Степанов П.А. Бржезовский А.В.

Функциональное программирование на языке LISP

версия 0.3

Методическое пособие

ГУАП
Санкт-Петербург
-2008-

Оглавление

1 Введение

В настоящее время большое развитие получили процедурные (итеративные) языки, в которых вычисления производятся путём последовательной записи операторов, выполняющих действия по занесению данных в ячейки памяти и передаче управления. Вместе с тем, необходимо знать, что существуют и другие виды языков программирования, зачастую в приложении к определённому классу задач обладающие весьма мощными выразительными средствами.

Среди таких видов языков можно отметить функциональные языки и конкретно язык Lisp. Этот язык был разработан Дж. Маккарти в 1962 году и с тех пор завоевал прочное место среди средств разработки программного обеспечения, особенно в области искусственного интеллекта. В частности, можно отметить, что набор редакторов текстов GNU EMACS написан на Lisp, а также весьма известным продуктом, написанным на этом языке, является AUTOCAD.

Чистый Lisp является весьма ограниченным набором функций, позволяющим выполнять все необходимые действия. Поэтому было разработано большое количество его последующих версий, в которых вводились дополнительные функции, опирающиеся на базовые и позволяющие сделать код более лаконичным. Среди таких версий можно, прежде всего, отметить Common Lisp, являющийся стандартом де-факто. В дальнейшем речь будет идти в основном про эту версию. Получить интерпретатор Common Lisp можно, например, по следующим адресам:

- ftp://ftp.gnu.org/gnu/clisp/release/2.30/clisp-2.28-win32.zip (для Windows)
- ftp://ftp.gnu.org/gnu/clisp/release/2.30/clisp-2.28.tar.bz2 (для Linux)

В своей работе автор также использовал демонстрационную версию Allegro Common Lisp, доступную (на момент написания учебного пособия) по адресу www.frantz.com.

Функциональное программирование, как и другие альтернативные императивному программированию модели, применяются для решения задач, которые трудно сформулировать в терминах последовательных операций. Это практически все задачи, связанные с искусственным интеллектом, в том числе задачи распознавания образов, общение с пользователем на естественном языке, реализация экспертных систем, автоматизированное доказательство теорем, символьные вычисления, построение трансляторов.

Методическое пособие предназначено для студентов специальностей 2204 и 3515, обучающихся в СПб ГУАП.

Понятие функционального программирования

Функциональное программирование является одной из альтернатив процедурному (итерационному) подходу. В итерационном программировании алгоритмы являются описаниями последовательностей выполнения операций, для которых определено текущее состояние и шаг выполнения программы, а моделью алгоритма является машина Тьюринга. В то же время функциональное программирование основано на более естественном с математической точки зрения формализме - лямбда-исчислении Черча, который имеет непосредственные корни в математических формулах. В функциональном программировании понятие шага или времени отсутствует. Программы являются выражениями, а исполнение программ заключается в вычислении этих выражений. Лямбда-исчисление будет подробнее рассмотрено в разделе 2.

Согласно этой модели, алгоритм решения задачи можно представить в виде последовательности вложенных друг в друга функций. При этом пользуются математическим определением функции, определяющим функцию как некое правило, отображающее область аргументов функции в область её значений, причём значения определяются по значениям аргументов единственным образом.

Программа на языке, использующем концепцию лямбда-исчисления, выглядит как композиция конечного набора функций, каждая из которых, в свою очередь, может быть композицией ещё более элементарных функций, и т.д. При этом каждая функция может прямо или косвенно осуществлять рекурсию, т.е. вызов самой себя. Данная модель лишена недостатков машин Тьюринга, описанных выше, в силу своей структуры, изначально требующей определения функций, значения которых зависят исключительно от аргументов функции и не зависят от контекста вызова и выполнения.

Грамматика лямбда-исчисления описывается следующим образом (формулами Бэкуса-Наура):

```
<Выражение> ::= <идентификатор>
 | <константа>
 | lambda <идентификатор> . <выражение>  -- абстракция функции
 | <Выражение> <Выражение>  --  применение функции к аргументу
 | ( <Выражение> )
```

Константами в расширенном лямбда-исчислении могут быть числа, кортежи, списки, имена предопределенных функций, и так далее. Результатом вычисление применения предопределенной функции к аргументам будет значение предопределенной функции в этой "точке". Словом "lambda" обозначена так называемая лямбда-абстракция - специальный оператор лямбда-исчисления. Результатом применения лямбда-абстракции к аргументу будет подстановка аргумента в выражение - "тело" лямбда-абстракции. Сами лямбда-абстракции так же являются выражениями, и, следовательно, могут быть аргументами.

Как правило, рассматривают так называемое "расширенное лямбда-исчисление". Его грамматику можно описать следующим образом:

Выражение ::= Простое выражение | Составное выражение
Простое выражение ::= Константа | Имя
Составное выражение ::=
абстракция | Применение | Квалифицированное выражение | Ветвление
Лямбда-абстракция ::= lambda Имя -> Выражение end
Применение ::= (Выражение Выражение)
Квалифицированное выражение ::=
let (Имя = Выражение ;)* in Выражение end
Ветвление ::=
if Выражение then Выражение
(elseif Выражение then Выражение)* else Выражение end

Как видно из грамматики, расширенное лямбда-исчисление включает в себя дополнительно функции условного вычисления. Впрочем, реализации Lisp пошли ещё дальше и поддерживают в определённой степени даже процедурное программирование.

Лямбда-абстракции имеют всего один аргумент. В то же время, обычно функции имеют гораздо большее их количество. Для разрешения этой проблемы обычно такой аргумент представляют как кортеж или подразумевают, что множество вычислимых функций X * Y -> Z взаимно однозначно отображается в множество вычислимых функций X -> (Y -> Z), где -> означает применение функции, а * - композицию (такие функции называются карризованными).

Чистое лямбда-исчисление Черча позволяет обходиться исключительно именами, лямбда-абстракциями от одного аргумента и применениями выражений к выражениям. В этих терминах можно описать и числа, и структуры данных, и логические значения, и условные операторы. Обходятся оно и без квалифицированных выражений, т.е. рекурсия выражается без использования имени функции в её теле. Более того, некоторые экспериментальные модели функционального программирования обходятся без имён в принципе.

Функциональное программирование (речь идёт о чистом функциональном программировании) обладает следующими двумя свойствами:

- **Аппликативность**, т.е. программа, а также любой программный объект представляется в виде выражения, состоящего из применения функций к аргументам.
- **Настраиваемость,** т.е. новые программные объекты могут быть порождены как применения соответствующих функций к соответствующим существующим объектам.

Для обеспечения корректности обработки разнотипных аргументов в функциональных языках существуют специальные системы типов, ориентированные именно на настраиваемость. Наиболее популярным решением можно считать самостоятельное определение типов аргументов функций, реализованное в большинстве таких языков.

Можно заметить, что, так как порядок вычисления подвыражений не имеет значения, функциональные языки совершенно естественно поддерживают распараллеливание вычислений.

Языки функционального программирования

В этом разделе приведено краткое описание некоторых языков функционального программирования (очень немногих из существующих).

Lisp (List processor). Считается первым функциональным языком программирования. Язык не типизирован и, хотя он содержит массу итеративных возможностей, основным для него является именно функциональный стиль программирования. При вычислениях использует вызов по значению. Существует объектно-ориентированный диалект языка - **CLOS**.

ISWIM (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным в 60-х годах для демонстрации того, каким может быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на ISWIM. Эта виртуальная машина, основанная на вызове по значению, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков, особенно на него похожи ML, и, в частности, Caml.

Scheme. Диалект Lisp, предназначенный для научных исследований в области компьютерной науки. При разработке Scheme был сделан упор на элегантность и простоту языка.

ML (Meta Language). Семейство строгих языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподается во многих западных университетах, иногда даже как первый язык программирования.

Standard ML. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения и поэтому не является чистым. При вычислениях использует вызов-по-значению. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка - Standard ML-97, существует формальное математическое определения синтаксиса, статической и динамической семантик языка.

Caml Light и **Objective Caml**. Язык принадлежит к семейству ML. Objective Caml отличается от Caml Light в основном поддержкой классического объектно-ориентированного программирования. Также как и Standard ML является строгим, но имеет некоторую встроенную поддержку отложенных вычислений.

Miranda. Язык разработан Дэвидом Тернером, в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет строгую полиморфную систему типов. Как и ML преподаётся во многих университетах.

Haskell. Один из самых распространенных нестрогих языков. Основан на комбинаторной логике Хаскелла Кэрри, в честь которого и получил свое название. Имеет очень развитую систему типизации. Последний стандарт языка - Haskell 98. Язык Haskell очень популярен в академических кругах. Haskell очень надежен - 99% ошибок выявляются на этапе компиляции. Тем не менее, он мало распространен в производстве (хотя есть промышленные экспертные системы, написанные на Haskell).

Gofer (GOod For Equational Reasoning). Упрощенный диалект Haskell. Предназначен для обучения функциональному программированию.
Clean. Язык специально предназначен для параллельного и распределенного программирования. Чисто функциональный язык. По синтаксису напоминает Haskell. Использует отложенные вычисления. С компилятором поставляется набор библиотек, позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.
РЕФАЛ. Алгоритмический язык Рефал (*Recursive functions algorithmic language*), был создан в конце 60-х - начале 70-х годов В.Ф.Турчиным. Рефал, как алгоритмический язык, отличается простотой, высокой степенью модульности и имманентной структурностью программ. Простая семантика Рефала повышает надежность программирования на нем, значительно облегчает трассировку и отладку программ. Рефал доказал свою состоятельность в области трансляции машинных языков, доказательства теорем, компьютерной алгебры, создания баз данных.

2 Теоретические основы функционального программирования

Рекурсивное программирование

Рекурсивное программирование является парадигмой, отличной от традиционного, или итеративного, программирования. Итеративное программирование предполагает выполнение программы как последовательность простых операций, таких как пересылка значений или передача управления, и реализуется в виде хорошо известного формализма Машины Тьюринга. Напротив, в рекурсивном программировании основой является исчисление так называемых частично-рекурсивных функций, что с математической точки зрения дает большое преимущество в простоте анализа алгоритма.

Термин «частично-рекурсивная функция» был предложен в 30х годах 20го века математиком Стивеном Клини. Фактически, этот термин является сокращением от фразы «частично-определенная рекурсивная функция», то есть рекурсивная функция, значение которой определено не на всей области значений аргументов. Понятие рекурсивной функции первоначально было описано Геделем как класс всех числовых функций, определимых в некоторой формальной системе. Позднее Алонзо Черч и Стивен Клини высказали утверждение о том, что “любая интуитивно вычислимая функция является частично-рекурсивной функцией”. Это утверждение, известное как тезис Черча-Клини или «слабый» тезис Черча, формально доказать невозможно, так как оно само по себе включает понятие интуитивности, поэтому принято за аксиому и используется в качестве основы всей теории. Значение тезиса Черча-Клини трудно переоценить, прежде всего, потому, что он позволяет применить результаты исследований математических функций к алгоритмам. В частности, важнейшее применение этот тезис нашел в теории вычислимости

Другой важной основой рекурсивного программирования является тезис Черча-Тьюринга, или «сильный» тезис Черча. Это утверждение, также не имеющее формального доказательства, гласит, что «все определения вычислимости эквивалентны». В частности, тезис Черча-Тьюринга утверждает эквивалентность машины Тьюринга и аппарата частично-рекурсивных функций. Естественным следствием является возможность использования методов формального анализа алгоритмов, заданных частично-рекурсивными функциями, для итеративного программирования.

По сравнению с итеративным программированием, рекурсивное программирование имеет ряд как преимуществ, так и недостатков. В качестве преимуществ следует, прежде всего, отметить, что многие задачи и структуры данных по своей сути имеют рекурсивную природу. Например, алгоритмы обработки деревьев намного более удобно строятся с помощью рекурсии, нежели при итеративном подходе. Также стоит отметить, что рекурсивные

алгоритмы обычно значительно превосходят итеративные в простоте и краткости записи.

С другой стороны, рекурсия обладает рядом недостатков. Во-первых, если язык, на котором реализован рекурсивный алгоритм, не имеет специальных средств для ее эффективного выполнения, то, за счет операций с передачей аргументов и управления временная эффективность программной реализации может быть ниже. Во-вторых, за счет необходимости использования стека, рекурсивные алгоритмы требуют больше памяти. Этот недостаток является достаточно большой проблемой при программировании, так как размер стека, вообще говоря, ограничен.

Введение в лямбда-исчисление

Чистое лямбда-исчисление является формальной теорией, в которой существуют лишь три конструкции – переменные, определения и применения функций. В чистом лямбда-исчислении изучаются редукции, причём теория не содержит никаких дополнительных выражений, за исключением альфа и бета-редукций. Существует также лямбда-эта теория, где дополнительно рассматривается эта-редукция.

Алонзо Черч

Лямбда-исчисление очень удобно для определения вычислимости. Например, Тьюринг показал, что вычислимость в терминах машины Тьюринга эквивалентна собственно возможности лямбда-описания системы. Клини доказал, что эта возможность описания также эквивалентна рекурсивности Гёделя-Гербрандта. Таким образом, если задача сформулирована с помощью лямбда-исчисления, то она имеет решение за конечное время.

Лямбда-исчисление оперирует следующими основными понятиями – лямбда-выражение, редукция и редекс. Лямбда-выражение – специального вида выражение, синтаксис которого с помощью формул Бэкуса-Наура можно записать так:

$$\mathbf{M ::= x \mid (M_1\ M_2) \mid (\lambda x.M)}$$

Где **x** – переменная, $\mathbf{(M_1\ M_2)}$ – применение функции $\mathbf{M_1}$ к аргументу $\mathbf{M_2}$ **(λx.M)** – абстракция (определение функции с параметром **x** и телом **M**).

В качестве примера приведём лямбда-выражение **(λx. x*x) 10.** Данное выражение означает применение функции возведения в квадрат к числу 10. Такая формула называется термом. Результатом редукции этого терма, очевидно, явится число 100. В чистом лямбда-исчислении рассматриваются только переменные, абстракции и применения, оно не рассматривает никаких

привязок к конкретным предметным областям. Поэтому приведённый пример **(λx. x*x) 10** не может считаться чистым лямбда-термом, так как содержит в себе арифметические знаки. В прикладном лямбда-исчислении допустимы константы, которые могут быть не только числовыми, но также и именами объектов и функций. Когда говорят о лямбда-исчислении, обычно имеют в виду именно прикладное лямбда-исчисление.

Для лямбда-термов вводится понятие бета-эквивалентности. **M $=_\beta$ N.** Для приведённого выше примера можно сказать, что **(λx. x*x) 10$=_\beta$10*10.** В более общем виде это записывается так:

$$(\lambda x.M)\ N\ =_\beta R$$

Упрощённо говоря, бета-эквивалентность означает, что левая и правая части выражения имеют одно и то же значение.

Одним из синтаксических соглашений лямбда-исчисления является то, что применение функции имеет более высокий приоритет, нежели абстракция, поэтому **(λx.x y)** должно рассматриваться как **(λx.(x y))**, т.е. определение функции x как применение x к переменной y. Также в лямбда-выражениях иногда раскрывают скобки, например, **((x y) z)** можно представить как **x y z**, а **(x (y z))** как **x (y z)**. При отсутствии функций применения группируются слева направо. Также выражения, содержащие несколько вложенных абстракций, могут быть преобразованы к одной абстракции, например, выражение **λx. λy. λz.M**, может быть записано как **λxyz.M**

Вхождение переменной **x** сразу за символом **λ** в **λx.M** называется связывающим вхождением или просто связыванием **x**. Все вхождения **x** в **(λx.M)** называются связанными в области этого связывания. Все не связанные вхождения переменных в терм являются свободными. Каждое вхождение переменной может быть либо связанным, либо свободным (не может быть и тем и другим). Надо иметь в виду, что переменная является свободной в терме, если хотя бы в одном из подтермов она является свободной, например, **(λx.y) (λy.y)** переменная y является свободной.

Применение функции может рассматриваться как подстановка значения, к которому она применяется, во все вхождения аргумента в тело функции. Например, **(λx.x+5*x*x) m** может рассматриваться как **m+m*m^2.** Подстановка терма. Подстановка терма **N** вместо переменной **x** в терме **N** записывается как **{N/x} M** и определяется следующими правилами:

1. Если свободные переменные в **N** не имеют связанных вхождений в **M**, то терм **{N/x}M** получается заменой всех свободных вхождений **x** в **M** на **N**
2. Если же существует такая переменная **y**, которая свободна в **N** и связана в **M**, то все связанные вхождения **y** в **M** заменяются на некоторую новую переменную **z**. Так продолжается до тех пор, пока не станет возможным применение предыдущего правила

Можно привести следующие примеры подстановок:

{u/x} x = u	**{u/x} y = y**	**{u/x}(λu.x) = {u/x}(λz.x) = (λz.u)**
{u/x} (x x) = (u u)	**{u/x} (y z) = (y z)**	**{u/x}(λu.u) = {u/x}(λz.z) = (λz.z)**
{u/x} (x y) = (u y)	**{u/x} (λy.y) = (λy.y)**	
{u/x} (x u) = (u u)	**{u/x} (λx.x) = (λx.x)**	
{(λx.x)/x} x = (λx.x)	**{(λx.x)/x} y = y**	

Точное определение подстановки таково:

- **{N/x} x = N**
- **{N/x} y = y y≠x**
- **{N/x}(P Q) = {N/x} P {N/x} Q**
- **{N/x}(λx.P) = λx.P**
- **{N/x}(λy.P) = λy.{N/x}P y≠x, y∉free(N)**
- **{N/x}(λy.P) = λz.{N/x}{z/y}P y≠x, z∉free(N) z∉free(P)**

где под free(X) понимается множество свободных переменных терма X.
В основе лямбда-исчисления лежат также две аксиомы – альфа и бета.

