Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

Н. Ю. Салмина

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ И ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ

Учебное пособие

Томск
2016

УДК 004.42.046 + 004.89
ББК 32.973.2-018я73 + 32.813я73
С 164

Рецензенты:

А. В. Ковшов, старший преподаватель кафедры
автоматизированных систем управления ТУСУР;
В. Ф. Тарасенко, д.т.н., профессор кафедры теоретической кибернетики ТГУ

Салмина Н. Ю.

С 164 Функциональное программирование и интеллектуальные системы : учебное пособие / Н. Ю. Салмина. – Томск : ФДО, ТУСУР, 2016. – 100 с.

В настоящем учебном пособии изложены основные положения функционального программирования на примере языка Лисп. Рассматриваются принципы функционального программирования, применения лямбда-выражений и написания собственных функций. Большое внимание уделено использованию рекурсии при написании программ. Проанализированны две модели представления знаний: фреймы и семантические сети, а также способы их построения и использования средствами языка Лисп.

Теоретический материал иллюстрируется многочисленными примерами.

Для студентов, обучающихся по направлению 38.03.05 «Бизнес-информатика» и родственным направлениям.

© Салмина Н. Ю., 2016
© Оформление.
ФДО, ТУСУР, 2016

Оглавление

Предисловие ... 5
Введение ... 7
1 Язык Лисп. Основы программирования ... 13
1.1 Лисп. Элементарные понятия ... 13
1.1.1 Атомы и списки как основные объекты языка Лисп ... 13
1.1.2 Внутреннее представление списков ... 14
1.1.3 Понятие функции ... 16
1.2 Программа на языке Лисп. Вычислимые выражения ... 17
1.3 Базовые функции языка ... 18
1.4 Лямбда-выражения и определение новых функций ... 27
2 Рекурсивные функции ... 30
2.1 Понятие рекурсии ... 30
2.2 Как работает рекурсивная функция ... 31
2.3 Правила записи рекурсивной функции ... 32
2.4 Рекурсия с несколькими терминальными или рекурсивными ветвями ... 34
2.5 Вспомогательные функции над списками ... 38
3 Технология программирования на языке Лисп ... 44
3.1 Передача параметров. Глобальные и локальные переменные ... 44
3.2 Диалоговый режим работы. Функции ввода-вывода ... 45
3.3 Разрушающие функции ... 50
3.4 Функционалы ... 53
3.5 Циклы и блочные функции ... 57
3.5.1 Блочные функции ... 57
3.5.2 Циклические предложения ... 60
3.6 Массивы ... 66
3.7 Свойства символов ... 68
3.8 Ассоциативные списки ... 70
3.9 Работа с файлами ... 72
3.9.1 Определение входных и выходных потоков ... 72
3.9.2 Чтение символов из файла ... 75
4 Модели представления знаний ... 77
4.1 Фреймы ... 77
4.1.1 Понятие и состав фрейма ... 77
4.1.2 Пример построения фреймовой структуры на Лиспе ... 79

4.2 Семантические сети 88
4.2.1 Представление знаний с помощью семантических сетей 88
4.2.2 Пример построения семантической сети с помощью Лиспа 93
Заключение 96
Литература 97
Глоссарий 98

Предисловие

Функциональное программирование называется так, потому что программа полностью состоит из функций. Сама программа тоже является функцией, которая получает исходные данные в качестве аргумента, а выходные данные выдает как результат. Как правило, основная функция определена в терминах других функций, которые в свою очередь определены в терминах еще большего количества функций, вплоть до функций – примитивов языка на самом нижнем уровне. Эти функции очень похожи на обычные математические функции.

Особенности и преимущества функционального программирования обычно излагаются следующим образом. Функциональные программы не содержат операторов присваивания, а переменные, получив однажды значение, никогда не изменяются. Более того, функциональные программы вообще не имеют побочных эффектов. Обращение к функции не вызывает иного эффекта кроме вычисления результата. Это устраняет главный источник ошибок и делает порядок выполнения функций несущественным: так как побочные эффекты не могут изменять значение выражения, оно может быть вычислено в любое время. Программист освобождается от бремени описания потока управления. Поскольку выражения могут быть вычислены в любое время, можно свободно заменять переменные их значениями и наоборот, то есть программы «прозрачны по ссылкам». Эта прозрачность ссылок делает функциональные программы более удобными для математической обработки, по сравнению с общепринятыми аналогами.

В данном пособии в качестве одного из функциональных языков рассматривается язык Лисп.

В практических интеллектуальных системах наибольшее внимание привлекают два блока: база знаний и решатель задач. Качество работы интеллектуальной системы в значительной мере определяется тем, как устроены эти блоки и насколько удачно организовано взаимодействие между ними.

Выбор модели представления знаний связан с видом зависимостей знаний с описанием тех или иных процессов во времени, прогнозом этих процессов. В данном учебном пособии рассмотрены два класса моделей представления знаний: фреймы и семантические сети.

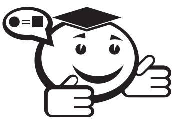
Другой проблемой в построении интеллектуальных систем является выбор инструментального средства и его реализация. В качестве инструментального средства предлагается использовать функциональный язык Лисп.

Для усвоения основных результатов и понятий, приведенных в учебном пособии, достаточно знания курса информатики в объеме университетской программы. Во всех главах содержатся многочисленные примеры, иллюстрирующие основные принципы функционального программирования.

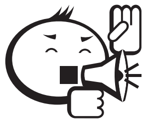
Основной задачей курса «Функциональное программирование и интеллектуальные системы» является формирование у студентов профессиональных знаний и практических навыков по разработке алгоритмов и написанию программ средствами функционального программирования, представлению знаний и разработке моделей интеллектуальных систем.

Соглашения, принятые в учебном пособии

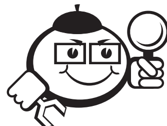
Для улучшения восприятия материала в данном учебном пособии используются пиктограммы и специальное выделение важной информации.



Эта пиктограмма означает определение или новое понятие.



Эта пиктограмма означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Контрольные вопросы по главе

Введение

Концепция функционального программирования

В основе функционального программирования лежит понятие функции. Вспомнив историю математики, можно оценить возраст понятия «функция»: ему уже около четырехсот лет, и математики придумали очень много теоретических и практических аппаратов для оперирования функциями, начиная от обыкновенных операций дифференцирования и интегрирования, заканчивая заумными функциональными анализами, теориями нечетких множеств и функций комплексных переменных.

Математические функции выражают связь между исходными данными и итоговым продуктом некоторого процесса. Процесс вычисления также имеет вход и выход, поэтому функция – вполне подходящее и адекватное средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования.

Функциональным называется *программирование* при помощи функций в математическом их понимании. Функциональное программирование основано на следующей идее: в результате каждого действия возникает значение, которое может быть аргументом следующего действия. Программы строятся из логически расчлененных определений функций. Каждое определение функции состоит из организующих вычисления управляющих структур и из вложенных, в том числе вызывающих самих себя (рекурсивных), вызовов функций. При выполнении программы функции получают параметры, вычисляют и возвращают результат, при необходимости вычисляя значения других функций. На функциональном языке программист не должен описывать порядок вычислений. Нужно просто описать желаемый результат как систему функций.

Выделим основные особенности и преимущества языков функционального программирования (ФП).

Особенности функционального программирования [1]:

1. Вызов функций является единственной разновидностью действий, выполняемых в функциональной программе.

2. В алгоритмических языках программа является последовательностью операторов, вызовов процедур в соответствии с алгоритмом. В функциональном программировании программа состоит из вызовов функций и описывает то,

что нужно делать и что собой представляет результат решения, а не как нужно действовать для получения результата.

3. Основными методами программирования являются суперпозиция функций и рекурсия.

4. Функциональное программирование есть программирование, управляемое данными. В строго функциональном языке однажды созданные (введенные) данные не могут быть изменены!

5. В алгоритмических языках с именем переменной связана некоторая область памяти, соответствие строго сохраняется в течение всего времени выполнения программы. В функциональном программировании переменная обозначает только имя некоторой структуры, имена символов, переменных, списков, функций и других объектов не закреплены предварительно за какими-либо типами данных. В ФП одна и та же переменная в различные моменты времени может представлять различные объекты.

6. В языках функционального программирования программа и обрабатываемые ею данные имеют единую списочную форму представления.

7. Функциональное программирование предполагает наличие функционалов – функций, аргументы и результаты которых могут быть функциями. Всякий язык функционального программирования предполагает наличие ядра, называемого строго функциональным языком.

Требования к строго функциональному языку:

1. Всякая функция должна однозначно определять результат по любому набору аргументов.

2. Отсутствует оператор присваивания.

3. Переменная обозначает только имя структуры.

4. В языке присутствуют функционалы.

Основные преимущества языков ФП [1]:

1. Краткость программы. Программы на функциональных языках обычно короче и проще, чем те же самые программы на императивных языках.

2. Функции – это значения. Функциональные программы поддаются формальному анализу легче своих аналогов на алгоритмических языках за счет использования математической функции в качестве основной конструкции: функции могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата.

3. Чистота. В императивных языках функция в процессе своего выполнения может читать и изменять значения глобальных переменных и осуществлять

операции ввода-вывода. Поэтому, если вызвать одну и ту же функцию дважды с одним и тем же аргументом, может случиться так, что в качестве результата будет вычислено два различных значения. Изменение функцией состояния программы иначе, чем через возвращение значения, называется побочным эффектом. В чистом функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые путем разбора и сбора уже существующих объектов. О ненужных объектах позаботится встроенный в язык сборщик мусора. Благодаря этому в чистых функциональных языках все функции свободны от побочных эффектов.

4. Возможность реализации на ЭВМ с параллельной архитектурой. Раз все функции для вычислений используют только свои параметры, мы можем вычислять независимые функции в произвольном порядке или параллельно, на результат вычислений это не повлияет. Причем параллелизм может быть организован не только на уровне компилятора языка, но и на уровне архитектуры. В нескольких научных лабораториях уже разработаны и используются экспериментальные компьютеры, основанные на подобных архитектурах. В качестве примера можно привести Lisp-машину.

Основные свойства функциональных языков:

- краткость и простота: программы на функциональных языках обычно короче и проще, чем те же самые программы на императивных языках;
- строгая типизация;
- модульность;
- функции – это значения;
- чистота (отсутствие побочных эффектов);
- отложенные (ленивые) вычисления.

Символьная обработка и искусственный интеллект

Искусственный интеллект (ИИ) – это область исследований по моделированию интеллектуальной деятельности человека для решения различных задач. ИИ находится на стыке ряда наук: информатики, языкознания (математической лингвистики), психологии (когнитивной науки) и философии. Задачи ИИ требуют работы с данными и знаниями в виде символьных структур. В обработке символьной информации важна не только форма рассматриваемых знаний, но и их содержание и значение. Причиной возникновения в конце 1950-х гг. потребности в специализированных инструментальных программных средствах символьной обработки послужила неспособность процедурных языков отражать

естественным образом в виде чисел и массивов объектов и ситуаций реального мира. Указанный недостаток процедурных средств, в частности, затруднял реализацию применяемых в решении задач ИИ эвристических методах.

Функциональное программирование, как и логическое программирование, нашло большое применение в теории искусственного интеллекта и ее приложениях.

Как известно, теоретические основы императивного программирования были заложены еще в 1930-х гг. Аланом Тьюрингом и Джоном фон Нейманом. Теория, положенная в основу функционального подхода, также родилась в 1920–1930-х гг. В числе разработчиков математических основ функционального программирования можно назвать Моисея Шейнфинкеля и Хаскелла Карри, разработавших комбинаторную логику, и Алонзо Черча, создателя λ-исчисления.

Теория так и оставалась теорией, пока в конце 1950-х гг. Джон Маккарти не разработал язык Лисп, который стал первым функциональным языком программирования и многие годы оставался единственным таковым. Лисп все еще используется, после многих лет эволюции он удовлетворяет современным запросам, которые заставляют разработчиков программ взваливать как можно бóльшую нóшу на компилятор, облегчив таким образом свой труд. Нужда в этом возникла из-за все более возрастающей сложности программного обеспечения.

Широкое распространение язык получил в конце 1970-х – начале 1980-х гг. с появлением необходимой мощности вычислительных машин и соответствующего круга задач. *В настоящее время Лисп является одним из главных инструментальных средств систем искусственного интеллекта*. Он принят как один из двух основных языков программирования для Министерства обороны США и постепенно вытесняет второй язык – Ада. На *Лиспе* разработана и система AutoCAD.

