нуль, а переменной **y** - исходный список;

2) Прибавляем к значению **x** голову **y**;

3) Присваиваем переменной **y** значение хвоста **y**;

4) Если значение **y** оказывается равным **Nil**, возвращаем значение **x**;

5) Переходим к шагу 2.

Ниже показывается, как работает такая конструкция:

```
(setq lst '(1 2 3 4 5 6 7 8 9 10))

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

(prog (x y) (setq x 0)
            (setq y lst)
        @   (setq x (+ x (car y)))
```

```
            (setq y (cdr y))
            (cond ((null y) (return x))
                  (t (go @)))
)

==> 55

(setq x 777)

==> 777

(prog (x y) (setq x 0)
            (setq y lst)
        @   (setq x (+ x (car y)))
            (setq y (cdr y))
            (cond ((null y) (return x))
                  (t (go @)))
)

==> 55

x

==> 777
```

Здесь сначала переменной **lst** присваивается суммируемый список. Далее **PROG**-конструкция вычисляет сумму его элементов. Как можно убедиться, получается правильный результат. Далее, переменной **x** "нарочно" присваивается значение **777**, после чего снова вычисляется сумма элементов списка. И, хотя переменная **x** меняет свое значение в теле **PROG**, после возврата значение **x** сохраняется.

Функции типа FEXPR. Функция DEFUNF.

Функции типа **FEXPR** отличаются от функций типа **EXPR** тем, что при их вычислении, ядро Лиспа не вычисляет значения аргументов, а передает их функции **как есть**.

Функции типа **FEXPR** не являются строго необходимыми, - если требуется передать в функцию не значение аргумента, а сам аргумент, то можно использовать функцию типа **EXPR** с квотированным аргументом. Если функции при любом вызове требуются **аргументы**, а не их**значения**, то **для простоты вызова**, можно создать функцию типа **FEXPR**.

Для создания функции типа **FEXPR** служит функция **DEFUNF**, являющаяся полным аналогом функции **DEFUN**, но порождающая функцию типа типа **FEXPR**. Использование функций типа **FEXPR** требует понимания и острожности. Рассмотрим в качестве примера простую функцию создающую точечную пару из двух аргументов:

```
(defunf cosq (x y) (cons x y))

==> consq
```

```
(consq 11 22)

==> (11 . 22)

(consq a b)

==> (a . b)

(consq a (b))

==> (a b)

(setq a 111)

==> 111

(consq a (b))

==> (a b)
```

Первые три вызова новой функции показывают ее работоспособность - функция строит точечную пару из своих аргументов не хуже, чем стандартная функция **CONS**. Но если присвоить атому **a** какое-либо значение, то попытка вызова **CONSQ** с атомом **a** в качестве одного из аргументов приводит к неожиданному результату: замены аргумента значением не происходит. Впрочем, почему неожиданному? Это - закономерный результат использования функции класса **FEXPR**. Значения аргументов, переданных функции класса **FEXPR**, можно вычислить в теле самой функции (используя универсальную функцию **EVAL**). Можно даже ввести в список параметров функции дополнительный параметр-флаг, управляющий процессом вычисления аргументов:

```
(defunf consq (x y f) (cond ((null (eval f)) (cons (eval x)
(eval y)))
                            ( T (cons x y))))

==> consq

(consq a b t)

==> (a . b)

(consq a b nil)

Символ a не имеет значения (не связан).
==> ERRSTATE

(setq a 111)

==> 111

(setq b 222)
```

```
==> 222

(consq a b nil)

==> (111 . 222)

(consq a b t)

==> (a . b)
```

Если значение параметра **f** (обязательно вычисляемого в теле функции!), равно **Nil**, строится точечная пара из **значений** аргументов. Если же значение параметра **f** равно **T**, то строится точечная пара из **самих** аргументов. Вызов **CONSQ** с параметром **f=Nil** требует вычисления значений аргументов (и вызывает ошибку, если хотя бы один из аргументов не имеет значения).

Легко видеть, что при необходимости **выборочного** вычисления значений аргументов проще использовать функцию класса **EXPR/SUBR** и просто **квотировать** аргументы, значения которых не нужно вычислять. Поэтому функции класса **FEXPR** используются не часто. Гораздо большее применение находит в Лиспе другой тип функций, не вычисляющий своих аргументов - функции типа **MACRO**.

Функции типа MACRO. Функция DEFMACRO.

Понятие **"макрообработка"** относится к достаточно давним понятиям информатики. Макрообработка - это преобразование произвольного**входного** текста (включающего кроме обычного содержания, **специальные команды**) с целью получения другого **выходного** текста. Важно отметить, что хотя специальные команды управляют процессом обработки, но в выходной текст не попадают. Сам процесс обработки называется **макрогенерацией**. Программа, осуществляющая преобразование, называется **макрогенератором**. Команды преобразования принято называть **макрокомандами**.

Если преобразуемый текст есть текст программы на каком-либо языке программирования, то применение средств макрогенерации позволяет, например, на основе одного исходного текста генерировать программы, рассчитанные на различные аппаратные платформы (Windows, Unix, Linux). Практически все современые средства разработки программ имеют в своем составе более или менее мощные макрогенераторы. Если такой макрогенератор автоматически запускается перед компиляцией, то его принято называть **препроцессором**.

Изобразительные возможности языка препроцессора, как правило, **заметно слабее** изобразительных возможностей базового языка. Исключение, пожалуй, составляет язык PL/I, в составе которого имеется мощный препроцессор, очень сходный по синтаксису с базовым языком. В языках C/C++, VB, Delphi возможности препроцессора весьма скромны.

Совсем по-иному обстоит дело в Лиспе. Поскольку в Лиспе нет принципиальной разницы между программами и данными (и то, и другое представляет собой списки), то появляется уникальная возможность создания функций, которые выполняются в два этапа. На первом этапе строится S-выражение, которое и вычисляется на втором этапе. Про такие функции

еще говорят, что они **вычисляют свое тело**. Такие функции в Лиспе называются **макросами**. Лисп не содержит в своем составе препроцессора, - он просто оказывается ненужным!

Функции-макросы начинают вычисляться, как обычные функции класса **FEXPR**, но **результат вычисления** вновь подается на вход универсальной функции **EVAL**. Введение таких функций в Лисп открывает любопытные возможности.

Вернемся к задаче вычисления суммы элементов произвольного списка (уже решенной выше). Оказывается, с использованием макро задачу о суммировании элементов списка можно решить значительно проще! В самом деле, если имеется числовой одноуровневый список произвольной длины, то для вычисления суммы элементов достаточно вычислить значение S-выражения, которое получается простым добавлением плюса в начало исходного списка.