- **(λx.M) N =β {N/x}M** (бета - аксиома)
- **(λx.M) =β λz.{z/x}M** (альфа - аксиома или аксиома переименования)

Бета-аксиома утверждает бета-эквивалентность исходного терма и терма, полученного в результате подстановки. Альфа-аксиома утверждает бета-эквивалентность исходного терма и терма, в котором некоторая переменная заменена на другую, не являющуюся свободной в исходном терме. Фактически, альфа-аксиома просто позволяет переименовать переменные некоторого терма для избежания, например, конфликта имён. На основе этих аксиом доказываются следующие свойства лямбда-термов:

M =β M	Свойство идемпотентности
M =β N ⇒ N =β M	Свойство коммутативности
M =β N, N =β P ⇒ M =β P	Свойство транзистивности
A =β B, C =β D ⇒ A C =β B D **M =β N ⇒ λx.M =β λx.N**	Свойства конгруэнтности

Считается, что подстановка приводит терм к более простому виду. Поэтому вместе с бета-эквивалентностью вводят понятие бета-редукции **(λx.M) N ⇒β {N/x} M.** Редексом называется применение абстракции к аргументу **(λx.M) N** .

С понятием редекса тесно связана **теорема Черча-Россера**. Если ~ - отношение, то пусть ~* означает его конечно-транзитивное замыкание. Пусть X → Y означает, что X редуцируется в Y. Обозначим X cnv Y <=> X → Y или Y → X. Тогда X cnv* Y => существует такое N, что X →* N и Y →* N

Следствием этой теоремы является **ромбическое свойство**. Оно гласит, что если выражение A может быть редуцировано либо к B, либо к C, то всегда существует такое выражение D, к которому могут быть редуцированы и B, и C.

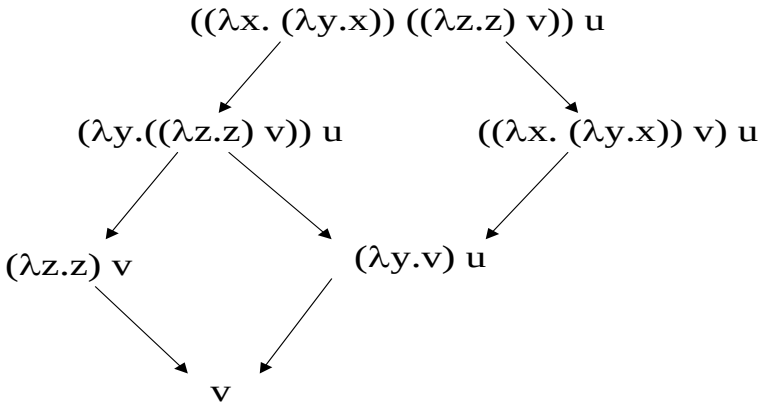


Иллюстрация ромбического свойства на примере

Если к терму нельзя применить бета-редукцию, то говорят, что он находится в нормальной форме. Например, **(λx.M) N** – не в нормальной форме, а **x** или **λx.x x** – в нормальной форме. Некоторые термы нельзя свести к нормальной форме. В качестве примера можно привести терм **(λx.x x) (λx.x x)** . Другие выражения в зависимости от стратегии вычисления могут давать или не давать результата, например, **(λy.0)((λx.x x)(λx.x x))**. Существует два основных метода получения нормальной формы - вызов по значению, когда сначала производится подстановка в самых внутренних редексах, и вызов по необходимости, когда сначала производится подстановка в самом внешнем редексе (слева). Если нормальная форма достижима, то она всегда может быть получена стратегией вызова по необходимости, однако вызов по значению обычно требует меньше редукций.

Прикладные лямбда-исчисления

Прикладное лямбда-исчисление отличается от чистого тем, что в него вводится набор констант c, характеризующий данную предметную область. В этом случае грамматика лямбда-выражения такова:

$$\mathbf{M = c \mid x \mid (M_1\ M_2) \mid (\lambda x.M)}$$

Константы иногда представляют в постфиксной форме, например, **X’ = ’(X)** или **A + B = (A +) B = ((+ A) B)**. Здесь следует понимать, что + является константой прикладного лямбда-исчисления.

В виду того, что прикладные лямбда-выражения обладают специфичными для предметной области зависимостями, вводят понятие сигма-редукции, которая преобразует константы исчисления. Данные и значения обычно ни к чему не редуцируются (т.е. редуцируются сами к себе).

Рассмотрим пример прикладного лямбда-исчисления. Определим следующий набор констант: **true false if not and or eq 0 next plus.** Определим правила сигма-редукции(предметные правила).

Правило	Комментарий
not true $\Rightarrow_\delta$ false	Данные правила частично являются

not false $\Rightarrow_\delta$ true	стандартными правилами математической логики, частично – арифметики, при соблюдении следующих условий. Под not следует понимать инверсию, or – дизъюнкцию, and – конъюнкцию. True и false соответствуют логическим единице и нулю. Оператор If может быть представлен более сложным образом, например в виде (or (and x y (not a)) (and y (not x) (not a)) (and a x (not y)) (and a x y))[1]. eq является выражением эквивалентности, а 0 – числом 0. plus представляет собой сложение, next - инкремент
or false x $\Rightarrow_\delta$ x	
or x false $\Rightarrow_\delta$ x	
or x true $\Rightarrow_\delta$ true	
or true x $\Rightarrow_\delta$ true	
if true x y $\Rightarrow_\delta$ x	
if false x y $\Rightarrow_\delta$ y	
and false x $\Rightarrow_\delta$ false	
and x false $\Rightarrow_\delta$ false	
and x true $\Rightarrow_\delta$ x	
and true x $\Rightarrow_\delta$ x	
eq true true $\Rightarrow_\delta$ true	
eq false false $\Rightarrow_\delta$ true	
eq true false $\Rightarrow_\delta$ false	
eq false true $\Rightarrow_\delta$ false	
eq 0 0 $\Rightarrow_\delta$ true	
eq (next x) 0 $\Rightarrow_\delta$ false	
eq 0 (next x) $\Rightarrow_\delta$false	
eq (next x) (next y) $\Rightarrow_\delta$eq x y	
plus 0 x $\Rightarrow_\delta$ x	
plus (next x) y $\Rightarrow_\delta$ next (plus x y)	

Большая часть данных правил соответствует обычной математической логике. В свете этого можно следующий пример, записанный в терминах языка Паскаль, вычислить с помощью лямбда-исчисления:

if 1+1=2 then return A else return B
if (eq (plus (next 0) (next 0)) (next (next 0))) A B =>
if (eq (next (plus 0 (next 0)) (next (next 0))) A B =>
if (eq (next (next 0)) (next (next 0))) A B=>
if (eq (next 0) (next 0)) A B =>
if (eq 0 0) A B=>
if true A B =>
A

Комбинаторная логика

[1] Считая для краткости, что and и or имеют произвольное число аргументов

3 Данные и выражения языка Lisp.

Понятия списков и атомов, вычисление функций.

В основе языка Lisp лежит использование двух основных конструкций - списков и атомов. Атомом в языке Lisp называется имя, состоящее из букв, цифр и специальных знаков. Атом, не являющийся числом, называется символом (т.е. атом *a10* это символ, а атом *10* - не символ). Рекомендуемыми к использованию специальными знаками являются следующие:

+ - * / @ $ % ^ & _ \ < >

В принципе, в зависимости от реализации допустимы и другие специальные знаки, и иногда даже русские буквы, однако их лучше не использовать по причинам совместимости, а также чтобы не вызывать путаницы.

Обычно интерпретатор Lisp не делает разницы между прописными и строчными буквами. Иными словами, символы Atom, ATOM или atom эквивалентны.

Как уже было сказано, наряду с символами в Lisp также используются числа, которые также представляют собой ограниченную пробелами последовательность знаков. Основными типами чисел являются целые и дробные. Все числа в Lisp являются константами, в то время как символы являются по умолчанию переменными.

Другой важной конструкцией языка являются списки. Список представляет собой заключённый в скобки набор атомов или списков, разделённых пробелами. Примером списка может быть, например, (a b) или ((a) (b)). Списки предназначены для хранения структурированной (в общем случае древовидной) информации и могут использоваться для хранения сложных данных.

В языке Lisp также имеется специальный символ для представления пустого списка, т.е. ( ). Это символ **NIL**, также означающий логическую ложь. Соответственно, логическую истину обозначает символ **T**.

Особенностью Lisp является то, что он не делает различий между данными и операторами. Это выражается в том, что Lisp интерпретирует список таким образом, чтобы первый символ списка считался некоторой функцией, а остальные - её аргументами. Иными словами, ввод с консоли списка (a b c) будет воспринят как предложение вычислить функцию A от аргументов B и C. Существует большой набор функций, заранее определённых в Lisp, а также возможность определять пользовательские. Надо также отметить, что в Lisp любая функция имеет своё значение, однако может также обладать побочным эффектом, например, менять значения ячеек памяти и структуру записанных в неё списков (типичным примером побочного эффекта является присвоение значения символу функцией SET, рассматриваемой в следующем разделе). При работе с консолью этот результат выводится на экран после выполнения функции.

Вычисление функции в программе представляет собой замену её вызова на её результат. Например, функция сложения + может быть вызвана следующим образом[2]:

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

Таким образом, мы видим, что функция была вычислена и возвращено её значение. Для лучшего восприятия материала приведём другой пример. Вычислим выражение (2*(5+7)):

```
> (* 2 (+ 5 7))
24
```

Используемая в Lisp форма записи выражений может показаться неудобной, однако с точки зрения машины она является идеальной для обработки.

В заключение приведём две стандартную функцию, одна из которых впоследствии будет очень часто использоваться. Это функции QUOTE и EVAL.

Quote Функция подавляет вычисление своего аргумента, заставляя Lisp работать с ним как с именем. Так как необходимость в этой функции возникает очень часто, для краткости её заменяют апострофом.

Таким образом,

```
>(+ 1 2)
3
>(QUOTE + 1 2)
+
>(QUOTE (+ 1 2))
(+ 1 2)
> ' (+ 1 2)
(+ 1 2)
```

Однако следующий пример ошибочен:

[2] Здесь и далее: символом > в примерах будет обозначаться приглашение консоли Lisp к вводу. В других реализациях этот символ может иметь иной вид, например, "USER[1]:" или "[1]>". Строки без символа > будут означать вывод программы на экран. Два символа > будут означать пользовательский ввод с клавиатуры. При написании кода (например, в случае проверки примеров) символ > не должен помещаться в текст. Также необходимо иметь в виду, что оператор в Lisp может записываться более чем на одной строке - место, в котором заканчивается оператор, может быть определено по закрывающей скобке, соответствующей самой первой открывающей.

```
> ('+ 1 2)
Error: Illegal function object: '+
```

Данное сообщение об ошибке появилось потому что, не смотря на знак апострофа, первый символ списка всё равно будет рассматриваться как функция. Таким образом, для её исправления апостроф должен быть вынесен за скобки, чтобы подавить вычисление списка целиком.

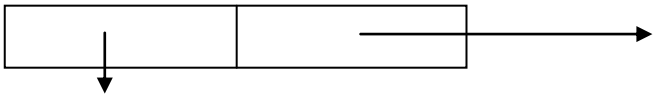
Eval Функция является противоположностью quote и аннулирует действие последней. В общем случае функция указывает на необходимость вычисления её аргумента.

```
> (eval (quote (+ 1 2)))
3
> (eval 3)
3
```

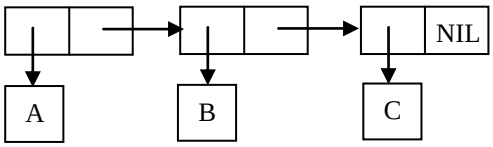
2.2 Организация памяти Lisp-машины

Как уже было сказано, каждый атом является переменной, если только не было определено обратное. Иными словами, если мы пишем список (+ 2 3) мы складываем то, что и пишем, т.е. 1+2, в то время как при записи (+ A B) вы пытаемся сложить значение A со значением B. Понимание механизма обработки данных при вычислении функции очень важно для избежания ошибок. в целях достижения этого понимания мы рассмотрим вопрос, как представляются данные в памяти интерпретатора Lisp.

Основой памяти виртуальной машины Lisp является ячейка. она содержит два указателя - на первый элемент списка (голову) и последующие (хвост). Необходимо обратить внимание, что хвост это всегда список, даже если он пустой или состоит из одного элемента.



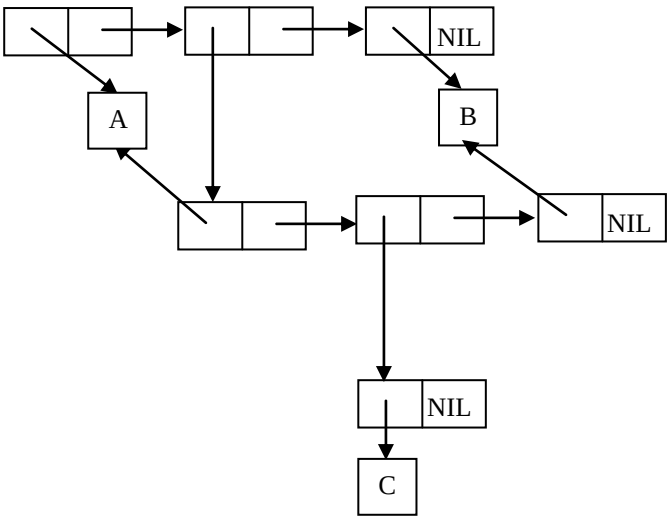
Например, если у нас имеется список (a b c) то первый указатель будет указывать на элемент A, а второй на список (B C). Соответственно, список (B C) также может быть представлен в виде подобной ячейки, где головой будет являться b, а хвостом (c). Наконец, список из одного элемента (C) может быть представлен как голова C и хвост NIL. Поясним это на рисунке:



Таким образом, мы видим, что переменные A, B и C хранятся в памяти в некоторых ячейках, в то время как списковая структура использует не сами

переменные, а ссылки на них. Важно понимать, что при создании другого списка, использующего переменные A, B или C, новые экземпляры этих переменных не создаются, а ссылки направляются в те же самые ячейки памяти, что и для первого списка. Это справедливо только для атомов – при создании нового списка (a b c) вся его структура будет продублирована заново.

Возникает естественный вопрос, как быть со списками со сложной структурой. В этом случае используется та же структура, в которой однако голова списка может указывать не только на переменную, но также и на список, обладающий в свою очередь сложной структурой. В качестве примера приведём список **(a (a (c) b) b):**



Как становится понятно из рисунка, значения элементов списка являются указателями на те ячейки памяти, в которых хранятся реальные данные. Несмотря на то, что наш список хранил в себе два элемента A и два элемента B, они в памяти имеют по одному экземпляру, а два экземпляра имеют указатели на них. Таким образом, переменные в языке Lisp хранятся в некоторых ячейках памяти, доступ в которые осуществляется через указатели. В дальнейшем будет показано, каким именно образом функции Lisp обрабатывают списки на основе их представления в памяти, здесь же мы подробнее остановимся на переменных.

Основные функции, присваивающие значения переменным, следующие.

set Функция присваивает значение некоторому символу. После этого использование в тексте символа с присвоенным значением будет интерпретироваться как использование его значения. Первый аргумент функции обязательно символ, в то время как второй может быть как символом, так и списком.

setq То же, что и set, однако подавляет вычисление первого аргумента

setqq То же, что и set, однако подавляет вычисление обоих аргументов. Эта функция определена не во всех версиях Lisp (например, существует в Interlisp).

Таким образом, при попытке вычисления функции set мы получаем следующий результат:

```
> (SET A B)
Error: Attempt to take the value of the unbound variable `A'.
```

Ошибка возникает в виду того, что символ A вычисляется, однако значение ему не было присвоено. Если же мы используем функцию setq, то получим

```
> (SETQ A B)
Error: Attempt to take the value of the unbound variable `B'.
```

Соответственно, эта функция уже корректно воспринимает свой первый аргумент, однако пытается присвоить ему значение второго аргумента, который также его не имеет. Наконец, последняя функция даёт верный результат.

```
> (SETQQ A B)
B
```

Таким образом, мы присвоили A значение B, и теперь при вызове A мы получим значение B:

```
>A
B
```

Подытоживая сказанное, мы можем считать следующие выражения эквивалентными:

```
>(SET 'A 'B)
B
>(SETQ A 'B)
B
>(SETQQ A B)
B
```

Также для пояснения механизма работы и отличий SET, SETQ и SETQQ приведём следующий пример:

```
> (SETQ A 'B)
B
> (SET A 'C)
C
> A
B
> B
C
>(EVAL A)
C
> (SETQ A 5)
5
> (SETQ B 'A)
A
> (EVAL B)
5
```

Таким образом, мы видим, что оператор (SET A 'C) сперва преобразовал свой первый аргумент в его значение (т.е. B) и уже после этого присвоил B значение своего второго аргумента. Любопытно также отметить, что хотя A имеет значение B, а B имеет значение C, при попытке вычислить A мы получаем B, а вовсе не C. Однако при попытке вычисления уже вычисленного

значения A мы получим C. Это связано с организацией памяти Lisp-машины, так как фактическим значением является указатель, и даже если в ячейку памяти занесено некоторое значение, второй раз по этому значению аргумент не вычисляется.

Рассмотрим ещё одну функцию, которая позволяет присваивать значение непосредственно ячейке памяти, на которую ссылается её первый аргумент (смысл этого действия будет объяснён при описании памяти lisp-машины)

setf Функция записывает значение непосредственно в ячейку памяти, на которую указывает первый аргумент, и является обобщённым вариантом присвоения символу значения.

