ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

Федеральное государственное
бюджетное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
им. проф. М. А. БОНЧ-БРУЕВИЧА

Т. В. Ермакова
Т. В. Губанова

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Часть 2
УЧЕБНОЕ ПОСОБИЕ

СПб ГУТ)))

САНКТ-ПЕТЕРБУРГ
2018

«Стиль – это отражение мышления»
А. Шопенгауер

“*Функциональное программирование ставит своей целью придать каждой программе простую математическую интерпретацию*”
Лоренс Паулсон

Введение

Языки программирования, которые мы используем для написания программ, всегда отражают не просто некоторую технологию, но *способ мышления программиста.*

В части 1 учебного пособия по ФЛП мы познакомились с различными стилями программирования и провели их краткий сравнительный анализ. (рис.2.) В той же части мы подробно разобрали логический стиль программирования.

Часть 2 учебного пособия по ФЛП посвящена изучению функционального стиля программирования. В последнее время и функциональное, и логическое программирование часто относят к *декларативному стилю* программирования, так как основным методом и в том, и в другом стилях является описание объектов без определения последовательности действий в отличие от императивного стиля программирования, в котором в явном виде (с помощью выполняемых операторов присваивания) определяются как сами действия, так и последовательность из выполнения.

1. Пространство различных парадигм программирования и их сравнительная оценка.

Пространство различных парадигм программирования разделено на два пересекающихся множества (рис.1.), что определяет возможность использования методов и инструментов различных парадигм программирования при решении задачи любым из этих стилей.

пространство различных парадигм программирования

Рис. 1

К *императивному* стилю принято относить *процедурную вычислительную модель* (структурное программирование (СП) и объектно-ориентированное программирование (ООП)). К декларативному стилю относят функциональное (аппликативное) программирование (ФП) и логическое (продукционное) программирование (ЛП).

Императивная модель связана с именами Алана Тьюринга, Джона фон Неймана. Декларативная модель – с именами АлонзаЧерча, Хаскелла Карри, Роберта Ковальского, Аллена Колмероэ и др..

Термин "*парадигма* " используется нами для определения *модели вычислений.* Модель вычислений в свою очередь определяет способ структурирования информации, организации вычислений и данных.

Вспомните курс " Алгоритмические основы программной инженерии" (АОПИ). Из него мы узнали, что понятие алгоритма связано с точным описанием специального класса функций, называемых рекурсивными. В то же время числовые функции, значения которых можно вычислить посредством алгоритма, называются вычислимыми функциями. При этом следует отметить, что понятие рекурсивной функции является математически строгим.

Так как в алгоритмических проблемах речь идет обычно не об алгоритмах, а о вычислимости некоторых специальным образом подобранных, решающих проблему функциях, то можно строго доказать, что если решающая конкретную задачу функция не может быть рекурсивной, то не существует и алгоритма решения этой задачи.

Мы показали в курсе АОПИ, что можно уточнить понятие алгоритма через *класс рекурсивных функций*. Сами рекурсивные функции были описаны Геделем, а затем и Черч пришел к тому же классу функций. Черч высказал гипотезу о том, что класс рекурсивных функций тождественен классу всюду определенных вычислимых функций.

В силу тезиса Черча вопрос о вычислимости функции равносилен вопросу о ее рекурсивности. Есть и другое направление, которое уточняет понятие алгоритма. Оно связано с именем Тьюринга. Основная идея его заключается в том, что алгоритмические процессы могут имитироваться на подходяще построенных машинах, которые описываются в точных математических терминах. Получается, что сложные процессы можно моделировать на простых устройствах. В этой ситуации алгоритм можно задать определенной функциональной схемой и реализовать его с помощью машины Тьюринга.

В курсе математики мы познакомились с достаточно богатой системой логики под названием исчисление предикатов первого порядка, которое позволяет производить весьма общие рассуждения, причем их можно запрограммировать для вычислительной машины, и тогда компьютер также сможет рассуждать в достаточно общем виде. Логика предикатов является основой логического программирования, которому и была посвящена первая часть учебного пособия по ФЛП.

2. Функциональное программирование (ФП)

ФП – раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значения функции *в математическом* понимании. Это значит, что на функцию мы смотрим, как на некоторый математический объект. В ФП при вызове функции с одними и теми же аргументами мы *всегда* получим одинаковый результат. Это значит, что выходные данные зависят только от входных данных. Что это нам дает? Это позволяет *кешировать* (сохранять) результаты функций и вызывать их тогда, когда это надо нам, а не в порядке, определяемом алгоритмом. Также можно распараллеливать их без каких-либо дополнительных действий со стороны программиста. А это как раз то, что нам обеспечивают функции без побочных эффектов. Возможность кеширования результатов – это один из способов оптимизации и увеличения скорости выполнения компьютерной программы. Для этого перед вызовом функции проверяется, вызывалась ли она ранее, и если вызывалась, то используется сохраненный результат.

Мы видим, что главным в парадигме функционального программирования является понятие “функция”, при этом есть языки, в которых используются функции “высших порядков”, которые дают функции в качестве результата и передают их другим функциям. Отсюда, собственно, и название – функциональное программирование. Из курса математики мы с вами знакомы с применением теоретических и практических аппаратов, позволяющих оперировать функциями. (Сюда можно отнести дифференцирование, интегрирование, функциональный анализ, теорию нечетких множеств и т. п.). Функциональные языки программирования позволяют нам связать *математическую теорию функций и технологию создания программного обеспечения (ПО)*, то есть мы получаем механизм, который дает нам возможность управлять поведением компьютера.

С точки зрения математики функция выражает связь между входными данными и результатом некоторого процесса, то есть позволяет нам отображать (*mapping)* элементы одного множества из области определения функции в другое множество – множество значений этой функции. А так как вычисление тоже является процессом, у которого есть входные данные и результат, то функцию можно считать вполне подходящим средством для описания вычислений. В процессе выполнения программы функция получает определенные параметры, вычисляет и возвращает полученный результат. Функции определяются через другие функции или рекурсивно – через самих себя. Все, что необходимо сделать программе для получения желаемого результата при решении задачи, должно быть представлено в виде системы взаимосвязанных функций. Это во многом определяет отличие функциональных языков программирования от императивных языков программирования. Одно из основных отличий заключается именно в том,

что порядок вычисления выражения, задающего отображение, управляется рекурсивными и условными выражениями, в то время как в императивных языках приходится работать с помощью итерации и последовательного выполнения операций.

Еще раз заметим, что важным свойством математических функций является то, что они всегда определяют одно и то же значение при заданном наборе аргументов, так как они не имеют побочных эффектов, которые обычно в процедурных языках связаны с переменными, моделирующими ячейки памяти. В функциональных языках функция определяет значение, а не задает последовательность операций над числами, хранящимися в ячейках памяти. А раз нет переменных, то нет и побочного эффекта.

Как мы уже отмечали, выполнение функции при одних и тех же параметрах всегда приводит к одному и тому же результату. Это делает семантику чисто функциональных языков значительно проще семантики языков, обладающих императивными свойствами.

2.1 λ *-исчисление Черча - математический механизм для описания вычислительных процессов.*

λ -исчисление Черча - это математическая абстракция, которая составляет базис почти всех языков функционального программирования, и не только функционального. Это – теоретическая база для описания и вычисления функций. Близкое к нему теоретическое понятие – *комбинаторная логика,* объединившись с *λ-исчислением* дают ясное и четкое описание принципов и основ математики, на которых и построено здание функционального программирования. Почему мы вспомнили все то, что пройдено нами в курсе АОПИ ? Потому, что работая на компьютере, нас с вами интересуют *вычислимые функции*.

Тьюринг показал, что вычислимость в терминах машины Тьюринга эквивалентна собственно возможности лямбда-описания системы. Клини доказал, что эта возможность лямбда-описания системы также эквивалентна рекурсивности Гёделя. Таким образом, если задача сформулирована с помощью лямбда-исчисления, то она имеет решение за конечное время. (Следует понимать разницу между чистым лямбда-исчислением и языками функционального программирования. Утверждение, что любая сформулированная на Lisp задача вычислима – неверно).

Таким образом, тезис Черча- Тьюринга, связанный с лямбда - термом, в нашей ситуации является одним из возможных определений вычислимости функций.

Тезис Черча - Тьюринга в его интерпретации для лямбда - исчисления гласит: *любая функция может быть определена при помощи соответствующего лямбда - терма*.

А это есть формальное обоснование возможности функциональной парадигмы программирования в качестве системы для проведения вычислений. Как и все прежние тезисы, этот тезис невозможно строго доказать или опровергнуть, так как он устанавливает “равенство” между *строго формализованным понятием частично рекурсивной функции и неформализованным понятием “ интуитивно вычислимой функции”*. Этот тезис очень важен при реализации многих языков программирования, которые основаны на лямбда- исчислении. Важен он и для Лиспа. Следует заметить, что большинство языков программирования (не только функциональных) основаны на лямбда- исчислении, просто многие языки расширены дополнительными синтаксическими конструкциями для упрощения написания функций. Питер Ландин еще в 1965 году писал, что лямбда - исчисление предоставляет возможности для представления в языках программирования *абстракции и аппликации*. Вот почему *формализм лямбда- исчисления фундаментален.*

Итак, *формализм лямбда - исчисления (λ) (lambda calculus) является фундаментом в области компьютерных наук и теории языков программирования*.

2.2 Вклад Алонзо Черча, Хаскелла Карри и Джона МакКарти в появление ФП и языка Лисп

Создателем лямбда-исчисления является Алонзо Черч. Интересна история его создания.

λ - исчисление не предполагалось использовать в качестве теории вычислений. Его создатель Алонзо Черч разработал *λ - исчисление* в целях разрешения парадокса Рассела. Мы сталкивались с парадоксом Рассела в курсе “Алгоритмические основы программной инженерии”.

Рассел рассматривал множество всех множеств, не принадлежащих самим себе, в качестве подмножества: $R=\{x \mid x \notin x\}$. При таком определении множества R выражение $R \in R$ истинно тогда и только тогда, когда $R \notin R$, и наоборот. Классическая теория множеств Кантора на этом парадоксе давала сбой, показывая свою противоречивость. Но Черч не смог разрешить парадокс Рассела с помощью *λ - исчисления*. Его разрешил Гильберт с помощью аксиоматической теории множеств. Именно эта теория оказалась устойчивой к парадоксу Рассела. Тут вмешался *Хаскелл Карри*, который первым понял, как *λ - исчисление* можно использовать в качестве теории вычислений. Но чтобы теория вычислений не осталась теорией, потребовалось еще вмешательство *Джона МакКарти*, который разработал язык Lisp.

Мы встречались с вами с понятием функции и в императивных языках программирования. Мы и там создавали наши собственные функции.

Но сам стиль создания и составления таких программ в функциональных языках программирования совершенно особенный.

В функциональном программировании вы имеете возможность принимать функцию в качестве аргумента, но можете и возвратить функцию в качестве результата.

Выдающийся ученый Джон МакКарти предложил концепцию символьной обработки на основе идей лямбда- исчисления Черча и разработал язык Лисп – первый функциональный язык.

Все данные, участвующие в работе программы, представляются в форме *символьных выражений*.(S-выражений).

Именно благодаря этому программист освобождается от распределения памяти под отдельные блоки данных, а программа использует для хранения таких структур всю доступную ей память.

МакКарти предложил рассматривать функции как некоторое общее понятие, к которому в программировании можно свести все другие понятия.

2.3 Рекурсивные функции и методы их реализации в системах программирования.

И здесь особо важно выделить рекурсивные функции и методы их реализации в системах программирования. Элегантность и лаконизм использования рекурсивных функций мы наглядно видим и в функциональном, и в логическом программировании. В функциональном программировании программы строятся из рекурсивных функций над символьными S- выражениями. Любая информация, в том числе определения функций и их вызовы, имеют вид S-выражений, а это означает, что они могут обрабатываться как обычные данные.

Лисп несмотря на то, что это один из старейших языков программирования, не утратил своей актуальности. Более того, мы сейчас работаем с языками, которые постарались заимствовать некоторые преимущества Лиспа. Это и Perl,и Python , и Ruby и др.

2.4 Особенности и преимущества Лиспа перед другими языками.