Для того, чтобы реализовать эту идею, необходимо в теле функции лишь построить список, состоящий из плюса и всех элементов суммируемого списка, а функция **EVAL** будет применена к этому списку автоматически:

```
(defmacro msumlist (x) (cons '+ x))

==> msumlist

(msumlist (1 2 3 4))

==> 10
```

В приведенном выше фрагменте использована функция **DEFMACRO**. Эта функция ничем не отличается от функций **DEFUN** и **DEFUNF**, кроме того, что порождает функции, вычисляемые в два этапа (как описано выше). Аргументы функций типа **MACRO** не вычисляются, поэтому при вызове список **(1 2 3 4)** не квотируется. Результат получается правильным (и заметно быстрее, чем при рекурсивном суммировании!).

При отладке **MACRO** бывает полезно увидеть **промежуточное S-выражение**, которое подается затем на вход **EVAL**. Увидеть это выражение позволяет встроенная функция **MACROEXPAND**:

```
(macroexpand msumlist (1 2 3 4))

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

Функция **MACROEXPAND** принадлежит классу **FSUBR** - ее аргументы не вычисляются. Первый аргумент функции должен быть именем**MACRO**; последующие - параметрами вызова этого **MACRO**. Функция возвращает промежуточное S-выражение (т.н. **макрорасширение**).

Поверхностный взгляд может привести к заключению, что функции типа **MACRO** бесполезны: ведь не составляет большого труда написать обычную функцию

(типа **EXPR/FEXPR**), которая готовит промежуточное S-выражение, а затем явно вызывает **EVAL** для его вычисления:

```
(defun esumlist (x) (eval (cons '+ x)))

==> esumlist

(esumlist '(1 2 3 4))

==> 10
```

Однако, на этом пути есть препятствие. Попробуем чуть усложнить задачу суммирования списка: пусть наша функция суммирования принимает два параметра - список и переменную, которой следует присвоить результат. Попробуем доработать уже написанную функцию**ESUMLIST** вполне очевидным образом:

```
(defun ssumlist (x y) (setq y (eval (cons '+ x))))

==> ssumlist

(ssumlist '(1 2 3 4) 'z)

==> 10

z

Символ z не имеет значения (не связан).
==> ERRSTATE
```

Функция верно считает сумму, но не присваивает результат в качестве значения второму параметру. Понятно, почему это происходит: ведь функция **SETQ** не вычисляет значения первого аргумента. В результате, присвоение выполняется, но значение получает не атом **z**, указанный при вызове, а атом **y** (связанная переменная). При выходе из функции связанная переменная **y** уничтожается, а с атомом **z**ничего не происходит. Исправить ситуацию можно, употребив вместо функции **SETQ** функцию **SET**:

```
(defun ssumlist (x y) (set y (eval (cons '+ x))))

==> ssumlist

(ssumlist '(1 2 3 4) 'z)

==> 10

z

==> 10
```

Все хорошо? Не будем торопиться с выводами... Проверим, как будет работать наша функция, если мы захотим присвоить результат переменной **y**:

```
(ssumlist '(1 2 3 4) 'y)

==> 10

y

Символ y не имеет значения (не связан).
==> ERRSTATE
```

Результат прежний - вычисление верное, но присвоения не происходит. И причина этой неудачи более глубокая: поскольку переменная **y**связана в теле функции **SSUMLIST**, то **отсутствует возможность присвоить из тела функции значение одноименной свободной переменной**. Для решения этой проблемы иногда предлагается давать параметрам функции экзотические названия типа **%1**, **%2** и т.д., в предположении, что никому не придет в голову называть переменные таким образом...

А вот что получится, если вместо функции **EXPR**-типа воспользоваться **MACRO**:

```
(defmacro msumlist (x y) (list 'setq y (eval (cons '+ (eval
x)))))

==> msumlist

(msumlist '(1 2 3 4) s)

==> 10

s

==> 10

(setq a '(1 2 3 4))

==> (1 2 3 4)

(msumlist a xx)

==> 10

xx

==> 10

(msumlist '(1 2 3 4) y)

==> 10
```

```
y

==> 10

(macroexpand msumlist '(1 2 3 4) y)

==> (SETQ y 10)
```

Видно, что функция работает правильно, причем при использовании любых имен. Пусть читатель обратит внимание на то, что в теле макро значение **x** перевычисляется (... **(cons '+ (eval x))** ...) Это позволяет обращаться к функции **MSUMLIST** с переменной на месте первого параметра, но при явном задании списка требуется квотировать первый аргумент. Последний вызов показывает результат макрорасширения. Видно, что собственно суммирование выполняется на этапе подготовки макрорасширения.

Но почему же присвоение переменной **y** дает эффект, недостижимый для функций типа **EXPR/FEXPR**? Все дело в том, что результат макрогенерации выполняется, когда связанные переменные уже уничтожены (как говорят лисперы - в исходном контексте). Поэтому вызов**(SETQ y 10)** присваивает значение не связанной переменной, а свободной. Вот почему и получается правильный результат.

Было бы неверно полагать, что поскольку **MACRO** выполняется в два этапа, выполнение макро-функций требует больше времени, чем вычисление функций типа **EXPR/FEXPR**. Это далеко не всегда так. Например, рекурсивное суммирование списка чисел от единицы до двухсот на компьютере автора выполняется примерно за 60 мсек, тогда, как функция **MSUMLIST** вычисляет тот же результат менее чем за 9 мсек. (почти в 7 раз быстрее). В данном случае это происходит потому, что рекурсивная программа много времени тратит на рекурсивные вызовы, тогда как при использовании макро рекурсии нет совсем, а вычисление происходит за один вызов функции **PLUS**, которая принадлежит классу **SUBR** и работает быстро. Сказанное, разумеется, не означает, что автор противопоставляет макро рекурсивному подходу; просто в ряде случаев макро может оказаться эффективнее.

Одним из наиболее важных применений макро является создание "новых конструкций" в языке. Кавычки здесь употребляются не случайно: макровызов все равно в процессе вычисления преобразуется в одну из базовых конструкций Лиспа.

В качестве иллюстрации сказанного попробуем составить макро, добавляющий к конструкциям базового Лиспа простой "оператор" **IF**. Синтаксис этого "оператора" определим так:

(IF (условие) THEN (...условие истино...) ELSE (...условие ложно...))

Напишем макро, которое реализует требуемые действия:

```
(setq a 100)
```

```
==> 100

(setq b 200)

==> 200

(defmacro if (cc dummy1 tr dummy2 fl) (list 'cond (list cc
tr) (list 'T fl)))

==> if

(if (= a b) Then (setq c a) Else (setq c (/ (+ a b) 2)))

==> 150.0

(if (= a b)
  Then
    (setq c a)
  Else
     (if (> a b)
       Then
           (setq c (/ (+ a b) 2))
       Else
           (setq c (/ (- a b) 2))
     )
)