Вообще пока что можно полагать, что (SETF a b) эквивалентно (SETQ a b), однако в дальнейшем будет показано, как эта функция используется для обновления значений в объектах различных типов языка Lisp.

2.3. Функции проверки типов и логические функции

Итак, мы присвоили нашей переменной какое-то значение и собираемся её использовать. Забегая немного вперёд, скажем, что тип значения связан в Lisp не с самой переменной, а с её значением, что делает невозможным проверку корректности информации на уровне передачи параметров. В таком случае уже программист должен предусмотреть проверку того, являются ли наши данные теми, с которыми мы умеем работать.

Первый вопрос, который может придти в голову, атом это или список? Для выяснения этого обстоятельства служит функция ATOM:

Atom Функция возвращает T, если её аргумент является атомом, и NIL в противном случае.

С помощью этой функции мы можем выяснить тип присвоенного символу значения (как использовать полученные логические значения будет объяснено в разделе 3). Для того же чтобы выяснить, есть ли у символа значение в принципе, мы можем использовать функцию

boundp возвращает значение T, если её первый аргумент является символом, связанным с каким-либо значением, и NIL в противном случае. Функция возвращает T, если её аргумент является атомом, и NIL в противном случае. Если аргумент является списком, функция возвращает ошибку.

Приведём пример, поясняющий работу этих функций:

```
>(BOUNDP 'a)
NIL
>(SETQ a 'b)
B
>(BOUNDP 'a)
T
```

```
>(BOUNDP '1)
Error: `5' is not of the expected type `SYMBOL'
>(BOUNDP 'T)
T
>(BOUNDP '(a))
Error: `(A)' is not of the expected type `SYMBOL'
```

Константы считаются всегда связанными со значениями, поэтому (BOUNDP 'T) даёт истину. В то же время для чисел мы получаем ошибку, так как числа являются атомами, но не символами. Аналогично мы не можем выяснить, связаны ли со значениями списки, так как список не указывает ни на какую ячейку в памяти в принципе. Наконец, мы можем проверить, не связана ли какая-то ячейка со значением NULL

null Функция возвращает T, если её аргумент равен NULL, и NIL в противном случае.

```
>(NULL T)
NIL
>(NULL NIL)
T
```

У нас также может возникнуть вопрос - а как сконструировать сложное условие? Для этого существуют две основные логические функции, которые мы сейчас рассмотрим.

and Логическое умножение. возвращает значение T, если **все** аргументы истинны

or Логическое сложение, возвращает T, если хотя бы один аргумент истинен

```
>(AND T T T NIL)
NIL
>(AND T)
T
>(OR NIL NIL T)
T
>(OR NIL)
NIL
```

Ещё одна обычная проблема связана с проверкой равенства двух символов. Lisp предоставляет достаточно большой набор функций для анализа эквивалентности:

eq Функция сравнивает два символа на предмет совпадения. С её помощью можно проверять только символы и константы T и NIL, а результат будет истиной только в том случае, если оба аргумента совпадают (для атомов просто проверяет совпадения адресов ячеек). Возникает проблема со сравнением вещественных чисел.

```
>(EQ 'A 'B)
NIL
>(EQ 'A 'A)
T
>(EQ 1 1)
T
>(EQ 1.0 1.0)
NIL
```

eql Сравнивает числа одного типа. Умеет корректно сравнивать вещественные числа.

equal Обобщение eql, позволяющее проверять совпадение двух списков. Считает объекты совпадающими, если совпадает их структура.

equalp Обобщение всех функций сравнения, которое умеет корректно понимать равенство между числами с плавающей запятой и целыми.

```
> (EQ 'b 'b)
T
> (EQ 1 1)
T
> (EQ 1.0 1.0)
NIL
>(EQL 1.0 1.0)
T
>(EQL 1.0 1)
NIL
> (EQL '(a) '(a))
NIL
> (EQUAL '(a) '(b))
NIL
> (EQUAL '(a) '(a))
T
>(EQUAL 1.0 1)
NIL
>(EQUALP 1.0 1)
T
```

2.4. Манипуляция элементами списков

Ещё одна простая задача заключается в манипуляции элементами списков (выделение элементов, получение из них новых списков и пр.). Она решается следующим набором функций.

car Получение головы списка. Аргумент всегда список

cdr Получение хвоста списка. Аргумент всегда список

cons Получение нового списка из его головы и хвоста. Первый аргумент может быть атомом или списком, второй аргумент тоже может быть не списком, в таком случае получается специальная структура, называемая *точечной парой.*

Три указанные функции являются основными при манипуляции элементами списков. Их работа напрямую связана с организацией памяти в Lisp-машине. Очевидно, имея указатель на первую ячейку структуры списка, car возвращает указатель на голову, а cdr - на хвост. Что касается cons, он конструирует из двух указателей новую ячейку, получая структуру нового списка. Разумеется, в случае, например, присвоения результирующего значения какой-либо переменной, структуру списка необходимо скопировать дабы не вызывать побочного эффекта изменения одного списка при работе с другим. Приведём примеры работы с CAR и CDR:

```
>(CAR '(a b c))
A
>(CAR '((a) (b) (c))
(A)
>(CDR '(a b c))
(B C)
>(CDR '((a) (b) (c))
((B) (C))
>(CDR '(a))
NIL
```

Необходимо обратить внимание, что результатом работы CDR всегда является список.

Для облегчения работы с функциями CAR и CDR были введены ещё несколько функций. Дело в том, что зачастую надо достать элемент откуда-нибудь из середины, что приводит к большой вложенности вызовов, например, получим элемент E из списка (B (D E) F):

```
>(CAR (CDR (CAR (CDR '(B (D E) F)))))
E
```

То же самое можно изложить с помощью функции, условно называемой CXXXXR, в которой XXXX собирается из вторых букв последовательно вложенных CAR и CDR, в нашем примере CADADR:

```
>(CADADR '(B (D E) F))
E
```

Таким образом, подобная функция просто представляет собой последовательное вычисление функций CAR и CDR. Сразу заметим, что они вычисляются справа налево, что и соответствует нормальному порядку вложенности. В большинстве реализаций Lisp число символов A и D в них не может быть больше 4х (в виду стремительного роста количества функций, реализующих оператор, при увеличении его длины).

Для сбора атома и списка или двух списков в один служит функция CONS. Она собирает голову и хвост в один список, и для неё можно установить соотношение следующего вида, что если L - некоторый список, то результат (CONS (CAR L) (CDR L)) есть L. Приведём примеры:

```
>(CONS 'a 'b)
(A . B)
>(CONS 'a '(b))
(A B)
>(CONS '(A) '(B))
((A) B)
```

Как видно из примера, использование атома в качестве второго операнда приводит к образованию специальной структуры Lisp, так называемой *точечной пары*. Это специальная структура, которая фактически представляет собой условную голову и условный хвост, объединённые точкой, образуется в том случае, если голова или хвост не являются элементами нужной структуры.

Также отметим одну специальную задачу, решаемую с помощью CONS. Помещение результата выполнения функции в скобки может быть выполнено следующим образом:

```
>(CAR '(A B))
A
>(CONS (CAR '(A B)) NIL)
(A)
```

Во многих реализациях Lisp функции CAR и CDR также имеют синонимы, называемые **FIRST** и **REST**. Их действие совершенно аналогично CAR и CDR. Также для удобства программиста иногда вводят функции **SECOND, THIRD** и так далее, вплоть до **TENTH** :

```
>(SECOND '(a b c))
B
>(THIRD '(a b c))
C
```

Обобщением подобных функций является функция **NTH**, выбирающая из списка элемент по его номеру (начиная от 0). Очевидно, (FIRST X) есть (NTH 0 X), (SECOND X) есть (NTH 1 X) и т.д. Если такого элемента нет, возвращается NIL. Как будет показано позднее при обсуждении типов данных языка Lisp, функция nth может использоваться не только для чтения элемента списка, в сочетании с setf она также может записывать в этот элемент новое значение.

Nth Возвращает элемент с индексом, заданным первым аргументом, из списка, заданного вторым аргументом

```
>(NTH 1 '((A)(B)))
(B)
>(NTH 2 '((A)(B)))
NIL
```

Последнее, о чём мы скажем в этом разделе - функция LIST, позволяющая конструировать список из произвольных элементов. List может иметь произвольное количество аргументов, из которых собирает список

list Преобразует все свои аргументы в список

```
>(LIST 'A '(B C) 'D)
(A (B C) D)
```

4 Основные конструкции

3.1. Лямбда - выражения и пользовательские функции

Введём понятие формы. Формой в языке Lisp называется любая управляющая конструкция (т.е. конструкция, которая может быть вычислена). Примерами форм может быть (+ 1 2) или (LIST 'A '(B C) 'D). Надо отметить, что оператор, содержащий в себе формы, сам может являться формой. Атомы также являются формами.

Форма является важным понятием, с помощью которого мы в дальнейшем будем описывать вызовы функций. Если в предыдущей главе все функции работали с атомами или списками, то здесь будут рассматриваться функции, принимающие своими параметрами управляющие конструкции языка Lisp.

Основой *любых* вычислений в языке Lisp являются лямбда-выражения, которые, в свою очередь, берут за основу теорию лямбда-исчисления Черча (описанную в разделе 2).

В Lisp общее представление лямбда-выражения имеет вид (LAMBDA (X1 X2 … XN) Fn), где символ LAMBDA указывает на определение функции.

lambda ***(LAMBDA ((формальные_параметры) тело_функции)***

Создаёт функцию. Список формальных параметров (или так называемый лямбда-список) представляет собой перечисление используемых в теле функции имён переменных. Тело функции может являться произвольной формой.

Для того чтобы понять, зачем это нужно, необходимо обратить внимание на форму записи вызовов функций (или лямбда-вызовов) в языке Lisp. Первый элемент списка всегда считается именем функции. Таким образом, можно в качестве некоторого первого элемента списка поставить лямбда-функцию и вычислить её от остальных элементов. С другой стороны, лямбда-вызов может быть также и фактическим параметром некоторой другой функции. Надо отметить, что если лямбда-выражение не используется одним из перечисленных способов, это приводит к ошибке.

В качестве примера приведём вычисления квадрата числа:

```
>(LAMBDA (X) (* X X))
#<Interpreted Function (unnamed) @ #x204d51ea>
>((LAMBDA (X) (* X X)) 2)
4
```

Как мы видим, использование ламбда-выражения без вызова привело к бессмысленному результату, однако правильно сформированный лямбда-вызов дал корректный результат.

В некотором приближении лямбда-выражение может считаться функцией без имени. Однако обычно гораздо удобнее формировать описание функции один раз, давать ему имя и впоследствии при вызове обращаться к функции по имени, нежели каждый раз подставлять лямбда-выражение, т.е.

полное описание последовательности действий. Некоторые версии Lisp позволяют связывать лямбда-выражения с символами, и впоследствии использовать эти символы в качестве имён, однако данный метод не является общепринятым. Наиболее общим определением имени функции является использование DEFUN:

defun ***(DEFUN имя_функции лямбда-смисок тело_функции)***
Определяет именованную функцию. Лямбда-список - список параметров, тело - произвольная форма. Впоследствии при обращении к имени функции на его место подставляется лямбда-выражение, составленное из лямбда-списка и тела функции.

Возвращаясь к вычислению квадрата числа, мы можем привести аналогичный код, но уже с использованием DEFUN:

```
>(DEFUN SQUARE (X) (* X X))
SQUARE
>(SQUARE 2)
4
```

В дальнейшем каждое обращение к SQUARE (до тех пор, пока символ не будет переопределён) будет вычислять квадрат аргумента.

Можно проверить, связан ли символ с функцией. Для этой цели используется функция FBOUNDP:

fboundp ***(FBOUNDP символ)***
Определяет, связано ли с символом определение функции и возвращает истину если связано и ложь в противном случае.

```
>(FBOUNDP 'SQUARE)
T
>(FBOUNDP 'SQUARE1)
NIL
```

Данный пример был выполнен в Clisp. В случае Allegro Common Lisp результат будет другой:

```
>(FBOUNDP 'SQUARE)
#<Interpreted Function SQUARE>
```

Тем не менее, будучи использован в аргументах других функций, этот вывод на экран соответствует возвращаемому значению T.

Ещё одна любопытная возможность связана со способностью Lisp показывать, каким именно образом функция была определена.

Symbol-function ***(symbol-function символ)***
Возвращает текстовое определение функции.

```
>(SYMBOL-FUNCTION 'SQUARE)
#<CLOSURE SQUARE (X)
        (DECLARE (SYSTEM::IN-DEFUN SQUARE))
(BLOCK SQUARE (* X X))>
```

Данный пример также выполнен в Clisp.

Для списка аргументов функций допустимо также указывать дополнительные свойства, которые начинаются со знака & и записываются перед объявляемым аргументом функции (использование таких слов – отличительная особенность диалекта COMMON LISP). Это свойства &OPTIONAL, &REST, &KEY и &AUX. Все параметры, объявленные до первого &KEY являются обязательными. Далее, значения параметров, объявленных через &OPTIONAL можно не указывать, и они будут переданы в тело функции как NIL или значением по умолчанию (если оно указано).

```
>(DEFUN SAMPLE (X &OPTIONAL (Y 5)) (* X Y))
SAMPLE
> (SAMPLE 1)
5
>(SAMPLE 1 2)
2
>(DEFUN SAMPLE2 (&OPTIONAL Y) Y)
SAMPLE2
>(SAMPLE2 1)
1
>(SAMPLE2)
NIL
```

Все параметры, которые указываются после слова &REST, передаются в тело программы через список несвязанных параметров (для организации функций с переменным числом параметров).

```
>(DEFUN SAMPLE3 (&REST PARAMS) PARAMS)
SAMPLE3
>(SAMPLE3)
NIL
>(SAMPLE3 'A)
(A)
>(SAMPLE3 'A 'B 'C)
(A B C)
```

Отметим, что наряду с (CONS x NIL) подобная функция также может использоваться для задачи помещения атома в скобки.

Параметры, объявленные со свойством &KEY являются необязательными и могут быть переданы в функцию только в виде пары :ИМЯ ЗНАЧЕНИЕ, зато в любом порядке:

```
>(DEFUN SAMPLE4 (&KEY X Y) (LIST X Y))
SAMPLE4
>(SAMPLE4)
(NIL NIL)
>(SAMPLE4 :X 1)
(1 NIL)
>(SAMPLE4 :Y 1)
(NIL 1)
```

```
>(SAMPLE4 :X 1 :Y 2)
(1 2)
```

Наконец, параметры, объявленные как вспомогательные, являются скорее локальными переменными, нежели аргументами функции и не требуют передачи при вызове.

3.2. Последовательности вычислений

В языке Lisp существует возможность организации последовательностей вычислений, вплоть до поддержки процедурного стиля программирования. Для этого служат различные специфичные конструкции, часть из которых будут описана в этом разделе.

let ***(LET ((локальная_переменная1 значение1) (локалная_переменная2_значение2)…) форма1 форма2 …)***
Создаёт локальную связь. Содержит в своём первом параметре список, каждый элемент которого представляет собой пару "локальная переменная - значение" и выполняется в начале вычисления оператора. Остальными параметрами LET являются вычисляемые формы, значение последней из которых возвращается в качестве результата. Необходимо отметить, что результат модификации локальных переменных по завершении оператора не сохраняется.

```
>(SETQ A 1)
1
>(LET ((A 2)) (+ A 1))
3
>A
1
```

Отметим, что в данном случае оператор LET содержал в себе одну форму (+ A 1) и, в то же время, естественно, сам являлся формой. В качестве параметра-формы LET вполне мог использоваться другой оператор LET или любой другой оператор.

prog1 ***(progn форма1 форма2 форма3…)***
prog2 Позволяет выполнять последовательные вычисления форм.
prong Возвращается значение, соответственно, первой, второй и последней формы.

```
>(PROG1 1 2 3)
1
>(PROG2 1 2 3)
2
>(PROGN 1 2 3)
3
```

Естественно, применение этой конструкции может быть и более сложным:

```
>(PROGN (SETQ A 5) (SETQ B 7))
7
>A
5
>B
7
```

3.3. Условные операторы и циклы

Для того, чтобы решать задачи, необходимо обеспечивать разветвление алгоритмов, в частности, языку особенно необходимы условные конструкции и циклы. Здесь мы рассмотрим некоторые из них, имеющие наиболее частое применение, и начнём с операторов условия.

cond ***(cond (условие1 форма1) (условие форма2)…)***
Реализует таблицу решений. В зависимости от выполняемого условия возвращает сопоставленное ему выражение. Аргументами COND являются списки из двух элементов, первый из которых указывает на условие, а второй определяет, какое значение при выполнении этого условия должен вернуть COND. Если ни одно условие не подойдёт, возвращается NIL. Ветви просматриваются последовательно, и как только одно из условий оказывается истинным, выполнение COND завершается

COND является наиболее общей формой условного оператора. Все остальные условные операторы могут быть выражены через него. Примером использования может быть некая таблица решений, например, вывести день недели по его номеру:

```
>(SETQ DAY 3)
3
>(COND
        ((EQL DAY 1)  "Monday")
        ((EQL DAY 2)  "Tuesday")
        ((EQL DAY 3)  "Wednesday")
        ((EQL DAY 4)  "Thursday")
        ((EQL DAY 5)  "Friday")
        ((EQL DAY 6)  "Saturday")
        ((EQL DAY 7)  "Sunday")
        (T "Check your calendar")
)
"Wednesday"
```

При попытке выполнить пример необходимо обратить внимание, что дни недели, написанные по-русски, некоторые интерпретаторы не понимают. Также заметим, что последним условием является T - это условие по умолчанию, оно всегда выполнится, если только до него не выполнилось уже какое-то другое. Естественно, условие NIL бессмысленно.

if ***(if условие форма-для-истинности форма-для-ложности)***
Реализует простой условный оператор. Содержит три аргумента - условие, результат, если условие истинно, результат, если

условие ложно. Соответствует оператору **(cond (Условие результат-при-истинном-условии) (T результат-при-ложном-условии))**

В качестве примера приведём следующий код, пытающийся определить, является ли число натуральным:

```
>(SETQ NUMBER 100)
100
>(IF (> NUMBER 0) "Natural" "Not Natural")
"Natural"
>( SETQ NUMBER -100)
-100
>(IF (> NUMBER 0) "Natural" "Not Natural")
"Not Natural"
```

Наконец, можно использовать также дополнительные виды условных операторов, такие как WHEN, UNLESS, CASE.

when ***(when условие значение-при-истинности)***
Модифицированный вариант оператора COND, содержит в качестве параметров одно условие (первый параметр) и несколько форм, которые вычисляются последовательно при истинности условия. Возвращается значение последней вычисленной формы.

unless ***(unless значение-при-ложности)***
То же самое, что и WHEN, однако формы вычисляются, если условие ложно.

case ***(case ключ (список-ключей форма1 форма2…) (список-ключей форма1 форма2…)…)***
Традиционная таблица решений (сходна с соответствующим оператором языка Pascal). Содержит первый параметр - ключ, который вычисляется перед обработкой всех остальных параметров. Остальные параметры состоят каждый из списка ключей и последовательности вычисляемых форм, причём вычисляются формы (и возвращается последняя из них) в том случае, если значение ключа CASE было найдено в списке ключей одного из параметров.