Лисп имеет целый ряд важных преимуществ перед другими языками:

- В Лиспе заложена постоянная возможность развиваться.
- В Лиспе можно определять новые операторы.
- При появлении каких-либо новых абстракций, они всегда могут быть реализованы в языке.
- Мы можем определять свои функции по собственному усмотрению, и эти функции могут стать такой же частью языка, как и лю-

бая другая стандартная лисповская функция.(например, +, * или eql.)

- На Лиспе мы легко можем модифицировать язык, подстраивая его под нашу конкретную задачу.
- Одна из интереснейших особенностей Лиспа – это возможность разрабатывать программы, которые будут писать другие программы. Это связано с тем, что все знания, представляемые в Лиспе – это списки.

  Программы, написанные на Лиспе представляются в виде его же структур данных. Их называют *макросы.*
- В качестве интересных необычных инструментов можно назвать и *замыкания.*

Это позволяет языку всегда находиться на передовых позициях.

Функции, которые создаются для расширения возможностей языка Лисп называют *утилитами*. Пример: написать функцию, которая принимает число n и возвращает функцию, которая добавляет n к своему аргументу.

```
(defun addn(n)          ; пример лексического замыкания
  #' (lambda (x)
    (+ x n)))
```

Пока это просто пример. Разбираться с замыканиями мы будем чуть позже. (Но попробовать решить эту же задачу на С вы можете уже сейчас. Это должно заставить вас задуматься.)

Замечание1

Уникальные особенности Лиспа -

- *интерактивность*
- *автоматическое управление памятью*
- *динамическая типизация*
- *замыкания (см. пример выше)*

Преимущества Лиспа заимствуются другими языками (Perl, который используется как скриптовый язык в Unix- подобных системах, Python, Ruby). Механизм *макросов* лежит в основе синтаксиса Лиспа, то есть он изначально разрабатывался таким, поэтому этот механизм трудно заимствовать. Реализации могут быть разными, с ними придется экспериментировать. Макросами мы тоже займемся позднее.

Замечание 2 Основные стили программирования

Прежде чем заняться лямбда- исчислением, вспомните, что мы знаем про исчисления.

Исчисление – это *совокупность правил* оперирования какими- либо символами. Например, дифференциальное исчисление- это совокупность правил нахождения касательной к кривым путем преобразования алгебраических выражений. В логическом программировании мы используем ограниченную форму исчисления предикатов первого порядка в виде логики хорновских дизъюнктов, которые являются логическими импликациями.

По сути, это и есть правила типа “если – то”. Наша программа на Прологе написана с помощью хорновских дизъюнктов.

В курсе АОПИ мы познакомились с теорией вычислимости и теорией сложности. Обе эти теории привели нас к модели вычислений как определению множества допустимых операций, использованных для вычислений, а также к необходимой оценке времени выполнения программы, объема памяти, ограничений самих алгоритмов или компьютеров, то есть вычислительных ресурсов.

Но узнать, какое количество необходимых вычислительных ресурсов понадобится, можно только выбрав определенную модель вычислений.

С моделями вычислений мы тоже уже знакомы. Вспомним четыре основные парадигмы программирования, представленные на рис.2. Алгоритмы, используемые в этих моделях, реализуются с помощью языков четырех типов – императивных, логических, функциональных и объектно-ориентированных. Вычислительная модель, в которой программа рассматривается как множество определений функций, называется функциональной моделью.

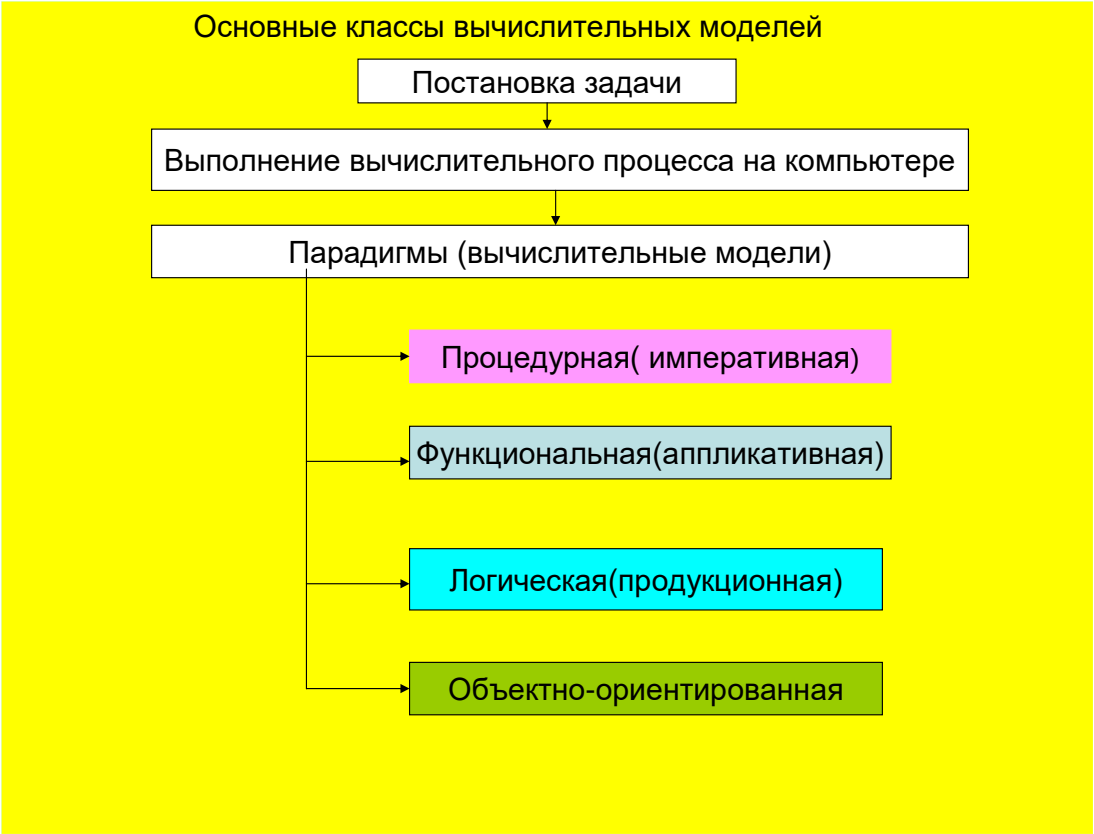


Рис. 2. Основные стили программирования

Наблюдая за развитием ЯВУ мы видим, что даже довольно абстрактные языки типа Java, C++ все еще наследуют свойство императивности. Однако здесь появляются и декларативные конструкции. Например, класс. Описание класса – это, конечно, декларация, но реализация методов класса осталось явным описанием императивного (процедурного) процесса. Мы приступаем к изучению языков, имеющих функциональный характер. Главным в парадигме функционального программирования является понятие функции, являющееся вполне подходящим средством описания вычислительного процесса. Объединяясь с логическим программированием, функциональное программирование тоже представляет собой парадигму декларативного программирования. Это означает, что функциональные программы носят описательный характер.

2.5 λ- исчисление и комбинаторная логика как теоретическая основа науки о вычислениях

Парадигма ФП возникла на базе двух математических направлений:

- λ- исчисление Черча (лямбда-исчисление)
- Комбинаторная логика Хаскелла Карри

Функциональное программирование основано на идее, что в результате каждого действия возникает значение. Это значение становится аргументом следующего действия. Конечный результат всей задачи выдается пользователю. Программа состоит из отдельных *определений функций*. Каждое определение состоит из *управляющих структур*, организующих порядок вычислений и из вложенных *вызовов функций* (часто рекурсивных).

- *Комбинаторную логику (*КЛ*) Хаскелл Карри* создавал для формализации вычислительных процессов. Это наука о природе подстановок, когда один объект подают на вход других, в результате чего *на основании определенных правил* получают новые объекты. Это наука об объектах и способах их комбинирования. В КЛ существует лишь одна операция – *апплика-ция* (применение). Аппликативный подход означает, что изначально нет разделения математических объектов на функции и аргументы, а есть только объекты одинаковой природы. *Эти объекты в зависимости от контекста могут использоваться по-разному – и в качестве функций, и в качестве аргументов.*

λ-исчисления Черча - это математический формализм, используемый при описании функций (по сути функциональная система обозначений). Лямбда-исчисление рассматривает применение функции и вычисление лямбда - выражений с помощью подстановки.

Базовым понятием в комбинаторной логике является *комбинатор.* Комбинатором называется некоторый константный заранее определенный объект, не содержащий внутри себя свободных переменных. Комбинатор декларирует правило комбинирования объектов друг с другом. Это значит, что комбинатор определенным образом комбинирует (склеивает) объекты, которые приложены к этому комбинатору. Существует несколько базовых комбинаторов, с помощью которых можно построить любые другие возможные комбинаторы. Их используют для описания различных формальных систем. Мы их используем для функционального программирования. Поэтому и говорят, что в комбинаторной логике существует единственная операция – применение (аппликация), то есть приложение одного объекта к другому. А система вычислений, основанная на применении комбинаторов друг к другу, носит название аппликативной вычислительной системы.

Комбинаторы, в сущности, играют роль связки, которая скрепляет программу, а любое имя функции или константы встречается теперь в качестве аргумента комбинатора. Функции получают теперь значения аргументов *на основе своих позиций в матрице комбинаторов, а не на основе имен этих аргументов*.

Это решает проблему с переменными, которая может возникнуть, если одно и то же имя может использоваться в различных контекстах. Для создания новых термов используются правила вывода, которые определяют характеристики отношения конвертируемости (=). В комбинаторной логике

используются пять правил вывода, обозначаемых строчными буквами греческого алфавита:

1.	***a=a;***
2.	***a=b=>b= a;***
3.	***a=b , b=c=>a=c;***
4.	***a=b=>ca=cb;***
5.	***a=b=>ac=bc.***

Наборы аксиом для вывода новых термов могут отличаться в зависимости от выбранного базиса. Именно базис определяет, через какие базовые комбинаторы будут выражаться остальные объекты. Самих базисов может существовать бесконечно много. Приведем в качестве простейшего базиса базис **K,S:**

Аксиомы для вывода термов в базисе **K,S** имеют вид:

Kxy=x;

S x y z=xz(yz)

При помощи комбинаторов можно моделировать итеративные и рекурсивные вычислительные процессы. Для этого Х. Карри разработал комбинатор неподвижной точки **Y.** Неподвижной точкой функции f называется такое значение x из области определения f, что f(x)=x. Например, если $f=x^2$, то неподвижными точками такой функции являются значения 0 и 1, так как $0^2=0$ и $1^2=1$.

Комбинаторов неподвижной точки существует неограниченное множество. Самый простейший комбинатор неподвижной точки, выраженный в базисе **K,S** :

$$Y_s = SSK(S(K(SS(S(SSK))))K).$$

Комбинаторная логика - это путь к аппликативному стилю мышления, где изначально нет разделения математических объектов на функции и аргументы, а все объекты имеют одинаковую природу. *А как будут рассматриваться эти объекты - как функции или как аргументы – зависит только от контекста*. Именно комбинаторная логика, положенная в основу функциональных языков, позволяет нам:

- синтезировать новые объекты с заданными свойствами,
- производить и использовать частичные вычисления,
- оптимизировать вычисления с помощью комбинирования параметров,
- создавать суперкомбинаторы, которые могут стать объектами для проведения “ленивых” вычислений некоторых выражений, а также решать ряд других немаловажных вопросов, например, с помощью типизации комбинаторов.

Лямбда - исчисление - это теоретическая основа функционального программирования. Аппарат λ - исчисления служит обоснованием и практической возможностью решать целый ряд задач:

- моделирование связывания переменных и ограничение их видимости
- возможность вызова функций по имени, *по значению (строгие вычисления)* и *по необходимости (ленивые вычисления).* В функциональном программировании ленивое вычисление (оно называется так, потому что при вызове по необходимости выполняется с задержкой) является очень важным понятием. С его помощью можно не только избежать ненужных оценок, но и *обрабатывать данные бесконечной структуры и бесконечные потоки данных.*

  При ленивом вычислении оценка аргумента функции выполняется не в момент применения функции, а при выполнении функции, когда возникает необходимость в его значении. Это значит, что оценка аргумента производится только один раз при первом использовании, а затем везде используется это первое полученное значение.
- λ - исчисление – это по сути универсальная машина Тьюринга. В соответствии с тезисом Черча все вычисляемые функции могут быть представлены при помощи λ -терма.
- Средства λ - исчисления позволяют смоделировать все обычные структуры данных, в том числе позволяют вычислять бесконечные объекты.
- λ - исчисление оперирует λ - термами и применяет к ним единственную операцию – аппликацию при которой вместо формальных параметров (связанных переменных) в тело λ -терма вносятся конкретные значения. Отсюда видно, что λ - исчисление очень похоже на комбинаторную логику.