Основные особенности Лиспа [2]:

- представление программы и данных производится одинаково – через списки: основными объектами, с которыми оперирует Лисп, являются списки. Отсюда происходит и название языка: словосочетание list processing означает «обработка списков». Это позволяет программе обрабатывать другие программы и даже саму себя;
- Лисп, как правило, является интерпретирующим языком;

- Лисп является бестиповым языком: это значит, что символы не связываются по умолчанию с каким-либо типом;
- Лисп имеет необычный синтаксис. Из-за большего числа скобок LISP расшифровывают как Lots of Idiotic Silly Parentheses;
- программы, написанные на Лиспе, во много раз короче, чем написанные на процедурных языках.

В основе Лиспа лежит функциональная модель вычислений, ориентированная прежде всего на решение задач нечислового характера.

В 1960-е гг. сформировался базовый диалект языка – LISP 1.5. Следует отметить, что, в отличие от других языков программирования (например, Си, Паскаля), не существует стандарта языка Лисп. Более того, различные диалекты Лиспа, хотя во многом и похожи друг на друга, имеют незначительные синтаксические различия, которые не позволяют выделить подмножество, общее для всех диалектов. Вместе с тем в языке Лисп имеются средства, позволяющие в рамках одного диалекта реализовать конструкции другого диалекта.

Рассмотрим *основные достоинства языка Лисп* [2].

1. Лисп ориентирован на решение задач символьной обработки. Это удобно при выполнении формульных преобразований.

2. В языке Лисп текст программы и данные представляются в единой форме – в виде списков. Более того, в процессе вычислений можно формировать фрагменты программы и выполнять их. Это позволяет легко организовывать преобразование программ и их хранение, что удобно при организации баз знаний, необходимых в системах искусственного интеллекта.

3. Лисп предоставляет богатые средства для хранения данных об объектах различной природы, в частности геометрических. Это достоинство языка используется в системе автоматизированного проектирования AUTOCAD.

4. Другая особенность Лиспа – это удобство работы с динамическими структурами данных. Программист может вводить новые структуры, «забывая» старые, не заботясь о том, чтобы очищать память от ненужных структур, как это приходится делать, например, в языках Паскаль, Си. В Лисп-системе есть специальное средство «сборщик мусора», которое автоматически освобождает память от неиспользуемых структур.

Обычно транслятор с языка Лисп представляет собой интерпретатор, что предоставляет готовые средства для создания различных диалоговых систем.

Как известно, вычисления в интерпретаторе выполняются медленнее, чем в скомпилированной программе. Поэтому было бы бессмысленно использовать

Лисп при решении больших вычислительных задач, тем более что запись арифметических выражений в Лиспе менее наглядна, чем в алгоритмических языках.

Вместе с тем есть диалекты Лиспа (COMMON LISP, MULISP-85), позволяющие выполнять точные вычисления в рациональных числах, практически любой величины. Такие диалекты можно использовать для нахождения точных значений различных коэффициентов, представляемых рациональными числами.

1 Язык Лисп. Основы программирования

1.1 Лисп. Элементарные понятия

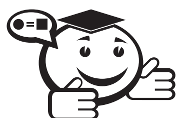
1.1.1 Атомы и списки как основные объекты языка Лисп

Основу Лиспа составляют символьные выражения, которые называются *S-выражениями* и образуют область определения для функциональных программ. S-выражение – это основная структура данных в Лиспе.

Примеры S-выражений:

```
(ДЖОН СМИТ 33 ГОДА)
((МАША 21) (ВАСЯ 24) (ПЕТЯ 1))
```

S-выражение может быть представлено либо атомом, либо списком, которые и являются основными объектами языка Лисп. Они служат как для представления данных, так и для представления программы на языке Лисп.



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

Атомы бывают двух типов: символьные и числовые.

Символьный атом – последовательность букв и цифр, при этом должен быть по крайней мере один символ, отличающий его от числа, например:

```
ДЖОН AB13 B54 10A
```

Символьный атом или символ (идентификатор) – это не идентификатор переменой в обычном языке программирования. Символ, как правило, обозначает какой-либо предмет, объект, вещь, действие.

Символьный атом рассматривается как неделимое целое. К символьным атомам применяется только одна операция: сравнение.

Числовой атом – это обыкновенное число, которое является константой:

```
345
-23
1.247
```

Список представляет собой заключенную в круглые скобки последовательность выражений языка Лисп, между которыми могут стоять пробелы. Если в списке рядом стоят два атома, то между ними следует поместить хотя бы один пробел. Несколько пробелов эквивалентны одному. Следующие выражения являются списками:

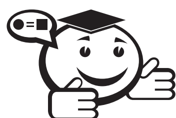
`()` – пустой список;

`(APPLE BOOK CAT)` – список из трех элементов-атомов;

`((AN APPLE) (Q (P R)))` – список из двух элементов;

`(+ 2 3)` – 3 атома;

`(((((первый) 2) второй) 4) 5)` – 2 элемента.



*Таким образом, **список** – это многоуровневая или иерархическая структура данных, в которой открывающиеся и закрывающиеся скобки находятся в строгом соответствии.*

Как видно, элементами списка могут быть другие списки. Выражение `()` представляет собой пустой список, не содержащий ни одного элемента. Он имеет также и другое представление – `NIL`. Указанные два обозначения пустых списков полностью эквивалентны.

Пустой список является одновременно и атомом, и списком.

Пустой список играет такую же важную роль в работе со списками, что и ноль в арифметике. Так же, как и другие S-выражения, он может быть элементом других списков:

`(NIL)` – список состоящий из атома `NIL` (этот список уже не пуст!);

`(())` – то же самое, что и `(NIL)`;

`((()))` – эквивалентно `((NIL))`;

`(NIL ())` – список из двух других списков.

Кроме того, *атом* `NIL` в логических выражениях обозначает логическую константу *ложь (false).*

Логическое *да (true)* задается *атомом* `T`.

1.1.2 Внутреннее представление списков

Мы рассмотрели внешнее представление списков. Рассмотрим внутреннюю структуру памяти в Лиспе и представление в ней S-выражений.

Каждый атом занимает ячейку.

Списки являются совокупностью атомов и связываются вместе специальными элементами памяти, называемыми списочными ячейками или cons-ячейками. Каждая списочная ячейка состоит из двух частей (полей). Левое поле содержит указатель на первый элемент списка, который называется головой (`CAR`), правое поле содержит указатель на хвост списка (`CDR`).

Таким образом, список можно изобразить в виде звеньев, соединенных ссылками; каждое звено состоит из пары ссылок. Так, для списка `((M N) 1.5 7)` можно изобразить внутреннее представление, отраженное на рисунке 1.1.

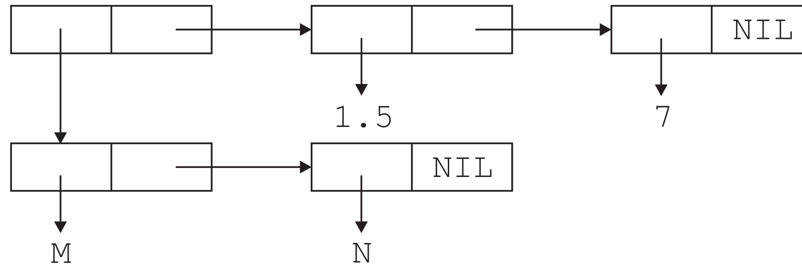


Рис. 1.1 – Внутреннее представление списка `((M N) 1.5 7)`

Здесь `NIL` указывает на конец списка.

В приведенном внутреннем представлении списка все правые ссылки в звеньях указывают на списки. Естественно возникает вопрос: «Какая конструкция получится, если некоторые из правых ссылок будут указывать на атомы, не являющиеся списками?». Такая конструкция будет списком, но с особым внешним представлением.

Рассмотрим внутреннее представление списка на рисунке 1.2.

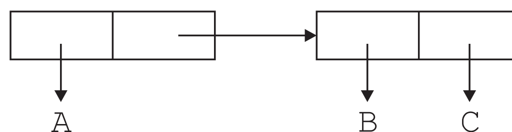


Рис. 1.2 – Внутреннее представление списка с точечной парой

Внешнее представление такого списка будет следующим: `(A B . C)`.

В записях таких конструкций используется точка, слева и справа от которой обычно ставятся пробелы. Обратите внимание на разницу конструкций `(1.5)` и `(1 . 5)`. Внутреннее представление этих списков представлено на рисунке 1.3.

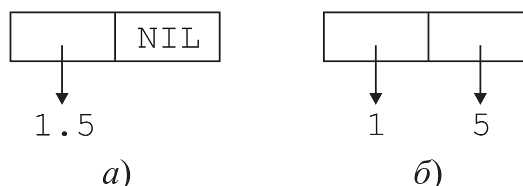


Рис. 1.3 – Разница во внутреннем представлении точечной пары и числа: *а*) внутреннее представление конструкции `(1.5)`; *б*) внутреннее представление конструкции `(1 . 5)`

1.1.3 Понятие функции

В математике *функция* отображает одно множество в другое. Обычная запись функции выглядит следующим образом: `Y = F(X)`, где `X` является аргументом функции, `Y` – ее значением, а `F` определяет вид функции.

У функции может быть любое количество аргументов, в том числе их может не быть совсем.

Функция без аргументов имеет постоянное значение.

Префиксная нотация

В математике принята префиксная нотация, в которой имя функции стоит перед аргументами, заключенными в скобках:

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

В арифметических выражениях используется инфиксная запись:

```
x + y
x – y
x * ( x + z )
```

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

```
( f x )
( g x y )
( h x ( g y z ) )
( + x y )
( - x y )
( * x ( + x z ) )
```

Такой вид записи имеет явные преимущества перед префиксной нотацией:

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

1.2 Программа на языке Лисп. Вычислимые выражения

Программа на языке Лисп представляет собой последовательность вычислимых выражений, т. е. выражений, имеющих значения. Значения вычислимых выражений могут быть получены в процессе вычислений. Выполнение программы заключается в последовательном вычислении входящих в нее выражений.

Транслятор Лиспа работает, как правило, в режиме интерпретатора.

В Лиспе сразу читается, а затем вычисляется (evaluate) значение функции и выдается значение. Для обозначения того, какое значение имеет выражение, будем справа от выражения помещать знак ==>, а следом за ним вырабатываемое значение. Например:

```
( + 2 3 ) ==> 5
```

Во вводимую функцию могут входить функциональные подвыражения:

```
(* ( + 1 2 ) ( + 3 4 )) ==> 21
```

Рассмотрим вычислимые выражения и правила получения их значений.

Число представляет собой вычислимое выражение, значением которого является само это число. Таким образом:

```
6.0   ==>  6.0
-8.32 ==> -8.32
```

Атомы `T` *и* `NIL` имеют значения, совпадающие с ними самими. Так:

```
T   ==> T
( ) ==> NIL
```

Напомним, что записи `( )` и `NIL` эквивалентны.

Другие *идентификаторы* имеют значения только тогда, когда являются именами переменных. В таком случае значением идентификатора будет значение соответствующей переменной. Например, если значением переменной `X` является список `(A B C)`, то

```
X ==> (A B C)
```

Список вида `(f a1 a2 ... an)` *или* `(g)` имеет значение, если `f` или `g`, соответственно, является обозначением функции, при этом `a1`, `a2`, `...`, `an` являются аргументами (фактическими параметрами) функции `f`, а функция `g` не имеет аргументов. Обозначение функции представляет собой либо имя функции, либо λ-выражение, которое мы рассмотрим в дальнейшем. Рассмотренный список, таким образом, представляет собой обращение к функции.

При выполнении большинства функций вначале вычисляются фактические параметры, а затем выполняются действия, предписанные данной функ-

ции. Такие функции будем называть *обычными*. Однако есть функции, у которых вычисляются лишь некоторые фактические параметры либо параметры вовсе не вычисляются. Такие функции назовем *особыми*.

Аргументом обычной функции будем называть значение фактического параметра, а аргументом особой функции – сам фактический параметр.

Далее мы рассмотрим базовые функции языка Лисп.

1.3 Базовые функции языка

***Функция* QUOTE**. Это особая функция, имеющая один фактический параметр, который и выдается в качестве значения, не будучи вычисленным. Эта функция нужна для того, чтобы явно задавать необходимые значения. Таким образом:

```
(QUOTE A) ==> A
(QUOTE (A B C)) ==> (A B C)
```

Очевидно, что значение `(QUOTE 7)` совпадает со значением выражения `7`, а значение `(QUOTE NIL)` со значением `( )`.

Наряду с функцией `QUOTE` можно использовать маркер `'` (апостроф), помещаемый непосредственно перед выражением, которое не следует вычислять. Например:

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

***Функция* EVAL** обеспечивает дополнительный вызов интерпретатора Лиспа. При этом вызов может производиться внутри вычисляемого S-выражения.

Функция `EVAL` позволяет снять блокировку `QUOTE`.

```
(QUOTE ( + 1 2 ) ) ==> ( + 1 2 )
(EVAL (QUOTE ( + 1 2 ) ) ) ==> 3
```

Функции `QUOTE` и `EVAL` действуют во взаимно противоположных направлениях и аннулируют эффект друг друга.