Приведём примеры использования соответствующих функций. Используем наш старый пример с определением того, является ли число натуральным (в противном случае возвращается NIL):

```
>(SETQ NUMBER 100)
100
>(SETQ NUMBER 100)
100
>(WHEN (> NUMBER 0) "Natural")
"Natural"
>(UNLESS (> NUMBER 0) "Not Natural")
NIL
>( SETQ NUMBER -100)
-100
>(WHEN (> NUMBER 0) "Natural")
NIL
>(UNLESS (> NUMBER 0) "Not Natural")
"Not Natural"
```

А также модифицированная версия таблицы решений относительно дней недели:

```
>(SETQ DAY 3)
3
>(CASE DAY
        (1  "Monday")
        (2  "Tuesday")
        (3  "Wednesday")
        (4  "Thursday")
        (5  "Friday")
        (6  "Saturday")
        (7  "Sunday")
        (T "Check your calendar")
)
"Wednesday"
```

Теперь рассмотрим операторы циклов

do ***(do ((переменная начальное-значение приращение)…) (условие-окончания форма1 форма2 …) форма 11 форма12…)***

Функция является наиболее общим вариантом организации циклов. Реализуется в форме DO Инициализация Условие Тело цикла. Инициализация является списком правил, каждое из которых состоит из имени переменной, начального значения и приращения, Условие состоит из собственно условия выхода и вычисляемых форм, которые вычисляются при возврате функцией значения (как обычно, возвращается последняя). Наконец тело представляет собой последовательность вычисляемых форм, которые выполняются каждый раз при новой итерации. К сожалению, не во всех версиях Lisp поддерживается приращение итератора. В таких слуяаях необходимо выполнять её приращение отдельным кодом.

Примером работы с оператором DO может служить программа, вычисляющая факториал числа (этот пример достаточно показателен в

следствие последующего изложения эквивалентности решений с помощью циклов и рекурсии).

```
>(DEFUN F1 (X)
        (DO
                ((RES 1) (RS 1))
                ((EQL X 0) RS)
                (SETQ RS (* RES RS))
                (SETQ X (- X 1))
                (SETQ RES (+ RES 1))
))
F1
>(F1 5)
120
```

Существует и более простой в обращении цикл LOOP:

loop ***(loop форма 1 форма2…)***

Функция повторяет вычисление своих форм до тех пор, пока среди них не встретится оператор RETURN

return ***(return результат)***

Функция позволяет выйти из цикла и вернуть значение для операторов LOOP и PROG.

Модифицируем факториал с помощью LOOP:

```
>(DEFUN F1 (X)
        (PROGN
        (SETQ RES 1)
                (LOOP
                        (SETQ RES (* RES X))
                        (SETQ X (- X 1))
                        (IF (EQL X 0) (RETURN RES))
)))
F1
>(F1 5)
120
```

Наконец, последним алгоритмическим средством, рассматриваемым в этом разделе, является оператор PROG, позволяющий выполнять программирование в традиционном процедурном стиле. Сразу отметим, что оператор PROG не имеет почти ничего общего с PROG1 и PROGN.

prog ***(prog(переменная1 переменная2…) оператор1 оператор2…)***

позволяет программировать в традиционном процедурном стиле, используя передачу управления между последовательными операторами. Позволяет использовать в качестве своих форм операторы GO и RETURN

go ***(go метка)***

Функция позволяет выполнить внутри оператора PROG переход на некоторую метку. Форма считается меткой если она является

символом или числом.

Приведём в качестве примера работы с функцией PROG следующий код, вычисляющий степень некоторого числа.

```
>(DEFUN EXP1 (X N)
      (PROG (RES)
                  (SETQ RES X)
                  LOOP
                  (IF (= N 1) (RETURN RES))
                  (SETQ RES (* RES X))
                  (SETQ N (- N 1))
                  (GO LOOP)
      )
 )
EXP1
>(EXP1 2 3)
8
```

3.4. Рекурсия

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя. В математике рекурсия применяется повсеместно, например, следующее определение рекурсивно: *"0 является натуральным числом; натуральное число плюс единица есть натуральное число"*.

С точки зрения реализации различают *прямую* рекурсию (когда процедура содержит ссылку на себя) и *косвенную* (когда процедура содержит ссылку на другую процедуру, содержащую ссылку на исходную). Более подробная информация о рекурсии вообще может быть найдена в источнике (Вирт, 1986).

В большинстве процедурных языков программирования рекурсия является скорее нежелательным методом программирования (вследствие возможности переполнения стека), однако в Lisp она является не только допустимой без ограничений, но, более того, является основным инструментом организации циклов. В чистом Lisp операторы DO и LOPP, а тем более PROG отсутствуют, и должны быть заменены рекурсией. Здесь мы рассмотрим основные приёмы использования этого механизма.

Простейшая рекурсия заключается в том, что некоторая функция вызывает сама себя для проведения некоторых вычислений. Рекурсия отражает рекуррентные соотношения (когда в ряду значений каждое новое вычисляется на основании предыдущих). Простейшим примером является факториал, который задаётся в частности как n!=n* (n-1)!. В примере с функцией DO было рассмотрено вычисление факториала с помощью цикла. С помощью рекурсии можно вычислить его так:

```
>(DEFUN FC2 (X)
        (COND
                ((EQL X 0) 1)
                ((EQL X 1) 1)
                (T (* X (FC2 (- X 1))))))
FC2
>(FC2 5)
120
```

Как видно из примера, рекурсивный вызов полностью отражает рекуррентное соотношение, описывающее факториал. Тем не менее, можно с помощью такого же метода организовывать и циклы, например, в следующем примере функция создаёт список из элементов от 1 до n

```
>(DEFUN LST2 (X LST)
        (COND
                ((EQL X 0) LST)
                (T (LST2 (- X 1) (CONS  X LST)))))
LST2
>(LST2 5 NIL)
(1 2 3 4 5)
```

Основной проблемой, которая может возникнуть при работе с таким механизмом, является бесконечная рекурсия. Бесконечная рекурсия возникает вследствие логических ошибок в программе и приводит к зацикливанию алгоритма и исчерпанию системных ресурсов машины. Примером бесконечной рекурсии может быть следующее выражение:

```
>(DEFUN ERR1 () (ERR1))
ERR1
>(ERR1)
Error: Stack overflow (signal 1000)
[condition type: SYNCHRONOUS-OPERATING-SYSTEM-SIGNAL]
```

В этом примере функция ERR1 выполняет свой рекурсивный вызов всегда, тем самым, вызывая переполнение стека и ошибку. Для того, чтобы избегать подобного, необходимо следовать следующим простым правилам:

- В теле функции всегда должно присутствовать правило, гарантирующее выход из рекурсии.
- Правила, гарантирующие выход из рекурсии, должны предшествовать правилам, содержащим рекурсию

Даже при соблюдении таких правил зацикливание всё равно возможно, поэтому необходимо использовать отладку. Отладка рекурсивных функций является, вообще говоря, не слишком простым делом. Для неё используется функция TRACE

trace ***(trace функция)***

Функция позволяет выводить на экран дерево вызовов некоторой функции.

untrace ***(untrace функция)***
Функция позволяет отменить режим вывода дерева вызовов некоторой функции, установленный функцией trace.

В качестве примера приведём всё то же вычисление факториала:

```
>(DEFUN FC2 (X)
        (COND
                ((EQL X 0) 1)
                ((EQL X 1) 1)
                (T (* X (FC2 (- X 1))))
))
FC2
>(FC2 5)
120
>(TRACE FC2)
(FC2)
>(FC2 5)
 0: (FC2 5)
  1: (FC2 4)
   2: (FC2 3)
    3: (FC2 2)
     4: (FC2 1)
     4: returned 1
    3: returned 2
   2: returned 6
  1: returned 24
 0: returned 120
 120
```

Подобный механизм способен существенно упростить отладку рекурсивных функций. К сожалению, не все версии Lisp позволяют выполнять отладку нескольких функций одновременно.

В заключение обсуждения простой рекурсии рассмотрим некоторые примеры операций над списками, реализуемые с помощью рекурсии.

1. Проверка на вхождения атома в список

```
>(DEFUN CHECKA (AT LST)
        (COND
                ((NULL LST) NIL)
                ((EQL AT (CAR LST)) T)
                (T (CHECKA AT (CDR LST)))
))
CHECKA
>(CHECKA 'A '(B C))
NIL
>(CHECKA 'A '(A B C))
T
```

2. Удаление из списка N-ного элемента

```
>(DEFUN DELAT (POS LST)
        (COND
                ((NULL LST) NIL)
                ((EQL POS 0) (CDR LST))
                (T (CONS (CAR LST) (DELAT (- POS 1) (CDR LST))))
        )
)
DELAT
>(DELAT 2 '(A B C D))
(A B D)
>(DELAT 5 '(A B C D))
(A B C D)
```

3. Добавление в список элемента в позицию N

```
>(DEFUN INSAT (POS LST AT)
        (COND
                ((NULL LST) (CONS AT NIL))
                ((EQL POS 0) (CONS AT LST))
                (T (CONS (CAR LST) (INSAT (- POS 1) (CDR LST)
        AT)))
        )
)
INSAT
>(INSAT 3 '(A B C D E) 'F)
(A B C F D E)
>(INSAT 30 '(A B C D E) 'F)
(A B C D E F)
```

В приведённых примерах рассматривалась рекурсия как таковая, однако в Lisp различают несколько вариантов рекурсии. Рассмотрим их подробнее.

- Параллельная рекурсия. Такая рекурсия возникает, когда в теле некоторой функции f1 содержится вызов функции f2, в качестве аргументов которой используется вызов функции f1.
- Взаимная рекурсия. Возникает при вызове в теле функции f1 функции f2, в теле которой, в свою очередь, существует вызов функции f1.
- Вложенная рекурсия - возникает, когда функция в своём теле вызывает саму себя, используя себя же в качестве параметра этого вызова.

Очевидно, что функции INSAT, DELAT и CHECKA, рассмотренные выше, имеют параллельную рекурсивную схему. В качестве ещё одного примера параллельной рекурсии приведём следующий код, осуществляющий инверсию списка любой глубины вложенности:

```
>(DEFUN REVERSEPLUS (LIST)
        (COND
                ((ATOM LIST) LIST)
                ((NULL (CDR LIST))
                (CONS (REVERSEPLUS (CAR LIST)) NIL))
                (T (APPEND (REVERSEPLUS (CDR LIST))
                        (REVERSEPLUS (CONS (CAR LIST) NIL))))))
REVERSEPLUS
>(REVERSEPLUS '(1 (2 3) (4 (5 6)) 7))
(7 ((6 5) 4) (3 2) 1)
```

Тем самым, мы видим, что функция развернула списки в обратном порядке на всех своих уровнях. То же самое можно выполнить с помощью взаимной рекурсии:

```
>(DEFUN REVERSEPLUS (LIST)
        (COND
                ((ATOM LIST) LIST)
                (T (CHANGE LIST NIL))
        )
)
REVERSEPLUS
>(DEFUN CHANGE (LIST RESULT)
        (COND
                ((NULL LIST) RESULT)
                (T (CHANGE (CDR LIST)
                (CONS (REVERSEPLUS (CAR LIST)) RESULT)))
        )
)
CHANGE
>(REVERSEPLUS '(1 (2 3) (4 (5 6)) 7))
(7 ((6 5) 4) (3 2) 1)
```

Вложенная рекурсия является достаточно редкой. Обычно этот вид рекурсии иллюстрируется с помощью специализированных математических функций, таких как функции Аккермана (Вирт, 1986). Такая функция определяется следующим образом:

A(0,n)=n+1;
A(m,0)=A(m-1,1); (m>0)
A(m,n)=A(m-1,A(m,n-1)); (m,n>0)

Очевидно, что функция Lisp, вычисляющая функцию Аккермана, будет построена в соответствии с определением:

```
> (DEFUN AKK (M N)
        (COND
                ((EQL M 0) (+ 1 N))
                ((EQL N 0) (AKK (- M 1) 1))
                (T (AKK (- M 1) (AKK M (- N 1))))))
AKK
>(AKK 2 2)
7
```

Автор рекомендует выполнить вызов (akk 2 2) после вызова (trace akk) и рассмотреть дерево функций, имеющее достаточно любопытный вид.

3.5. Функционалы

В математике под функционалом понимается функция, имеющая своим аргументом другую функцию. Соответственно, в языках программирования, в том числе в Lisp, функционалом является функция, принимающая некоторыми своими аргументами "процедурный" тип данных.

Первым делом параметр-функцию надо передать, подавив её вычисление. Для этого используется функция FUNCTION

function ***(function функция аргументы)***
Функция позволяет подавить вычисление своего параметра. Действует аналогично QUOTE, но для аргументов функционального типа. В большинстве случаев функциональный параметр используется вместе с QUOTE, однако FUNCTION может быть использована для указания, что в качестве параметра следует ждать именно функцию.

Другая задача - вычислить переданный таким образом аргумент.

funcall ***(funcall функция)***
Обратная FUNCTION функция, позволяющая отменить подавление вычисления.

```
>(FUNCALL
      (FUNCTION
            (LAMBDA (X Y) (* X Y))
      )
  2 4)
 8
```

В приведённом примере с помощью LAMBDA задаётся функция, принимающая два аргумента и возвращающая их произведение. Вычисление этой функции подавляется с помощью FUNCTION (иначе мы получили бы ошибку о недопустимом числе аргументов). Затем над результатом, а именно функциональным аргументом, выполняется FUNCALL, с указанием двух аргументов - 2 и 4. Как видно, результатом было искомое произведение.

Приведём пример использования функционала. Например, мы хотим вычислить экстремум в некоторой последовательности:

```
>(DEFUN EXTREMUM (LST FN EL)
      (COND
            ((EQ (CDR LST) NIL) (CAR LST))
            ((FUNCALL FN (CAR LST) (EXTREMUM (CDR LST)
            FN EL)) (CAR LST))
            (T (EXTREMUM (CDR LST) FN EL))))
EXTREMUM
```

Теперь вычислим минимум и максимум в последовательностях:

```
>(EXTREMUM '(6 -3 1 2 3) (FUNCTION <) 1000)
-3
>(EXTREMUM '(6 -3 1 2 3) (FUNCTION >) -1000)
6
```

Таким образом, в зависимости от переданного в функцию функционального аргумента (< или >) мы вычисляем минимум или максимум. Третьим параметром функции является условно наибольшее и наименьшее число. В принципе, можно таковым считать первый символ последовательности (пример апеллирует к определённой ранее функции EXTREMUM).

```
>(DEFUN EXTREMUM2 (LST ARG)
        (EXTREMUM (CDR LST) ARG (CAR LST)))
EXTREMUM2
>(EXTREMUM2 '(1 2 3) (FUNCTION <))
1
>( EXTREMUM2 '(1 2 3) (FUNCTION >))
3
```

В Lisp присутствует также некоторое количество стандартных функций с функциональными аргументами, которые в основном также как и в нашем примере реализуют работу с последовательностями. Различают два основных класса функционалов - отображающие и применяющие. Применяющими функционалами являются уже известный нам FUNCALL, а также APPLY.

apply ***(apply функция аргумент)***

Функционал применяет некоторую функцию к аргументам, которые заданы списком в качестве второго параметра APPLY

```
>(APPLY 'CAR '((A B)))
A
>(FUNCALL 'CAR '((A B)))
(A B)
```

Отметим, что не все функции могут использоваться в качестве функциональных аргументов, например, использование OR и AND даёт ошибку (интерпретатор аргументирует это тем, что OR, AND и некоторые другие функции имеют некую специальную реализацию).