В результате в ФП λ - исчисление оперирует λ - термами, применяя к ним единственную операцию – аппликацию, или подстановку, когда вместо формальных параметров (за счет связывания переменных) в тело λ -терма вносятся конкретные значения. Аппликация в λ- исчислении – это естественный способ описания частичного применения. И комбинаторная логика, и λ - исчисление оперируют объектами одинаковой природы (только в комбинаторной логике нет дифференциации объектов на термы и переменные, а в λ - исчислении такая дифференциация есть).

λ -терм – это выражение, которое принимает одну из следующих форм:

- x – переменная;
- λx . M – абстракция, где x- переменная, а M - λ -терм, при этом для того, чтобы указать, что именно представляет собой вычислительный процесс, описываемый λ -термом, можно использовать любые средства, предоставляемые математическим аппаратом.

- В λ -терме λx . M – переменная x называется связанной, если она присутствует внутри M, а выражение M называется телом абстракции. В теле может присутствовать свободные переменные, которые не связываются в абстракции при помощи символа λ.

  Например, λx. (x y) -> переменная x –связана, y – свободна

  λ -терм λx . M представляет собой описание функции f : f(x)=M для любых x из ее области определения. Применение функции f к некоторому значению N представляет собой подстановку N вместо формального параметра x. Результатом будет являться λ -терм M, в котором все вхождения переменной x заменены на значение N: (λx . M) [x <- N])
  - (MN) –аппликация, где M и N - λ -термы. (рекурсивное определение)

На этой основе и построены функциональные языки программирования, такие как Lisp, Haskell и др. Из лямбда-исчисления взяли абстрактное и простое лямбда- выражение. Это дало возможность естественным образом описывать функции и передавать им параметры. Так как применение функции к аргументу (аппликация) позволяет применять только уже имеющиеся функции, то ввели понятие *конструктора* (оператор для конструирования) новых функций. В λ - исчислении эту роль играет *греческая буква* λ : m=λ x . 2*x+3.

Применим эту функцию к разным числам:

m(6)=15

m(1)=5

p=λ x . if x>4 then m(x) else -x

p(10)=23

p(1)= - 1

В действительности λ - *выражение обозначает неименованную функцию.* Самой важной была идея об одинаковом представлении программ и данных в виде списка. Именно благодаря этому функции можно передавать в качестве параметров другим функциям. Стало также возможным *порождение новых программ во время их исполнения* (а это по сути есть *обучение и адаптация).* Только в языках типа Lisp или Haskell , основанных на λ - *исчислении* есть возможность *создавать новые функции во время исполнения программы.*

Связь лямбда- исчисления с комбинаторной логикой.

Итак,

Алонзо Черч разработал λ *- исчисление. Хаскелл Карри ввел* в рассмотрение *комбинаторную логику*, которая упростила вариант λ - исчисления. Джон МакКарти *разработал функциональный язык программирования Lisp*, который перевел теоретические исследования в практическую плоскость. Этот язык жив и широко используется. (Это происходит потому, что функции, определяемые пользователем, выглядят так же, как и примитивы, и это приводит к нескончаемым попыткам расширения языка).

Таким образом, само по себе функциональное программирование является частью дискретной математики, которое потом благодаря МакКарти нашло непосредственное отображение в прикладной области технологии программирования.

Lisp наряду с логическими языками широко применяется :

- в системах искусственного интеллекта,
- в качестве языка макросов в системах ПО (текстовый редактор *EMacs*, система автоматизированного проектирования *Autocad)*;
- большинство компиляторов функциональных языков реализовано на *Lisp*. Это естественно, так как что лучше компиляторов приспособлено к повторному использованию?

λ - *исчисление* интересно своей простотой для формализации функций, потому что языка функционального программирования Лисп вполне достаточно для вычисления всего, что *в принципе* можно вычислить.

λ - *исчисление* популярно не менее машины Тьюринга или цепей Маркова.

3. ФУНКЦИОНАЛЬНАЯ ВЫЧИСЛИТЕЛЬНАЯ МОДЕЛЬ

В этой модели программа рассматривается как *множество определений взаимосвязанных функций.*

Функции определяются через другие функции или рекурсивно через самих себя. В основу положена математическая модель, называемая "*лямбда-исчислением " Черча.* Эта модель позволяет с помощью простых синтаксиса и семантики представлять все функции функционального программирования. По сути лямбда-исчисление является естественным выражением вычислительного процесса.

Для представления обычной функции используется выражение f(x).При этом неясно: то ли оно означает функцию f,то ли ее значение при заданном параметре x. Для четкого описания функции было введено выражение λxf(x).То есть, когда хотят рассматривать выражение P как функцию от x, следует использовать запись λxP (лямбда-абстракция). Таким образом, выражение λx(x+y) является функцией от x, а не от y. При этом x называют связанной переменной, а y- свободной переменной.

Таким образом, главным в парадигме функционального программирования является понятие функции (отсюда и название – функциональное программирование Теперь появился раздел, который напрямую связал математическую теорию функций и технологию создания программного обеспечения (ПО) для компьютеров.

Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. То есть, функцию можно рассматривать как некоторый черный ящик, входные значения которого порождают значения на выходе (выражают связь между входом и выходом некоторого процесса).

Так как вычисление – это тоже процесс, имеющий вход и выход, то функция является вполне адекватным средством описания вычислений.

Именно этот простой принцип и положен в основу функционального программирования. Функции определяются через другие функции или рекурсивно – через самих себя. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций.

Все, что необходимо сделать программе для получения желаемого результата при решении задачи, должно быть представлено в виде системы взаимосвязанных функций.

Lisp (list processing – обработка списков) - первый, но не единственный язык функционального программирования.

3.1 Языки функционального программирования:

- ***ML(****Meta Language)* – основан на типизированном λ - *исчислении и исчислении предикатов первого порядка.* Самый важный диалект – *Standard ML*, из которого вышли языки семейства *Caml*. ***Caml*** - язык программирования общего назначения (Диалекты- *Caml Light, Objective Caml*).
- ***Miranda*** – нестрогий функциональный язык программирования, своими идеями связанный с языками *ML* и *Hope*. Использует частичные вычисления, алгебраические типы, ленивые вычисления, двумерный синтаксис.
- ***Haskell*** –современный мощный язык функционального программирования. Берет начало в языке *Miranda*.
- ***Clean*** (синтаксис похож на синтаксис *Haskell*, но имеет другой способ вычислений)
- ***Curry*** – объединяет две парадигмы – функциональную и логическую
- ***Erlang*** - язык общего назначения. Используют для систем параллельных вычислений и организации взаимодействия между процессами. Возможна переносимость между платформами исполнения программ. Язык разработан компанией Ericsson для телекоммуникационных проектов. Имеет много прикладных библиотек для решения практических задач из области сетевого взаимодействия и телекоммуникаций - библиотеки OPT.

Кроме этого можно назвать языки ***Hope, ISWIM, Unlambda, Рефал***. Универсальный робототехнический язык **GRL**(Generic Robot Language) для создания больших модульных систем управления. Программы на GRL компилируются в эффективные программы на таких языках, как C. Язык **CES**(C++ for embedded [im'bedid] systems) – C++ для встроенных систем. В нем объединены вероятностные средства и средства обучения. **ALisp** применяет средства определения правильного действия с помощью индуктивного обучения – обучения с подкреплением. С его помощью предполагают создавать роботов, способных к обучению в результате взаимодействия со своей средой.

Модель вычислений – *функция*, а не отношение, как в языках логического программирования. Язык – функциональный, использует аппликацию. Аппликация – это *операция применения функции к аргументу*. Результат выражается в терминах функции. Вместо цикла или итерации используется *рекурсивный вызов функции*.

В математике функция – это отображение (*mapping*) между множествами, которое однозначно отображает одни значения на другие. Например, y=f(x) ставит в соответствие каждому элементу x из множества определения единственный элемент y из множества значений функции f. У функции может быть произвольное количество аргументов, в том числе их вовсе может не быть. Функция, не имеющая аргументов, будет иметь постоянное значение. В общем случае функция может задавать отображение из не-

скольких множеств в множество значений: f(x,y)->z – отображение пар элементов x,y в множество z. Типы аргументов x, y , а также тип значения z можно обозначить следующим образом:

f : A*B -> C. Знак “ * “ означает прямое произведение множеств.

Это значит, что тип первого аргумента функции f принадлежит множеству A, тип второго аргумента принадлежит множеству B, а тип значения – множеству C. Функция f ставит в соответствие упорядоченным парам, образованных из элементов множеств A и B ,элемент из множества C.

3.2 Композиция функций. Составной оператор

Пусть есть функции f и g , такие, что f: D -> R и g: R->S ,где D и R – области определений функций f и g соответственно, а R и S –области значений этих функций. Можно определить новую функцию g.f : D->S, которая называется композицией функций g и f : (g.f) (x) = g(f(x)).

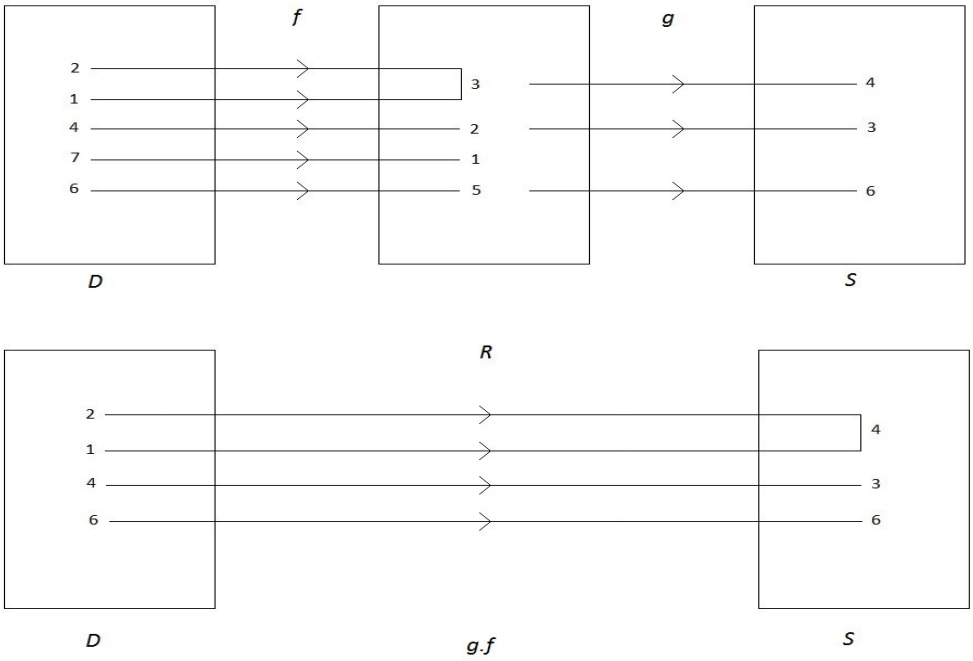


Рис. 3 Композиция отображений

Отсюда: D={2,1,4,7,6} R={3,2,1,5} S={4,3,6} f={<2,3>,<1,3>,<4,2>,<7,1>,<6,5>} и т.д. Обобщим понятие композиции функций:

Пусть имеются две функции: z=h(y) и y=g(x) , причем область значений функции g принадлежит области определения функции h. Тогда функция z=h(g(x)) называется композицией функций h и g, или сложной функцией, или суперпозицией функций. Аналогично можно рассматривать

композицию любого конечного числа функций. Композицией двух преобразований называют последовательное выполнение этих преобразований, а также полученное в результате этого итоговое преобразование, что мы и видели на рисунке выше.

Например, Составим из двух функций y=f(u) и u=g(x) сложную функцию y=f(g(x)) y=(sin lnx) -> эта функция получена суперпозицией функций –натурального логарифма и синуса.

Замечание

Суперпозиция - это функция, полученная из некоторого множества функций путем подстановки одной функции в другую (что в сущности и есть *композиция функций).* Аппликация в нашем случае представляет собой применение функции к аргументу.

Аппликация позволяет применять только уже имеющиеся функции, поэтому нужен *конструктор* (оператор для конструирования) новых функций. В λ-исчислении роль конструктора играет греческая буква λ, а сам процесс называется λ - абстракцией.