==> -50.0
```

Легко убедиться, что макро **IF** просто генерирует простую **COND**-конструкцию. Атомы **Then** и **Else** при вызове макро связываются в переменных **dummy1** и **dummy2**. Эти переменые в теле макро никак не анализируются. При необходимости можно было бы ужесточить синтаксис "оператора" **IF**, анализируя значения переменных **dummy1** и **dummy2**.

Из приведенного примера видно, что макро **IF** допускает вложенный вызов и работает правильно. Кстати, понятно, почему при вызове макро не вычисляются аргументы. Если бы это было бы не так, то в приведенном выше примере пришлось бы квотировать атомы **Then** и**Else**, что нельзя признать изящным...

Лисп - язык символьного программирования.

Автор обращает внимание на важное отличие **сути** арифметики Лиспа от арифметики традиционных языков программирования (таких, как С, Паскаль, Бэйсик и т.д.) В традиционных языках для хранения значений служат переменные, которые являются просто областями памяти (ячейками с размерами, соответствующими типу данных). Присвоение переменной значения означает занесение этого значения в соответствующую ячейку. Если переменная **x** имеет значение **1**, а переменная **y** - значение **2**, то оператор **x=x+y** вызовет сложение значений переменной **x** и переменной **y** с последующим занесением результата в ячейку **x**.

В Лиспе все обстоит не так. Конструкция, соответствующая оператору **x=x+y**, в Лиспе

имеет вид **(SETQ x (+ x y))**. При вычислении выражения **(+ x y)** (при значениях **x=1** и **y=2**) получится **3**, что в идеологии Лиспа означает, что **будет создан атом 3**. Именно этот**атом** и будет присвоен в качестве значения атому **x**.

Возникает вопрос: атом **3** может получаться в результате самых разных вычислений, не означает ли сказанное, что в системе может существовать несколько идентичных атомов? Ответ звучит так: **"категорически - нет"!**. Несмотря на то, что любой атом Лиспа может встречаться в самых **разных S-выражениях**, в системе он существует **в единственном экземпляре**. В S-выражении, содержащем атом, хранится не сам атом, а ссылка на него.

Прежде, чем создать новый атом, Лисп-машина проверяет, не существует ли этот атом в системе. Если атом не существует, Лисп-машина его создает. В любом случае, при построении списка или другого S-выражения, содержащего атом, в самом выражении хранится не атом, а ссылка на него.

Все сказанное выше означает, что Лисп является **символьным** языком: первичными объектами Лиспа являются **символы** (атомы). Атомы могут иметь вид привычных человеку чисел, строк и т.д., но могут быть и совершенно произвольными цепочками литер. Символьный характер Лиспа порождает довольно любопытные ситуации, иногда не имеющие аналогов в традиционных (не символьных) языках программирования. Один из таких моментов рассматривается подробно в следующем разделе.

Контекст вычисления в HomeLisp.

Хорошо известно, что в современных языках программирования часто оказывается, что одно и то же **имя переменной** соответствует совершенно разным **значениям** (в зависимости от того, где происходит обращение к переменной, или, как пишут в современной литературе - в зависимости от **контекста вычислений**). Символьный характер языка Лисп усложняет проблему контекста; она требует более глубокого анализа, а непонимание может привести к труднораспознаваемым ошибкам. К сожалению, в известных автору курсах Лиспа затронутой проблеме не уделяется достаточного внимания. Ситуация усугубляется тем, что в разных реализациях Лиспа проблема контекста может решаться по-разному.

Ниже речь будет идти в основном о **HomeLisp**; тонкости других известных автору реализаций будут оговорены явно.

Рассмотрим очень простой пример:

```
(defun g1 (x y) (cons x y))

==> g1

(setq x 1)

==> 1

(setq y 2)

==> 2
```

```
(setq z 3)

==> 3

(g1 x y)

==> (1 . 2)

(g1 y y)

==> (2 . 2)

(g1 z z)

==> (3 . 3)

(g1 'x 'y)

==> (x . y)
```

Здесь определена функция, которая строит из своих аргументов точечную пару. Функция ведет себя абсолютно предсказуемо. Рассмотрим, к примеру, вызов **(g1 x y)**. Поскольку функция **g1** принадлежит классу **EXPR**, то значения аргументов **x** и **y** будут вычислены ядром Лиспа до передачи управления в функцию **g1**; связанные в теле функции переменные **x** и **y** получат значения **1** и **2** соответственно.

Вызов **(g1 'x 'y)** приведет к тому, что связанные в теле функции переменные **x** и **y** получат значения **x** и **y**. В результате вычисления будет получен верный результат **(x . y)**.

А теперь чуть-чуть подправим функцию **g1** и создадим функцию **g2**:

```
(defun g2 (x y) (cons (eval x) (eval y)))

==> g2
```

Отличие заключается в том, что к значениям аргументов дополнительно применяется функция **eval**. Испытаем нашу функцию, предполагая, что переменные **x**, **y** и **z** имеют прежние значения (**1**, **2** и **3**) соответственно:

```
(g2 x y)

==> (1 . 2)

(g2 'x 'y)

==> (x . y)

(g2 'x 'z)
```

```
==> (x . 3)

(g2 'y 'x)