Общая идея, объединяющая применяющие функционалы, заключается в применении функций к спискам аргументов. Отображающие функционалы предназначены для повторения вычисления функции на элементах списка.

mapcar ***(mapcar функция список)***

Функция, позволяющая выполнить некоторое вычисление последовательно надо всеми элементами списка. Результатом является список, построенный из результатов применения функционального аргумента к элементам списка.

```
>(MAPCAR 'NULL '(() (A B) (NIL C D)))
(T NIL NIL)
>(MAPCAR '(LAMBDA (X) (CONS X NIL)) '(A B C))
((A) (B) (C))
```

maplist ***(maplist функция список)***

Функция, позволяющая выполнить некоторое вычисление последовательно над хвостовыми частями списка. Сперва вычисляется функция от списка, затем (CDR список), затем (CDDR список), (CDDDR список) и так далее. Результатом является список, построенный из результатов применения функционального аргумента к хвостовым элементам списка.

```
>(MAPLIST '(LAMBDA (X) X) '(A B C))
((A B C) (B C) (C))
```

С помощью таких функционалов можно существенно сократить тексты функций в некоторых специальных случаях. Например, так выглядит вычисление максимума:

```
>(DEFUN MAX2 (LST)
        (PROGN
                (SETQ X (CAR LST))
                (MAPCAR '(LAMBDA (Y) (IF (< X Y) (SETQ X Y)))
                LST)
                X))
MAX2
>(MAX2 '(1 2 3))
3
```

mapcan ***(mapcan функция список)***

mapcon ***(mapcon функция список)***

Аналоги MAPCAR и MAPLIST, однако при своей работе строят список, используя функцию NCONC, т.е. потенциально структура исходных данных может разрушиться. Эти функционалы называются объединяющими, и одно из основных проявлений их особенностей заключается в уничтожении всех вхождений NIL в результирующий список.

```
>(SETQ X '(A B C))
(A B C)
>(MAPCAR 'LIST X)
((A) (B) (C))
>(MAPCAN 'LIST X)
(A B C)
>(MAPLIST 'LIST X)
(((A B C)) ((B C)) ((C)))
>(MAPCON 'LIST X)
((A B C) (B C) (C))
```

Используем способность объединяющих функционалов уничтожать NIL для написания функции, проверяющей вхождение элемента в список:

```
>(DEFUN ISINLIST (LST EL)
        (CAR
                (MAPCAN
                        (FUNCTION
                                (LAMBDA (Y)
                                        (IF (EQL Y EL) '(T) NIL)))
                LST)))
ISINLIST
>(ISINLIST '(1 2 3) 2)
T
>(ISINLIST '(1 2 3) 4)
NIL
```

mapc ***(mapc функция список)***

mapl ***(mapl функция список)***

Аналоги MAPCAR и MAPLIST, однако эти функции не строят никакого нового результата, а просто возвращают исходный список. Таким образом, для них может быть важен только некоторых побочный результат, в том случае, если использованная в качестве параметра функция содержит в той или иной форме присвоение, вывод на экран и т.п.

В качестве примера такого побочного действия рассмотрим вывод на экран.

```
>(MAPC 'PRINT '(A B C))

A
B
C
(A B C)
```

Таким образом, все элементы списка были по очереди выведены, а в качестве результата вернулся исходный список. Очевидно, этот же эффект мог быть достигнут и MAPCAR, и MAPCAN, однако при этом не тратились ресурсы на конструирование новых структур в памяти.

Функционалы не обязательно работают с функциями одного аргумента. Если аргументов больше, требуется подача на вход нескольких списков. В качестве примера рассмотрим преобразование двух списков в третий, содержащий в себе произведения элементов исходных списков, находящихся в одинаковых позициях.

```
>(MAPCAR '* '(1 2 3) '(4 5 6))
(4 10 18)
```

Или, например, попарное объединение элементов двух списков:

```
>(MAPCAR 'LIST  '(1 2 3) '(4 5 6))
((1 4) (2 5) (3 6))
```

Расширением этого примера может быть вычисление скалярного произведения двух векторов (в примере [1 2 3] и [4 5 6]):

```
>(APPLY '+ (MAPCAR '* '( 1 2 3) '(4 5 6)))
32
```

В случае, если у списков параметров не совпадают размеры, лишние элементы списков будут просто проигнорированы:

```
>(MAPCAR 'LIST  '(1 2 3) '(4 5))
((1 4) (2 5))
```

Для MAPC и MAPL возвращаемым значением всегда будет первых аргумент-список.

3.6. Макросы

Макросом называется специальная функция, которая вычисляется не сразу а в два этапа: сперва вычисляется её тело, и только потом тело применяется к аргументам. Это бывает полезно, например, если мы хотим сформировать последовательность вычислений программно, а затем эту последовательность выполнить.

defmacro ***(defmacro имя лямбда-список тело)***
Определяет макрос. Синтаксис определения макроса в целом сходен с синтаксисом определения функции, однако при вызове макроса не происходит предварительное вычисление его аргументов.

Первый этап вычисления макроса называется этапом расширения. На этом этапе макрос преобразуется в вычисляемую форму. На втором этапе эта форма вычисляется и возвращается результат.

В качестве примера макроса определим функцию SETQQ, которая была бы аналогична SETQ, но не требовала явного подавления вычисления второго аргумента.

```
>(DEFMACRO SETQQ (X Y) (LIST 'SETQ X (LIST 'QUOTE Y)))
SETQQ
>SETQQ X (A B C))
(A B C)
>X
(A B C)
```

macroexpand ***(macroexpand макровызов)***
Функция возвращает результат расширения макроса. Имеет важное значение для тестирования.

Для того, чтобы посмотреть, какая именно форма была вычислена, применим следующий вызов:

```
>(MACROEXPAND '(SETQQ X (A B C)))
(SETQ X '(A B C))
T
```

В том, что, несмотря на выведенное на печать значение T, функция возвращает именно результат раскрытия макроса, можно убедиться на следующем примере:

```
>(SETQ X (MACROEXPAND '(SETQQ X (A B C))))
(SETQ X '(A B C))
>X
(SETQ X '(A B C))
```

Контекст вычисления макроса отличается от контекста вычисления функции. На этапе раскрытия в макросе могут использоваться все статические связи, созданные при его определении, однако когда происходит вычисление, текст макроса полностью подменяется на результат расширения его определения, и все статические связи становятся недоступны.

В принципе, макросы могут при расширении давать формы, содержащие в себе макровызовы, в том числе и рекурсивные. Эти формы будут расширены и вычислены на этапе вычисления основного макроса.

Макросы могут быть рекурсивными. Это означает, что расширение макроса даёт нам новый макрос. Рекурсивные макросы сродни рекурсивным функциям, за исключением того, что вычисление нового макровызова также производится в два этапа. В качестве примера рекурсивного макроса приведём следующий макрос, считающий сумму чисел в списках специального вида:

```
>(DEFMACRO UNWRAP (X Y)
        (COND
            ((ATOM Y) (LIST '+ X Y))
            (T  (LIST '+ X (LIST 'UNWRAP (CAR Y) (CADR Y))))))
UNWRAP
>(UNWRAP 1 2)
3
>(UNWRAP 1 (2 3))
6
>(UNWRAP 1 (2 (3 4)))
10
>(MACROEXPAND '(UNWRAP 1 (2 (3 4))))
(+ 1 (UNWRAP 2 (3 4)))
```

Естественно, отладка подобных макросов является достаточно трудоёмкой задачей.

Также как и для функций, для параметров макросов можно использовать зарезервированные слова &OPTIONAL, &KEY, &REST и &AUX. Дополнительно можно использовать свойство параметра &WHOLE, которое свяжет параметр со всей формой макровызова. Особенно полезно это в макросах, способных динамически менять свой код с помощью псевдофункций, таких как RPLACA, RPLACD, SETF м другие. В качестве примера приведём макрос, который сам себя превращает в функцию CAR:

```
>(DEFMACRO FRST (ARG &WHOLE CALL)
      >(RPLACA CALL 'CAR))
FRST
>(FRST '(1 2))
1
```

С макросами тесно связан механизм обратной блокировки, позволяющий упростить их запись. Обратная блокировка изображается символом ` (апостроф, наклонённый вправо) и отличается от обычной тем, что внутри такого выражения можно управлять вычислением или невычислением секций. Например, запятая указывает, что следующее выражение должно быть вычислено. Символ @ позволяет включить идущее после него выражение в результирующий список.

```
>`(A ,(+ 1 2) 3)
(A 3 3)
>(SETQ X '(+ 1 2))
(+ 1 2)
>`(A ,@X 3)
(A + 1 2 3)
```

3.7. Ввод и вывод

Одной из основных задач для каждого языка программирования является организация взаимодействия с пользователем. В Lisp для этой цели предусмотрены функции ввода-вывода для стандартного устройства, а также функции работы с файлами.

read ***(read [поток])***

Функция читает выражение из консоли и возвращает его в качестве результата. Функция никак не показывает пользователю, что ожидается ввод. В этой функции и далее: если поток не указан, чтение (или для других функций запись) производится со стандартного устройства ввода - консоли.

```
>(READ)
>>a+b
A+B
```

Здесь при вызове функции READ пользователь ввёл выражение A+B (символ >> символизирует приглашение к вводу и не должен переноситься в текст при попытке выполнить пример). Результатом вычисления явилось собственно введённое выражение.

В качестве примера рассмотрим вычисление суммы от двух введённых чисел:

```
>(* (READ) (READ))
>>3
>>8
24

>(* (READ) (READ))
>>3 8
24
```

Иными словами, Lisp, как и большинство языков высокого уровня, выполняет синтаксический разбор ввода с консоли и выделяет параметры, идущие через пробел. При этом READ вводит именно выражения, т.е. если бы в примере мы ввели, например, (+ 5 7) (- 6 4) то этот ввод также был бы распознан как два параметра, другое дело что подобное выражение недопустимо в рамках примера. Модифицируем пример следующим образом:

```
>(* (EVAL (READ)) (EVAL (READ)))
>>(+ 5 7)  (- 6 4)
24
```

Некоторые символы в Lisp при вводе указывают интерпретатору на выполнение определённых действий (например, скобки, пробел, точка). Lisp хранит информацию о таких символах в специальной таблице, называемой таблице чтения. Эти символы называются макрознаками, и их набор может быть изменён (вследствие чего, естественно, может измениться и синтаксис языка Lisp).

set-macro-character ***(set-macro-character символ функция)***
Функция сопоставляет символу некоторую функцию, обрабатывающую этот символ при синтаксическом разборе выражений. Функция разбора должна иметь два аргумента - поток чтения и собственно символ, который мы обрабатываем.

Следующий пример заставляет Lisp интерпретировать символ % так же, как апостроф (т.е. как функцию QUOTE)

```
>(SET-MACRO-CHARACTER #\%
        (LAMBDA (STREAM CHAR) (LIST 'QUOTE (READ))))
T
> %(A)
(A)
```

print ***(print выражение [поток])***
prin1 ***(prin1 выражение [поток])***
princ ***(princ выражение [поток])***
Функция выводит данные на консоль. PRINT выводит данные, печатает пробел и переводит строку. В различных диалектах указанные действия производятся в различной последовательности. Кроме того, иногда функция не печатает пробел. PRIN1 работает точно также, но не переводит строку. PRINC работает также, как и PRIN1, но преобразует при выводе некоторые типы данных (например, строковые) к более приятному для восприятия виду.

```
>(PROGN (SETQ DATA "A B C D")
        (PRINT DATA) (PRIN1 DATA) (PRINC DATA))

"a b c d" "a b c d"a b c d
"a b c d"
```

Как видно из примера, в Allegro Common Lisp функция PRINT сперва перевела строку, а затем вывела данные и пробел. Затем функция PRIN1 вывела те же самые данные, но строку не перевела и пробел не напечатала. Наконец, PRINC вывела всё ту же самую строку, но при выводе исключила кавычки. Однако, результатом этой функции, как мы видим, является всё та же строка.

terpri ***(terpri [поток])***
Функция переводит строку в стандартном устройстве вывода.

format ***(format поток образец [аргументы])***

Функция выводит строку в соответствии с образцом. Если вы качестве потока задаётся T, то вывод производится на экран, в противном случае необходимо указать ассоциированный с файлом и открытый поток вывода. В качестве образца в функцию подаётся строка, определяющая формат вывода (в чём-то сходная со строкой форматирования функции printf языка C). Значения управляющих символов в строке даны в таблице 1. Текст в образце, не являющийся управляющими символами, также печатается. Функция возвращает в качестве своего значения nil.

Таблица 1

Символы управления выводом функции format

Символ	Смысл
~%	Перевод строки
~S	Вывод очередного аргумента функцией PRIN1
~A	Вывод очередного аргумента функцией PRINC
~nT	Начинает вывод с колонки n. Если она уже была достигнута, выводится пробел.
~~	(Два знака тильды подряд) Вывод самого знака тильды

```
>(FORMAT T "~~ПЕРВЫЙ ПАРАМЕТР ~A И ВТОРОЙ ПАРАМЕТР ~S" 5 7)
~ 5   7
NIL
```

Данная функция весьма удобна при выводе таблиц:

```
>(FORMAT T
        "~10T~%~A~10T~A~%~A~10T~A~%~A~10T~A~%"
        1 2 3 4 5 6)
1       2
3       4
5       6
```

Файловый ввод и вывод несколько отличается от работы с консолью. Во-первых, для работы с файлом его надо ассоциировать с потоком и открыть. Во-вторых, по окончании работы с файлом его надо закрыть. Также различаются файлы последовательного доступа, где записи могут быть получены последовательным чтением с начала файла, и файлы прямого доступа, в которых в любой момент можно обратиться к каждой записи.

Для работы с файлами в Lisp предусмотрены следующие средства. Во-первых, определены специальные символы, указывающие на стандартное устройство ввода и стандартное устройство вывода. Эти символы, например, могут применяться в функциях PRIN1, PRINC и READ, а также FORMAT.

STANDARD-INPUT
STANDARD-OUTPUT Символы указывают на стандартные устройства ввода и вывода соответственно (т.е. клавиатуру и экран).

Можно также связать какой-либо поток с файлом, используя специальные функции.

open ***(open файл направление_обмена)***
Функция открывает файл для чтения или записи. Если файл необходимо открыть для чтения, в качестве направления обмена указывается :**INPUT,** если для записи -:**OUTPUT.** Для одновременного чтения и записи указывают направление обмена :**IO.** В качестве возвращаемого значения передаётся открытый поток, который может быть использован в других функциях.

close ***(close поток)***
Функция закрывает поток, открытый ранее с помощью OPEN

with-open-file ***(with-open-file (имя_потока имя_файла направление_обмена) формы)***
Функция открывает поток, выполняет все формы и закрывает поток, не требуя от пользователя никаких дополнительных действий.

Отдельно следует отметить функцию, существенно облегчающую написание Lisp-программ и позволяющую создавать их во внешних редакторах и дробить на части:

load ***(load имя_файла)***
Функция загружает и исполняет код на языке Lisp, хранящийся в указанном файле.

4. Типы данных

4.1. Иерархия типов

Типы данных являются классами и подклассами используемых в программах на Lisp объектов. Как и в большинстве других языков, в Lisp различают простые и составные типы данных.

Зато важным отличием является то, что тип данных связывается не с именем символа, а с его значением, что в общем-то характерно для интерпретаторов (аналогичный механизм используется, например, в весьма популярном языке JavaScript). Присвоение переменной значения всегда является корректным, независимо от того, что это за переменная и что это за значение. Поэтому Lisp иногда называют нетипизированным (в другом варианте бестиповым) языком. Тем не менее, типы данных в нём есть, а термин обозначает лишь отсутствие жёсткой типизации.

В виду отсутствия жёсткой типизации важное значение имеет преобразование типов и проверка на принадлежность какому-либо типу.

typep ***(typep объект тип)***
Функция возвращает T, если объект имеет указанный тип, и NIL в

противном случае.

type-of ***(typep объект)***
Функция возвращает имя типа, который имеет объект.

coerce ***(coerce объект тип)***
Функция преобразует тип объекта в указанный параметром функции COERCE.

В качестве примера приведём следующий код:

```
>(TYPE-OF 'ATOM)
SYMBOL
>(TYPE-OF 1)
FIXNUM
>(TYPE-OF '(A B C))
CONS
>(TYPEP 'A 'ATOM)
T
>(TYPEP 'A 'SYMBOL)
T
>(TYPEP 'A 'FIXNUM)
NIL
```

Таким образом, мы видим, что один и тот же символ может принадлежать к нескольким типам одновременно, т.е. между типами существуют отношения иерархии.

В таблице 2 приводится полный список всех простых типов языка Lisp, а в таблице 3 - составных (в соответствии со спецификацией ANSI Lisp):

Таблица 2

Полный перечень простых типов языка Lisp.

arithmetic-error	division-by-zero	function	real	simple-condition
array	double-float	generic-function	restart	simple-error
atom	echo-stream	hash-table	sequence	simple-string
base-char	end-of-file	integer	serious-condition	simple-type-error
base-string	error	keyword	short-float	simple-vector
bignum	extended-char	list	signed-byte	simple-warning
bit	file-error	logical-pathname	simple-array	single-float
bit-vector	file-stream	long-float	simple-base-string	standard-char
broadcast-stream	fixnum	method	simple-bit-vector	standard-class
built-in-class	float	method-combination	style-warning	standard-generic-function
cell-error	floating-point-	nil	symbol	standard-method

	inexact			
character	floating-point-invalid-operation	null	synonym-stream	standard-object
class	floating-point-overflow	number	t	storage-condition
compiled-function	floating-point-underflow	package	two-way-stream	stream
complex	random-state	package-error	type-error	stream-error
concatenated-stream	ratio	parse-error	unbound-slot	string
Condition	rational	pathname	unbound-variable	string-stream
Cons	reader-error	print-not-readable	undefined-function	structure-class
control-error	readtable	program-error	unsigned-byte	structure-object
			vector	warning

Таблица 3

Полный перечень составных типов языка Lisp.

and	double-float	member	satisfies	simple-string
array	eql	mod	short-float	simple-vector
base-string	float	not	signed-byte	Single-float
bit-vector	function	or	simple-array	String
complex	integer	rational	simple-base-string	unsigned-byte
cons	long-float	real	simple-bit-vector	values
				vector

В отличие от большинства языков типы языка Lisp в своей иерархии не образуют выраженного дерева, хотя обычно и говорят об иерархии типов. Во-первых, имеются два специальных типа - T и NIL, первый из которых является наиболее общим типом для всех остальных, а второй является потомком любого из определённых типов. Иными словами, каждый символ имеет тип T, а NIL является значением, принадлежащим всем типам. Кроме того, некоторые типы имеют несколько предков. Например, тип NULL является одновременно потомком SYMBOL и LIST. Кроме того, взглянув на таблицы 2 и 3, становится понятно, что некоторые типы одновременно являются и простыми, и составными. Таким образом, иерархия достаточно запутана и в дальнейшем мы будем рассматривать типы данных отдельными фрагментами общей иерархии.