Например, можно образовать функцию одного аргумента x, поставив символ lambda перед этим аргументом, точку(.) после аргумента и выражение, дающее значение функции, после точки:

```
    g = lambda x . 2 * x+ 3
Применим эту функцию к разным значениям аргумента:
    g(1)=5
    g(3)=9
```

Лямба - выражение изображает параметризированные вычисления. Лямбда-вызов соответствует вызову функции:

```
(лямбда-выражение  a1 a2 …. an )
```

```
((lambda (x y)
   (* x y))
 5 2)
10
```

Применение лямбда - выражения к фактическим параметрам производится в два этапа. Сначала вычисляются значения фактических параметров и соответствующие формальные параметры связываются с полученными значениями. На следующем этапе с учетом новых связей вычисляется форма, являющаяся телом лямбда - выражения, а полученное значение воз-

вращается в качестве значения лямбда - вызова. Формальным параметрам после окончания вычисления возвращаются те связи, которые у них, возможно, были перед вычислением лямбда - преобразования. *Лямбда- выражение – это безымянная функция, которая пропадает сразу после вычисления значения формы.* Но ее можно использовать при передаче функции в качестве аргумента другой функции, или при формировании функции в результате вычислений, то есть при синтезе программ. λ-выражение в действительности обозначает неименованную функцию. λ -выражение можно поместить в любое место, куда мы могли бы поместить функциональный символ. λ -выражение может иметь функции - аргументы и выдавать функции в качестве результата. В обычном программировании мы встречали функции, допускающие аргументы - функции, но не функции, дающие результаты - функции. Еще раз напомним: только в языках типа Лисп, основанных на λ -исчислении, есть возможность создавать новые функции во время исполнения программы.

3.3 Особенности функциональных языков

1. В чистых аппликативных языках не используются присваивания. Если переменная получила значение, то оно не может быть изменено. Могут создаваться новые структуры, но при этом сохраняются старые. Выражение строится с помощью аппликации, где результат, выданный одной функцией, подается на вход другой.
2. Возможно использование алгебраических подстановок для упрощения и оптимизации. Программа в функциональной форме похожа на обычные алгебраические выражения, которые легко упрощать и преобразовывать.
3. Функции сами по себе могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата. Это называется *функциями высших порядков*. В математике в качестве аналога может быть приведен интеграл или производная.
4. Программы значительно короче и проще, чем императивные. Кроме этого у нас есть возможность интерактивной разработки, а это означает, что в программу легко внести исправления и не надо как в императивных языках проводить редактирование, компиляцию и длительное тестирование.
5. Отсутствуют побочные эффекты и присутствует детерминированность. Это называется – чистые функции. Они зависят только от *своих параметров* и возвращают *только свой результат*. Если результат такой функции не используется, то его можно удалить без вреда для других выражений. Если функция снова вызывается с этим же параметром, то возвращается тот же результат. Если между этими двумя

функциями нет никакой зависимости по данным, то порядок их вычисления можно менять, то есть их можно распараллелить.

6. Возможны отложенные (*ленивые*) вычисления. Это характеристика исключительно функциональных языков. Это свойство говорит о том, что могут отсутствовать вычисления в случаях, когда результат этих вычислений не требуется. Если функциональный язык не поддерживает отложенных вычислений, то его называют *строгим*. Языки, использующие отложенные вычисления, называют *нестрогими или ленивыми языками*.(например, Lisp,Haskell, Miranda, Gofer)

3.4 Применение функциональных языков

Решение задач в рамках *искусственного интеллекта*, то есть постановка задач на компьютере без точной формализации. Математические формулы просто переводятся на функциональный язык, после чего в дело вступает компилятор. Для задач искусственного интеллекта характерна работа с символьной информацией. В таких задачах обрабатываются чаще всего рекурсивные типы данных, такие как списки, деревья, графы и другие структуры, которые могут быть приведены к ним.

Сейчас приходится работать с очень крупными формированиями, такими как

- *маршрутизация сетей,*
- *многоканальный обмен информацией,*
- *многоуровневые протоколы,*
- *работа с многопроцессорными системами* и т. п.

Переход к работе со столь сложно устроенными данными, требует более глубокого абстрагирования, к чему идеально приспособлен функциональный стиль программирования. Такие языки как Лисп очень подходят для моделирования задач системного программирования.

Программы в функциональной форме похожи на обычные алгебраические выражения. Корректность таких программ легче доказать с помощью стандартных математических приемов. Эти программы легко поддаются распараллеливанию**.**

Итак, лямбда-исчисление рассматривает применение функций и вычисление лямбда - выражения с помощью подстановки.

4. Основы языка Лисп

4.1 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, напи-

санные 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).

Лисп-машина – это программа, исполняющая команды Лиспа.

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.

4.2. 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".

4.3 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.

Таблица 1 – Правила построения атомов Лиспа

Строка	Описание
Abc	Возможный вариант атома
1Abc	Возможный вариант атома (имя может начинаться с цифры)
Q$W	Возможный вариант атома (имя может начинаться с цифры)
123	Возможный вариант атома - число
-12.3	Возможный вариант атома - число
6.02E+23	Возможный вариант атома - число
A.A	Возможный вариант атома
A A	*Это не может быть атомом, так как пробел в имени недопустим!*
A(	*Это не может быть атомом, так как скобки в имени недопустимы!*
A'B	*Это не может быть атомом, так как апостроф в имени недопустим!*
()	Возможный вариант атома
"Первая программа"	Это - атом-строка. Внутри строки пробелы вполне допустимы
"Первая "программа""	*Это - не атом. Кавычки, стоящие внутри строки, при записи должны удваиваться.*
"Первая ""программа"""	Теперь верно.
"Первая 'программа'"	Можно и так. Апостроф внутри строки - обычный символ.
&HFFFFFF	Это - атом-битовая шкала.
&H1122334455667788	Это - просто атом

4.4 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 2, в которой представлены все типы точечных пар, а также приведены ошибочные построения.

Таблица 2 – Правила построения точечных пар Лиспа

Строка	Описание
(a . b) ((1 . 2) . b) ((1 . 2) . (3 . 4)) (x . (y . z)) (name . "Анатолий")	Правильные точечные пары
(1 .)	Не точечные пары после точки до скобки должен стоять атом или точечная пара
(. 2)	после скобки до точки должен стоять атом или точечная пара
(1 . 2	скобки должны быть сбалансированы

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.

4.5 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.

4.6 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.

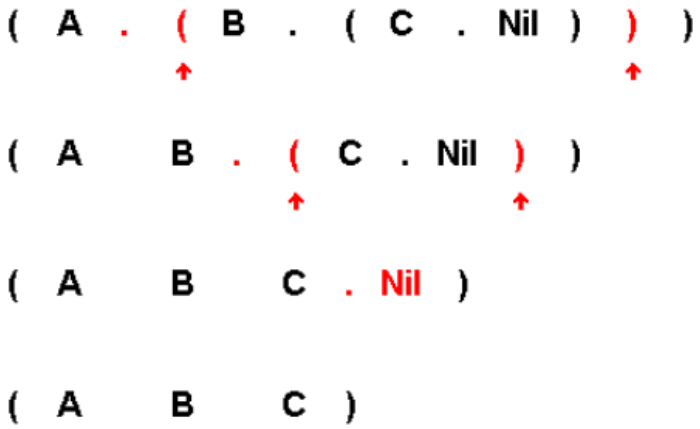
Этих правил всего два:

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

Рассмотрим применение этих правил к записи S-выражения:

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

На приведенной ниже схеме 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, 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. 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.

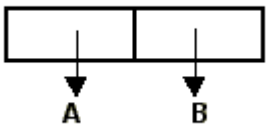
- Часть списка, за исключением головы называется **хвостом** списка. Если кроме головы список не содержит других элементов, то хвост такого списка есть пустой список.

4.7 Внутреннее представление списков

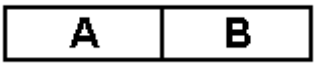
Оперативная память, в которой хранятся S-выражения, обычно делится на две больших области: список объектов и область списочных ячеек. В списке объектов хранятся атомы. Каждый атом занимает блок памяти переменного размера. В этом блоке хранится символьное изображение атома и ряд его дополнительных характеристик. 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.

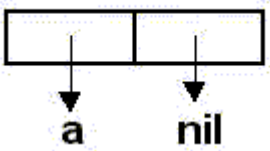
Графически точечную пару изображают так:



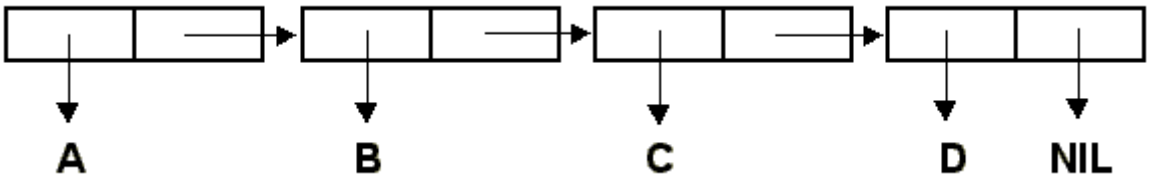
или так:



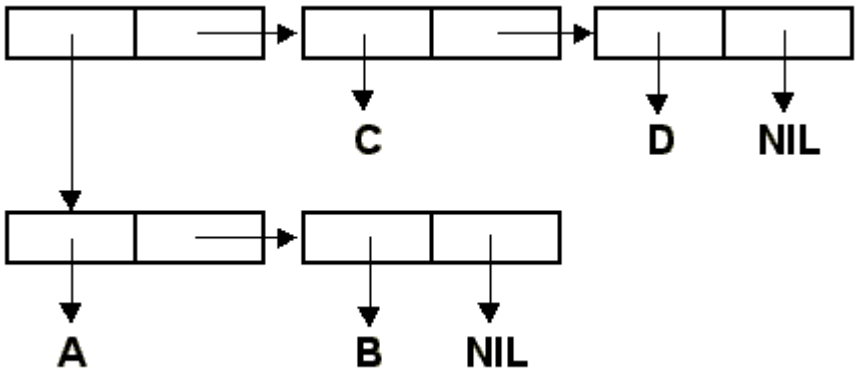
А как можно представить графически список (A)? Поскольку список (A) есть точечная пара (A . Nil), то графическое изображение строится сразу:



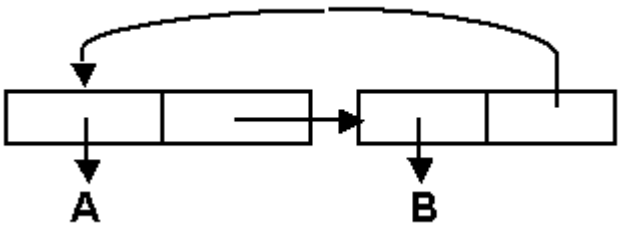
Не представляет труда построить графическое изображение и более сложных списков. Так, например, список (A B C D) устроен так:



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:



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. Говоря о внутреннем представлении S-выражений, следует добавить, что каждый атом уникален. Даже если в каком-либо списке атом A встречается многократно, в списке объектов он представлен в единственном экземпляре (т.е. во всех списочных ячейках из которых исходят стрелки, указывающие на атом A, будет содержаться один и тот же адрес). Синтаксис S-выражений на первых порах проще всего представлять, как формальную знаковую систему.

4.8. 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.

4.9 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 аргу-

мент 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)).**

4.10 Классификация функций Лиспа

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. Отмеченная проблема решается в Лиспе двумя способами: введением специального класса функций, которые не вычисляют свои аргументы, а также выборочным использованием функции QUOTE. 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.