`EVAL` – это универсальная функция Лиспа, которая может вычислить любое правильно составленное S-выражение.

Рассмотрим теперь простейшие функции над списками. Будем при описании функции после ее имени указывать общий вид обращения к ней.

***Функция* CAR**

`(CAR x)`. Это обычная функция, ее аргумент – непустой список. Значением функции является первый элемент этого списка (голова списка). Напри-

мер:

```
(CAR '(A B C)) ==> A
(CAR '(1 . 2)) ==> 1
(CAR '((A B) C)) ==> (A B)
```

***Функция* CDR**

(CDR x). Это обычная функция, аргумент x – непустой список. Значением функции является выражение, на которое указывает правая ссылка первого звена, т. е. хвост списка. Например:

```
(CDR '(A B C)) ==> (B C)
(CDR '(A)) ==> NIL
(CDR '(1 . 2)) ==> 2
(CAR (CDR '(A B C))) ==> B
```

Первым выполняется CDR, а затем CAR, т. е. в Лиспе первым выполняются внутренние функции, а затем внешние. Исполнение идет «изнутри наружу».

Отметим, что CAR и CDR – очень быстрые функции.

Фактически действие CAR сводится к выдаче левой ссылки первого звена списка, а CDR – правой ссылки того же звена.

***Функция* CONS**

(CONS x y). Это обычная функция, которая строит новый список из своих аргументов. Аргументы x и y – произвольные выражения. Функция создает звено, левая ссылка которого указывает на x, а правая – на y: первый аргумент становится головой второго аргумента. Например:

```
(CONS 'A '(1 2)) ==> (A 1 2)
(CAR (CONS 1  '(B C))) ==> 1
(CDR (CONS 1  '(B C))) ==> (B C)
(CONS '(A B) '(C D)) ==> ((A B) C D)
(CONS 'A 'B) ==> (A . B)
(CONS 'A NIL) ==> (A) /* позволяет превращать элемент в список */
```

С точки зрения внутреннего представления списков функция CONS создает новую списочную ячейку CAR, поле которой указывает на первый элемент, а CDR – на второй. Например, конструкция (CONS `X `(A B C)) создает список, внутреннее представление которого показано на рисунке 1.4.

Селекторы CAR *и* CDR *являются обратными для конструктора* CONS. *Список, разбитый с помощью функции* CAR *и* CDR *на голову и хвост, можно восстановить функцией* CONS*:*

```
(CONS (CAR '(A B)) (CDR '(A B))) ==> (A B)
```

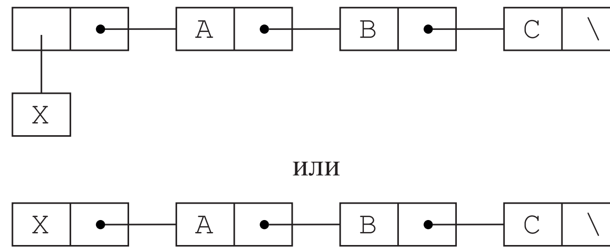


Рис. 1.4 – Работа CONS с точки зрения внутреннего представления

Предикаты

Рассмотрим теперь основные предикаты – функции, проверяющие наличие некоторых свойств аргумента. Отметим, что в Лиспе в качестве логического значения «ложь» используется атом NIL, а в качестве значения «истина» – любое другое значение. Так, следующие значения считаются истинными:

```
5
(A B)
T
```

Для определенности в качестве истинного значения предикатов используется атом T (значением T является тоже T).

Предикаты являются обычными функциями. Рассмотрим сначала два базовых предиката (справа от имени предиката указываем условие, при котором значением предиката будет T):

ATOM – аргумент является атомом

NULL – аргумент есть пустой список

Примеры:

```
(ATOM (CAR '(A B C))) ==> T
(ATOM ()) ==> T
(ATOM (CDR '(A B C))) ==> NIL
(NULL (CDR '(A))) ==> T
```

В таблице 1.1 приведены основные предикаты Лиспа.

Таблица 1.1 – Основные предикаты Лиспа

Предикат	**Действие**	**Истина (T)**	**Ложь (NIL)**
SIMBOLP	аргумент символ?	(symbolp 'a)	(symbolp '10)
LISTP	аргумент список?	(listp '(a))	(listp 'a)
NUMBERP	аргумент число?	(numberp 10)	(numberp 'a)
ZEROP	$arg = 0$	(zerop 0)	(zerop 1) (zerop `(a b c))
PLUSP	$arg > 0$	(plusp 1)	(plusp -1)
MINUSP	$arg < 0$	(minusp -1)	(minusp 1)

Следующие функции являются базовыми предикатами сравнения S-выражений в Лиспе.

***Функция* EQ**

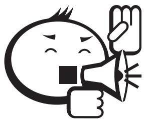
`(EQ e1, e2)`. Это обычная функция. Если известно, что один из аргументов – идентификатор, то значение функции будет истинным при равенстве ее аргументов и ложным в противном случае. Если аргументы не идентификаторы, то значение функции может быть и истинным, и ложным.

Примеры:

```
(EQ 'A 'B) ==> NIL
(EQ 'A  '(A B)) ==> NIL
(EQ 'M (CAR '(M N P))) ==> T
(EQ `(Q W)  `(Q W)) ==> NIL /* !!! списки, а не содержимое, разные!!! */
```

***Функция* EQUAL**

`(EQUAL e1 e2)`. Это – обычная функция. Она принимает истинное значение, если у нее одинаковые аргументы. Функция `EQUAL` позволяет сравнивать на равенство любые значения (идентификаторы, числа, списки). Функция `EQUAL` выполняется медленнее функции `EQ`, поэтому использовать функцию `EQUAL` следует в случае особой необходимости. Тем не менее, для сравнения списков следует использовать `EQUAL`.



Примечание. В большинстве Лисп-систем результатом сравнения вещественного и целого числа будет всегда значение «ложь», даже если эти значения арифметически равны друг другу (например, 5.0 и 5).

Примеры:

```
(EQUAL 5 (CAR '(5 A))) ==> T
(EQUAL 0 -0) ==> T
(EQUAL '(A B) (CDR '(1 A B))) ==> T /* EQ даст NIL !!! */
(EQUAL (CONS 'A 'B) '(A . B)) ==> T /* EQ даст NIL !!! */
```

Операции отношения

Перейдем теперь к арифметическим операциям отношения. Имена функций, определяющих операции отношения:

= (равно);

/= (не равно);

< (меньше);

> (больше);

<= (не больше);

>= (не меньше).

Примеры:

```
(=  3.0 3.0) ==> T
(<  3 5.5) ==> T
(>= 7.2 7) ==> T
```

Арифметические функции

Арифметические функции могут быть использованы с целыми или действительными аргументами. Число аргументов для большинства арифметических функций может быть разным.

Существуют следующие арифметические функции:

+ (сложение);

- (вычитание);

* (умножение);

/ (деление);

`TRUNCATE` (деление нацело).

Примеры:

```
(+ 5 6 7.0) ==> 18.0
(+ 5 6 7) ==> 18
(/ 5 2) ==> 2.5
(TRUNCATE 5 2) ==> 2
(- 7 3.0) ==> 4.0
(* 2 3 4) ==> 24
```

Логические функции

Для объединения предикатов в сложные выражения и для выделения элементов `NIL` в Лиспе используются логические функции `AND`, `OR` и `NOT`.

***Функция* NOT**

`(NOT e)`. Это обычная функция. Она определяет операцию логического отрицания: возвращает значение, противоположное значению аргумента. Если аргумент `NIL`, `NOT` возвращает `T`. Если аргумент любое значение не `NIL`, `NOT` возвращает `NIL`. По сути, значение функции всегда совпадает со значением функции `NULL`. Обычно для наглядности в качестве предиката применяется функция `NULL`, а в качестве логического отрицания – `NOT`. Однако это не ме-

шает отдавать предпочтение только одной из этих функций.

Примеры:

```
(NOT NIL) ==> T
(NOT '(A B C)) ==> NIL
```

***Функция* AND**

(AND e1 e2 ... en). Это особая функция. Эта функция последовательно вычисляет аргументы (слева направо) до тех пор, пока не появится значение, равное NIL; в этом случае функция прекращает вычисление аргументов и заканчивает свое выполнение со значением NIL. Если же значения всех аргументов отличны от NIL, то значением функции будет значение последнего аргумента. Таким образом, функция AND может использоваться как следующее условное выражение: «если e1, и e2, ..., и en-1, то выполнить en».

Примеры:

```
(AND 'A () (5 B C)) ==> NIL
(AND (+ 5 6) 3 7) ==> 7
(and 2 3 (car `(q w e))) ==> q
```

***Функция* OR**

(OR e1 e2 ... en). Это особая функция. Эта функция последовательно вычисляет аргументы до тех пор, пока не появится значение, отличное от NIL; в этом случае функция прекращает вычисление аргументов и выдает последнее вычисленное значение. Если же значения всех аргументов равны NIL, то и значением функции будет NIL. Таким образом, функция может использоваться для задания такого условного выражения: «если не e1, не e2, ..., не en-1, то выполнить en».

Примеры:

```
(OR 'A 'B 'C) ==> A
(OR 'NIL 'B 'C) ==> B
(OR (ATOM '(A)) (NUMBERP 'A)) ==> NIL
```

Разветвление вычислений

Рассмотрим теперь функцию COND, которая служит для задания условных выражений произвольного вида и является основным средством организации разветвления вычислений.

***Функция* COND**

```
(COND (p1 e11...e1n1)
      (p2 e21...e2n2)
      (pk ek1...e1nk))
```

Эта функция является особой. Как видно, и сами аргументы имеют особый вид. Функция последовательно вычисляет значения `pi (i=1,k)` до тех пор, пока не встретится `pj`, значение которого отлично от `NIL`; в этом случае функция вычисляет последовательно все `ej1,...,ejnj`, а ее значением становится значение `ejnj`. Если значения всех `p1`, `p2,...`, `pk` равны `NIL`, то функция `COND` заканчивает свое выполнение со значением `NIL`.

Обычно в качестве последнего условия пишется `T`, соответствующее ему выражение будет вычисляться в тех случаях, когда ни одно другое условие не выполняется.

Таким образом, выражение `(COND (a b) (T c))` представляет собой условное выражение вида: «если `a`, то `b`, иначе `c`».

Примеры:

```
(COND ((ATOM '(A B C)) A) (T 'B)) ==> B
(COND (T 'A) (Q P R S T)) ==> A
(COND ((< 5 3) 5) (T 3)) ==> 3
```

Суперпозиции CAR и CDR

Существуют обычные функции от одного аргумента, действия которых эквивалентны выполнению некоторой суперпозиции `CAR` и `CDR`. Например, выражение `(CADAR x)` имеет то же значение, что и `(CAR (CDR (CAR x)))`. Последовательность `A` и `D` в имени функции отвечает порядку следования `CAR` и `CDR` в эквивалентной суперпозиции (справа налево).

Так, функция `CDDR` выдает список без двух первых элементов, а `CDDDR` – без трех первых элементов. Функция `CADR` выбирает из списка второй элемент, а `CADDR` – третий.

Примеры:

```
(CADR '(A B C D)) ==> B
(CDADR '((1 Q 3) (M N P) (ONE TWO))) ==> (N P)
(CDDAR '((1 2 3) A)) ==> (3)
```

Рассмотрим теперь некоторые производные функции над списками, действия которых реализуются через простейшие функции `CAR`, `CDR` и `CONS`.

***Функция* `LIST`**

`(LIST e1 e2...en)`. Это обычная функция. Значением ее является список из аргументов. Таким образом, значение функции совпадает со значением выражения `(CONS e1 (CONS e2 ... (CONS en NIL)))`.

Примеры использования:

```
(LIST `A `C `N) ==> (A C N)
(LIST (+ 5 6) (CAR '(A B))) ==> (11 A)
(LIST 'A) ==> (A)
(LIST NIL) ==> (NIL)
(EQUAL (LIST NIL) '(())) ==> T
```

Рассмотрим работу функции на уровне внутреннего представления списков (рис. 1.5). Создадим список с помощью функции LIST:

```
(LIST 'a '(b c)) ==> (a (b c))
```

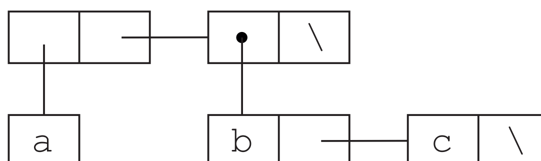


Рис. 1.5 – Внутреннее представление списка (a (b c))

Список получается следующим образом:

1. Создается списочная ячейка для каждого аргумента функции.
2. В CAR поле ставится указатель на соответствующий элемент.
3. В CDR поле ставится указатель на следующую списочную ячейку.