4.2. Числа

Реализация численной арифметики первоначально не была одной из основных задач при разработке языка Lisp, однако, в виду необходимости конкурировать с другими языками в области научных вычислений, Lisp был существенно расширен именно в части работы с числами. Наиболее общим типом является тип NUMBER. Этот тип обозначает любое число, будь то целое, с плавающей запятой или комплексное. Также имеются следующие типы: RATIONAL (рациональные), FLOAT (с плавающей запятой) COMPLEX (комплексные) INTEGER (целые - подмножество RATIONAL) и RATION (натуральные дроби - подмножество RATIONAL). Также рассматривается работа с "целыми числами обыкновенного диапазона" - FIXNUM, которые наиболее эффективно поддерживаются языком.

Важным моментом является задание числового значения соответствующего типа. Целые числа в языке Lisp задаются в следующей форме:

[знак]модуль[точка]

Например: 5, +5, -5 и 5. являются целыми числами. Для определения дробного числа используется черта, например, 1/5, -5/7 и т.п. Сложнее дело обстоит с плавающей запятой. Обычная в таких случаях буква E может быть заменена буквой, указывающие точность числа (зависит от реализации). Допустимы следующие модификаторы:

Таблица 4

Модификаторы типа FLOAT

Буква	Тип
S	SHORT-FLOAT (укороченное число)
F	SINGLE-FLOAT (одинарная точность)
E	SINGLE-FLOAT (одинарная точность)
D	DOUBLE-FLOAT (двойная точность)
L	LONG-FLOAT (удлинённое число)

Например, 5F3 является числом 5000, взятым с обычной точностью. Наконец, комплексные числа могут быть заданы с помощью модификатора #C(действительная часть, мнимая часть). Эти числа существуют не во всех модификациях Lisp.

```
>#C(5 5)
#c(5 5)
```

Для типа NUMBER определены следующие операции:

plusp ***(plusp число)***
minusp ***(minusp число)***
zerop ***(zerop число)***
Функции возвращают T если число является, соответственно, положительным, отрицательным и нулём.

abs ***(abs число)***
Функция возвращает модуль числа.

Также определена операция унарного минуса, изменяющая знак:

```
>(SETQ X 5)
5
>(PLUSP X)
T
>(MINUSP X)
NIL
>(MINUSP (- X))
T
>(MINUSP (ABS (- X)))
NIL
```

Отдельно следует отметить функции сравнения (табл. 5). В Lisp они могут иметь произвольное количество аргументов:

Таблица 5

Основные арифметико-логические языка :Lisp

Мнемоника	Смысл
=	равенство
/=	неравенство
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно
min	минимум из аргументов
max	максимум из аргументов
+	Сложение
-	Вычитание
*	Умножение
/	Деление

```
>(< 1 2 3)
T
>(< 2 1 3)
NIL
>(< 1 3 2)
NIL
>(= 1 1 1 2)
NIL
>(= 1 1 1 1)
T
>(/= 1 1 1 2)
NIL
>(/= 1 2 3 4)
T
```

Как видно, оператор < проверяет, упорядочен ли список по возрастанию, оператор = сравнивает все свои аргументы и т.п. Аналогичной особенностью обладают арифметические функции. Сложение складывает все аргументы, вычитание вычитает из первого все остальные и так далее.

Несколько отдельно стоят функции инкремента. Также как и SETF они могут изменять сам символ, а не только формировать возвращаемое значение:

incf ***(incf переменная приращение)***
decf ***(decf переменная приращение)***
Функции соответственно увеличивают и уменьшают значение ячейки памяти на величину приращения.

```
>(SETQ A 5)
5
>(INCF A)
6
>A
6
>(DECF A)
5
>A
5
```

Очевидно, что конструкция (DECF A) эквивалентна (SETF A(- A 1)).
Рассмотрим ещё несколько математических функций:

exp ***(exp x)***
expt ***(expt x y)***
log ***(log x [y])***
sqrt ***(sqrt x)***
Функции работы со степенями и логарифмами: EXP вычисляет экспоненту в степени X, EXPT вычисляет X степени Y, LOG вычисляет логарифм X, если второй параметр указан, то по основанию Y, иначе по основанию E. SQRT вычисляет квадратный корень из X.

sin ***(sin x)***
cos ***(cos x)***
tan ***(tan x)***
asin ***(asin x)***
acos ***(acos x)***
atan ***(atan x)***
sinh ***(sinh x)***
cosh ***(cosh x)***
tanh ***(tanh x)***
asinh ***(asinh x)***
acosh ***(acosh x)***
atanh ***(atanh x)***
Тригонометрические функции: синус, косинус, тангенс, арксинус, арккосинус, арктангенс, гиперболические синус, косинус, тангенс, арксинус, арккосинус, арктангенс.

В качестве примера проверим правильность известной тригонометрической формулы $\sin^2 x + \cos^2 x = 1$:

```
>(SETQ X 12)
12
>(+ (EXPT (SIN X) 2) (EXPT (COS X) 2))
1.0
```

random ***(random x)***
Генерирует случайные целые числа, неотрицательные и меньшие X.

```
>(RANDOM 10)
3
>(RANDOM 10)
7
```

Для отдельных подтипов типа NUMBER предусмотрены специальные операции. Отметим следующие операции, определённые только для целых чисел:

evenp ***(evenp x)***
oddp ***(oddp x)***
gcd ***(gcd x [x1 x2 x3…])***
lcm ***(lcm x [x1 x2 x3…])***
EVENP возвращает T, если число чётно, ODDP - если нечётно. GCD вычисляет наибольший общий делитель, а LCM - наименьшее общее кратное.

```
>(GCD 8 14)
2
>(LCM 8 14)
56
```

4.3. Символы

Символ является достаточно необычной структурой данных. Во-первых, с символом может быть связано некоторое значение, с помощью использования функций SET, SETQ и SETF. Кроме того, символ может указывать на лямбда-выражение. И, наконец, символ может иметь свойства. Таким образом, символ является структурированным объектом, состоящим из нескольких элементов - имени, значения, лямбда-выражения и списка свойств, а также данные о пространстве имён, которому принадлежит этот символ.

Некоторые функции, такие как BOUNDP, FBOUNDP и SYMBOL-FUNCTION мы уже рассматривали, когда говорили о переменных и функциях. Здесь мы рассмотрим ещё несколько функций, связанных с получением значений символов.

symbol-name ***(symbol-name символ)***
Возвращает имя символа в виде строки.

```
>(SETQ X 5)
5
>(SYMBOL-NAME 'X)
"X"
```

Любопытной особенностью символов является возможность их связывания со специальными ячейками памяти, называемыми свойствами. Свойство символа характеризуется его именем и значением. У подобного механизма есть нечто общее с понятиями структур и записей таких языков, как C и Pascal, однако символ может иметь динамический набор свойств. Кроме того, свойства символа не связаны с его значением, независимо от изменения значения символа его свойства не меняются.

get ***(get символ свойство)***

Возвращает ссылку на свойство символа, если такого свойства не существует, оно не создаётся, возвращается NIL. По умолчанию ссылка может использоваться для чтения его значения, однако в случае применения функции SETF становится возможно создание свойства и запись в него нового значения.

В следующем примере мы создаём у символа у свойство property, И записываем в него значение 5. Отметим, что способ, которым это было выполнено, специфичен для Common Lisp, в других диалектах для этого используется специальная функция PUTPROP:

```
>(GET 'Y 'PROPERTY)
NIL
>(SETF (GET 'Y 'PROPERTY) 5)
5
>(GET 'Y 'PROPERTY)
5
```

У символа может быть несколько свойств, различающихся именами. Тем самым, с помощью символов можно организовывать достаточно сложные структуры данных.

symbol-plist ***(symbol-plist символ)***

Возвращает список всех свойств и их значений для некоторого символа.

remprop ***(remprop символ имя_свойства)***

Удаляет присвоенное ранее символу свойство. Функция возвращает T, если такое свойство есть, и NIL, если такого свойства нет.

```
>(SETF (GET 'SYM 'SIZE) 'BIG)
BIG
>(SETF (GET 'SYM 'COLOR) 'RED)
RED
>(SYMBOL-PLIST 'SYM)
(SIZE BIG COLOR RED)
>(REMPROP 'SYM 'COLOR)
T
>(SYMBOL-PLIST 'SYM)
(SIZE BIG)
```

4.4. Списки

Наряду с атомом, список является основной структурой данных Lisp. Как уже упоминалось, любой список состоит из ячеек, включающих в себя голову и хвост. Абстрактной списковой ячейкой является точечная пара, у которой хвост может представлять собой не только список, но и атом. Важно понимать, что точечная пара является обобщением всех списковых ячеек, и её голова и хвост сами могут быть точечными парами.

Список может быть определён рекуррентно следующим образом:

1. NIL или () является списком.
2. Структура, состоящая из точечных пар, в каждой из которых хвост является списком, сама является списком.

Таким образом, результат объединения (CONS A (B)) является списком, а (CONS A B) нет, так как B это атом, а не список. Тем не менее, обе функции в результате дадут нам точечную пару.

Списки в Lisp имеют базовый тип LIST. Два основных его подтипа это пустой список NIL и непустой список CONS. Элементом списка может быть объект произвольного типа.

Многие операции со списками были уже рассмотрены ранее, поэтому здесь мы остановимся только на некоторых, ранее не упоминавшихся:

rplaca ***(rplaca список значение)***
rplacd ***(rplacd список значение)***

Функции изменяет значение соответственно головы и хвоста списка, указанного первым аргументом, на значение второго аргумента.

```
>(SETQ X '(A B (C)))
(A B (C))
>(RPLACA X '(M N))
((M N) B (C))
>X
((M N) B (C))
>(RPLACD X 'NIL)
((M N))
```

append ***(append список1 список2)***

Функция объединяет свои аргументы-списки в один список. В отличие от LIST, аргументы APPEND считаются списками элементов, а не элементами списка.

```
>(APPEND '( A (B) C) '(D (E) F))
(A (B) C D (E) F)
>(LIST '( A (B) C) '(D (E) F))
((A (B) C) (D (E) F))
```

nconc ***(nconc список1 список2)***

Функция объединяет свои аргументы-списки в один список. В отличие от APPEND, функция разрушает структуру исходных данных - список, заданный первым аргументом, принимает значение результата NCONC.

```
>(APPEND '( A (B) C) '(D (E) F))
(A (B) C D (E) F)
>(NCONC '( A (B) C) '(D (E) F))
(A (B) C D (E) F)
>(SETQ ARG1 '(A (B) C))
(A (B) C)
>(SETQ ARG2 '(D (E) F))
(D (E) F)
>(NCONC ARG1 ARG2)
(A (B) C D (E) F)
>ARG1
(A (B) C D (E) F)
```

Структуроразрушающие функции надо использовать с большой осторожностью, так как их вызов может неожиданным образом влиять на другие функции. Например, следующий вызов приводит к результатам, на первый взгляд достаточно странным :

```
>(SETQ X '(A))
(A)
>(NCONC X X)
(A A A A A A A A A A ...)
```

Таким образом, объединяя, казалось бы, два элементарных списка, мы получили список бесконечной длины[3].

remove ***(remove образец список)***

Функция удаляет из списка все элементы, соответствующие её первому аргументу. Рассматриваются элементы, находящиеся на

[3] Сразу оговоримся, что далеко не все версии Lisp выдерживают этот пример, в частности GNU Clisp просто зависает. Слегка модифицированный пример (NCONC (NCONC X X) X) привёл к зависанию даже тех версий Lisp, который выдержали исходный.

самом верхнем уровне списка - аргумента, т.е. если мы ищем атом A, в списке (A B C) он будет найден, а в списке ((A) B C) - нет.

```
>(REMOVE 'B '( A B C B A))
(A C A)
>(REMOVE 'B '( A (B C B) A))
(A (B C B) A)
```

Функция REMOVE определяется с помощью различных методов сравнения элементов списка, по умолчанию это EQL. Если необходимо работать, например, со списками, функцию сравнения можно задать непосредственно:

```
>(REMOVE '(A B) '((A B) (C D)))
((A B) (C D))
>(REMOVE '(A B) '((A B) (C D)) :TEST 'EQUAL)
((C D))
```

substitute ***(substitute образец новое_значение список)***

Функция заменяет в списке все элементы, соответствующие образцу, на новое значение. Рассматриваются элементы, находящиеся на самом верхнем уровне списка - аргумента, т.е. если мы ищем атом A, в списке (A B C) он будет найден, а в списке ((A) B C) - нет. Функция работает аналогично REMOVE.

```
>(SUBSTITUTE 'B 'A '(A C (A)))
(B C (A))
>(SUBSTITUTE 'B 'A '(A C (A)) :TEST 'EQUAL)
(B C (A))
>(SUBSTITUTE 'B '(A) '(A C (A)) :TEST 'EQUAL)
(A C B)
```

reverse ***(reverse список)***

Функция переворачивает список, т.е. формирует результат как список элементов исходного списка, следующих в обратном порядке. Также как APPEND, REMOVE и SUBSTITUTE рассматривает только первый уровень вложенности.

```
>(REVERSE '(A (B C) D))
(D (B C) A)
```

length ***(length список)***

Функция возвращает длину списка в элементах первого уровня.

```
>(LENGTH '(A B C))
3
>(LENGTH '(A (B C)))
2
```

4.5. Строки

Строковый тип данных STRING является одним из основных при отображении данных. Строка в Lisp состоит из последовательности знаков, имеющих, в свою очередь, тип CHARACTER. В общем случае знак отображается в Lisp так:

#\x

где x указывает на отображаемое данной записью слово или символ. В большинстве случаев этот символ повторяет x, однако бывают и исключения, например, #\Tab выводит символ табуляции а #\Return переводит строку. Знак является своего рода константой и его значением является он сам. Таким образом, строка является дочерним типом для последовательности.

Строка представляет собой последовательность символов, ограниченную с обеих сторон двойными кавычками, например, "String".

Stringp ***(sptingp объект)***
Функция проверяет, является ли объект строкой. Если да, возвращает T, если нет, NIL.

```
>(STRINGP 1)
NIL
>(STRINGP "1")
T
```

char ***(char строка символ)***
Функция возвращает определённый символ из строки. Индексация начинается с нуля.

Функция char может использоваться в качестве параметра SETF.

```
>(SETQ STR "ABCD")
"ABCD"
>(SETF (CHAR STR 0) #\E)
#\E
>STR
"EBCD"
```

4.6. Последовательности

Последовательностью в Lisp называется упорядоченное множество, которое принадлежит одному из двух типов - списку или вектору (одномерному массиву). Таким образом, и списки, и вектора могут использовать функции, определённые для последовательностей. Более того, многие из тех функций, которые были рассмотрены в разделе про списки, на самом деле определены и для последовательностей.

Базовым типом для них является тип SEQUENCE. Вектора, являющиеся потомками типа SEQUENCE, могут быть обработаны не только с помощью стандартных операций для последовательностей, но также операций над массивами (см. 4.7.). Строки, которые рассматривались в предыдущей главе, также являются частным случаем векторов:

```
>(TYPE-OF #(1 2 3))
(SIMPLE-ARRAY T (3))
>(TYPE-OF "123")
(SIMPLE-ARRAY CHARACTER (3))
```

Сразу оговоримся, что символ # используется для идентификации массива.

Таким образом, последовательности являются общим типом данных, предком других типов, крайне важных для пользователя. Здесь мы рассмотрим только наиболее общие операции для последовательностей.

Make-sequence ***(make-sequence тип_последовательности количество [значение_по_умолчанию])***

Функция создаёт последовательность. Если указано значение по умолчанию, созданная последовательность заполняется элементами с данным значением. Типом последовательности могут быть, например, LIST, ARRAY или VECTOR, т.е. типы, являющиеся потомками типа SEQUENCE.

```
>(MAKE-SEQUENCE 'ARRAY 4)
#(NIL NIL NIL NIL)
>(MAKE-SEQUENCE 'LIST 4)
(NIL NIL NIL NIL)
>(MAKE-SEQUENCE 'SEQUENCE 4)
ERROR: TYPE SEQUENCE IS NOT VALID OR DOES NOT
    SPECIFY A SEQUENCE.
>(MAKE-SEQUENCE 'SYMBOL 4)
ERROR: TYPE SYMBOL IS NOT VALID OR DOES NOT SPECIFY A
    SEQUENCE.
```

elt ***(elt последовательность номер_элемента)***

Функция возвращает заданный номером элемент последовательности. Нумерация начинается от нуля.

```
>(SETQ SEQ (MAKE-SEQUENCE 'LIST 5))
(NIL NIL NIL NIL NIL)
>(ELT SEQ 5)
NIL
(SETF (ELT SEQ 4) 1)
1
>SEQ
(NIL NIL NIL NIL 1)
>(SETF (ELT SEQ 5) 1)
ERROR: IN SETF OF NTH THE INDEX 5 EXCEEDS THE LIST
  LENGTH.
```

Как видно из примера, ELT позволяет читать значения, находящиеся за пределами последовательности, возвращая в качестве результата NIL, однако попытка присвоения значения элементу с индексом, превосходящим размер последовательности, была неудачной и размер последовательности не увеличился.

subseq ***(subseq последовательность номер_первого_элемента [номер_последнего_элемента])***
Функция создаёт подпоследовательность из последовательности. Выбираются элементы, начиная с индекса, заданного вторым параметром, и до индекса, заданного вторым параметром, либо до конца последовательности.