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

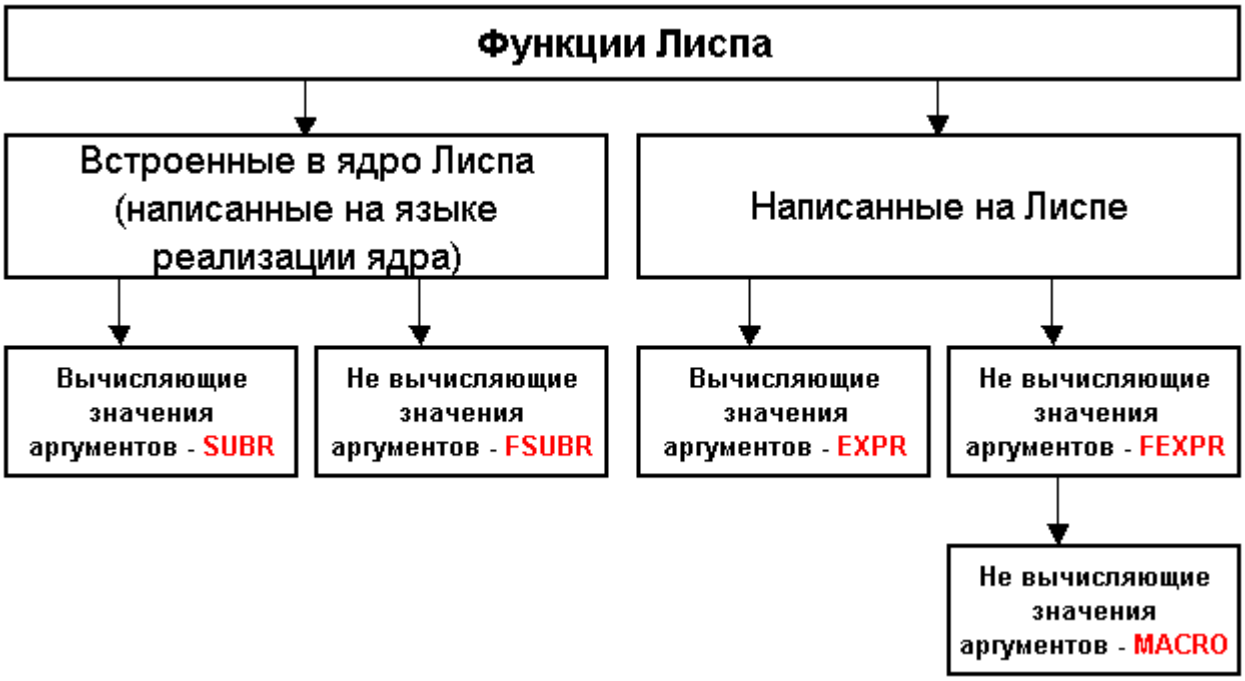


Рисунок 4 – Классификация функций Лиспа

4.11 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). Пользователь вводит S-выражение в область ввода и получает ответ в области вывода. 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 Lisp возвращает в качестве результата специальный атом ERRSTATE (ведь результат должен быть тоже S-выражением!). Если вы-

числения 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.

4.12 Блокировка вычисления

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. Например, существует функция SUMLIST, вычисляющая 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.

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 правилам Лиспа вместо того, чтобы писать:

(QUOTE Нечто)

допустимо писать:

'Нечто

Далее 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 являет-

ся самоопределенный атом, то квотирование не требуется (хотя и не приведет к ошибке, а только чуть удлинит вычисления).

4.13 Присвоение значений атомам

Произвольному атому Лиспа можно присвоить значение.
Функция SET класса SUBR требует ровно два аргумента. Значениемпервого 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.

4.14 Разбор списков на составные части

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.

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 '(1 2 3))

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 в таблице 4

Таблица 4 – Комбинации CAR и CDR

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

4.15 Построение списков из составных частей

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 жирным):

((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 3соответственно, 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.

4.16 Проверка на атомность

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.

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

В третьем примере 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).

4.17 Сравнение атомов

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

4.18 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 Lisp; 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.

4.19 Арифметические функции

Лисп умеет выполнять все привычные действия с числами: сложение, вычитание, умножение, деление, возведение в степень, сравнения и т.д. Все арифметические функции принадлежат к классу SUBR - они встроены в ядро Лиспа и вычисляют значения всех своих аргументов. В приведенной ниже таблице 5 представлены элементарные арифметические функции.

Таблица 5 – Арифметические функции

Имя функции	Знак	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

4.20 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.

4.21 Создание собственных функций

Язык программирования, который не позволяет программисту создавать собственные функции, бесполезен. В Лиспе пользователь может создавать свои функции трех классов - **EXPR**, **FEXPR** и **MACRO**. 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 вместе, получим:

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

Можно создать функцию, тело которой не обращается к формальным параметрам. Однако, такая функция при любых параметрах давала бы один и тот же результат. 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

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 функ-

ции **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

4.22 Рекурсия

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. 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 повоторное 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.

Чтобы изучить работу рекурсивной функции, удобно использовать возможность Лиспа, называемую **трассировкой**. Трассировка заключается в том, что при входе в трассируемую функцию печатаются значения вход-

ных 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**.

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))))))
```

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 бу-

дет 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).

4.23 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, **Lisp** 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.

4.24 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
(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 функцию 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
```

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. 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)

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**.

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:

(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

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.

4.25 Парадигмы программирования на Лиспе

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...**.

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
...

)

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 сказаь, 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** 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**.

Приложение А
Примеры программирования на Лиспе

1) Построение списка целых

Одной из первых задач, которую предлагают студенту или школьнику при изучении Паскаля, является задача заполнения массива последовательными целыми числами от единицы до заданного **N**. Напомним решение этой задачи:

- заводим массив **M**;
- заводим целую переменную **i**;
- в цикле по **i** от единицы до **N** **i**-му элементу массива величину **i** (**M[i]:=i**).

Аналогичным образом можно поступить и в Лиспе, но это решение не будет **лисповским** по духу. Приведем рекурсивное, **исконно лисповское** решение задачи.

Будем рассуждать так:

- Очевидно, что если **N=0**, то результирующий список должен быть пустым списком (список из нуля элементов);
- Пусть у нас уже построен список из **N-1**-го числа (обозначим его L_{n-1}). Тогда очевидно, что список из **N** чисел получается из списка L_{n-1} простым добавлением в хвост числа **N**: $L_n = L_{n-1} + (N)$.

Как можно убедиться, приведенные выше рассуждения в точности повторяют выкладки метода математической индукции. Теперь осталось воплотить эти выкладки в программу на Лиспе. Объединение списков выполняет функция **append**, а построить список из одного элемента можно с помощью функции **list**. Собирая все вместе, получаем:

```
(defun alist (n) (cond ((= n 0) Nil)
                       (T (append (alist (- N 1)) (list N)))
                 )
)
==> alist
(alist 100)
==> (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
    23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
    42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
    61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
    80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
    99 100)
```

Функция работоспособна. Теперь усложним задачу. Предположим, что требуется построить список не всех чисел от единицы до заданного **N**, а список всех чисел из интервала N_1 - N_2. Повторим наши рассуждения, применительно к новой задаче:

- Очевидно, что если $N_1 = N_2$, то результирующий список должен быть списком из одного элемента (неважно, какого N_1 или N_2);
- Пусть у нас уже построен список чисел из диапазона $N_1 - (N_2-1)$. Тогда очевидно, что искомый список получается из последнего просто добавлением в хвост числа N_2.

Этот подход реализуется на Лиспе вполне очевидным образом:

```
(defun clist (n1 n2) (cond ((= n1 n2) (list n1))
                           (T (append (clist N1 (- N2 1)) (list N2)))
                     )
)
==> clist
(clist 5 55)
==> (5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
     27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
     47 48 49 50 51 52 53 54 55)
```

2) Чередование элементов двух списков ("два гребня")

Пусть даны два произвольных списка. Обозначим элементы этих списков A_i и B_i соответственно. Длины списков могут различаться. Требуется построить новый список в котором элементы исходных списков чередовались бы следующим образом: A_1 , B_1 , A_2 , B_2 ... и т.д. до исчерпания более короткого списка. Допуская вольность, можно представить себе результат, как два два гребня, вставленных один в другой.

Рассмотрим лисповское (функциональное) решение этой задачи. Назовем функцию, которая будет строить список-результат **grb**. У этой функции будет два параметра (исходные списки); обозначим их **a** и **b**. Представляется очевидным, что результирующий список будет совпадать со списком **a**, если список **b** пуст. Соответственно, если пуст список **a**, то результат будет совпадать со списком **b**. Если же оба исходных списка не пусты, то результат можно представить как объединение следующих двух списков:

- двухэлементного списка, состоящего из **первых элементов** списков **a** и **b**;
- результата применения функции **grb** к **хвостам** списков **a** и **b**.

В Лиспе это выглядит следующим образом:

```
(defun grb (a b) (cond ((null a) b)
                       ((null b) a)
                       (T (append (list (car a) (car b)) (grb (cdr a) (cdr b))))
                 )
)
==> grb
(grb '(1 2 3 4 5 6) '(a b c d e f g h))
```

==> (1 a 2 b 3 c 4 d 5 e 6 f g h)

Определяющее выражение функции получается дословным переводом на Лисп приведенных выше математических рассуждений (что является большим преимуществом рекурсивного функционального подхода).

3) Получение n-го элемента списка

Встроенные в Лисп функции **CAR** и **CDR** позволяют "разобрать" произвольный список на составные части. Однако, в стандартный набор функций Лиспа не входит функция, позволяющая получить элемент списка по его номеру. Попробуем заполнить этот пробел. Построим функцию **n-th**, имеющую два параметра: список и номер извлекаемого элемента. Очевидно, что если входной список пуст, то функция должна возбудить состояние ошибки (у пустого списка нет элементов). Далее, если запрошен **первый** элемент непустого списка, то он просто равен **голове** этого списка. И, наконец, если запрошен n-й элемент непустого списка, то он в точности равен (n-1)-му элементу хвоста исходного списка.

Воплощаем все сказанное в Лиспе:

```
(defun n-th (x n) (cond ((null x) (raiseError "n-th: список пуст(исчерпан)"))
                        ((= n 1) (car x))
                        (T (n-th (cdr x) (- n 1)))
                  )
)
==> n-th
(n-th '(1 2 3) 5)
n-th: список пуст(исчерпан)
==> ERRSTATE
(n-th '(q w e r t y) 4)
==> r
(n-th '(q w e r t y) -4)
n-th: список пуст(исчерпан)
==> ERRSTATE
```

Использованная выше функция **RAISEERROR** позволяет установить состояние ошибки и задать текст разъясняющего сообщения. Приведенное сообщение может возникнуть в двух случаях: когда явно запрашивается элемент пустого списка или когда запрашиваемый элемент списка не существует (у трехэлементного списка запрошен, например, пятый элемент). Может показаться, что построенная нами функция не защищена от зацикливания в случае, если она будет вызвана с отрицательным или нулевым вторым аргументом. Однако это не так, что подтверждается последним

примером запуска. Дело в том, что параллельно с вычитанием единицы здесь происходит "укорачивание" длины входного списка. Рано или поздно список исчерпается, что вызовет состояние ошибки с соответствующим сообщением. Тем не менее, чтобы функция была более дружественной, вероятно стоит добавить в **COND**-конструкцию ветвь для анализа положительности аргумента:

```
(defun n-th (x n) (cond ((<= n 0) (raiseError "n-th: аргумент неположителен"))
                        ((null x) (raiseError "n-th: список пуст(исчерпан)"))
                        ((= n 1) (car x))
                        (T (n-th (cdr x) (- n 1)))
                  )
)
==> n-th
(n-th '(q w e r t y) 4)
==> r
(n-th '(q w e r t y) -4)

n-th: аргумент неположителен
==> ERRSTATE
```

4) Арифметика неограниченной разрядности

Действия с целыми числами высокой разрядности всегда доставляли определенные трудности. Дело в том, что стандартные целые, которые обрабатывает компьютер, имеют достаточно большую, но все же ограниченную разрядность. Так, для популярных интеловских 32-х битных процессоров максимальная величина целого составляет $\mathbf{2^{31} = 2\ 147\ 483\ 648}$. Если попробовать вычислить, например, величину **50!=1*2*3*...*49*50**, то переполнение неминуемо. Хорошим тоном при реализации Лиспа считается обеспечение арифметики неограниченной разрядности. Долгое время (примерно, до середины 90-х годов прошлого века) только Лисп и позволял вычислить, например, все значащие цифры **100!**. Сейчас с появлением языков новых поколений (таких, как **Haskell** или **Python**) ситуация изменилась. Тем не менее, арифметика неограниченной разрядности остается привлекательной строной Лиспа. В данном разделе будет разобрано две задачи, в которых арифметика высокой разрядности играет главную роль.