Разница работы функций LIST *и* CONS:

```
(list `(1 2) `(3 4)) ==> ((1 2) (3 4))
(cons `(1 2) `(3 4)) ==> ((1  2)  3  4)
```

Использование символов в качестве переменных

Изначально символы в Лиспе не имеют значения. Значения имеют только константы:

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

Если попытаться вычислить символ, то система выдает ошибку.

Значения символов хранятся в ячейках, закрепленных за каждым символом. Если в эту ячейку положить значение, то символ будет связан (bind) со значением. В процедурных языках говорят «будет присвоено значение».

Для Лиспа есть отличие:

Не оговаривается, что может храниться в ячейке: целое, атом, список, массив и т. д. В ячейке может храниться что угодно.

С символом может быть связана не только ячейка со значением, но и многие другие ячейки, число которых не ограничено.

Для связывания символов используются следующие функции:

SET

SETQ

***Функция* SET**

Функция SET связывает символ со значением, предварительно вычисляя значения аргументов.

В качестве значения функция SET возвращает значение второго аргумента. Если перед первым аргументом нет апострофа, то значение будет присвоено значению этого аргумента.

```
(SET  'd   '( x y z ) ) ==> ( x y z )
(SET  'a  'b ) ==> b
(SET   a   'e ) ==> e
a ==> b
b ==> e
```

На значение символа можно сослаться, записав его без апострофа.

***Функция* SETQ**

Она аналогична SET, но не вычисляет значение первого аргумента. Об автоматическом блокировании первого аргумента напоминает буква Q.

```
(SETQ  m   d ) ==> (x y z)
m ==> (x y z)
```

Эти функции имеют побочный эффект: образуется связь между символом и его значением, а значением функции является связываемое значение.

Примечание. В Лиспе все действия возвращают некоторое значение. Значение имеется даже у таких действий, основное предназначение которых заключается в осуществлении побочного эффекта, например, вывод результатов на печать. Функции, обладающие побочным эффектом, в Лиспе называются псевдофункциями. В чистом функциональном программировании псевдофункции не нужны и не используются.

Обратная блокировка

Обычная блокировка не позволяет вычислять выражения. Вспомним работу функции QUOTE:

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

Таким образом, функция QUOTE рассматривает любой атом или список как невычислимое выражение. Если же мы не используем функцию QUOTE, то атом рассматривается как переменная, которая должна иметь значение, а список – как вычислимое выражение. В случае вычислимого выражения первым элементов списка должна выступать функция.

Тем не менее, иногда возникает необходимость вычислить только часть выражения. Пусть атом b будет связан со списком:

```
(setq b '(x y z))
```

Если мы запишем в любом невычислимом списке перед b запятую, то b будет вычислено:

```
`( a ,b c) ==> (a (x y z) c)
```

Любая блокировка QOUTE *может частично сниматься запятой.*

Обратная блокировка может использоваться и при печати:

```
(setq n 'john)
(setq m 'Robert)
(print `(boys ,n and ,m)) ==> (boys john and roberts)
```

1.4 Лямбда-выражения и определение новых функций

λ-выражение

В языке Лисп есть возможность в качестве обозначения функции использовать наряду с идентификатором функции специальное выражение, называемое лямбда-выражением (λ-выражением). Оно используется, в частности, для явного задания определения функции в обращении к функции. Лямбда-выражение имеет такой вид:

```
(LAMBDA a e1 e2 ... en),
```

где a – это список идентификаторов формальных параметров (он может быть и пустым), а e1, e2, ..., en – вычислимые выражения.

Выполнение обращения к λ-выражению вида ((LAMBDA (a1 a2 ... ak) e1 e2 ... en) p1 p2 ... pk), заключается в следующих трех этапах:

1) последовательно вычисляются фактические параметры p1, p2, ..., pk;
2) значение каждого pi (i=1,k) присваивается соответствующему формальному параметру ai, который внутри λ-выражения служит локальной переменной;
3) последовательно вычисляются e1, e2,..., en, а значением обращения к λ-выражению становится значение en.

Вычисление обращения к λ-выражению вида (LAMBDA NIL e_1 e_2 ... e_n) сводится к последовательному вычислению выражений e1, e2, ..., en и к выдаче значения en.

Рассмотрим примеры использования λ-выражений:

```
((LAMBDA () (+ 5 6)) ==> 11
((LAMBDA (X Y) (+ (* X X) (* Y Y))) 3 4) ==> 25
((LAMBDA (X Y)
     (COND ((> X Y) X) (T Y)))
          (* 3 7 8 9) (* 10 11 14)) ==> 1540
```

В последнем примере вычисляется максимальное из значений фактических параметров.

Лямбда-выражение позволяет сократить число вычислений и вводить локальные переменные для дальнейшего использования. Так, без λ-выражения конструкцию из последнего примера можно было бы представить следующим образом:

```
(COND ((> (* 3 7 8 9) (* 10 11 14)) (* 3 7 8 9))
     (T (* 10 11 14)))
```

В этом случае, как видно, необходимо произвести больше вычислений.

Определение функций

Однако, если нам часто приходится вычислять максимум, то неудобно каждый раз записывать одно и то же λ-выражение. Хотелось бы поставить ему в соответствие имя и указывать это имя всякий раз, когда нужно вычислить максимум. А имя вместе с поставленным ему λ-выражением есть не что иное, как функция. Тем самым мы приходим к определению обычной функции.

Чтобы определить функцию, необходимо:

- дать имя функции – атом;
- определить параметры функции – список;
- определить, что должна делать функция – набор выражений.

В Лиспе описание функции задается следующим образом:

```
(DEFUN  f  a  e1 e2 ... en),
```

где `f` – имя определяемой функции,

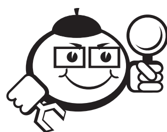
`a` – список идентификаторов формальных параметров,

`e1, e2, ..., en` – выражения, вычисляемые при выполнении функции и составляющие тело функции.

Указанная конструкция ставит в соответствие имени `f` λ-выражение вида `(LAMBDA a e1 e2 ... en)`.

Вызов функции: после того, как функция определена, при выполнении любой конструкции вида `(f p1 p2 ... pn)` (в том числе и входящей в одно

из e1, e2, ..., en) будет вычисляться выражение ((LAMBDA a e_1 e_2 ... e_n) p_1 p_2 ... p_k), значение которого и станет значением f.



Пример 1.1

Необходимо создать функцию, которая будет вставлять новый элемент на второе место в списке.

Назовем функцию insert-second.

Определим два аргумента: x – вставляемый элемент, y – старый список.

Описание функции:

```
(defun insert-second (x y)
     (cons (car y ) ( cons x (cdr y) ) )
```

Вызов функции:

```
(insert-second 'b  '( a c d ) ) ==> (a b c d)
```

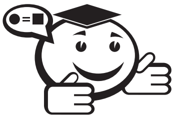
Контрольные вопросы по главе 1

1. Что является S-выражением?
2. Чем отличается внутреннее представление списка и атома?
3. Что такое точечная пара?
4. Чем отличаются функции CAR и CDR?
5. Для чего используется лямбда-выражение?
6. Как определяется функция в Лиспе?

2 Рекурсивные функции

2.1 Понятие рекурсии

При написании функций мы часто сталкиваемся с необходимостью описания повторяющихся действий. В алгоритмических языках для этого используются циклы. В функциональной парадигме программирования, строго говоря, нет такого понятия, как цикл. В функциональных языках цикл обычно реализуется в виде рекурсии. ЛИСП основан на рекурсивном подходе.



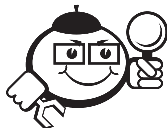
Объект *называется* ***рекурсивным****, если он содержит сам себя или определен с помощью самого себя.*

Функция *называется* ***рекурсивной****, если во время выполнения ее тела может встретиться обращение к этой функции.*

В частности, рекурсивной является функция, в теле которой находится обращение к ней самой. Такая рекурсия называется ***прямой****.*

Если же в теле некоторой функции `f` *есть обращение к функции* `f1`*, а в теле* `f1` *– обращение к* `f2`*,* `...`*, а в теле* `fn` *– обращение к* `f`*, то такая рекурсия называется* ***косвенной****.*

Рассмотрим случаи прямой рекурсии.



Пример 2.1

Функция **MEMBER**

`(MEMBER x y)`. Это обычная функция. Значением функции является подсписок списка `y`, начинающийся с элемента, совпадающего с атомом `x`. Если элемента, совпадающего с `x`, в списке `y` нет, то значением функции будет `NIL`. Поскольку в Лиспе любое значение, отличное от `NIL`, является истинным, функция `MEMBER` может использоваться как предикат, проверяющий, принадлежит ли `x` списку `y`. Ее можно определить так:

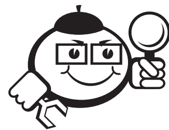
```
(DEFUN MEMBER (X Y)
    (COND ((NULL Y) NIL)
          ((EQL X (CAR Y)) Y)
          (T (MEMBER X (CDR Y)))))
```

Пример вызова функции:

```
(MEMBER 'Q '(A S Q E D)) ==> (Q E D)
```

Рассмотрим поэтапное вычисление следующего обращения к функции: `(MEMBER 'Q '(A S Q E D))`. В теле функции будет вычисляться следующее выражение:

1. Так как `Y` не является пустым списком и `Q` не равно `A`, то `(MEMBER `Q `(S Q E D))`.
2. Так как новый `Y` не является пустым списком и `Q` не равно `S`, то `(MEMBER `Q `(Q E D))`.
3. Так как новый `Y` не является пустым списком, но `Q` равно `Q`, то функция завершает работу и возвращает `Y3`: `(Q E D)`.



Пример 2.2

Предположим, что необходимо написать функцию `sumall`, которая имеет один аргумент `N` – целое положительное число и возвращает сумму всех целых чисел между нулем и этим числом.

Например `(sumall 9)` должно вернуть 45.

Отметим два факта:

1. Если `N=0`, сумма чисел между нулем и `N` равна `0`.
2. Если `N>0`, сумма чисел между нулем и `N` равна `N` плюс сумма чисел между нулем и `N-1`.

Эти два факта переводятся непосредственно в определение функции:

```
(defun sumall (n)
     (cond ((zerop n) 0)
          (t (+ n (sumall (- n 1))))))
```

Производится проверка `N`: равно нулю или нет. Если значение `N=0`, то функция возвращает `0`. В противном случае функция вызывает сама себя для вычисления суммы чисел между `0` и `N-1` и добавляет `N` к этой сумме.

2.2 Как работает рекурсивная функция

Посмотрим, как работает рекурсивная функция. Проследим за несколькими вызовами функции, описанной в примере 2.2.

Ясно, как работает `(sumall 0)`. Функция возвращает `0`. Эта ветвь в

`cond` называется терминальной (*terminating*), или завершающей, так как функция дает значение без рекурсивного вызова.

Если `(sumall 1)`, то расчет идет по второй ветке, которая называется рекурсивной, так как идет вызов самой себя. В этом случае выполняется выражение `(+ 1 (sumall 0))`, и значение функции равно `1`.

Если `(sumall 2)`, то по рекурсивной ветке выполняется `(+ 2 (sumall 1))` и функция возвращает `3`.

Если значение аргумента равно `3`, то

`(sumall 3)` вызывает `(sumall 2)`

`(sumall 1)` вызывает `(sumall 0)`.

После того как `(sumall 0)` вернет `0`,

`(sumall 1)` вернет `1`,

`(sumall 2)` вернет `3`,

`(sumall 3)` (это вызов верхнего уровня) даст значение `6`.

2.3 Правила записи рекурсивной функции

Рассмотренный выше простой случай иллюстрирует несколько правил в записи рекурсивной функции.

Правило 1. В рекурсивном определении *существенен порядок следования условий*. При определении порядка следования основным моментом является то, что сначала проверяются всевозможные условия окончания, а затем ситуации, требующие продолжения вычислений.

Правило 2. Терминальная ветвь необходима для окончания вызова. Без терминальной ветви рекурсивный вызов был бы бесконечным. Терминальная ветвь возвращает результат, который является базой для вычисления результатов рекурсивных вызовов.

Правило 3. После каждого вызова функцией самой себя мы должны приближаться к терминальной ветви. В нашем случае вызовы уменьшали `N`, и была гарантия, что на некотором шаге будет вызов `(sumall 0)`. Всегда должна быть уверенность, что рекурсивные вызовы ведут к терминальной ветви.

Проследить вычисления в рекурсии чрезвычайно сложно. Очень трудно мысленно проследить за действием рекурсивных функций. Это практически невозможно для функций более сложных, чем `sumall`.

Таким образом мы должны уметь писать рекурсивные функции, без того чтобы представлять точно порядок вычисления.