Приведём пример выделения подпоследовательности из последовательности, определённой в предыдущем примере:

```
>(SUBSEQ SEQ 2)
(NIL NIL 1)
>(SUBSEQ SEQ 2 3)
(NIL)
>(SUBSEQ SEQ 2 5)
(NIL NIL 1)
>(SUBSEQ SEQ 2 1)
ERROR: START IS GREATER THAN END
```

sort ***(sort последовательность предикат)***
Функция создаёт подпоследовательность из последовательности. Выбираются элементы, начиная с индекса, заданного вторым параметром, и до индекса, заданного вторым параметром, либо до конца последовательности.

```
>(SORT '(1 2 3 4) '>)
(4 3 2 1)
```

Наконец, для последовательности заданы многие функции, которые были рассмотрены ранее при описании последовательностей - это LENGTH, REVERSE, REMOVE, SUBSTITUTE, RPLACA.

Для последовательностей также задано некоторое количество функционалов, наиболее простыми из которых (не считая рассматривавшихся ранее MAPCAR, MAPLIST и пр.) являются EVERY и SOME.

every ***(every функция_условие последовательность)***
Функционал возвращает истину, если функция-условие даёт истину для всех элементов списка. В

some ***(some функция-условие последовательность)***
Функционал возвращает истину, если функция-условие даёт истину хотя бы для одного из элементов списка

```
>(EVERY 'ATOM '(1 2 3))
T
>(EVERY 'ATOM '(1 (2) 3))
NIL
>(SOME 'ATOM '(1 (2) 3))
T
>(SOME 'LISTP '(1 (2) 3))
T
```

Отметим, что для функций, проверяющих значения, не обязательно возвращать NIL или T.

Отображающие функционалы обобщены для последовательностей таким образом, что теперь первым аргументом функционала является тип результата, например,

```
(MAP 'LIST '(LAMBDA (X) (+ X 1)) #(1 2 3 4))
(2 3 4 5)
(MAP 'ARRAY '(LAMBDA (X) (+ X 1)) #(1 2 3 4))
#(2 3 4 5)
```

Наконец, для наиболее комфортной работы с последовательностями в Lisp реализованы специфические функционалы, позволяющие производить условную обработку последовательностей с помощью добавления к функциям REVERSE, SUBSTITUTE, FIND, POSITION и COUNT структур вида -IF и -IF-NOT. Рассмотрение этих функционалов выходит за рамки данного пособия, желающие могут найти их описание в литературе [1].

4.7. Массивы

Также как и в других языках, массивом в языке Lisp называется многомерная структура данных, состоящая из однотипных элементов, идентифицируемых набором индексов. Базовым типом для массива является ARRAY. Объект такого типа может быть создан следующим образом:

make-array ***(make-array (список_размерностей) [:element-type тип_элементов] [:initial-element значение_по_умолчанию])***
Функция создаёт массив из элементов заданного типа, присваивая им заданные начальные значения.

В следующем примере разными способами создаётся массив размером 2 на 4:

```
>(MAKE-ARRAY '(2 4))
#2A((NIL NIL NIL NIL) (NIL NIL NIL NIL))
>(MAKE-ARRAY '(2 4) :ELEMENT-TYPE 'INTEGER)
#2A((NIL NIL NIL NIL) (NIL NIL NIL NIL))
>(MAKE-ARRAY '(2 4) :ELEMENT-TYPE 'INTEGER :INITIAL-ELEMENT 1)
#2A((1 1 1 1) (1 1 1 1))
```

Массивы могут иметь произвольное количество измерений, но не меньшее нуля. Массив с нулевой размерностью состоит из одного элемента. Массивы с размерностью 1 называются векторами. Вектора кроме обычных для

массивов действий могут также обрабатываться специальными функциями, например, работы с последовательностями.

aref ***(aref массив индекс_1 [индекс_2 […]])***
Функция возвращает ссылку на элемент массива (которая может быть использована как аргумент SETF).

```
>(SETQ ARR (MAKE-ARRAY 3))
#(NIL NIL NIL)
>(SETF (AREF ARR 1) 5)
5
>ARR
#(NIL 5 NIL)
```

4.8. Структуры

В разделе, посвящённом символам, рассматривалась возможность связывания символов и свойств, что делало их похожими на некие структуры. Однако в Lisp существует и отдельный тип STRUCTURE, предназначенный для описания сложных объектов.

defstruct ***(defstruct имя_структуры [поля_входящие_в_структуру])***
Функция создаёт тип-структуру, и связывает определение этого типа с символом, заданным первым аргументом. Кроме того, создаётся функция MAKE-имя_структуры, служащая для создания экземпляров типа имя_структуры, COPY-имя_структуры, копирующей структуру, и имя_структуры-P, проверяющей принадлежность объекта к классу имя_структуры. Также создаются функции доступа к полям структуры вида имя_структуры-имя_поля_входящего_в_структуру.

```
>(DEFSTRUCT MYSTR X Y)
MYSTR
>(SETQ STR (MAKE-MYSTR))
#S(MYSTR :X NIL :Y NIL)
>(MYSTR-X STR)
NIL
```

```
> (SETF (MYSTR-X STR) 1)
1
> (MYSTR-X STR)
1
> STR
#S(MYSTR :X 1 :Y NIL)
>(MYSTR-P STR)
T
>(MYSTR-P 1)
NIL
```

5 Лабораторные работы

Лабораторная работа 1

Цель работы - первоначальное ознакомление с системой Lisp и устройством виртуальной Lisp-машины.
Указания. Необходимо выполнить задание, используя только функции CAR, CDR и CONS.

Пример. Из списка (a (b c (d e) f g)) получить список ((g f (e d) c b) a)
Решение.

Соответствующие элементы списка могут быть получены с помощью следующих функций:

G	(CADDR (CDDADR lst))
F	(CADR (CDDADR lst))
E	(CADAR (CDDADR lst))
D	(CAAR (CDDADR lst))
C	(CADADR lst)
B	(CAADR lst)
A	(CAR lst)

С другой стороны, результирующее выражение может быть сконструировано так:

```
(cons
   (cons 'g
      (cons 'f
         (cons
            (cons 'e
               (cons 'd nil))
            (cons 'c
               (cons 'b nil)))))
   (cons 'a nil))
```

Подставляя в формулу функции из таблицы, получаем

```
>(SETQ LST '(A (B C (D E) F G)) )
>(CONS
    (CONS
      (CADDR    (CDDADR LST))
        (CONS (CADR (CDDADR LST))
          (CONS
            (CONS (CADAR (CDDADR LST))
              (CONS (CAAR (CDDADR LST)) NIL))
            (CONS (CADADR LST)
              (CONS (CAADR LST) NIL)))))
```

(CONS (CAR LST) NIL))

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

1. Запишите последовательность вызовов CAR и CDR, выделяющие из списков символ "студент":
 1.1. (WHERE STUDENT IVANOV ALEXEY IS STUDYING)
 1.2. ((4110) (GROUP OF) (STUDENT IVANOV (ALEXEY)))
 1.3. ((DMITRY (ANDREY (MAXIM ALL ARE STUDENTS))))

2. Получите список (X Y Z) из заданных списков с помощью вызова функций CONS, CAR, CDR:
 2.1. (A X C), (F Y), (Z)
 2.2. ((A) (B X) (C D)), ((Y (Z)))
 2.3. (A X B Y), (F Z)

3. Проверьте, сдал ли Иванов экзамен по БД на "отлично", если о нем известна следующая информация:
 3.1. ((SUBJECT DB) (IVANOV EXELLENT) (PETROV SATISF))
 3.2. ((DB IVANOV SATISF) (БЗ PETROV EXELLENT))

4. Дан список ((K L) (M N) A B C (D (E))) получить:
 4.1 элемент K
 4.2 элемент M
 4.3 элемент E
 4.4 список (K D N)
 4.5 список (A D K)
 4.6 список (D (M N) A)

5. Дан список (A (B C) (D (E) K L)) получить:
 5.1 элемент K
 5.2 элемент E
 5.3 список (A B C)
 5.4 список (K D L)
 5.5 список ((B C) K)
 5.6 список (C A K)

6. Дан список (((A) B) (C) D E (K L)) получить:
 6.1 элемент A
 6.2 список (A)
 6.3 элемент C
 6.4 список (A B C)
 6.5 список (D A C)
 6.6 список (C A K)

7. Дан список (A ((B) (C)) (D) (K) L) получить
 7.1 элемент A
 7.2 элемент D

7.3 список (D)
7.4 список (A B C)
7.5 список (A (D) (C))
7.6 список (D A B)

8. Дан список (A (B C) (D E) (K)) получить
8.1 элемент E
8.2 элемент K
8.3 список (K A D)
8.4 список (B A K)
8.5 список (C D K)
8.6 список ((B C) A)

9. Дан список ((A) (B (C D) E (K L)) получить:
9.1 элемент C
9.2 список (C D)
9.3 получить элемент K
9.4 получить список (A B C)
9.5 получить список (B A K)
9.6 получить список (E (K L))
9.7 получить список ((A) (K L))

10. Дан список ((A B (C)) (D (E) (K L M))) получить:
10.1. элемент E
10.2. элемент K
10.3. элемент M
10.4. список (C)
10.5. список (A B C D)
10.6. список (K A C)

11. Дан список ((A (B C)) (D (E) K) L) получить:
11.1. элемент K
11.2. элемент E
11.3. список (A B C)
11.4. список (D K L)
11.5. список ((B C) K)
11.6. список (C A K)

12. Дан список (K L (M N) O (P R S) A) получить:
12.1. элемент P
12.2. элемент R
12.3. элемент A
12.4. список (M A)
12.5. список (A K O P)
12.6. список (A (P R S))

Лабораторная работа 2

Цель работы - ознакомление с лямбда - выражениями и функциями языка Lisp.
Указания. В работе **не допускается** использовать следующие функции: операторы циклов; функции прямого доступа к элементам список, такие как nth, elt, aref, высокоуровневые операторы обработки списков, такие как append, reverse, nconc, функционалы, все виды оператора set, оператор prog.

Пример. Разработать функцию, разворачивающую список любой вложенности в линейный в обратном порядке

Функция plainlist осуществляет первичную обработку списка, приводя его в плоский вид. Res – промежуточный результат, lst –список

```
>(DEFUN PLAINLIST (RES LST)
        (COND
                ((ATOM LST) (CONS LST RES))
                ((NULL LST) RES)
                ((NULL (CAR LST)) RES)
                ((NOT (ATOM (CAR LST))) (PLAINLIST
                        (PLAINLIST RES (CAR LST)) (CDR LST)))
                (T (PLAINLIST (CONS (CAR LST) RES) (CDR LST)))))
```

Например,

```
>PLAINLIST NIL '(A B C))
(NIL C B A)
>PLAINLIST NIL '(A (B) C))
(NIL C NIL B A)
>PLAINLIST NIL '((A (B)) C))
(NIL C NIL NIL B A)
```

Функция cutnulls вырезает из списка все значения nil:

```
>(DEFUN CUTNULLS (LST)
        (COND
                ((NULL LST) LST)
                ((NULL (CAR LST)) (CUTNULLS (CDR LST)))
                (T (CONS (CAR LST) (CUTNULLS (CDR LST))))))
>(CUTNULLS '(NIL A B C NIL))
(A B C)
```

Наконец, функция PLAINLISTREVERSE возвращает искомый результат:

```
>(DEFUN PLAINANDREVERSE (LST)
        (CUTNULLS
                (PLAINLIST NIL LST)))
```

Примеры:

```
>(PLAINANDREVERSE '(A B C))
(C B A)
>(PLAINANDREVERSE '(A (B) C))
(C B A)
```

>(PLAINANDREVERSE '((A (B)) C))
(C B A)

1. Разработать функцию, объединяющую два списка в результирующий список, в котором чередуются элементы исходных списков.
   Например:
   Вход: (1 2 3 4 5), (a b c).
   Выход: (1 a 2 b 3 c 4 5).

2. Разработать функцию, выделяющую из исходного списка подсписок, начиная с элемента с номером ***N*** и заканчивая элементом ***N + K***. ***N*** и ***K*** — аргументы функции.
   Например:
   Вход: (1 2 3 4 5 6 7 8 9), ***N*** = 3, ***K*** = 4.
   Выход: (3 4 5 6 7).

3. Разработать функцию, находящую теоретико-множественное объединение двух списков.
   Например:
   Вход: (1 2 3 4 5), (4 5 6 7).
   Выход: (1 2 3 4 5 6 7).

4. Разработать функцию, находящую теоретико-множественное пересечение двух списков.
   Например:
   Вход: (1 2 3 4 5), (4 5 6 7).
   Выход: (4 5).

5. Разработать функцию, находящую теоретико-множественную разность двух списков.
   Например:
   Вход: (1 2 3 4 5), (4 5 6 7).
   Выход: (1 2 3).

6. Разработать функцию, производящую удаление из исходного списка всех элементов с четными номерами.
   Например:
   Вход: (a b c d e).
   Выход: (a c e).

7. Разработать функцию, находящую сумму элементов с нечетными номерами в заданном списке чисел.
   Например:
   Вход: (2 4 3 1 7 2 4).
   Выход: 16.

8. Разработать функцию, аргументом которой является список, возвращающую список пар: (<элемент исходного списка> <количество его вхождений в исходный список>).

Например:
Вход: (1 2 1 1 3 5 2 5).
Выход: ((1 3) (2 2) (3 1) (5 2)).

9. Разработать функцию, аргументом которой является список, возвращающую список, содержащий два подсписка. В первый подсписок включается ***N*** очередных элементов исходного, а следующие ***K*** элементов — во второй. Затем все повторяется. ***N*** и ***K*** — аргументы функции.
   Например:
   Вход: (1 2 3 4 5 6 7 8 9 10 11) , ***N*** = 2, ***K*** = 3.
   Выход: (1 2 6 7 11), (3 4 5 8 9 10).

10. Разработать функцию, осуществляющую вставку в исходный список подсписка за элементом с номером ***N***.
    Например:
    Вход: (1 2 3 4 5 6), (a b c), ***N*** = 3.
    Выход: (1 2 3 a b c 4 5 6).

11. Разработать функцию, аргументом которой является список, а выходом список, элементами которого являются произведения 1-го и последнего элемента исходного списка, 2-го и предпоследнего и т. д.
    Например:
    Вход: (1 2 3 4 5 6).
    Выход: (6 10 12).

12. Разработать функцию, удаляющую из исходного списка элементы, порядковые номера которых заданы во втором списке.
    Например:
    Вход: (a b c d e f g), (2 5).
    Выход: (a c d f g).

13. Разработать функцию, формирующую на основе исходного списка длины ***N*** список, содержащий суммы элементов с номерами 1 и ***N***/2+1, 2 и ***N***/2+2 и т. д. ***N*** — аргумент функции.
    Например:
    Вход: (1 2 3 4 5 6 7 8 9 10).
    Выход: (7 9 11 13 15).

14. Разработать функцию, преобразующую исходный список, в список «луковицу».
    Например:
    Вход: (3 2 1 2 3).
    Выход: (3 (2 (1) 2) 3).

15. Разработать функцию, преобразующую арифметическое выражение, заданное в форме списка, в польскую обратную запись.
    Например:
    Вход: (3 * 2 - 5).
    Выход: (- * 3 2 5).

16. Разработать функцию, выполняющую преобразование, обратное предыдущему.

17. Разработать функцию, перемещающую в исходном списке последовательность элементов, начиная с элемента с номером ***N*** и длины ***N*** + ***K***, на позицию за элементом с номером ***M***. ***N***, ***K***, ***M*** — аргументы функции.
    Например:
    Вход: (1 2 3 4 5 6 7 8 9), ***N*** = 3, ***K*** = 4, ***M*** = 8.
    Выход: (1 2 7 8 3 4 5 6 9).

18. Разработать функцию, возвращающую количество четных (по значению) элементов в списке чисел.
    Например:
    Вход: (1 2 3 4 5 6 7 8 9).
    Выход: 4.

19. Разработать функцию, увеличивающую в исходном списке нечетные элементы (по значению) на 1 и уменьшающую четные на 2.
    Например:
    Вход: (1 2 3 4 5 6 7 8 9).
    Выход: (2 0 4 2 6 4 6 6 10).

20. Разработать функцию, инвертирующую в исходном списке последовательность элементов, начиная с элемента с номером ***N*** и заканчивая элементом с номером ***N*** + ***K***. ***N*** и ***K*** — аргументы функции.
    Например:
    Вход: (1 2 3 4 5 6 7 8 9), ***N*** = 4, ***K*** = 3.
    Выход: (1 2 3 7 6 5 4 8 9).

21. Разработать функцию, заменяющую в исходном списке символов и всех его подсписках последовательность … a b a … на последовательность … a b b a … .
    Например:
    Вход: (a b c d a b a c (e f (d a b a) b a b) b a c).
    Выход: (a b c d a b b a c (e f (d a b b a) b a b) b a c).

22. Разработать функцию, возвращающую ***t***, если в исходном списке из нулей и единиц, больше нулей, ***nil*** — в противном случае.
    Например:
    Вход: (1 0 0 1 0 1 1 0 0).
    Выход: ***t***.

23. Разработать функцию, вычисляющую среднее арифметическое для заданного списка чисел.
    Например:
    Вход: (1 2 3 4 5).
    Выход: 3.

24. Разработать функцию, транспонирующую матрицу, заданную списком списков.
    Например:
    Вход: ((1 2 3)(4 5 6)(7 8 9)).
    Выход: ((1 4 7)(2 5 8)(3 6 9)).