==> (x . y)
```

Результат вызова **(g2 x y)** не отличается от результата вызова **(g1 x y)** и это неудивительно: при вычислении **значений** атомов **x** и **y**получается **1** и **2** соответственно. Связанные в теле функции переменные **x** и **y** получат значения **1** и **2**. И, поскольку **(eval 1)** равно **1**, а**(eval 2)** равно **2**, результат получается такой же, как и при вызове **g1**.

А вот результат вызова **(g2 'x 'y)** уже вызывает удивление: получается, что дополнительные вызовы **eval** в теле функции **g2** не срабатывают. Но на самом деле, при вычислении значений аргументов перед вычислением тела функции **g2** связанные в теле функции переменные **x** и **y** получат значения **x** и **y**. Теперь вызов **(eval x)** вернет результат **x**, а вызов **(eval y)** - соответственно **y**. Из этих значений и образуется точечная пара **(x . y)**.

Еще интереснее результат вызова **(g2 'x 'z)**: атом **x** не заменяется своим значением (по причине, описанной выше), а атом **z** почему-то заменяется... Хотя в этом нет ничего удивительного: перед вызовом **(g2 'x 'z)** происходит вычисление аргументов; получается **x** и **z**. Далее связанная переменная **x** получает значение **x**, а связанная переменная **y** - значение **z**. При вычислении **(eval x)** снова получится **x**, а при вычислении **(eval z)** получится **3**. Этим и объясняется результат.

Последний вызов **(g2 'y 'x)** приводит к совсем парадоксальному результату. На первый взгляд кажется, что должно получиться **(y . x)**, а получается снова **(x . y)**.

Разберемся, почему это происходит.

При вычислении значений аргументов перед передачей управления в функцию **g2** значение первого аргумента (связанной переменной **x**) будет равно **y**, а значение второго (связанной переменной **y**) - соответственно **x**. При вычислении выражения **(eval x)** в теле функции **g2**сначала будет вычислено значение связанной переменной **x**. Поскольку в теле функции переменная **x** имеет значение **y**, то на вход функции **eval** будет подан аргумент **y**. Функция **eval** вычислит значение атома **y**. Оно равно **x**. Первым атомом результирующей точечной пары снова будет **x**. Совершенно аналогично можно убедиться, что вторым элементом результирующей точечной пары будет **y**.

Получается, что функции типа приведенной выше **g2** ("довычисляющие" значения аргументов в теле функции) будут работать **неверно**, если значение фактического параметра совпадет с именем какого-либо формального параметра. В действительности дело обстоит еще хуже: если значение фактического параметра есть **выражение**, содержащие **свободные** переменные, имена которых совпадают с именами **связанных** переменных, то при вычислении в теле функции будут подставлены значения **связанных** переменных. Крайне неприятная ситуация, совершенно немыслимая для традиционных языков типа Паскаля, Си или Бэйсика (несимвольных языков)! Правда, в Лиспе "довычисление" значений аргументов характерно скорее для класса **FEXPR**, но ситуация от этого не становится более легкой.

Очевидный способ борьбы с отмеченной проблемой состоит в том, чтобы выбирать для связанных переменных такие имена, которые не придет в голову использовать при вызове функции (благо в Лиспе имя переменной может содержать любые символы). Этот способ, правда нельзя признать изящным. Пусть читатель представит, что он разработал функцию на Си, в спецификации которой говорится: "не вызывать с фактическим параметром A"... Лавров и Силагадзе в своей книге **[1]** предупреждают читателей, что одна из функций библиотеки будет работать неверно, если, например, имя формального параметра совпадет с именем фактического. Э.Хювенен и Й.Сеппянен **[6]** также отмечают эту проблему и указывают надежный способ ее предупреждения - использовать вместо функции **макрос**.

Определим новую функцию-макрос **g2m** следующим образом:

```
(defmacro g2m (x y) (list 'cons (list 'eval x) (list 'eval
y)))

==> g2m
```

то ситуация нормализуется (предполагается, что переменные **x**, **y** и **z** имеют прежние значения):

```
(g2m x y)

==> (1 . 2)

(g2m y x)

==> (2 . 1)
```

Использование макросов в Лиспе - совершенно законный прием, однако написание и отладка макроса заметно более трудоемкий процесс, чем написание и отладка обычной функции.

Ситуация дополнительно усложняется при использовании конструкции **PROG** и локальных переменных. Рассмотрим следующую врезку:

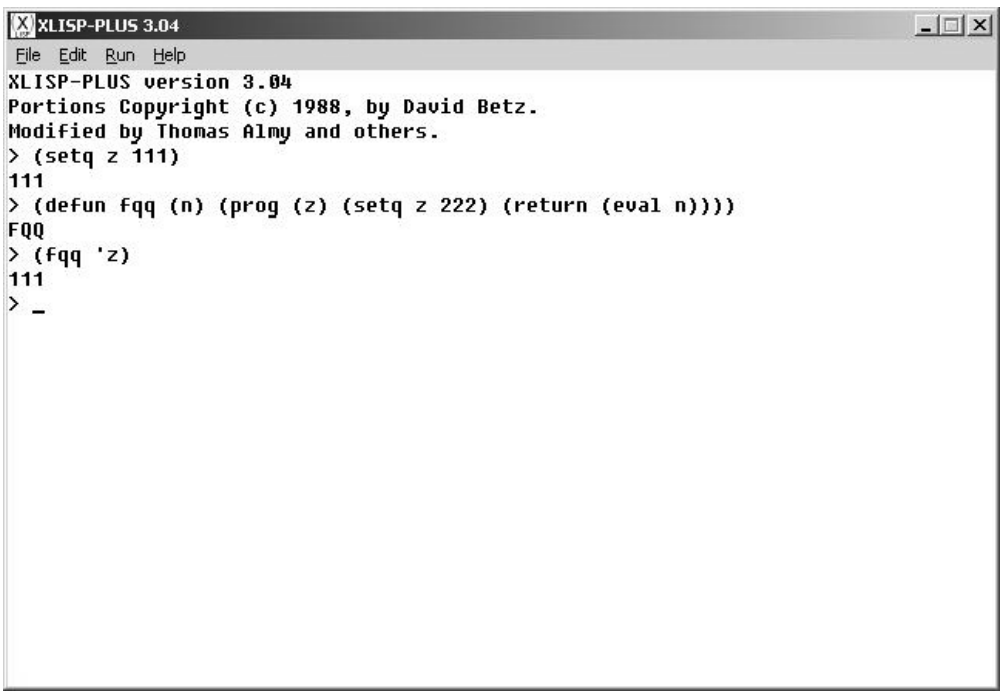
```
(setq z 111)

==> 111

(defun fqq (n) (prog (z) (setq z 222) (return (eval n))))

==> fqq

(fqq 5)

==> 5
```

```
(fqq 'z)