При написании рекурсивной функции необходимо:

1. *Планирование терминальной ветви.* При написании рекурсивной функции мы должны решить, когда функция может вернуть значение без рекурсивного вызова.

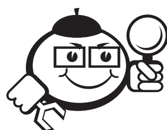
2. *Планирование рекурсивной ветви.* В этом случае мы вызываем функцию рекурсивно с упрощенным аргументом и используем результат для расчета значения при текущем аргументе.

Таким образом, мы должны решить:

1. Как упрощать аргумент, приближая его шаг за шагом к конечному значению.

2. Кроме этого необходимо построить форму, называемую рекурсивным отношением, которая связывает правильное значение текущего вызова со значением рекурсивного вызова. В рассмотренном примере это `(sumall N)` и `(sumall (- N 1))`. Иногда просто найти это отношение, а если не получается, то надо выполнить следующую последовательность шагов:

- определить значение некоторого простого вызова функции и ее соответствующего рекурсивного вызова;
- определить соотношение между парой этих функций.

 Пример 2.3

Определим функцию возведения в степень `power`. Она берет два численных аргумента `M` и `N` и вычисляет значение `M` в степени `N`.

Вначале составим рекурсивную таблицу.

Шаг 1. Завершение (терминальная ветвь) `N=0` – аргумент, `(power 2 0)=1` – значение функции.

Шаг 2. Рекурсивная ветвь – рекурсивные отношения между `(power M N)` и `(power M (- N 1))`.

Примеры рекурсии:

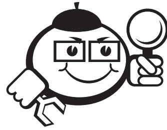
```
(power 5 3) ==> 125
(power 5 2) ==> 25
(power 3 1) ==> 3
(power 3 0 ) ==> 1
```

Характеристическое рекурсивное отношение `(power M N)` может быть получено из отношения `(power M (- N 1)` путем его умножения на `M`.

Полученная функция:

```
(defun power (m n)
     (cond ((zerop n) 1)
           (t (* m (power  m (- n 1))))))
```

Логика и структура CDR рекурсии сходна с численной рекурсией.



Пример 2.4

Написать функцию `list-sum`, которая берет один аргумент `Y` – список чисел, и возвращает сумму этих чисел.

Последовательно упрощающимся аргументом в этом случае будет список. Упрощение списка `(cdr Y)`. Последнее значение аргумента `nil`.

Составим рекурсивную таблицу для `(list-sum Y)`.

Шаг 1. Завершение (терминальная ветвь) `(list-sum nil)=0` – значение.

Шаг 2. Рекурсивная ветвь: рекурсивные отношения между выражениями `(list-sum Y)` и `(list-sum (cdr Y))`.

Примеры рекурсии:

```
(list-sum '(2 5 3)) ==> 10
(list-sum '(5 3)) ==> 8
(list-sum '(3)) ==> 3
(list-sum nil ) ==> 0
```

Характеристическое рекурсивное отношение `(list-sum Y)` может быть получена из `(list-sum (cdr Y))` сложением с `(car Y)`.

Текст функции:

```
(defun list-sum (Y)
     (cond ((null Y) 0)
          (t (+ (car Y) (list-sum (cdr Y))))))
```

2.4 Рекурсия с несколькими терминальными или рекурсивными ветвями

Несколько терминальных ветвей

Мы рассмотрели случай рекурсии с одной терминальной и одной рекурсивной ветвью. Однако в определении рекурсивной функции может быть не-

сколько терминальных ветвей. Две терминальные ветви будут в том случае, когда ведется поиск цели в последовательности значений, и мы желаем получить результат, как только цель найдена.

Ветвь 1. Цель найдена, и надо вернуть ответ.

Ветвь 2. Цель не найдена, и элементов больше нет.

На рисунке 2.1 показан принцип построения функции с несколькими терминальными ветвями.

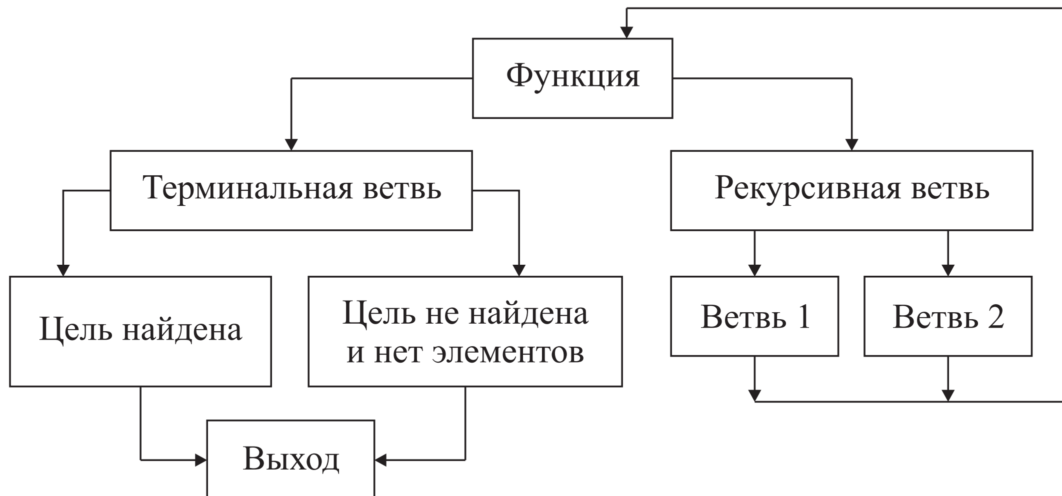
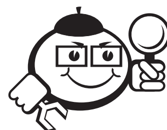


Рис. 2.1 – Функция с несколькими терминальными ветвями

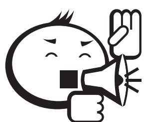


Пример 2.5

Необходимо написать функцию `greaternum`. Она имеет два аргумента: список чисел `L` и заданное число `X`. Функция возвращает первое число в списке, превышающее заданное. Если этого числа нет, возвращается заданное число.

Программа

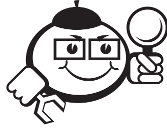
```
(defun greaternum (lis num)
     (cond ((null lis) num)
           ((> (car lis) num) (car lis))
           (t (greaternum (cdr lis) num))))
```



Порядок ветвей в рекурсивном определении существенен.

Несколько рекурсивных ветвей

Несколько рекурсивных ветвей могут понадобиться, если функция обрабатывает все элементы в структуре, но использует некоторые элементы отлично от других. В этом случае составляются два рекурсивных отношения.



Пример 2.6

Необходимо написать функцию `negnums`, которая получает список чисел `L` и возвращает список, который содержит только отрицательные числа (0 положителен).

Шаг 1. Завершение (терминальная ветвь) `(negnums nil)=nil`.

Шаг 2. Рекурсивная ветвь: рекурсивные отношения между `(negnums l)` и `(negnums (cdr l))`.

Примеры рекурсии:

```
(negnums '(-5 3)) = (-5)
(negnums '(3)) = nil
(negnums '(-5 3 -6 0 )) = (-5 -6)
(negnums '(3 -6 0 )) = (-6)
```

Ситуация `(car l ) < 0`

Характеристическое рекурсивное отношение `(negnums L)` может быть получено из `(negnums (cdr L))` следующим образом: `(cons (car L) (negnums (cdr L)).`

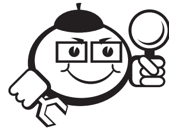
Ситуация `(car l ) >= 0`

Характеристическое рекурсивное отношение `(negnums L)` может быть получено из `(negnums (cdr L))` через `(negnums L)=(negnums (cdr L)).`

Программа

```
(defun negnums (L)
     (cond ((null L) nil)
           ((< (car L) 0) (cons (car L) (negnums (cdr L))))
           (t (negnums (cdr L)))))
```

Минимальное значение k *для заданного аргумента, при котором выполняется условие завершения, будем называть* ***глубиной рекурсии****.*



Пример 2.7

Рассмотрим функцию SUM, которая суммирует все числа, содержащиеся в некотором произвольном выражении (будем рассматривать выражения, состоящие из любых атомов и списков). Определение функции можно записать так:

```
(DEFUN SUM (X)
     (COND ((NUMBERP X) X)
           ((ATOM X) 0)
           (T (+ (SUM (CAR X)) (SUM (CDR X))))))
```

Рассмотрим поэтапное вычисление следующего обращения к функции: (SUM '(A (3 5) . 7)). В теле функции будет вычисляться следующее выражение:

1. `(+ (SUM (CAR '(A (3 5) . 7))) (SUM (CDR '(A (3 5) . 7)))`

В результате вычисления CAR получаем следующее:

2. `(+ (SUM A) (SUM (CDR '(A (3 5) . 7)))`

Значением первого слагаемого становится 0. После этого вычисляется CDR, в результате чего получаем следующее:

3. `(+ 0 (SUM '((3 5) . 7)))`

Теперь рассматриваем список ((3 5) . 7):

4.
```
(+ 0
   (+ (SUM '(3 5)) (SUM (CDR '((3 5) . 7)))))
```

Далее порядок вычислений будет следующим:

5.
```
(+ 0
   (+ (+ (SUM '3) (SUM (CDR '(3 5))))
      (SUM (CDR '((3 5) . 7)))))
```

6.
```
(+ 0
   (+ (+ 3 (SUM '(5)))
      (SUM (CDR '((3 5) . 7)))))
```

7.
```
(+ 0
   (+ (+ 3 (+ 5 (SUM 'NIL)))
      (SUM (CDR '((3 5) . 7)))))
```

8.
```
(+ 0
   (+ (+3 (+ 5 0))
      (SUM (CDR '((3 5) . 7)))))
```

9.
```
(+ 0
   (+ 8
      (SUM '7)))
```

10.
```
(+ 0
    (+ 8 7))
```

После суммирования получается результат – 15. Для рассмотренного примера глубина рекурсии равна 4.

. . . .

2.5 Вспомогательные функции над списками

В этом разделе мы рассмотрим ряд часто встречающихся вспомогательных функций. Большинство из этих функций обычно являются встроенными. Если же в какой-то реализации Лиспа отсутствует необходимая функция, то можно воспользоваться приводимым ниже определением.

***Функция* APPEND**

(APPEND x y). Это обычная функция. Значением функции является список, получающийся в результате конкатенации (соединения) списков x и y. Определить функцию можно так:

```
(DEFUN APPEND (X Y)
     (COND ((ATOM X) Y)
          (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

Примеры обращения к функции:

```
(APPEND '(A B) '(C (1) D)) ==> (A B C (1) D)
(APPEND '(1 2 3) '(7 8)) ==> (1 2 3 7 8)
```

Ниже приведен пример, иллюстрирующий разницу работы трех функций – LIST, CONS *и* APPEND.

```
(LIST `(1 2) `(3 4)) ==> ((1 2) (3 4))
(CONS `(1 2) `(3 4)) ==> ((1 2) 3 4)
(APPEND `(1 2) `(3 4)) ==> (1 2 3 4)
```

Таким образом, отличие объединяющих функций LIST, CONS *и* APPEND *заключается в следующем:*

- CONS всегда берет два аргумента и помещает первый в начало второго;
- LIST берет один или больше аргументов и образует список, помещая аргументы в скобки;
- APPEND образует новый список, убирая скобки вокруг аргументов и помещая их в один список.

На самом деле, *на внутреннем уровне представления списков* тоже существуют различия в работе трех указанных функций. Работа функций CONS и

LIST на внутреннем уровне была рассмотрена ранее. Покажем теперь работу функции APPEND.

Рассмотрим следующую последовательность действий:

```
(setq first '(a b)) ==> (a b)
(setq second '(c d)) ==> (c d)
(setq both (append first second)) ==> (a b c d)
```

Эти действия показывает диаграмма, приведенная на рисунке 2.2.

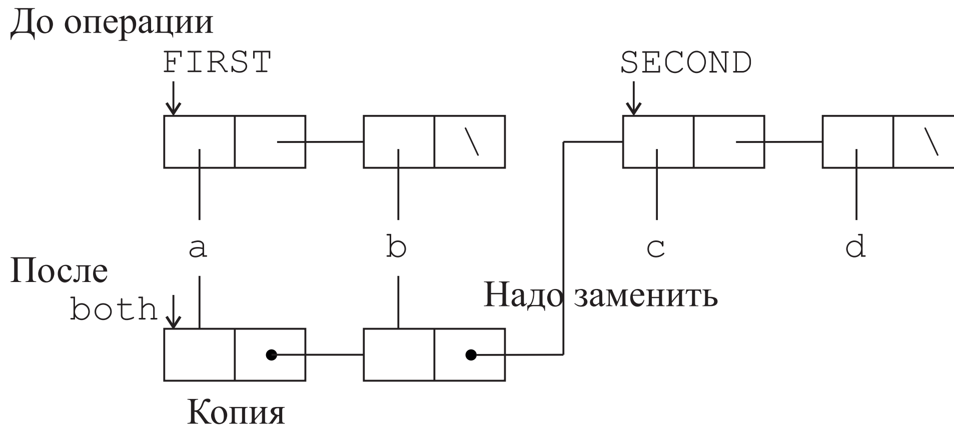


Рис. 2.2 – Работа функции APPEND на внутреннем уровне

APPEND *создает копии всех списочных ячеек для каждого элемента во всех аргументах, исключая последний аргумент.*

CONS *создает только одну списочную ячейку.*

LIST *создает списочную ячейку для каждого элемента функции.*

Если складывают два списка в 1 000 и 1 элемент с помощью APPEND, то будет создано 1 000 копий списочных ячеек, вместо того чтобы исправить один указатель.

LIST в этом случае создаст только две списочных ячейки (два списка – два элемента).

***Функция* REVAPPEND**

(REVAPPEND x y). Это обычная функция. Она выполняет конкатенацию перевернутого списка x со списком y.

Ее определение может быть таким:

```
(DEFUN  REVAPPEND (X Y)
     (COND ((ATOM X) Y)
           (T (REVAPPEND (CDR X) (CONS (CAR X) Y)))))
```

Примеры использования:

```
(REVAPPEND '(A (B C) D) '(1 2)) ==> (D (B C) A 1 2)
(REVAPPEND '(A B C) '(2)) ==> (C B A 2)
```

```
(REVAPPEND '(ONE TWO) ()) ==> (TWO ONE)
```

***Функция* REVERSE**

(REVERSE x). Это обычная функция. Значением функции является список, полученный из элементов списка x, расположенных в обратном порядке. Необходимо отметить, что данная функция реверсирует только верхний уровень списка.

Определение функции:

```
(DEFUN REVERSE (X)
     (COND ((NULL X) NIL)
           (T (APPEND (REVERSE (CDR X)) (LIST (CAR X))))))
```

С использованием функции REVAPPEND функцию REVERSE можно определить значительно короче:

```
(DEFUN REVERSE (X)
     (REVAPPEND X NIL))
```

Пример использования:

```
(REVERSE '(10 (A B) 20)) ==> (20 (A B) 10)
```

***Функция* NTH**

(NTH i x). Это обычная функция. Значением функции является i-й элемент списка x. Элементы нумеруются начиная с нуля. Ее можно определить следующим образом:

```
(DEFUN NTH (I L)
     (COND ((ATOM L) NIL)
           ((ZEROP I) (CAR L))
           (T (NTH (- I 1) (CDR L)))))
```

Примеры использования:

```
(NTH 1 '(1 2 3)) ==> 2
(NTH 3 '(A B C (D E))) ==> (D E)
(NTH 0 '(A B C)) ==> A
```

***Функция* NTHCDR**

(NTHCDR i x). Эта функция обычная. Значением ее является список x без i начальных элементов. Определение функции может быть таким:

```
(DEFUN NTHCDR (I L)
     (COND ((NULL L) NIL)
           ((ZEROP I) L)
           (T (NTHCDR (- I 1) (CDR L)))))
```

Примеры:

```
(NTHCDR 0 '(1 2 3)) ==> (1 2 3)
(NTHCDR 2 '(1 2 3)) ==> (3)
```

```
(NTHCDR 2 '(A (B C))) ==> NIL
```

***Функция* LENGTH**

`(LENGTH x)`. Это обычная функция. Ее значением является число элементов в списке `x`, т. е. длина этого списка. Функцию можно определить так:

```
(DEFUN LENGTH (X)
     (COND ((ATOM X) 0)
           (T (+ 1 (LENGTH (CDR X))))))
```

Примеры:

```
(LENGTH NIL) ==> 0
(LENGTH '((A B) C)) ==> 2
```

***Функция* BUTLAST**

`(BUTLAST x i)`. Это обычная функция. Она формирует список, содержащий элементы списка `x`, кроме `i` последних. Приведем определение этой функции.

```
(DEFUN MCAR (X N)
     (COND ((<= N 0) NIL)
           (T (CONS (CAR X) (MCAR (CDR X) (- N 1))))))
(DEFUN BUTLAST (L I)
     (MCAR L (- (LENGTH L) I)))
```

Примеры использования:

```
(BUTLAST '(A B C) 1) ==> (A B)
(BUTLAST '(JOHN MARY JANE) 2) ==> (JOHN)
```

***Функция* SUBSTITUTE**

`(SUBSTITUTE x y z)`. Эта функция обычная. Она формирует новый список, получающийся из списка `z` путем подстановки выражения `x` вместо всех элементов, совпадающих с атомом `y` на верхнем уровне. Определение можно дать такое:

```
(DEFUN SUBSTITUTE (X Y L)
     (COND ((ATOM L) L)
           ((EQL Y (CAR L)) (CONS X (SUBSTITUTE X Y (CDR L))))
           (T (CONS (CAR L) (SUBSTITUTE X Y (CDR L))))))
```

Пример использования:

```
(SUBSTITUTE 1 'ONE '(ONE TWO (ONE TWO) ONE)) ==>
                        ==> (1 TWO (ONE TWO) 1)
```

***Функция* SUBST**

`(SUBST x y z)`. Это обычная функция. Она формирует новый список, получающийся из списка `z` путем подстановки выражения `x` вместо всех выражений на всех уровнях списка `z`, совпадающих с атомом `y`. Если `z` – атом, сов-

падающий с y, то значением функции будет x. Если же z – атом, не совпадающий с y, то значением функции будет z. Можно дать следующее определение:

```
(DEFUN SUBST (X Y Z)
    (COND ((EQL Y Z) X)
          ((ATOM Z) Z)
          (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))))
```

Примеры использования:

```
(SUBST 1  'ONE '(ONE TWO (ONE TWO) ONE)) ==>
                          ==> (1 TWO (1 TWO) 1)
(SUBST 'B 'A '(A C (C A) . A)) ==> (B C (C B) . B)
```

***Функция* POSITION**

(POSITION x y). Обычная функция. Ее значением является номер первого элемента списка y, совпадающего с атомом x. Если такого элемента в списке нет, то значением функции будет NIL. Ее определение может быть таким:

```
(DEFUN POSITION (X L)
     (COND ((ATOM L) NIL)
           ((EQL X (CAR L)) 0)
           (T ((LAMBDA (N) (COND (N (+ N 1))))
                                (POSITION X (CDR L))) )))
```

Подобно MEMBER, эта функция может использоваться как предикат.

Примеры:

```
(POSITION 'A '(C (A B) A)) ==> 2
(POSITION  5 '(1 2 3 4 5)) ==> 4
```

***Функция* LAST**

(LAST x). Обычная функция. Ее значением является список из последнего элемента списка x. Если список x пустой, то значением функции будет NIL. Ее можно определить так:

```
(DEFUN LAST (L)
     (COND ((ATOM L) L)
           ((ATOM (CDR L)) L)
           (T (LAST (CDR L)))))
```

Примеры использования:

```
(LAST '(A1 B2 C3)) ==> (C3)
(LAST '(1 2 3 (4 5))) ==> ((4 5))
```

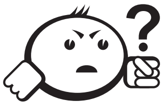
***Функция* REMOVE**

(REMOVE x y). Это – обычная функция. Значением ее является список, состоящий из элементов списка y, за исключением совпадающих с атомом x. Эту функцию можно определить так:

```
(DEFUN REMOVE (X L)
     (COND ((ATOM L) L)
           ((EQL X (CAR D) (REMOVE X (CDR L)))
           (T (CONS (CAR L) (REMOVE X (CDR L))))))
```

Пример:

```
(REMOVE 'A '(A C B A 6 A)) ==> (C B 6)
```



Контрольные вопросы по главе 2

1. Какая функция называется рекурсивной?
2. В чем отличие прямой и косвенной рекурсии?
3. Как работает рекурсивная функция?
4. Что такое терминальная ветвь?
5. Каков должен быть порядок терминальной и рекурсивной ветвей при написании рекурсивной функции?
6. В чем разница работы функций LIST, CONS, APPEND?

3 Технология программирования на языке Лисп

3.1 Передача параметров. Глобальные и локальные переменные

В Лиспе передача параметров производится в функцию по значению, т. е. формальный параметр в функции связывается с тем же значением, что и значение фактического параметра.

Изменение значения формального параметра не оказывает влияния на значения фактических параметров. После вычисления функции созданные на это время связи параметров ликвидируются и происходит возврат к тому состоянию, которое было до вызова функции. Параметры функции являются *локальными переменными* и имеют значение только внутри функции.

Если в теле функции есть переменные, не входящие в число ее формальных параметров, они называются *свободными*. Значения свободных переменных остается в силе после ее выполнения. По сути, свободная переменная является глобальной.

Рассмотрим теперь, как определяется доступ к *глобальным переменным*.

В языке Лисп глобальные переменные определяются в том месте, где находится обращение к функции, а не там, где стоит определение:

- связь между именем переменной и самой переменной осуществляется в момент доступа к переменной, т. е. при изменении или выборке значения переменной;
- в момент доступа используется переменная из ближайшей объемлющей выполняемой функции.

Рассмотрим подробнее работу функции `SETQ`, которая позволяет изменять значения переменных (описание функции было приведено в гл. 1).

Если к моменту выполнения функции `(SETQ x v)` не существовало переменной с именем `x`, то такая *переменная* заводится в самом широком, глобальном контексте и, таким образом, *становится глобальной* по отношению ко всем используемым функциям.

Пусть имеются следующие определения функций:

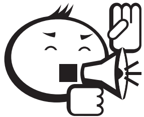
```
(DEFUN F1 (N) (SETQ X 5) (+ N X))
(DEFUN F2 (Q) (* Q X))
```

Вызов этих функций приводит к следующим результатам:

```
(F1 2) ==> 7
X      ==> 5
```

```
(F2 3) ==> 15 /* переменная X является свободной (глобаль-
ной)*/
(SETQ X 10) /* переопределили X*/
(F2 3) ==> 30
(F1 1) ==> 6 /* функция F1 снова переопределила значение X */
(F2 3) ==> 15
(F2 X) ==> 25
```

. . . .



Необходимо помнить, что использование функции SETQ не всегда определяет глобальное значение переменной. Если мы присваиваем значение формальному параметру функции, то после окончания работы описанной функции значение соответствующего параметра не определенно, так как такая переменная не является свободной!

. . . .

Например, определим следующую функцию:

```
(DEFUN F3 (A) (SETQ A (+ A 2)))
```

Вызов этой функции даст следующий результат:

```
(F3 3) ==> 5
```

При этом, если мы попытаемся обратиться к переменной A вне функции F3, то Лисп выдаст ошибку. Это связано с тем, что A является локальной переменной и имеет значение только внутри функции F3:

```
(+ A 2) ==> Ошибка! Переменная A не связана.
```

3.2 Диалоговый режим работы. Функции ввода-вывода

Как отмечалось выше, Лисп-программа представляет собой последовательность вычислимых выражений. При работе в режиме диалога пользователь вводит необходимое выражение с терминала. Оно обрабатывается Лисп-системой. Если это выражение синтаксически правильное, то оно вычисляется. Если во время вычисления не было обнаружено ошибок, то значение выражения выдается на экран дисплея. В случае ошибки на экран выдается соответствующая диагностика.

Функция и определенные в глобальном контексте переменные доступны в течение всего сеанса работы (если не было выполнено специальных действий по их уничтожению).

Порядок определения функций может быть произвольным, поскольку связь между именем функции и определяющим ее выражением осуществляется

в момент ее выполнения. Конечно, к началу выполнения функция должна быть определена.

Иногда требуется производить выдачу на экран дисплея промежуточных результатов, полученных во время вычисления некоторого выражения. Для этого служит следующая функция.

***Функция* PRINT**

(PRINT e). Это обычная функция. Она выдает на экран дисплея с новой строки значение выражения e. Значением функции становится выдаваемое на экран выражение.

Например, если ввести выражение (CONS 'A (PRINT '(1 2 3))), то на экран будет выдано следующее:

```
(1 2 3)
(A 1 2 3)
```

Первая строчка выдается функцией PRINT, а вторая – Лисп-системой.

Если необходимо вычислить одно и то же выражение с разными начальными данными, то удобно осуществлять ввод данных с терминала непосредственно при вычислении выражения. Такой ввод можно осуществить с помощью следующей функции.

***Функция* READ**

(READ). При выполнении функции на экран дисплея выдается символ приглашения на ввод. При этом необходимо ввести выражение, которое и станет значением функции READ.

READ отличается от операторов ввода-вывода других языков программирования тем, что он обрабатывает вводимое выражение целиком, а не одиночные элементы данных.

Как только интерпретатор встречает READ, вычисления приостанавливаются до тех пор, пока пользователь не введет какой-либо символ или выражение. Окончанием ввода считается либо пробел (если мы вводим атом), либо закрывающаяся скобка (если мы вводим список).

Так, если ввести выражение (CONS (READ) (READ)), то на экране дисплея появится символ приглашения на ввод. Если после этого ввести выражение (+ 2 3), то вновь появится символ приглашения. Если теперь ввести идентификатор A, то на экран будет выдано выражение ((+ 2 3) . A), которое и является значением выражения, содержащего обращение к READ.

Например, определим следующую функцию:

```
( DEFUN TR (ARG) (LIST ( + ARG ( READ ) ) ( READ ) ) )
```

При обращении к функции и вводе данных получаем:

```
(TR 8)
     14         /* ввод данных
     CAT        /* ввод данных
==> (22 CAT)
```

PRINT и READ – это псевдофункции, у которых кроме значения есть побочный эффект. Значение функции – значение ее аргумента. Побочный эффект – печать этого значения.

Это не значит, что при печати всегда выводятся две строки. Такое происходит, только когда функция PRINT используется на высшем уровне, чего практически не бывает.

Кроме основной функции вывода существуют дополнительные функции печати, которые позволяют делать вывод более гибким.

***Функция* PRINC**

(PRINC Q) печатает значение аргумента Q без пробела и перевода на другую строку:

```
(DEFUN F () (PRINC 1) (PRINC `A) (PRINC 2))
(F) ==> 1A2
```

***Функция* TERPRI**

(TERPRI) – функция без аргументов, производит перевод строки, как значение возвращает NIL.

В Лиспе существует специальный тип данных, который называется *строка*. Строки – это последовательность знаков, заключенная в кавычки.

Строка является атомом, но не может быть переменной. Как и у числа, значением строки является сама строка.