Проблема 3n+1

Эта математическая проблема, суть которой сообщил автору П.Л. Бахрах, состоит в следующем: Берется произвольное целое число **N**. Из него получается число **M** по следующему алгоритму:

- Если **N** - четное, то **M** полагается равным **N\2**;
- Если **N** - нечетное, то **M** полагается равным **3*N+1**;

Далее полагается **N = M** и процесс повторяется. При этом оказывается, что какое бы исходное число **N** ни было бы взято, процесс рано или поздно сойдется к единице. Строгого математического доказательства этого факта (на момент написания этих строк) пока не найдено. Удивительно здесь то, что число или уменьшается вдвое или увеличивается втрое. Казалось бы, увеличение должно "перевесить", и число должно бесконечно возрастать. Вот простая Лисповская программа, реализующая приведенный выше алгоритм:

```
(defun 3n+1 (n) (prog nil
                (printline n)
                (cond ((= n 1) (return 'OK))
                      ((= (% n 2) 0) (return (3n+1 (\ n 2))))
                      (T (return (3n+1 (+ (* n 3) 1))))
                )
           )
)
==> 3n+1
```

Функция снова получена простым переводом на Лисп алгоритма "3n+1". Единственное отличие заключается в том, что тело функции оформлена как **PROG**-конструкция. При входе в функцию сначала выполняется печать значения аргумента. Это позволяет проследить, какие числа получаются в процессе работы программы. А вот и результат вызова:

```
(3n+1 10000)
10000
5000
2500
1250
625
1876
938
469
1408
704
352
176
88
44
22
11
34
17
52
```

26
13
40
20
10
5
16
8
4
2
1
==> OK

А если не требуется видеть все промежуточные результаты, а хочется получить только количество циклов до сходимости? В этом случае функция будет выглядеть еще проще:

```
(defun 3n+1 (n k) (cond ((= n 1) k)
                        ((= (% n 2) 0) (3n+1 (\ n 2) (+ k 1)))
                        (T  (3n+1 (+ (* n 3) 1) (+ k 1)))
                  )
)
```

==> 3n+1

Здесь функции добавлен второй параметр **k**. При рекурсивном обращении функции к самой себе к значению параметра прибавляется единица. При терминировании рекурсии накопленное значение параметра **k** возвращается в качестве результата. Легко видеть, что это и будет в точности количество выполненных циклов. Подобный прием использования параметра в рекурсивном программировании имеет специальное название - **накапливающий параметр**. Единственным неудобством новой редакции функции **3n+1** является то, что вызывать ее нужно **с двумя параметрами** - исследуемым числом и нулем:

(3n+1 10000 0)
==> 29
(3n+1 (fact 50) 0)
==> 1295

Впрочем, это неудобство легко преодолеть: достаточно **погрузить** функцию **3n+1** в функцию-**облочку**:

```
(defun *3n+1 (n) (3n+1 n 0))
==> *3n+1
(*3n+1 10000)
==> 29
```

Накапливающий параметр теперь скрыт в оболочке.

5) Численный перебор

Сколько существует натуральных чисел **n**, меньших **1000**, для которых 2^n-n делится на 7? Решение ее кажется простым: завести цикл по **i** от единицы до 1000 и для каждого **i** проверить остаток от деления 2^i-i на 7. Да вот незадача: Visual Basic не позволит вычислить 2^i при **i** большем 31... А в Лиспе такой проблемы нет! Вот простая программа решения задачи:

```
(prog (i n)
    (setq n 0)
    (setq i 1)
  @1 (cond ((= (% (- (^ 2 i) i) 7) 0) (setq n (+ 1 n))))
    (setq i (+ 1 i))
    (cond ((<= i 1000) (go @1)))
    (return n)
)
```

==> **142**

PROG-конструкция использует две локальные переменные **i** и **n**. Для очередного **i** вычисляется остаток от деления на **7** величины 2^i-i. Если остаток оказывается нулевым, значение переменной **n** увеличивается на единицу. Далее величина **i** увеличивается на единицу и проверяется, не превысила ли величина **i** граничного значения (1000). Если нет - вычисления повторяются. Когда **i** получает значение, большее 1000, цикл разрывается и оператор **return** возвращает количество чисел, удовлетворяющих заданному условию. Используя макро **for** этот же алгоритм можно реализовать короче:

```
(prog (i n)
    (setq n 0)
    (for i 1 1000
      ( (cond ((= (% (- (^ 2 i) i) 7) 0) (setq n (+ 1 n))) )
    )
    (return n)
)
```

==> **142**

Обе программы дают верный результат, но работают ощутимо долго - около минуты. Производительность программы может быть повышена почти в 10 раз, если оптимизировать приведенный код. Легко видеть, что "узким местом" является вычисление степени двойки. Предположим, на **j**-м витке цикла вычислена величина 2^j. На следующем витке будет вычислятся величина 2^{j+1} с помощью встроенной функции **EXPT(^)**, хотя для вычисления 2^{j+1}достаточно умножить **уже вычисленную ранее величину** 2^j на 2! Если не перевычислять степень двойки на каждом витке, а завести рабочую

переменную, значение которой просто умножать на 2, тот производительность программы возрастет более, чем в 9 раз:

```
(prog (i n m)
    (setq n 0)
    (setq m 2)
    (for i 1 1000
      ( (cond ((= (% (- m i) 7) 0) (setq n (+ 1 n)))) (setq m (* 2 m)) )
    )
    (return n)
)
```

==> **142**

Результат получается не за минуту, а за 7.5 сек! А если мы хотим не просто подсчитать, а увидеть числа **n**, для которых $\mathbf{2^n-n}$ делится на 7 без остатка? Нет проблем:

```
(prog (i n m)
    (setq n 0)
    (setq m 2)
    (for i 1 1000
      ( (cond ((= (% (- m i) 7) 0) (prog nil (printline i) (setq n (+ 1 n)))))
        (setq m (* 2 m))
      )
    )
    (return n)
)
11
15
16
32
36
37
53
57
58
74
78
79
95
99
...
960
961
977
981
```

982
998

==> 142

Здесь просто добавлена печать очередного числа, удовлетворяющего условиям задачи. Желающие могут убедиться, что все приведенне числа условиям задачи удовлетворяют. Задача решена.

Теперь рассмотрим решение задачи, пригодное для реализации в языке, не поддерживающем арифметики неограниченной разрядности. Поскольку верно равенство $7=2^3-1$, то $2^n-n=7*2^{n-3}+2^{n-3}-n$. Поэтому, остаток от деления (2^n-n) на 7 равен остатку от деления на 7 величины $(2^{n-3}-n)$. Далее: $(2^{n-3}-n) = 7*2^{n-6}+2^{n-6}-n$. Остаток от деления $(2^{n-3}-n)$ на 7 равен остатку от деления на 7 величины $(2^{n-6}-n)$. Таким образом, чтобы оценить остаток от деления (2^n-n) на 7 нужно вычитать из показателя степени по тройке до тех пор, пока показатель не станет меньше 32 (когда степень можно вычислить явно).

Вот программа на Лиспе, реализующая эту идею:

```
(prog (m n i k)
    (setq n 0)
    (for i 1 1000
      (
       (setq m i)
       (setq k (+ 1 (\ (- m 31) 3)))
       (setq m (- m (* 3 k)))
       (cond ((= (% (- (^ 2 m) i) 7) 0) (setq n (+ 1 n))  ))
      )
    )
    (return n)
)
```

==> 142

6) Различные математические объекты

Встроенными типами Лиспа являются только целые и вещественные числа. Более сложные математические объекты (**комплексные числа, векторы, матрицы**) числа можно смоделировать. В нижеследующих разделах показывается, что это совсем не сложно, а динамическая природа списков позволяет легко создавать векторы и матрицы любых размеров. Неограниченная размерность целочисленной арифметики Лиспа позволяет построить точную **рациональную арифметику** (арифметику дробей).

Рациональная арифметика

С рациональными числами читатель, несомненно, знаком еще по школьному курсу арифметики. Тем не менее, рассмотрим строгое **математическое**определение рациональных чисел.

Рассмотрим множество упорядоченных пар целых **взаимно простых** чисел **(n , d)**. Первое число пары будем называть **числителем**, второе - **знаменателем**. Предполагаем, что **d** отлично от нуля. Взаимная простота чисел **n** и **d** означает, что числа **n** и **d** не имеют общих делителей, кроме единицы. Будем считать, что пары (n_1 , d_1) и (n_2 , d_2) равны тогда и только тогда, когда $n_1 = n_2$ и $d_1 = d_2$.

Следует особо подчеркнуть, что в наше множество пар целых включаются не всякие пары, а только пары взаимно-простых чисел. В частности, пара **(0 , d)**входит в наше множество только при **d=1**. Соответственно, пара **(n , n)** входит в наше множество только при **n=1**. Пары **(0 , 1)** и **(1 , 1)** играют в нашем множестве особую роль.

Теперь определим **алгебраические операции** на множестве пар:

- **Суммой** пар (n_1 , d_1) и (n_2 , d_2) будем называть пару **(n , d)** , числитель которой равен:
  $n = (n_1 * d_2 + n_2 * d_1)/НОД((n_1 * d_2 + n_2 * d_1) , (d_1 * d_2))$
  а знаменатель:
  $d = (d_1 * d_2)/НОД((n_1 * d_2 + n_2 * d_1) , (d_1 * d_2))$
- **Разностью** пар (n_1 , d_1) и (n_2 , d_2) будем называть пару **(n , d)** , числитель которой равен:
  $n = (n_1 * d_2 - n_2 * d_1)/НОД((n_1 * d_2 + n_2 * d_1) , (d_1 * d_2))$
  а знаменатель:
  $d = (d_1 * d_2)/НОД((n_1 * d_2 + n_2 * d_1) , (d_1 * d_2))$
- **Произведением** пар (n_1 , d_1) и (n_2 , d_2) будем называть пару **(n , d)** , числитель которой равен:
  $n = (n_1 * n_2)/НОД((n_1 * n_2) , (d_1 * d_2))$
  а знаменатель:
  $d = (d_1 * d_2)/НОД((n_1 * n_2) , (d_1 * d_2))$

В приведенных выше формулах символ **НОД(x , y)** означает наибольший общий делитель целых **x** и **y**. Использование НОД необходимо, поскольку при вычислении числителя и знаменателя может нарушиться взаимная простота.

- **Обратной** паре **(n , d)** при **n** отличном от нуля, будем называть пару **(d , n)**. Легко видеть, что произведение пар **(n , d)** и **(d , n)** дает пару **(1 , 1)**.
- **Противоположной** паре **(n , d)** будем называть пару **(-n , d)**. Легко видеть, что сумма пар **(n , d)** и **(-n , d)** дает пару **(0 , d)**.

Можно убедиться, что для определенных выше алгебраических операций будут справедливы сочетательный, распределительный и переместительный законы арифметики. Роль **нуля** будет играть пара **(0 , 1)**, а роль единицы - пара **(1 , 1)**. Осталось сказать, что пара **(n , d)** традиционно запи-

сывается в виде **n/d** и называется **дробью**. Множество целых чисел естественным образом вкладывается во множество дробей, если отожествить целое **p** с дробью **p/1**.

Множество дробей с двумя операциями, определенными выше, называется **полем рациональных чисел** и обозначается **Q**.

Переходим к моделированию рациональной арифметики на Лиспе. Целочисленная арифметика в нашем распоряжении уже имеется - она встроена в ядро Лиспа. Как следует из приведенных выше определений, нам прежде всего понадобится функция, вычисляющая **НОД**. Для этого существует классический **алгоритм Евклида**. Пусть требуется найти **НОД** двух целых **X** и **Y**. Очевидно, что если одно из чисел равно нулю, то **НОД** равен другому (ненулевому) числу. Если оба числа **X** и **Y** равны нулю, то **НОД** этих чисел неопределен. А при ненулевых **X** и **Y** следует поступать так:

1) Вычислить **R** - остаток от деления **X** на **Y**.
2) Если **R=0**, то **НОД(X , Y)=Y**
3) Если **R <> 0**, то положить **X=Y**; **Y=R** и перейти к шагу 1).

Алгоритм гарантированно сходится. Ниже приведен нерекурсивный вариант программы вычисления **НОД** двух чисел.

```
(defun gcd (x y) (cond
                 ((AND (eq y 0) (eq x 0)) nil)
                 ((eq x 0) y)
                 ((eq y 0) x)
                 (T
                   (prog (xx yy r)
                        (setq xx x)
                        (setq yy y)
                    loop
                        (setq r (remainder xx yy))
                        (cond ((eq r 0) (return yy)))
                        (setq xx yy)
                        (setq yy r)
                        (go loop)
                   )
                 )
              )
)
==> gcd
(gcd 1212 6543)
==> 3
```

Название функции происходит от английского "the **g**reat **c**ommon **d**ivisor", что по-русски как раз и означает "наибольший общий делитель".