==> 222
```

Здесь тело функции **fqq** представляет собой **PROG**-конструкцию с локальной переменной **z**. Этой локальной переменной в теле функции присваивается значение **222**. Когда функция **fqq** вычисляется со значением аргумента, равным **5**, не происходит ничего неожиданного: результат равен пяти. А вот если вычисляется выражение **(fqq 'z)**, то происходит следующее: сначала связанная переменная **n**(формальный параметр функции) получает значение **z**; когда в теле функции будет выполняться оператор **RETURN**, произойдет обращение к функции **eval** с аргументом **z**. В даный момент атом **z** будет иметь значение **222**, поэтому функция вернет результат **222**. То, что свободная переменная **z** имеет значение **111**, не окажет никакого влияния на результат: значение свободной переменной "перекрыто" значением одноименной локальной переменной.

Снова получается, что значение выражения зависит от места, где оно вычисляется (т.е. от контекста вычисления). Если функция "довычисляет" значения своих аргументов, то имена локальных **PROG**-переменных могут повлиять на результат... Поэтому, некоторые версии Лиспа не поддерживают **PROG**-переменных.

А вот в системах **XLisp** Дэвида Бетца (David Betz) и Томаса Алми (Thomas Almy), и в Российской реализации Лиспа **YLisp** Дмитрия Иванова и Арсения Слободюка аналогичный пример дает другой результат:



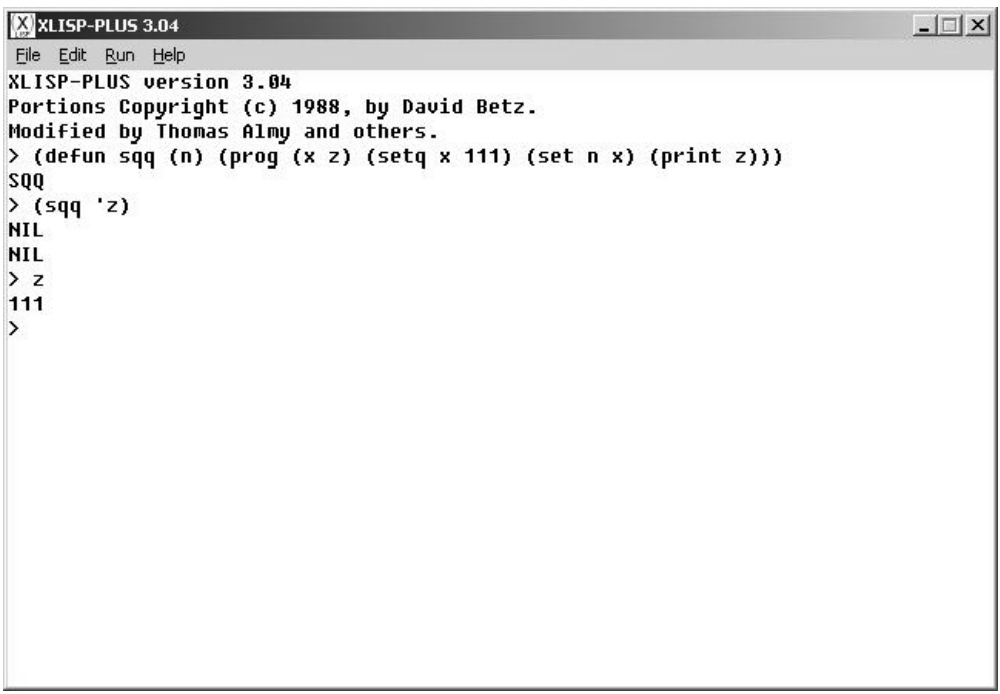
В этих системах использование локальных переменных с любыми именами не окажет влияния на "довычисление" значений аргументов в теле функции. Видно, что при вычислении **(eval n)** использовалось значение **z**, как глобальной переменной.

Вот еще одна достаточно сложная ситуация, которая сводится к проблеме контекста. Рассмотрим следующую врезку:

```
(defun sqq (n) (prog (x z) (setq x 111) (set n x) (printline
z)))

==> sqq

(sqq 'z)
111

==> NIL

z

Символ z не имеет значения (не связан).
==> ERRSTATE
```

Здесь происходит неявное "довычисление" значения параметра в теле функции. Поскольку функция **SET** принадлежит к классу **SUBR**, то при вычислении **(set n x)** ядро Лиспа сначала вычислит значение **n** (при этом получится **z**), затем вычислит значение второго аргумента (получится **111**). Функция **SET** будет вычисляться с аргументами **z** и **111**. Естественно, атом **z** получит значение **111**, что и будет напечатано. Поскольку **z** является локальной **PROG**-переменной, то после выхода из функции атом **z** не будет иметь никакого значения (или не изменит прежнего значения, если до вызова символ **z** был переменной).

Совсем по-другому эта функция будет вычисляться в среде **XLisp**:



Вызов функции **SET** создает здесь переменную, которая сохраняет значение после выхода из функции. Казалось бы, вызов **(set n x)** при значении **n**, равном **z**, должен был бы изменить значение локальной переменной **z**, а вместо этого создается свободная переменная **z**, а локальная переменная не меняет значения. Получается, что **set** действует **в другом контексте**.

Отмеченные расхождения долго не давали покоя автору этих строк. Наконец было принято решение **ввести в ядро языка** две функции**geval** (вычислить в глобальном контексте) и **gset** (присвоить в глобальном контексте).

Функция **geval** работает совершенно аналогично функции **eval** за одним исключением: при вычислении значений атомов сначала проверяется, не существует ли свободная переменная с соответствующим именем. И, если такая переменная существует, используется ее значение; в противном случае используется значение локальной **PROG** переменной. Если же одноименная локальная переменная не найдена, то функция **geval** "полагает", что атом не имеет значения.

Приведенный выше пример, в котором вместо **eval** используется **geval**, дает точно такой же результат, что и в XLisp:

```
(setq z 111)

==> 111

(defun fqq (n) (prog (z) (setq z 222) (return (geval n))))

==> fqq

(fqq 'z)

==> 111
```

Функция **gset** работает аналогично функции **set** за одним исключением: при присвоении значения атому (значению первого аргумента) локальные переменные игнорируются. Если соответствующая глобальная переменная существует - она изменит значение; если глобальной переменной не существует - она будет создана.

Вот как выглядит модифицированная функция **sqq**:

```
(defun sqq (n) (prog (z x) (setq x 111) (gset n x) (printline
z)))

==> sqq

z

Символ z не имеет значения (не связан).
==> ERRSTATE
```

```
(sqq 'z)
NIL

==> NIL

z

==> 111
```

Предполагается, что до вызова **(sqq 'z)** переменная **z** отсутствовала. Вызов **(gset n x)** создает переменную **z** в глобальном контексте. Вызов **(printline z)** не имеет доступа к переменной **z** в глобальном контексте, - он печатает
значение **локальной** переменной **z** (равное, естественно, атому **Nil**). А свободная переменная **z** продолжает жить и после выхода из функции **sqq**.

Динамические и лексические переменные.

Рассмотрим следующий пример:

```
(defun f (x) (+ x y))

==> f

(f 5)

Символ y не имеет значения (не связан).
==> ERRSTATE

(defun g (y) (f 5))

==> g

(g 10)