```
"(+ 1 2)" ==> "(+ 1 2)"
```

Строки удобно использовать для вывода с помощью оператора PRINC: PRINC печатает строки без кавычек.

Например, определим функцию:

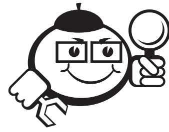
```
(DEFUN FT () (SETQ X 4) (PRINC " X = ") (PRINC X) (PRINC
" m "))
```

При вызове функции получаем:

```
(FT) ==> X = 4 m
```

Заметим, что если записать ту же функцию, использовав вместо PRINC функцию PRINT, мы получим следующий вывод:

```
(FT) ==> X =
         4
         m
```



Пример 3.1

Рассмотрим задачу поиска пути в лабиринте. Представим лабиринт в виде отдельных комнат, соединенных проходами. Комнаты обозначим идентификаторами.

Составим список, в котором после каждого имени комнаты укажем список из имен комнат, с которыми данная комната непосредственно соединена проходами. Полученный список присвоим переменной `LABYRINTH`. Так будет задан лабиринт в программе.

Например, рассмотрим лабиринт, представленный на рисунке 3.1.

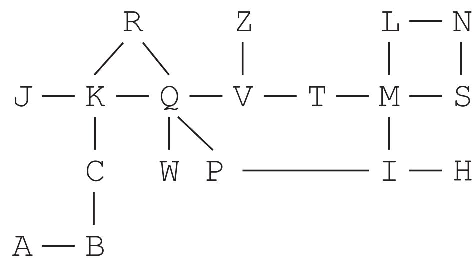


Рис. 3.1 – Пример лабиринта

Он будет представлен в программе так:

```
(SETQ LABYRINTH '(A (B) B (C A) C (B K) H (I) I (P M H) J (K)
K (J C Q R) L (M N) M (T I L S) N (L S) P (Q I) Q (V P R K W) R (Q
K) S (M N) T (M V) V (Q T Z) W (Q) Z (V)))
```

Задача состоит в том, чтобы найти все пути без циклов (где нет комнат, проходимых более одного раза) из одной заданной комнаты в другую. Путь будем задавать списком из пройденных комнат.

Определим функцию `WAY`, которая выполняет поиск путей:

```
(DEFUN WAY (A B1) (SETQ B B1) (PRLISTS (PATH A ())))
```

Параметр `A` определяет исходную комнату, а параметр `B1` – конечную.

Функция `PATH` выдает список, содержащий все пути из комнаты, определяемой первым аргументом, в комнату `B` (`B` – глобальная переменная в теле `PATH`). Второй аргумент задает пройденный путь. В начальный момент – это пустой список.

Функция `PRLISTS` служит для выдачи на экран найденных ею путей. Ее можно определить так:

```
(DEFUN PRLISTS (L)
     (COND ((NULL L) 0)
           (T (PRINT (CAR L)) (+ 1 (PRLISTS (CDR L)))) ))
```

Значением PRLISTS является число найденных путей.

Функцию PATH определим следующим образом:

```
(DEFUN PATH (THIS PATH)
     (COND ((EQ THIS B) (LIST (REVERSE (CONS B PATH))))
           ((MEMBER THIS PATH) ())
           (T (NEXT (CADR (MEMBER THIS LABYRINTH))
                                   (CONS THIS PATH)))
     ))
```

В функции PATH вначале проверяется, совпадает ли имя текущей комнаты THIS с именем конечной комнаты B. Если это так, то выдается список, содержащий найденный путь. Поскольку в процессе вычислений заносить элементы удобно в начало списка, а не в конец, то имена комнат будут перечислены в обратном порядке. Для получения правильного порядка комнат применяется функция REVERSE.

Во втором условии функции COND проверяется, не была ли ранее пройдена эта комната. Если была, то значением PATH будет NIL, что означает: путь не найден, текущая вершина найдена неверно.

Третий случай предполагает продвижение в соседние комнаты. Это осуществляется функцией NEXT, которая, исходя из текущего пути P, формирует список путей, получаемых при переходе в соседние комнаты. Приведем определение функции NEXT:

```
(DEFUN NEXT (N P)
     (COND ((NULL N) NIL)
           (T (APPEND (PATH (CAR N) P)  (NEXT (CDR N) P)))  ))
```

Если теперь Лисп-системе дать на ввод выражение (WAY 'A 'Z), то на экран будет выдано следующее:

```
(A B C K Q V Z)
(A B C K Q P I M T V Z)
(A B C K R Q V Z)
(A B C K R Q P I M T V Z)
4
```

Таким образом, будут получены четыре пути без циклов из комнаты A в комнату Z в заданном лабиринте LABYRINTH.

Задавая другие параметры в обращении к функции WAY, можно получить пути, соединяющие другие комнаты. Можно исследовать другой лабиринт, изменив содержимое переменной LABYRINTH. При этом не требуется изменять или вводить заново используемые в программе функции.

. . . .

3.3 Разрушающие функции

До сих пор мы рассматривали такой способ обработки списков, при котором формируются списки, но не изменяется структура уже существующих.

Однако если нам приходится часто изменять отдельные элементы списка, тем более если список при этом большой длины, то частое формирование новых списков приведет к долгому выполнению программы.

Хотелось бы иметь средство, позволяющее изменять структуру списков подобно тому, как мы изменяем, например, элементы массивов в других языках программирования.

Такой способ в Лиспе есть. Он, в противовес формированию, называется *разрушением*, а функции, которые изменяют структуру списков, называются *разрушающими*.

Следующие две разрушающие функции являются основными.

***Функция* RPLACA**

(RPLACA X E). Обычная функция. Она заменяет левую ссылку в первом звене непустого списка X ссылкой на выражение E. Значением функции является измененный список.

Например, если значением переменной A является список (ONE TWO), то после выполнения (RPLACA A 'ONLY) значением переменной A станет список (ONLY TWO).

Приведем другой пример. Пусть значением переменной X является список (BLUE PEN). В результате выполнения (SETQ Y (CONS 'ONE X)) значением Y становится список (ONE BLUE FEN).

Представим схематично, что получилось, на рисунке 3.2.

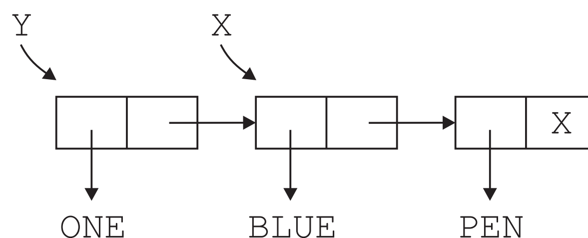


Рис. 3.2 – Внутреннее представление списка в результате работы функции CONS

Выполним теперь (RPLACA X 'GREEN). Рассмотрим схему полученной структуры на рисунке 3.3.

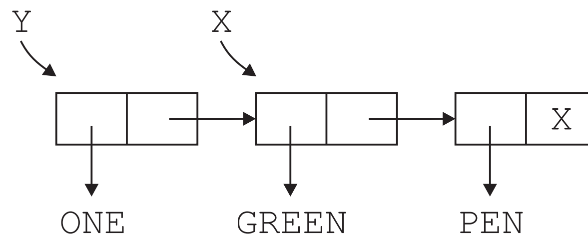


Рис. 3.3 – Внутреннее представление списка в результате работы функции RPLACA

Таким образом, изменилось значение и у переменной Y:

```
Y ==> (ONE GREEN PEN)
```

Отметим, что точно такой же эффект получился бы, если бы мы выполнили вместо (RPLACA X 'GREEN) выражение (RPLACA (CDR Y) 'GREEN), при этом значение X также изменилось бы.

***Функция* RPLACD**

(RPLACD X E). Обычная функция. Она заменяет правую ссылку в первом звене непустого списка X на выражение E. Значением функции является измененный список X.

Например, если значением переменной Z является список (A B), то после выполнения (RPLACD Z 2) значением Z станет список (A . 2).

Рассмотрим другой пример. Пусть значения X и Y такие же и так же структурно связаны, как во втором примере с функцией RPLACA. Тогда результатом выполнения (RPLACD X '(PENCIL)) будет структура, приведенная на рисунке 3.4.

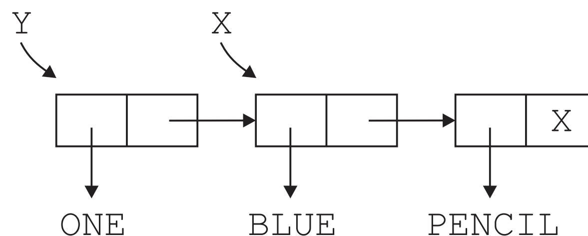


Рис. 3.4 – Внутреннее представление списка в результате работы функции CONS

Рассмотрим теперь ряд полезных производных разрушающих функций, которые являются встроенными во многих современных диалектах.

***Функция* NCONC**

(NCONC X Y). Обычная функция. Она заменяет ссылку в последнем звене списка X ссылкой на Y. Значением функции при этом становится изме-

ненный список X. Если исходный список X пустой, то значением функции будет Y.

Список не копируется! Вместо этого NIL в последней списочной ячейке меняется на указатель к первой списочной ячейке второго списка.

Ее можно определить так:

```
(DEFUN NCONC (X Y)
     (COND ((ATOM X) Y)
           (T (RPLACD (LAST X) Y) X)))
```

Для многих разрушающих функций существуют *неразрушающие двойники* – функции, которые формируют значения, во многих случаях такие же, как и соответствующие разрушающие, но при этом не изменяют структуру аргументов. Вместе с тем разрушающие функции являются *разрушающими двойниками* по отношению к своим неразрушающим двойникам. Так, функция NCONC является разрушающим двойником для функции APPEND.

Пусть, например, значением переменной X является список (A B C), а значением Y – список (1 2), тогда в результате выполнения (NCONC X Y) значением функции и переменной X станет список (A B C 1 2).

***Функция* NREVERSE**

(NREVERSE X). Обычная функция. Значением ее является перевернутый список X. При этом функция разрушает исходную структуру аргумента, причем новым значением аргумента вовсе не является перевернутый список.

Определить функцию можно так:

```
(DEFUN NREVAPPEND (A B)
     (COND ((ATOM A) B)
           (T (NREVAPPEND (CDR A) (RPLACD A B)))))
(DEFUN NREVERSE (X) (NREVAPPEND X NIL))
```

Мы определили вспомогательную функцию NREVAPPEND, которая приписывает к перевернутому списку A список B, причем исходный список A при этом разрушается.

Для функции NREVERSE неразрушающим двойником является функция REVERSE.

Если, например, значением переменной A является список и необходимо заменить ее значение на перевернутый список, то следует выполнить (SETQ A (NREVERSE A)), поскольку разрушенное значение аргумента не совпадает со значением функции NREVERSE.

***Функция* DELETE**

(DELETE X Y). Обычная функция. Ее значением является список Y, из которого удалены все атомы X. Функция разрушает прежнее значение аргумента Y, причем новое значение этого аргумента может не совпадать со значением функции.

Для функции DELETE неразрушающим двойником является функция REMOVE.

При выполнении разрушающих функций могут возникать циклические структуры. Для работы с такими структурами необходимо проверять совпадение ссылок на структуры. Для этого служит рассмотренная ранее функция EQ.

Дело в том, что совпадающие по написанию идентификаторы присутствуют в Лисп-системе в единственном экземпляре, поэтому при их сравнении достаточно сравнивать ссылки. Однако внешне одинаковые числа и списки могут присутствовать в системе в нескольких экземплярах. Поэтому для их сравнения совпадения ссылок недостаточно, приходится использовать другие функции сравнения (= и EQUAL). Однако для отыскания циклов в списочных структурах необходимо сравнивать именно ссылки.

Пусть, например, значением переменной X является список (A B C). Тогда

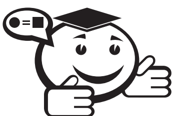
```
(LIST (SETQ Y (CDR X)) (SETQ Z (CDR X)) (EQ Y Z)) ==>
                                   ==> ((B C) (B C) T)
```

С другой стороны,