Как представлять дробь в Лиспе? Наиболее естественное представление дроби - в виде двухэлементного списка. Первый элемент будет числителем, второй - знаменателем. А чтобы Лисп мог отличить дробь от произвольного двухэлементного списка, добавим в начало списка атом **RATIONAL**. Окончательно дробь будет представляться списком из **трех** элементов: атом **RATIONAL**, числитель и знаменатель. В процессе работы нам понадобится программа "сокращения" дроби. Суть сокращения проста - деление числителя и знаменателя на их **НОД**. Вот соответствующая программа на Лиспе:

```
(defun  simplify (x) (prog (num denom g sgn)
                (setq denom (cadr x))
                (cond ((= denom 0) (raiseerror "Знаменатель равен нулю!")))
                (setq num (car x))
                (setq sgn 1)
                (cond ((And (< num 0) (> denom 0)) (setq sgn -1))
                    ((And (> num 0) (< denom 0)) (setq sgn -1))
                    ((= num 0) (setq denom 1))
                )
                (setq num (abs num))
                (setq denom (abs denom))
                (setq g (gcd num denom))
                (cond ((eq g 1) (return (list (* num sgn) denom)))
                    (T (return (list (\ (* num sgn) g) (\ denom g))))
                )
               )
)

==> simplify
(simplify '(6 3))
==> (2 1)
(simplify '(3 6))
==> (1 2)
(simplify '(0 6))
==> (0 1)
(simplify '(-6 6))
==> (-1 1)
(simplify '(6 -6))
==> (-1 1)
```

Функция **simplify** кроме деления на **НОД** числителя и знаменателя, выполняет еще два дополнительных действия: относит знак дроби к числителю и препятствует созданию дроби со знаменателем, равным нулю. Пусть читателя не удивляет, что функция **simplify** работает с двухэлемент-

ным списком: в приводимых ниже функциях манипулирования дробями **simplify** всегда применяется к **хвосту** списка, моделирующего дробь. Теперь можно приступать к созданию функций для работы с дробями. Имена всех этих функций будут иметь префикс **rat** (рациональный). Начнем с функции **создания** дроби. Она оказывается очень простой:

```
(defun ratCreate (num denom)
          (cond ((null (flagp num 'FIXED)) (raiseerror "Числитель - не целое"))
                ((null (flagp denom 'FIXED)) (raiseerror "Знаменатель - не целое"))
                (T (cons 'RATIONAL (simplify (list num denom))))
          )
)
==> ratCreate
(ratCreate -1 2)
==> (RATIONAL -1 2)
(ratCreate -6 3)
==> (RATIONAL -2 1)
(ratCreate 6 -3)
==> (RATIONAL -2 1)
```

Функция ожидает два параметра - числитель и знаменатель. Возвращает функция трехэлементный список: атом **RATIONAL** (признак дроби), числитель и знаменатель. К списку **(числитель знаменатель)** применяется функция **simplify** (что обеспечивает несократимость дроби и отнесение знака к числителю). Перед созданием списка функция проверяет наличие флага **FIXED** у значений числителя и знаменателя. Если числитель или знаменатель не являются числом типа **FIXED** - возбуждается состояние ошибки. Теперь написать функции манипулирования рациональными оказывается совсем просто. Как ни странно, самой сложной функцией будет функция **печати**рационального числа. При печати желательно предусмотреть все возможности: чтобы рациональное со знаменателем 1 печаталось как целое, чтобы дробь с числителем 0 печаталась как 0, чтобы из неправильной дроби выделялась целая часть и печаталась отдельно. Ниже приводится функция **ratPrint**, которая печатает рациональные числа в привычном для человека виде.

```
(defun ratPrint (x)
    (cond ((<> (car x) 'RATIONAL) (raiseerror "ratPrint: Не дробь"))
        (T
          (prog (tmp num denom sgn)
            (setq tmp (simplify (cdr x)))    ;; на всякий случай...
            (setq num (car tmp))           ;; числитель
            (setq denom (cadr tmp))          ;; знаменатель
```

```
            (setq sgn (sign num))          ;; знак дроби
            (setq num (abs num))           ;; берем абс. величину
            (cond ((= num 0) (go @1) ))    ;; числитель=0
            (cond ((< sgn 0) (prints "-")))  ;; если знак отрицат.
            (cond ((= denom num) (go @2) ))  ;; числитель=знаменатель
            (cond ((= denom 1) (go @3) ))    ;; знаменатель=1 - целое
            (cond ((> num denom)           ;; дробь неправильная
                  (prog nil              ;; печатаем целую часть
                    (print (\ num denom))
                    (setq num (% num denom))
                  )
                 )
            )
            (prints "(")                 ;; печатаем дробную часть
            (print num)
            (prints "/")
            (print denom)
            (prints ")")
            (go @Exit)
        @1    (print 0)
            (go @Exit)
        @2    (print 1)
            (go @Exit)
        @3    (print num)
        @Exit  (return x)
           )
         )
      )

)
==> ratPrint
(ratPrint (ratCreate 1 2))
(1/2)
==> (RATIONAL 1 2)
(ratPrint (ratCreate 45 75))
(3/5)
==> (RATIONAL 3 5)
(ratPrint (ratCreate 6 1))
6
==> (RATIONAL 6 1)
```

А вот и остальные функции манипулирования рациональными числами:

```
;;
;; Сложить две дроби
;;

(defun ratAdd (x y)
     (cond ((<> (car x) 'RATIONAL) (raiseerror "ratAdd: Не дробь"))
          ((<> (car y) 'RATIONAL) (raiseerror "ratAdd: Не дробь"))
          (T (cons 'RATIONAL
               (simplify (list (+ (* (caddr y) (cadr x))
                                  (* (caddr x) (cadr y))
                               )
                               (* (caddr x) (caddr y))
                         )
               )
             )
          )
     )
)

;;
;; Вычесть две дроби
;;

(defun ratSub (x y)
     (cond ((<> (car x) 'RATIONAL) (raiseerror "ratSub: Не дробь"))
          ((<> (car y) 'RATIONAL) (raiseerror "ratSub: Не дробь"))
          (T (cons 'RATIONAL
               (simplify (list (- (* (caddr y) (cadr x))
                                  (* (caddr x) (cadr y))
                               )
                               (* (caddr x) (caddr y))
                         )
               )
             )
          )
     )
)

;;
;; Перемножить дроби
;;

(defun ratMult (x y)
```

```
    (cond ((<> (car x) 'RATIONAL) (raiseerror "ratMult: Не дробь"))
        ((<> (car y) 'RATIONAL) (raiseerror "ratMult: Не дробь"))
        (T (cons 'RATIONAL
            (simplify (list (* (cadr x) (cadr y))
                            (* (caddr x) (caddr y))
                      )
            )
          )
        )
    )
)

;;
;; Разделить дроби
;;

(defun ratDiv (x y)
    (cond ((<> (car x) 'RATIONAL) (raiseerror "ratDiv: Не дробь"))
        ((<> (car y) 'RATIONAL) (raiseerror "ratDiv: Не дробь"))
        (T (cons 'RATIONAL
            (simplify (list (* (cadr x) (caddr y))
                            (* (caddr x) (cadr y))
                      )
            )
          )
         )
    )
)

;;
;; Обратить дробь
;;

(defun ratInv (x)
    (cond ((<> (car x) 'RATIONAL) (raiseerror "ratInv: Не дробь"))
        (T (cons 'RATIONAL
            (simplify (list (caddr x)
                            (cadr x)
                      )
            )
          )
        )
    )
```

```
)

;;
;; Из рационального в плавающее
;;

(defun rat2flo (x)
    (cond ((<> (car x) 'RATIONAL)(raiseerror "rat2flo: Не дробь"))
        (T (/ (cadr x) (caddr x)))
    )
)

;;
;; Модуль дроби
;;

(defun ratAbs (x)
    (cond ((<> (car x) 'RATIONAL)(raiseerror "ratAbs: Не дробь"))
        (T (cons 'RATIONAL
            (list (abs (cadr x))
                (caddr x)
            )
          )
        )
    )
)
```

Следует обратить внимание на то, что функция деления дробей не анализирует случай деления на нуль. В этом нет необходимости, поскольку соответствующую ситуацию "отлавливает" функция **simplify**. Мы построили вполне корректную рациональную арифметику, причем, достаточно скромными средствами.

Комплексные числа

Прежде, чем строить систему комплексных чисел, напомним их математическое определение. Комплексным числом будем называть упорядоченную пару вещественных чисел **(a , b)**. Две таких пары **(a_1 , b_1)** и **(a_2 , b_2)** будем считать **равными** в том и только в том случае, когда $a_1 = a_2$ и $b_1 = b_2$. Теперь введем на множестве пар естественные алгебраические операции **сложение** и **умножение**.

- **Суммой** комплексных чисел **(a_1 , b_1)** и **(a_2 , b_2)** будем называть новое комплексное число **(c , d)**, такое, что:

$\mathbf{c} = \mathbf{a_1} + \mathbf{a_2}$

$\mathbf{d} = \mathbf{b_1} + \mathbf{b_2}$

Достаточно легко убедиться, что для так определенного сложения будут соблюдаться сочетательный и переместительный законы.

- **Произведением** комплексных чисел $\mathbf{(a_1 , b_1)}$ и $\mathbf{(a_2 , b_2)}$ будем называть новое комплексное число **(c , d)**, такое, что:

  $\mathbf{c} = \mathbf{a_1} * \mathbf{a_2} - \mathbf{b_1} * \mathbf{b_2}$

  $\mathbf{d} = \mathbf{a_1} * \mathbf{b_2} + \mathbf{a_2} * \mathbf{b_2}$

Для умножения комплексных чисел будет также будут выполняться сочетательный и переместительный законы. А для сложения и умножения - распределительный закон. Рассмотрим теперь пары вида **(a , 0)**. Легко убедиться, что сумма и произведение таких пар тоже имеет вид **(a , 0)**. Причем, произведение пар **(x , 0)** и **(y , 0)** равно **(xy , 0)**, а сумма - **(x+y , 0)**. Кроме того, произведение пары **(a , 0)** на произвольную пару **(x , y)** даст в результате пару **(ax , ay)**. Все это позволяет отождествить пары вида **(a , 0)** с вещественными числами.

А теперь рассмотрим комплексное число **(0 , 1)**. Легко убедиться, что квадрат этого числа (в смысле определенного выше умножения) равен числу **(-1 , 0)**. Последнее число мы договорились отождествить с обычным вещественным числом **-1**. Число **(0 , 1)** в математике обозначается символом **i**. Таким образом, оказывается верным равенство:

$$\mathbf{i^2 = -1}$$

Кроме того, произвольное комплексное число **(a , b)** можно представить в виде: **a + b*i**, где **a** - сокращенная запись числа **(a , 0)**, а **b** - сокращенная запись числа **(b , 0)**.

Для произвольного комплексного числа **X = (a , b)** вещественное число **a** называется действительной частью **X** (обозначается **Re(X)**); а вещественное число **b** называется мнимой частью **X** (обозначается **Im(X)**).

Для произвольного комплексного числа **X = (a , b)** вещественное число $\mathbf{r = (a^2 + b^2)^{1/2}}$ называется модулем комплексного числа, а число **fi = arctg(b/a)** - его аргументом. С использованием модуля и аргумента любое комплексное число может быть представлено в эквивалентной **тригонометрической форме**:

$$\mathbf{(a , b) = r*(cos(fi) + i * sin(fi))}$$

При возведении комплексного числа в целую степень имеет место замечательное соотношение (формула Муавра):

$$\mathbf{(a , b)^n = (r^n)*(cos(n*fi) + i * sin(n*fi))}$$

Здесь **n** - целый показатель степени. **Сопряженным** к произвольному комплексному числу **(a , b)** будем называть число **(a , -b)**. Легко убедиться, что для произвольного **(a , b)** имеет место:

$$\mathbf{(a , b)*(a , -b) = a^2 + b^2}$$

Произведение числа на сопряженное равно квадрату модуля числа.