==> 15
```

Здесь символ **x** связан в теле функции **f** (является параметром), а символ **y** свободен. Если не существует глобальной переменной **y**(созданной вызовом **SET/SETQ** на верхнем уровне), то вычисление **(f x)** завершится ошибкой (т.к. свободный символ символ **y** не имеет значения). При вычислении функции **g** символ **y** получит значение и последующее вычисление значения **f** завершится нормально. Чтобы разобраться, почему это происходит, имеет смысл включить дампирование и посмотреть на пары, возникающие в ассоциативном списке при вычислении выражения **(+ x y)**:

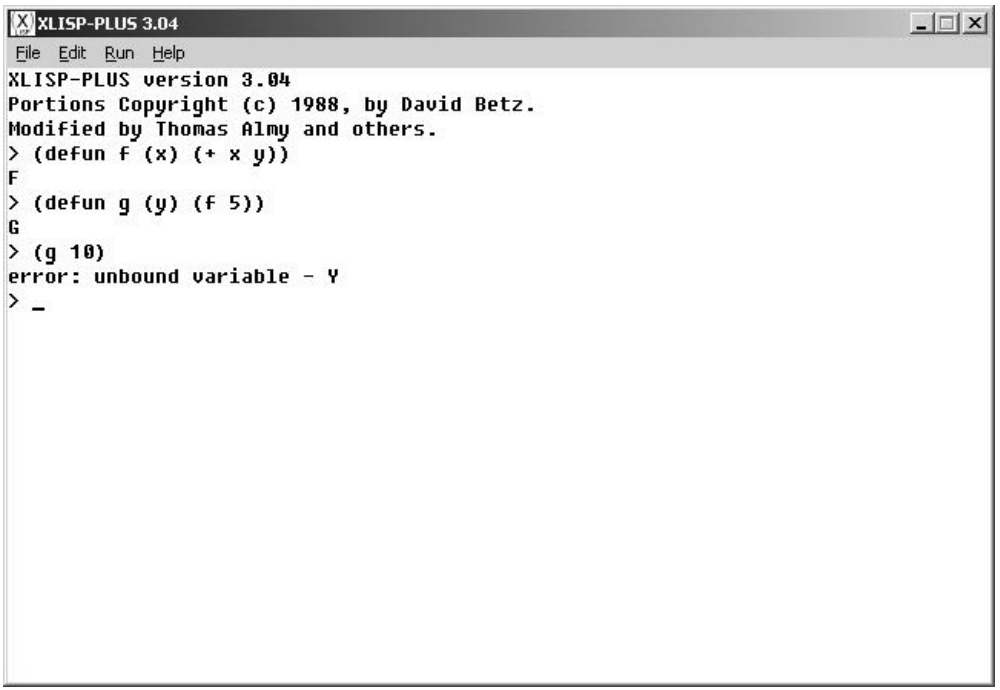
```
........EVLAM вход: A1=((x) (+ x y)) A2=(5)
.........PAIRLIS вход: ptrAasso=1 A1=(x) A2=(5) Flg= 1
.........
......... Ассоциативный список:
.........
```

```
.........  2  -> (x . 5) Параметр
.........  1  -> (y . 10) Параметр
```

Теперь при вычислении значения символа **x** получится **5**, а при вычислении значения символа **y** - 10. Получается, что переменная **y**видима во всех формах, которые вычисляются в течение времени жизни этой переменной. Если переменная создана вызовом **SET/SETQ** на верхнем уровне (является глобальной), то она будет видима во всех вычисляемых формах текущего сеанса.

Переменная, которая видима во всех формах в течение своего времени жизни, называется **динамической**. Как можно убедиться, **все переменые в HomeLisp (по крайней мере в редакциях ядра 11.*.*) являются динамическими**.

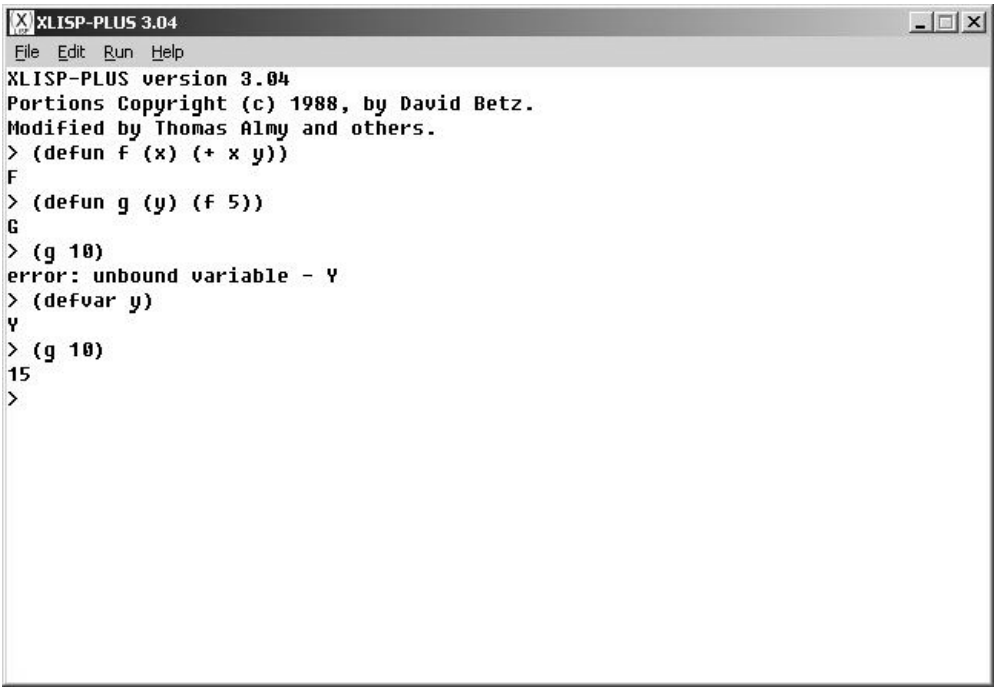
Если попытаться выполнить приведенный выше пример в системе **XLisp** или **YLisp**, то результат будет другим:



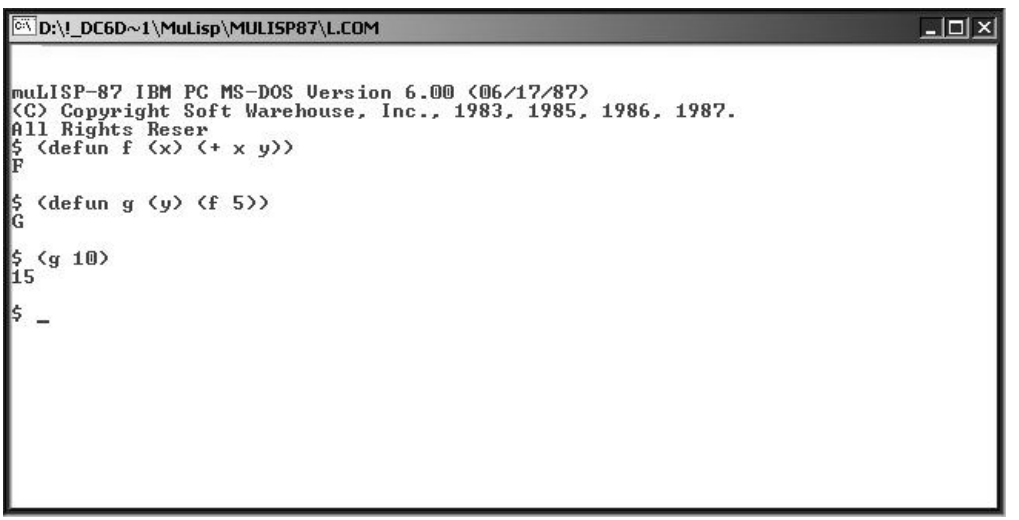
Здесь переменная **y**, связанная в функции **g** оказывается **невидимой** в форме **(f 5)**.

Переменная, которая видима только в форме, в которой определена, называется **лексической**. В системах **XLisp**, **YLisp** (и всех других, следующих стандарту **CommonLisp**) все переменные по умолчанию являются **лексическими**. Соблюдения стандарта **CommonLisp** в части лексических переменных планируется реализовать в редакциях ядра **HomeLisp**, начиная с версии **13.1.***.

Кстати, в стандарте **CommonLisp** лексическую переменную легко превратить в динамическую. Для этого достаточно вызвать функцию**defvar** в форме: **(defvar Имя_переменной)**. После успешного завершения этой команды переменная, имя которой задано при вызове**defvar** начинает вести себя как динамическая:



Пару функций **f** и **g**, приведенную выше, можно использовать в качестве теста, проверяющая тип переменных той или иной реализации Лиспа. Вот пример проверки известной в недалеком прошлом системы **MuLisp**:



Сразу видно, что в **MuLisp** переменные по умолчанию динамические.

Лексические, динамические и глобальные переменные в 13-й редакции ядра HomeLisp

Понятия лексической и динамической переменной принадлежат к числу наиболее тонких понятий Лиспа. В принципе, лексические переменные Лиспа являются аналогами

локальных переменных традиционных языков (Паскаля, Бэйсика, Си). В Лиспе, однако, имеются ньюансы, поэтому сформулировать точное определение лексической переменной не так просто, как может показаться. Именно поэтому, И. Болдырев в статье **[20]** предпочитает определять лексические переменные "от противного" (определяя сначала понятие динамической переменной, и заявляя потом, что все остальные переменные - лексические).

Классическое определение гласит: **Лексической** переменной (или переменной с **лексической** областью видимости является переменная, видимая только в той форме, где она определена. Добавим, что если в форме, определяющей переменную, есть вызов функции, а переменная не входит в список формальных параметров этой функции, то переменная **не будет видима** в теле функции.

Чтобы сказанное не казалось слишком абстрактным, рассмотрим простой пример:

```
(defun f (x) (* x a))

==> f

(let ((a 1)) (f 5))

Внутри LET: Assoc: Символ a не имеет значения (не связан).
==> ERRSTATE

(let ((a 1)) (printline (+ a 7)) (f 5))
8

Внутри LET: Assoc: Символ a не имеет значения (не связан).
==> ERRSTATE
```

Определяется функция **f** с параметром **x**. Символ **a** является свободным по отношению к функции **f**. Далее следует вызов **f** в составе **LET**-предложения. Казалось бы, вызов **(f 5)** должен вернуть **5**, но на самом деле возникает ошибка (**Символ a не имеет значения**). Это происходит потому, что переменная **a** имеет лексическую область видимости - она видима только в форме **LET**; видимость "не просачивается" в тело вложенного вызова. Следующая строка показывает, что переменная **a** в **LET**-предложении "жива"; можно вычислить и напечатать **a+1**, но в теле функции **f** переменная **a** невидима.

Такое поведение переменной **a** вовсе не связано с тем, что эта переменная создана при помощи **LET**-конструкции. Вот другой пример, дающий тот же результат:

```
(defun g (a) (f 5))

==> g

(g 1)

Assoc: Символ a не имеет значения (не связан).
==> ERRSTATE
```

Здесь переменная **a** (равная после вызова **g** единице) снова невидима в теле функции **f**. Описанное и задает суть понятия **лексическая видимость**. Можно несколько "оживить" данное выше определение:

Лексическая переменная, это такая переменная, которая, будучи свободной относительно функции, невидима в этой функции.

Все переменные в **HomeLisp** по умолчанию являются лексическими.

В противоположность лексическим переменным, **динамические** переменные видимы во всех вызовах функций. Чтобы сделать переменную динамической, нужно вызвать функцию **DEFDYN** или **DEFVAR**. Отличие между этими функциями состоит в том, что **DEFDYN** устанавливает у своего аргумента признак "динамической области видимости", не трогая значение аргумента, тогда как **DEFVAR** создает динамическую переменую и сразу же присваивает ей значение (подробности - в разделе, посвященном **DEFVAR**).

Если переменную **a** из предыдущих примеров сделать динамической, то результаты вычислений изменятся:

```
(defdyn 'a)

==> a

a

Assoc: Символ a не имеет значения (не связан).
==> ERRSTATE

(g 1)

==> 5

(let ((a 1)) (printline (+ a 7)) (f 5))
8

==> 5
```

Легко убедиться, что переменная **a** теперь видима в теле функции **f**. Следует отметить, что превращение переменной в динамическую вызовом **DEFDYN** не присваивает этой переменной никакого значения; попытка вычислить значение **a** завершается ошибкой. Важно отметить, что став динамической, переменная будет оставаться таковой до завершения работы Лисп-машины.

В **HomeLisp** в дополнение к динамическим есть также **глобальные** переменные. Если переменная объявлена глобальной, то она будет видима в любой точке программы (как и положено глобальной переменной). Создать глобальную переменную можно либо вызовом функции **DEFGLOB**, либо вызвав **SET/SETQ** на верхнем уровне. При этом "глобальность" и "лексичность/динамичность" не исключают друг друга. Если глобальная переменная не

объявлена динамической, то она будет вести себя, как лексическая: установленное выше значение не "просочится" в вызов, а будет просто взято глобальное значение. Глобальная динамическая переменная практически не отличается от просто динамической переменной.

Вот комплексный пример:

```
(setq a 1)

==> 1
Создана глобальная переменная a

(defun f (x) (* x a))

==> f

(defun g (a) (f 5))

==> g

(g 111)

==> 5

(defdyn 'a)

==> a

(g 111)