25. Разработать функцию, осуществляющую сортировку списка чисел по возрастанию.
    Например:
    Вход: (1 9 4 7 3 6 8 5 2).
    Выход: (1 2 3 7 6 5 4 8 9).

26. Разработать функцию, осуществляющую сортировку списка методом двухфазного слияния.

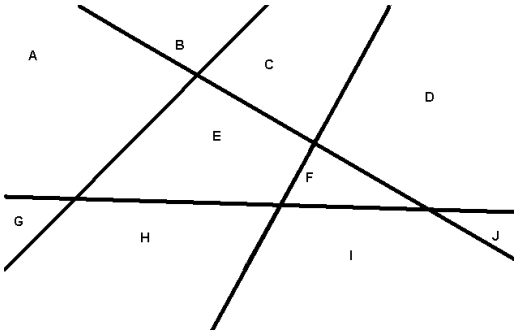
Лабораторная работа 3

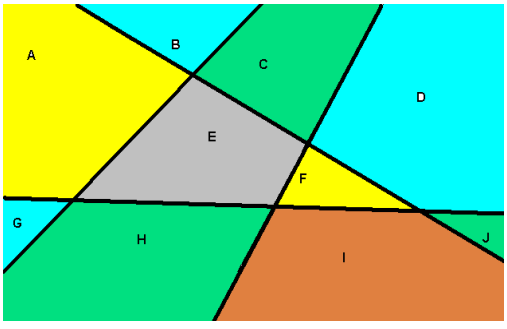
Цель работы - исследование класса задач, решаемых полным перебором и методов их решения средствами Lisp.

Указания. Используя списковое представление графов и рекурсивные функции, разработать программу, находящую решение, в соответствии с приведенными ниже вариантами. Не разрешается использовать операторы цикла и оператор PROG.

Пример решения.

Дан некоторый произвольный плоский граф. Необходимо раскрасить все его внутренние области так, чтобы никакие две соседние области не имели одинаковый цвет (см. рис)





.

В общем случае данная задача решается полным перебором, однако существует алгоритм, позволяющий подобрать достаточно хорошее решение за гораздо меньшее число шагов. Решим задачу с помощью этого алгоритма.

Подбор “*достаточно хорошего решения*” “*жадным*” алгоритмом. Алгоритм решения заключается в последовательном повторении следующих двух шагов:

а) выбираем новый цвет

b) закрашиваем им все возможные области, выбирая их в произвольном порядке

В тот момент, когда все вершины будут закрашены, нужное нам решение и будет найдено.

Зададим граф, изображённый на рисунке в примере к задаче, следующим образом:

```
>(SETQ MAP '((A 0 (B C G H E))
             (B 0 (A C E))
             (C 0 (B A E F D))
             (D 0 (C E F I J))
             (E 0 (A B C D F G H I))
             (F 0 (C D E H I J))
             (G 0 (A E H))
             (H 0 (A E F G I))
             (I 0 (C D E F J H))
             (J 0 (D F I))
      )
)
```

Каждый из элементов списка представляет собой тройку вида (имя_области цвет_области список_смежных_областей). Таким образом, например, область a имеет цвет 0 (т.е. не задан) и граничит с областями b,c,g,h и e.

Напишем несколько вспомогательных функций. Первая функция определяет номер области в списке областей по её имени:

```
>(DEFUN GETNODEBYNAME (NAME MAP)
      (COND
            ((NULL MAP) 0)
            ((EQ (NTH 0 (NTH 0 MAP)) NAME) 1)
            (T (SETQ NODEID
            (GETNODEBYNAME NAME (CDR MAP)))
            (IF (EQL NODEID 0) 0 (+ 1 NODEID)))))
```

Функция возвращает номер записи в списке считая от единицы, нуль означает, что такой записи не найдено.

Следующая функция проверяет, является ли некоторые две области смежными:

```
>(DEFUN ARENEIGHBOURS (NODE1 NODE2 MAP)
      (COND
            ((NULL (MEMBER NODE2 (NTH 2 (NTH
                  (- (GETNODEBYNAME NODE1 MAP) 1) MAP)))) NIL)
            (T T)))
```

Следующая функция проверяет, можно ли закрасить некоторый узел указанным цветом, и возвращает в зависимости от этого T или NIL

```
>(DEFUN ISNODEPAINTABLE (NODE COLOR MAP MAPREST)
        (COND
                ((NULL MAPREST) T)
                ((EQ (CAAR MAPREST) NODE)
                (ISNODEPAINTABLE NODE COLOR MAP (CDR MAPREST)))
                ((AND (ARENEIGHBOURS NODE (CAAR MAPREST) MAP)
                (EQ (CADAR MAPREST) COLOR)) NIL)
                                (T(ISNODEPAINTABLE NODE COLOR MAP
                                (CDR MAPREST)))))
```

Ещё одна функция получает произвольную область, которую можно закрасить текущим цветом, если же такой области нет, возвращает nil.

```
>(DEFUN GETFIRSTPAINTABLE (COLOR MAP MAPREST)
        (COND
          ((NULL MAPREST) NIL)
          ((AND (ISNODEPAINTABLE (CAAR MAPREST)
                COLOR MAP MAP)  (NOT (EQ (CADAR MAPREST) COLOR) )
                (EQ (CADAR MAPREST) 0) )(CAAR MAPREST))
         (T (GETFIRSTPAINTABLE COLOR MAP (CDR MAPREST)))))
```

Ещё две сервисные функции – закрасить область и проверить, есть ли ещё хоть одна незакрашенная область

```
>(DEFUN PAINTREGION (NODE COLOR MAP)
        (PROGN
                (SETF (NTH 1 (NTH (- (GETNODEBYNAME NODE MAP)  1)
                        MAP)) COLOR)
                MAP))

>(DEFUN UNPAINTEDYET (MAP)
   (COND
    ((NULL MAP) NIL)
    ((EQ (CADAR MAP) 0) T)
    (T (UNPAINTEDYET (CDR MAP)))))
```

Следующая функция закрашивает некоторым цветом все доступные узлы:

```
>(DEFUN PAINTMAPWITHCOLOR (COLOR MAP)
   (COND
        ((NULL (GETFIRSTPAINTABLE COLOR MAP MAP)) MAP)
        (T (PROGN (PAINTREGION (GETFIRSTPAINTABLE
                COLOR MAP MAP) COLOR MAP) (PAINTMAPWITHCOLOR
                COLOR MAP)))))
```

Наконец, собственно тело программы

```
>(DEFUN PAINTALL (MAP COLOR)
   (PROGN
        (PAINTMAPWITHCOLOR COLOR MAP)
```

(IF (UNPAINTEDYET MAP) (PAINTALL MAP (+ 1 COLOR)) MAP)))

Контрольный пример.

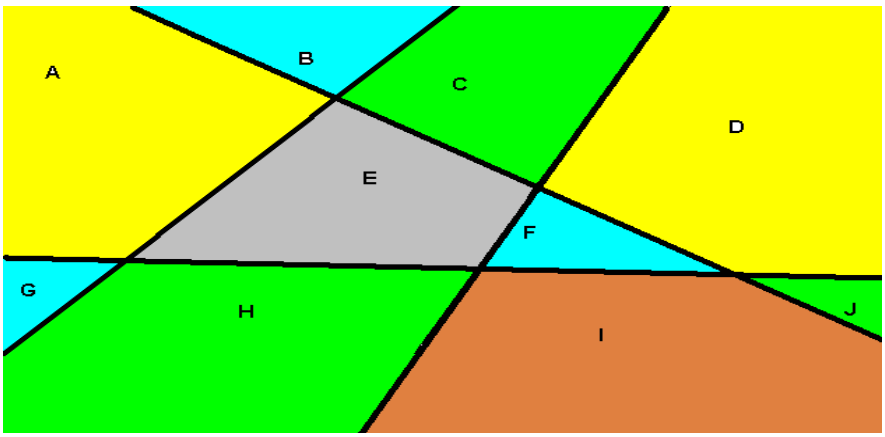
```
>(SETQ MAP '((A 0 (B C G H E))
             (B 0 (A C E))
             (C 0 (B A E F D))
             (D 0 (C E F I J))
             (E 0 (A B C D F G H I))
             (F 0 (C D E H I J))
             (G 0 (A E H))
             (H 0 (A E F G I))
             (I 0 (C D E F J H))
             (J 0 (D F I))
    )
)
```

((A 0 (B C G H E)) (B 0 (A C E)) (C 0 (B A E F D)) (D 0 (C E F I J)) (E 0 (A B C D F G H I)) (F 0 (C D E H I J)) (G 0 (A E H)) (H 0 (A E F G I)) (I 0 (C D E F J H)) (J 0 (D F I)))

> (PAINTALL MAP 1)

((A 1 (B C G H E)) (B 2 (A C E)) (C 3 (B A E F D)) (D 1 (C E F I J)) (E 4 (A B C D F G H I)) (F 2 (C D E H I J)) (G 2 (A E H)) (H 3 (A E F G I)) (I 5 (C D E F J H)) (J 3 (D F I)))

Таким образом, для раскраски графа нам понадобилось пять цветов. Результат выполнения программы приведён на рисунке:



Задания для лабораторной работы

1. **Дана схема метрополитена, найти кратчайший путь между станциями.**

Схема метрополитена задаётся с помощью матрицы смежности или матрицы инциденций. Каждому перегону соответствует некоторый вес (длительность перегона). Каждой пересадке также соответствует некоторый вес (длительность пересадки). Необходимо для заданной преподавателем схемы вывести самый короткий путь или все такие пути, если их несколько.

2. Игра в 15.

Задача состоит в том, что на прямоугольном поле 4x4 расставлены прямоугольные фишки с номерами от 1 до 15 в произвольном порядке, также имеется одно пустое поле. За каждый ход можно передвинуть на пустое поле одну фишку. Задача игры состоит в том, чтобы упорядочить фишки по номерам от 1 до 15. Обратите внимание, что не для всех начальных состояний задача имеет решение. Начальная позиция вводится преподавателем и программа должна вывести наикратчайший алгоритм решения задачи, или все такие алгоритмы, если их несколько, или сообщить, что таких алгоритмов нет.

3. Задача о 8-ми ферзях.

Необходимо расставить на шахматной доске 8 ферзей в соответствии со стандартными правилами, так, чтобы никакие два ферзя не били друг друга (ферзь бьёт любую фигуру, находящуюся с ним на одной вертикали, горизонтали и диагонали)

4. Задача о 8-ми ферзях 2.

Необходимо расставить на шахматной доске фигуры, которые сочетают в себе одновременно свойства ладьи и коня и определить, какое наибольшее число таких фигур можно расставить на доске 8x8 (ладья бьёт любую фигуру, находящуюся с ней на одной вертикали или горизонтали, конь бьёт любую фигуру находящуюся через две клетки по горизонтали или вертикали и одну клетку по диагонали)

5. Задача о Ханойской башне.

Имеются три стержня, на один из них нанизано n дисков, остальные пустые. Диски имеют различный диаметр, упорядоченный от самого узкого наверху до самого широкого внизу. Разрешается перекладывать диски с одного стержня на другой, при условии, что ни при каких обстоятельствах более широкий диск не будет лежать сверху на более узком. Необходимо вывести последовательность действий, при которой пирамида будет перенесена с одного стержня на другой[4].

6. Задача о составлении расписания

Проводится турнир по круговой схеме, в котором участвуют N игроков. Необходимо составить расписание встреч участников так, чтобы каждый игрок сыграл с каждым из своих противников, и при этом играл ежедневно один матч.

[4] При этом запрещается экспериментировать с числом дисков, равным 64, так как если кому-либо это удастся, согласно легенде о ханойских башнях, наступит конец света

7. **Раскрасить плоскую карту четырьмя цветами, так что бы любые две смежные области не были окрашены в один цвет**

Одно из решений данной задачи разобрано в примере, здесь следует предложить решение, которое даст наименьшее количество цветов и выведет все возможные варианты окраски ними карты.

.

8. **Задача о коммивояжере.**

Коммивояжер должен посетить клиентов, находящихся в разных городах. Коммивояжер возвращается в тот же город, из которого он выехал. Коммивояжер никогда не бывает дважды в одном и том же городе. Необходимо предложить порядок посещения городов, так, чтобы пройденный путь был минимальным. Данная задача решается только методом полного перебора. Необходимо по заданной преподавателем топологии местности вывести все кратчайшие маршруты движения коммивояжера либо сообщить, что таких маршрутов нет.

9. **Нахождение центральной вершины орграфа**

Дан некоторый связный ориентированный граф. Необходимо найти в нём центральную вершину (наиболее равноудалённую ото всех остальных). Наиболее равноудалённая вершина может быть получена как вершина, среднее расстояние от которой до других вершин наиболее близко к среднему значению этой величины для всех вершин графа. Если таких вершин несколько, вывести их все.

10. **Нахождение центральной вершины неориентированного графа**

Дан некоторый связный неориентированный граф. Необходимо найти в нём центральную вершину (наиболее равноудалённую ото всех остальных). Наиболее равноудалённая вершина может быть получена как вершина, среднее расстояние от которой до других вершин наиболее близко к среднему значению этой величины для всех вершин графа. Если таких вершин несколько, вывести их все.

11. **Нахождение минимально функционирующего сегмента сети.**

Дана некоторая сеть (в виде графа). Каждой её дуге сопоставлена некоторая пропускная способность. Также две вершины помечены как источник и приёмник данных. Необходимо определить, какой минимальный фрагмент этой сети обладает той же пропускной способностью между источником и приёмником, что и исходная сеть.

12. **Построить остовное дерево минимальной стоимости для произвольного неориентированного связного графа**

Остовным деревом для графа называется любое свободное дерево, являющееся его подграфом. Под стоимостью понимается совокупный вес всех оставшихся в этом подграфе дуг. Необходимо для заданного неориентированного графа вывести все его остовные деревья минимальной стоимости.

13. **Для заданного орграфа определить минимальный набор узлов, удаление которых (вместе с входящими или исходящими рёбрами) приведёт к разделению исходного графа на заданное число подграфов.**

Необходимо привести все решения с минимальным набором узлов, либо сообщить, что таких решений нет.

14. Для заданного неориентированного графа определить минимальный набор узлов, удаление которых (вместе с входящими или исходящими рёбрами) приведёт к разделению исходного графа на заданное число подграфов.

Необходимо привести все решения с минимальным набором узлов, либо сообщить, что таких решений нет.

15. Необходимо определить степень связности орграфа

Под степенью связности графа понимается некоторое число R, такое, что между любыми двумя узлами графа имеются не менее R путей

16. Необходимо определить степень связности неориентированного графа

Под степенью связности графа понимается некоторое число R, такое, что между любыми двумя узлами графа имеются не менее R путей

17. Задача о триангуляции многоугольника.

Дан выпуклый многоугольник, определенный через координаты вершин на плоскости. Необходимо провести непересекающиеся хорды внутри этого многоугольника так, чтобы он весь был разбит на треугольники и при этом суммарная длина хорд была минимальна.

18. Построить дерево игры в крестики-нолики на доске MxM

Необходимо из заданной позиции построить дерево игры и указать, существует ли алгоритм, позволяющий выиграть игроку, делающему ход.

Для решения задачи можно, например, построить дерево игры (т.е. все возможные осмысленные продолжения) после чего определить выигрышный путь следующим образом:

- позиция считается выигрышной, если игрок поставил 5 фишек в ряд
- позиция считается выигрышной, если существует ход, после которого независимо от хода противника игра переходит в выигрышную позицию.

19. Определить, является ли заданный граф свободным деревом

Свободное дерево – дерево, не содержащее циклов

20. Вывести в произвольном орграфе все имеющиеся циклы без дублирования

21. Вывести в произвольном неориентированном графе все имеющиеся циклы без дублирования

22. Задача о покрытии плоскости тетрадами

Тетрада – квадрат, на каждой стороне которого написано некоторое число. Имеется бесконечное количество тетрад нескольких заданных типов. Необходимо вывести все возможные варианты покрытия прямоугольной области заданного размера, избегая повторений. Основное условие – тетрады

могут соприкасаться только теми гранями, на которых написаны одинаковые числа.

23. Задача об устойчивых браках. (см. Шапиро, стр. 173, п. 2).

Лабораторная работа 4

Цель работы. Ознакомиться с применением функционалов в языке Lisp.

Указания. Используя связывание свойств с символами, структуры, последовательности и функционалы реализовать программу для предложенного ниже варианта, выполняющую:

1. включение/исключение/замену данных;
2. 4-5 поисковых запроса (разработать самостоятельно и согласовать с преподавателем).

   Формальным требованием является использование последовательностей и функционалов при выполнении задания

1. Расписание авиарейсов.
2. Железнодорожное расписание.
3. Купля/продажа/сдача/наем/обмен недвижимости.
4. Телефонный справочник организации.
5. Учет результатов экзаменационных сессий.
6. Учет технических осмотров автомобилей в автопарке.
7. Учет выданных книг в библиотеке.
8. Инвентаризация лабораторий университета: аудитории, установленная техника.
9. Маршруты городского транспорта.
10. Репертуар театров.
11. Котировки ценных бумаг.
12. Запись к врачам в поликлинике.
13. Проведение научной конференции.
14. Учет товаров на складе.
15. Учёт ошибок при разработке программного обеспечения

Указатель функций

Список литературы

1 Хювенен Э. Сеппенян И. “Мир Лиспа”. М, Мир, 1990.

2 http://www.elwood.com/alu/table/bibliography.htm

3 Ахо А, Хопкрофт Д, Ульман Дж, Структуры данных и алгоритмы. Вильямс, 2001

4 Головешкин В.А. Ульянов М.В. Теория рекурсии для программистов. Москва, ФИЗМАТЛИТ 2006