После этого математического введения построить систему комплексных чисел на Лиспе будет совсем просто. Как и в случае рациональных чисел, будем моделировать комплексное число списком из трех элементов. Первый элемент этого списка - всегда атом **COMPLEX** - признак комплексного числа; два последующих элемента представляют собой действительную и мнимую части комплексного числа. Снова, как и в случае рациональных чисел, самой сложной функцией будет функция вывода комплексного числа. Остальные функции реализуются достаточно элементарно. Имена всех функций манипулирования комплексными числами будут начинаться с префикса **"cplx"**. Вот как выглядит функция **cplxCreate** - создать комплексное число:

```
;;
;;  Создать комплексное число
;;
(defun cplxCreate (x y)
     (cond ((not (numberp x)) (raiseerror "cplxCreate: нечисловой аргумент"))
           ((not (numberp y)) (raiseerror "cplxCreate: нечисловой аргумент"))
           (T (cons 'COMPLEX (list (+ x 0.0) (+ y 0.0))))
     )
)
==> cplxCreate
```

Функция проверяет, чтобы значения аргументов были числами (типа **FIXED** или **FLOAT**). В теле функции сложение значений аргументов с числом **0.0**приводит к тому, что в списке действительная и мнимая часть хранятся как числа с плавающей точкой. Как и при реализации рациональной арифметики, самой громоздкой функцией является функция печати комплексного числа. Тут важно предусмотреть печать комплексного числа без мнимой части, как действительного; чисто мнимого комплексного числа - без нулевой действительной части; у мнимой части не выводить коэффициент **1** при мнимой части. Собирая все вместе, получим:

```
;;
;;  Печать
;;

(defun cplxPrint (x)
    (cond ((<> (car x) 'COMPLEX) (raiseerror "cplxPrint: неверен тип аргумента"))
          (T
            (prog (re im)
```

```
        (setq re (cadr x))

        (setq im (caddr x))

        (cond ((and (<= (abs re) 1.0E-15) (<= (abs im) 1.0E-15)) (go
@PrZ)))

        (cond ((and (> (abs re) 1.0E-15) (<= (abs im) 1.0E-15)) (go
@PrR)))

        (cond ((and (<= (abs re) 1.0E-15) (> (abs im) 1.0E-15)) (go
@PrI)))

        (print re)

        (cond ((> im 0.0)
               (prog nil
                  (prints "+")
                  (cond ((> (abs (- im 1)) 1.0E-15) (prog nil  (print im))))
                  (prints "J")
               )
              )
              ((< im 0.0)
               (prog nil
                  (prints "-")
                  (setq im (abs im))
                  (cond ((> (abs (- im 1)) 1.0E-15) (prog nil  (print im))))
                  (prints "J")
               )
              )
        )

        (go @Ret)

@PrZ    (print 0)  (go @Ret)
@PrR    (print re) (go @Ret)
@PrI
        (cond ((> im 0.0)
               (prog nil

                  (cond ((> (abs (- im 1)) 1.0E-15) (prog nil (print im))))
                  (prints "J")
               )
```

```
                )
                ((< im 0.0)
                  (prog nil
                     (prints "-")
                     (setq im (abs im))
                     (cond ((> (abs (- im 1)) 1.0E-15) (prog nil  (print im))))
                     (prints "J")
                  )
                )
            )

    @Ret        (return x)

          )
        )
     )
)

==> cplxPrint
```

Реализация остальных функций выглядят достаточно просто:

```
;;
;;  Действительная часть комплексного числа
;;

(defun cplxRe (x) (cond ((<> (car x) 'COMPLEX)
                (raiseerror "cplxRe: неверен тип аргумента"))
                (T (cadr x))
            )
)

;;
;;  Мнимая часть комплексного числа
;;

(defun cplxIm (x) (cond ((<> (car x) 'COMPLEX)
                (raiseerror "cplxIm: неверен тип аргумента"))
                (T (caddr x))
            )
)

;;
```

```
;;  Сопряжение
;;

(defun cplxConj (x) (cond ((<> (car x) 'COMPLEX)
                           (raiseerror "cplxConj: неверен тип аргумента"))
                          (T (cons 'COMPLEX (list (cadr x) (- (caddr x))))
                          )
                    )
)

;;
;; Модуль
;;

(defun cplxAbs (x) (cond ((<> (car x) 'COMPLEX)
                           (raiseerror "cplxAbs: неверен тип аргумента"))
                         (T (sqr (+ (* (cadr x) (cadr x)) (* (caddr x) (caddr x)))))
                   )
)

;;
;; Аргумент
;;

(defun cplxArg (x) (cond
                       ((<> (car x) 'COMPLEX)
                        (raiseerror "cplxArg: неверен тип аргумента"))
                       (T
                        (cond ((<= (abs (cadr x)) 0.1E-16) (* 0.5 _Pi (sign (caddr x))))
                              (T (atn (/ (caddr x) (cadr x))))
                        )
                       )
                   )
)

;;
;;  Сложение
;;

(defun cplxAdd (x y) (cond ((<> (car x) 'COMPLEX)
                            (raiseerror "cplxAdd: неверен тип аргумента"))
                           ((<> (car y) 'COMPLEX)
                            (raiseerror "cplxAdd: неверен тип аргумента"))
```

```
                (T (cplxCreate (+ (cadr x) (cadr y)) (+ (caddr x) (caddr y))))
            )
)

;;
;;  Вычитание
;;

(defun cplxSub (x y) (cond ((<> (car x) 'COMPLEX)
                            (raiseerror "cplxSub: неверен тип аргумента"))
                           ((<> (car y) 'COMPLEX)
                            (raiseerror "cplxSub: неверен тип аргумента"))
                           (T (cplxCreate (- (cadr x) (cadr y)) (- (caddr x) (caddr y))))
            )
)

;;
;;  Умножение
;;

(defun cplxMult (x y) (cond ((<> (car x) 'COMPLEX)
                             (raiseerror "cplxMult: неверен тип аргумента"))
                            ((<> (car y) 'COMPLEX)
                             (raiseerror "cplxMult: неверен тип аргумента"))
                            (T (cplxCreate (- (* (cadr x) (cadr y)) (* (caddr x) (caddr
y)))
                                           (+ (* (cadr x) (caddr y)) (* (caddr x) (cadr y))))
                            )
            )
)

;;
;;  Деление
;;

(defun cplxDiv (x y) (cond ((<= (cplxAbs y) 0.1E-15)
                            (raiseerror "cplxDiv: Деление на нуль"))
                           (T
                             (prog (d Z)
                               (setq d (+ (* (cadr y) (cadr y)) (* (caddr y) (caddr y))))
                               (setq Z (cplxMult x (cplxConj y)))
                               (return (cons 'COMPLEX (list (/ (cadr Z) d) (/ (caddr z)
d))))
```

```
                    )
                  )
               )
)

;;
;; Возведение в степень (формула Муавра)
;;

(defun cplxPow (x n)

     (cond ((or (<> (car x) 'COMPLEX)
                (Not (fixedp n)))
            (raiseerror "cplxPow: неверен тип одного из аргументов")
           )
           (T
            (prog (rr fi)
              (setq rr (cplxAbs x))
              (setq fi (cplxArg x))
              (return  (cplxCreate (* (^ rr n) (cos (* fi n))) (* (^ rr n) (sin (* fi
n)))))
            )
           )
     )
)
```

Снова, как и в случае рациональных чисел, мы без особого напряжения построили алгебру комплексных чисел. Читатель может убедиться, что функции корректно работают. Некоторое удивление может вызвать результат возведения числа в степень:

```
(cplxPrint (cplxPow (cplxCreate 0 1) 2))
-1.0-6.98296672221876E-15J
==> (COMPLEX -1.0 -6.98296672221876E-15)
```

Результат вызова **(cplxCreate 0 1)** есть просто мнимая единица (**J**). Возведение мнимой единицы в квадрат должно дать **-1**. У результата, приведенного выше, действительная часть действительно равна **-1**, а мнимая часть есть очень маленькая величина, но не нуль. Причина заключается в погрешности при вычислении тригонометрических функций (**sin** и **cos**).

Векторы и матрицы

n-мерным вектором над полем действительных чисел называется упорядоченный набор из **n** чисел вида:

$$(x_1, x_2, \dots, x_n)$$

Числа x_1, x_2 и т.д. называются **координатами** вектора. Два вектора считаются равными тогда и только тогда, когда они имеют одинаковую размерность (число координат) и при всех **i** от **1** до **n** **i**-я первого вектора равна **i**-й координате второго вектора.

С векторами можно выполнять различные действия. Простейшими из этих действий являются сложение векторов и умножение вектора на действительное число. Суммой двух векторов **одной и той же размерности** называется вектор, координаты которого получаются сложением соответствующих координат первого и второго векторов. Так, если заданы два вектора:

$$(x_1, x_2, \dots, x_n)$$

и

$$(y_1, y_2, \dots, y_n)$$

то их суммой будет вектор с координатами:

$$((x_1+y_1), (x_2+y_2), \dots, (x_n+y_n))$$

Результатом умножения вектора на вещественное число является вектор, координаты которого получаются из координат исходного вектора умножением на заданное число.

Несколько более сложной операцией является **скалярное произведение** векторов. По определению скалярное произведение векторов

$$(x_1, x_2, \dots, x_n)$$

и

$$(y_1, y_2, \dots, y_n)$$

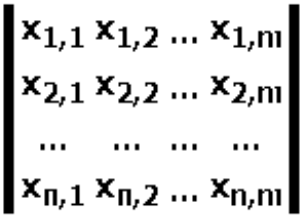
есть **вещественное число**, вычисляемое следующим образом:

$$(x_1*y_1) + (x_2*y_2) + \dots + (x_n*y_n)$$

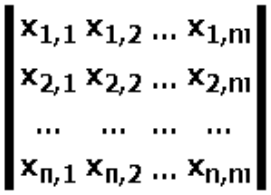
Модулем (или длиной) вектора называется квадратный корень из суммы квадратов координат вектора.

Как хорошо известно, векторы с двумя компонентами можно естественным образом отождествить с векторами на плоскости, а векторы с тремя компонентами - с векторами в пространстве. Теорема Пифагора показывает, что для векторов на плоскости и в пространстве модуль вектора совпадает с его обычной евклидовой длиной. Можно также убедиться, что скалярное произведение вектров есть произведение их модулей на косинус угла между ними.

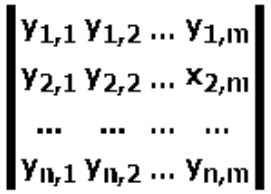
Прямоугольной матрицей размерности (**n*m**) называется прямоугольная таблица вещественных чисел:



Если количество строк матрицы равно количеству столбцов, то матрица называется квадратной. Две матрицы одинаковой размерности:



и



равны в том и только в том случае, когда при всех **(1 <= i <= n)** и **(1 <= j <= m)** имеет место $x_{i,j}=y_{i,j}$.

Суммой (разностью) двух матриц одинаковой размерности называется матрица, элементы которой вычисляются "покоординатным" сложением (вычитанием) элементов исходных матриц. Для приведенных выше матриц матричная сумма будет новой матрицей, элементы которой вычисляются в соответствии с соотношением: $z_{i,j}=x_{i,j} + y_{i,j}$, а разностью - матрица, с элементами $w_{i,j}=x_{i,j} - y_{i,j}$.

Произведение матрицы на вещественное число есть новая матрица, каждый элемент которой есть соответствующий элемент исходной матрицы, умноженный на заданное число.

Литература

1. Душкин, Р.В. Функциональное программирование на языке Haskell / Р.В. Душкин. - М.: ДМК, 2016. - 608 с.
2. Ездаков, А.Л. Функциональное и логическое программирование: Учебное пособие / А.Л. Ездаков. - М.: Бином, 2016. - 119 с.
3. Сеппанен Й., Хьювенен Э. Мир Лиспа.Введение в язык Лисп и функциональное программирование. -М.: Мир,1990. Т1,2 -447с.
4. 4.Сошников, Д.В. Функциональное программирование на F# / Д.В. Сошников. - М.: ДМК, 2011. - 192 с.
5. Грем П. ANSI Common Lisp. – СПб.: Символ – Плюс, 2016. – 448с.
6. Губанова Т. В. Функциональное и логическое программирование.- СПб, РИО СПбГУТ,2010. – 103 с.
7. Файфель Б. Л. , Файфель А. Б. Примеры программирования и избранные задачи [Электронный ресурс]. – Электрон. дан. – режим доступа: http://homelisp.ru/help/samples.html
8. Файфель Б. Л. , Файфель А. Б. Введение в Лисп [Электронный ресурс]. – Электрон. дан. – режим доступа: http://homelisp.ru/help/lisp.html