==> 555
```

Сначала создана глобальная переменная **a**. Затем создается функция **f** относительно которой переменная **a** свободна. Затем создается функция **g** и вызывается с параметром **111**. В теле **g** значение переменной **a** будет равно **111**, однако, поскольку **a** - не динамическая переменная, в функцию **f** попадет значение **глобальной** переменной. Если же сделать **a** динамической переменной, то при вызове **f** будет передано значение **111** (и получится соответствующий результат). Отличить лексическую переменную от динамической можно с помощью функции **SPECIALP**.

Списки свойств атомов.

При изложении основ Лиспа предполагалось, что ядро Лиспа каким-то образом "знает", что некоторые атомы являются числами, некоторые - строками, некоторые - битовыми константами и т.д. Сейчас пришло время объяснить механизм этого "знания".

Каждый атом **может иметь** т.н. **список свойств**. Атомы-числа, атомы-строки, атомы-битовые константы имеют этот список в обязательном порядке. Имеют список свойств и атомы-имена функций (всех типов - от **SUBR** до **MACRO**). Нетрудно догадаться, что этот список связывается с атомом **при его создании** ядром Лиспа. Как устроены эти списки? Очень просто. При инициализации ядра вместе с самоопределенными атомами типа **Nil**, **T**,

создаются стандартные **атомы-флаги**:

- **SUBR** - признак встроенной функции на языке ядра Лиспа, вычисляющей аргументы;
- **FSUBR** - признак встроенной функции на языке ядра Лиспа, не вычисляющей аргументы;
- **EXPR** - признак функции, написанной на Лиспе, вычисляющей аргументы;
- **FEXPR** - признак функции, написанной на Лиспе, не вычисляющей аргументы;
- **APVAL** - признак наличия определяющего выражения;
- **FIXED** - признак целого числа;
- **BITS** - признак битовой шкалы;
- **STRING**- признак строки;
- **FLOAT** - признак числа с плавающей точкой;
- **MACRO** - признак функции типа макро;
- **WINDOW**- признак графического окна;
- **FILE** - признак файла;
- **BLOB** - признак большого двоичного объекта;
- **COM** - признак COM-объекта;
- **DIALOG**- признак диалога;
- **CONTROL** - признак элемента управления

Когда создается атом, являющийся числом, битовой константой или строкой, то ядро Лиспа сразу же строит список свойств, содержащий соответствующие флаги. Ссылка на этот список хранится вместе с атомом. Для других атомов соответствующий признак попадает в список свойств, когда выполняется функция создания соответствующего программного объекта (диалога, графического окна, файла и т.д.)

Чтобы увидеть список свойств атома нужно вызвать функцию **PROPLIST**. Эта функция показывает список свойств своего единственного аргумента, который должен являться атомом. Вот примеры вызова функции **PROPLIST**:

```
(proplist 7)

==> (FIXED)

(proplist 7.0)

==> (FLOAT)
```

```
(proplist &H7)

==> (BITS)

(proplist "7")

==> (STRING)

(proplist 7,0)

Символ 7,0 не имеет значения (не связан).
==> ERRSTATE

(proplist '7,0)

==> NIL

(proplist 'b)

==> NIL

(proplist nil)

==> (APVAL)

(proplist T)

==> (APVAL)

(proplist _WHITE)

==> (BITS)
```

Из приведенной врезки вполне очевидно, что атом **7** имеет тип **FIXED**, атом **7.0** имеет тип **FLOAT**, атом **&H7** имеет тип **BITS**, а атом **"7"** - тип **STRING**. Как видим, список свойств числовых, битовых и строковых констант состоит из единственного элемента.

Некоторое удивление может вызвать результат вычисления **(proplist 7,0)**. Но ничего сверхъестественного не произошло. Цепочка символов **7,0** является **атомом** Лиспа, но этот атом (не являющийся корректным изображением числа) не имеет значения, а функция**PROPLIST** принадлежит классу **SUBR**; попытка **вычислить** значение атома **7,0** вызывает ошибку. Квотирование исправляет ситуацию. Список свойств атома **7,0** пуст. Точно так же пустым оказывается и список свойств атома **b**. А вот списки свойств атомов **Nil** и **T**оказываются непустыми - они содержат атом **APVAL**. Интересно отметить, что список свойств атома **_WHITE** содержит атом **BITS** (другими словами, атом **_WHITE** есть битовая константа). Это нисколько не удивительно, поскольку можно убедиться, что атом **_WHITE** создается вызовом **(csetq _WHITE &HFFFFFF)**).

Рассмотрим еще несколько примеров вызова функции **PROPLIST**:

```
(proplist 'car)
```

```
==> (SUBR)

(proplist 'quote)

==> (FSUBR)

(defun f (x y) (* x y))

==> f

(proplist 'f)

==> (EXPR)

(defunf g (x y) (list x y))

==> g

(proplist 'g)

==> (FEXPR)

(proplist 'for)

==> (MACRO)
```

Хорошо видно, что атом **CAR** есть имя функции типа **SUBR**; атом **QUOTE** - имя функции типа **FSUBR**; атом **f** - имя только что созданной функции типа **EXPR**; атом **g** - имя только что созданной функции типа **FEXPR**, а атом **for** - имя функции типа **MACRO**.

Все сказанное выше не означает, что список свойств атома может строиться **только** ядром Лиспа. Имеется встроенная функция **SPROPL**, которая может задать список свойств у **любого** атома. Эта функция принимает два параметра. Первый параметр должен быть атомом, а второй - списком. После возврата значение второго параметра **значение** второго параметра **замещает** список свойств атома, заданного первым параметром. При этом стандартные флаги **не создаются и не замещаются**. Вот примеры вызова функции **SPROPL**:

```
(spropl 'a1 '(q w e r t y))

==> (q w e r T y)

(proplist 'a1)

==> (q w e r T y)

(spropl 2 '(a b c))

==> (a b c)

(proplist 2)
```

```
==> (FIXED a b c)

(+ 2 2)

==> 4
```

В первом примере у атома **a1** устанавливается список свойств **(q w e r t y)**. Последующий вызов функции **PROPLIST** показывает, что список успешно установлен. В следующем примере делается попытка установить список свойств у атома **2**. И попытка оказывается успешной: последующий вызов **PROPLIST** показывает, что у атома **2** список свойств изменился. Стандартный флаг **FIXED** при этом никуда не делся, - как было отмечено выше, стандартные флаги неуничтожимы. Пусть читатель приведет еще какой-нибудь язык программирования, в котором у **двойки** могут быть **еще** какие-либо свойства! (Разве что **Forth**. Может быть, и до него доберемся...) Кстати, наличие списка свойств нисколько не мешает арифметике - последний пример это подтверждает: **2 + 2** по-прежнему **4**.

Список свойств атома может иметь сколь угодно сложную структуру, однако обычно он является одноуровневым списком.

В принципе двух описанных функций (**PROPLIST** и **SPROPL**) достаточно для любых манипуляций со списками свойств. При необходимости другие функции можно реализовать на Лиспе (см. **PUTFLAG**, **PUTPROP**, **FLAG**, **FLAGP**).

Важное замечание

Совершенно естественным применением списков свойств могло бы быть моделирование математических объектов (рациональные числа, векторы, матрицы и т.д.). К сожалению, использование списков свойств здесь не так удобно, как может показаться на первый взгляд. Проблема состоит в том, что каждый атом в Лиспе уникален, соответственно уникален и список его свойств. А вот **значения** атома зависят от **контекста вычислений**. Если некий атом используется как локальная переменная, то его значение при выходе из **PROG**-конструкции восстановится. Но если внутри **PROG**-конструкции у этого атома **модифицировался** список свойств, то этот список свойств **не изменится**при выходе из **PROG**. Подобная ситуация совершенно недопустима, поэтому в главе, посвященной приемам программирования на Лиспе, признаки создаваемых объектов (массивов, рациональных чисел и т.д.) хранятся в общем списке с данными, составляющими сущность моделируемого объекта.