```
(LIST (SETQ Y (CDR X)) (SETQ Z (CONS 'B (CDDR X)))
(EQ Y Z)) ==> ((B C) (B C) NIL)
```

3.4 Функционалы

До сих пор мы рассматривали функции, которые в качестве аргументов использовали данные. Однако в качестве аргумента функции можно указывать и функцию.



*Аргумент, значением которого является функция, называют **функциональным аргументом**, а функцию, имеющую функциональный аргумент, – **функционалом**.*

В Лиспе различие между понятиями «данные» и «функция» определяется не на основе их структуры, а в зависимости от их использования.

Если аргумент используется в функции как объект, участвующий в вычислениях, то это данные.

Если аргумент используется как средство, определяющее вычисления, то это функция.

В языке Лисп есть возможность выполнять сформированные выражения. Для этого служит функция EVAL.

***Функция* EVAL**

(EVAL e). Обычная функция. Ее значением является вычисленное значение аргумента (значение значения фактического параметра).

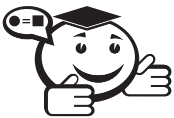
Пример:

```
(EVAL (LIST 'CAR '(QUOTE (A B)))) ==> A
(EVAL '(+ 1 2 3)) ==> 6
(QUOTE (EVAL (+ 1 2 3))) ==> (EVAL (+1 2 3))
```

Последние два примера показывают взаимосвязь EVAL и QUOTE.

С помощью функции EVAL можно определять *функционалы* – функции, аргументами которых являются имена других функций, применяемых функционалами для обработки данных.

Различают два типа функционалов: применяющие и отображающие функционалы.



*Функционал, который применяет функциональный аргумент к остальным аргументам, называется **применяющим**.*

*Функционал, который применяет функциональный аргумент к элементам списка, в результате чего строится новый список, называется **отображающим**: исходный список отображается на результирующий.*

Основными отображающими функционалами являются функции MAPCAR и MAPLIST. К этому классу функционалов также относятся функции MAPCON, MEMBER-IF, REMOVE-IF, SUBSTITUTE-IF, SUBST-IF.

Рассмотрим работу этих функций.

***Функция* MAPCAR**

(MAPCAR F X). Обычная функция. Она применяет обычную функцию F (от одного аргумента) к каждому элементу списка X, причем элементы списка при этом не вычисляются. Значением MAPCAR является список из полученных значений.

Примеры использования:

```
(MAPCAR 'REVERSE '((1 2) (A B C))) ==> ((2 1) (C B A))
(MAPCAR '(LAMBDA (X) (+ X 10)) '(7 8 9)) ==> (17 18 19)
```

MAPCAR может обрабатывать больше списков, если в функциональном аргументе несколько аргументов: (MAPCAR F X1 X2 … XN).

Например:

```
(DEFUN ADDLIST (L1 L2)  (MAPCAR `+  L1 L2))
(ADDLIST `(1 2 3) `(4 5 6)) ==> (5 7 9)
```

Если списки разной длины, то длина результата будет равна длине наименьшего.

***Функция* MAPLIST**

(MAPLIST F X). Обычная функция. Она применяет функцию F (одного аргумента) вначале ко всему списку X, затем к списку X без начального элемента, потом к списку X без двух первых элементов и т. д., в конце концов к последнему звену списка X. Функция MAPLIST выдает в качестве результата список из значений, выработанных функцией F.

Примеры использования:

1) попарное сравнение двух соседних элементов списка:

```
(MAPLIST '(LAMBDA (X) (COND ((CDR X) (< (CAR X) (CADR X)))))
                  '(7 2 1 5 3 8)) ==> (NIL NIL T NIL T NIL)
```

2) пусть задан список: (setq zx `(d e 4 5 1 g))

Тогда, если мы применим CAR, то получим тот же список:

```
(maplist `car zx) =>  (D E 4 5 1 G)
```

При применении CDR получаем:

```
(maplist `cdr zx) => ((E 4 5 1 G) (4 5 1 G) (5 1 G) (1 G) (G)
NIL)
```

***Функция* MAPCAN**

(MAPCAN F X). Обычная функция. Она подобна MAPCAR, только соединяет результаты выполнения F, используя NCONC.

Пример:

```
(MAPCAN '(LAMBDA (X) (AND (> X 0) (LIST X))) '(-1 2 -7 3 -9 8
5)) ==> (2 3 8 5)
```

***Функция* MAPCON**

(MAPCON F X). Обычная функция. Она подобна MAPLIST, только для соединения результатов выполнения F применяет NCONC.

Пример:

```
(MAPCON '(LAMBDA (X) (COND ((MEMBER (CAR X) (CDR X) NIL)
                           (T (LIST (CAR X)))))
'(A B A C C D)) ==> (B A C D)
```

В данном примере из списка удаляются повторяющиеся элементы.

***Функция* MEMBER-IF**

(MEMBER-IF F X). Обычная функция. Она ищет элемент, для которого одноместный предикат F принимает истинное значение, и выдает часть списка X, начиная с этого элемента. При применении предиката элементы списка не вычисляются. Если для всех элементов списка значение F будет NIL, то значением функции станет также NIL.

Пример:

```
(MEMBER-IF 'ATOM '((+ 2 3) 7 (A B))) ==> (7 (A B))
```

***Функция* REMOVE-IF**

(REMOVE-IF F X). Обычная функция. Она формирует новый список, который получается из X удалением всех элементов, для которых значение предиката F является истинным. При применении предиката элементы списка не вычисляются.

Пример:

```
(REMOVE-IF '(LAMBDA (X) (> X 5)) '(10 7 3 8 2 1)) ==> (3 2 1)
```

***Функция* SUBSTITUTE-IF**

(SUBSTITUTE-IF X F Y). Обычная функция. Она формирует новый список, получающийся из списка Y путем подстановки выражения X вместо каждого элемента, для которого истинен предикат F. Элементы списка при выполнении предиката не вычисляются.

Пример:

```
(SUBSTITUTE-IF 9 'ZEROP '(0 5 7 0 2)) ==> (9 5 7 9 2)
```

***Функция* SUBST-IF**

(SUBST-IF X F Y). Обычная функция. Она формирует новое выражение из Y путем подстановки выражения X вместо тех выражений на всех уровнях Y, для которых одноместный предикат F принимает истинное значение. Выражения, к которым применяется предикат, не вычисляются.

Пример использования:

```
(SUBST-IF 'A '(LAMBDA (X) (NOT (ATOM X))) '(M (1 2) 3)) ==> A
(SUBST-IF 'Q 'NUMBERP '(M (1 2) 3)) ==> (M (Q Q) Q)
```

К применяющим функционалам относятся функции FUNCALL и APPLY.

Функция FUNCALL

(FUNCALL X Y1 Y2 … YN). Принимает произвольное количество аргументов YN. Эта функция применяет свой первый (функциональный) аргумент X к оставшемуся списку аргументов Y1, Y2, ..., YN.

Например:

```
(FUNCALL '* 1 2 3 4 5 6 7 8 9 10) ==> 3628800
```

Использование FUNCALL позволяет, например, давать именам функций синонимы:

```
(SETQ сложить '+)
(FUNCALL сложить 1 2 3) ==> 6
```

Функция **APPLY**

(APPLY X Y). Обычная функция. Вызов APPLY заключается в том, что вычисляется функция, заданная аргументом X, со списком параметров, заданным аргументом Y.

Пример использования:

```
(SETQ MULT  '*)
 MULT ==> *
  (APPLY MULT  '(1 2 3 4 5 6 7 8 9 10)) ==> 3628800
```

3.5 Циклы и блочные функции

3.5.1 Блочные функции

В COMMON LISP, а также во многих более ранних диалектах языка Лисп есть PROG-выражение, позволяющее вводить в употребление блок с метками. Функция PROG относится к управляющим структурам, которые объединяют последовательные вычисления.

Функция **PROG**

(PROG (i1 i2 ... in) s1 s2 ... sk). Это особая функция. Здесь i1, i2, ..., in (n>=0) – идентификаторы, а s1, s2, ..., sk (k>=1) – идентификаторы или вычислимые выражения.

При выполнении функции PROG вначале объявляются локальные переменные с именами i1, i2, ..., in и начальными значениями NIL. После этого последовательно вычисляются s1, s2, ..., sk, не являющиеся идентификаторами. Выражения sj, представляющие собой идентификаторы, называются метками. Во время выполнения PROG-выражения может встретиться обращение к функции GO, которая осуществляет переход по метке, или обращение к

RETURN, осуществляющей выход из функции PROG с заданным значением. Если при выполнении PROG-выражения обращения к функции RETURN не было, то значением функции PROG становится NIL.

Пример использования:

```
(PROG (X Y) (SETQ X 'A)
            (PRINT (CONS X Y))) ==> NIL
```

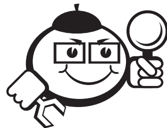
При этом на печать будет выдано (A).

***Функция* RETURN**

(RETURN E). Это обычная функция. Она осуществляет выход из ближайшего объемлющего PROG-выражения, причем значением PROG-выражения становится значение выражения E.

***Функция* GO**

(GO X). Особая функция. Аргумент должен быть идентификатором. Он не вычисляется. Функция осуществляет переход по метке X в ближайшем объемлющем PROG-выражении. Таким образом, после выполнения функции GO будет вычисляться первое из выражений sj (1<=j<=k), которое не является идентификатором и следует за s0, совпадающим с X.

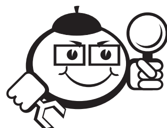


Пример 3.2

Рассмотрим реализацию функции REVERSE без рекурсии, но с использованием PROG-выражения.

```
(DEFUN REVERSE (X)
   (PROG (Y)
     L (COND ((ATOM X) (RETURN Y))
           (T (SETQ Y (CONS (CAR X) Y))(SETQ X (CDR X))(GO L))
        )
))
```

При использовании PROG-выражений часто встречаются конструкции вида (SETQ x (CONS e x)) и (SETQ y (CDR y)). Для упрощения записи таких конструкций используются функции (PUSH e x) и (POP y), которые осуществляют те же действия, что и рассмотренные конструкции.



Пример 3.3

Рассмотрим определение функции REVERSE с использованием функций POP и PUSH.

```
(DEFUN REVERSE (X)
     (PROG (Y)
          L (COND ((ATOM X) (RETURN Y))
                 (T (PUSH (CAR X) Y) (POP X) (GO L)))   ))
```

. . . .

В том случае, когда используется вычисление последовательности форм, удобно бывает ввести локальные переменные, сохраняемые до окончания вычислений. Это делается с помощью предложения LET.

***Функция* LET**

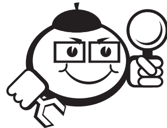
(LET ((i1 v1) (i2 v2) ... (ik vk)) e1 e2... en). Это особая функция. Она вводит в употребление локальные переменные i1, i2, ..., ik, присваивая им соответственно значения выражений v1, v2, ..., vk. После этого последовательно вычисляются значения выражений e1, e2, ..., en. Значение en становится значением LET.

После выхода из LET связи переменных i1, i2, ..., ik ликвидируются.

Предложение LET удобно использовать, когда надо временно сохранять промежуточные значения.

Пример использования:

```
(LET ((A (+ 2 3)) (B '(6 7 8))) (CONS A B)) ==> (5 6 7 8)
```

. . . .  Пример 3.4 . . . .

Рассмотрим функцию RECTANGLE, которая имеет один аргумент – список из двух элементов, задающих длину и ширину прямоугольника. Функция рассчитывает и печатает площадь и периметр прямоугольника.

Описание функции:

```
(DEFUN RECTANGLE (DIM)
     (LET ((LEN (CAR DIM)) (WID (CADR DIM)))
          (PRINT (LIST 'AREA (* LEN WID)))
          (PRINT (LIST 'PERIMETR (* (+ LEN WID) 2)))) ))
```

Вызов функции:

```
(RECTANGLE '(4 5)) ==>
          (AREA 20)
          (PERIMETR 18)
```

. . . .

3.5.2 Циклические предложения

До сих пор при определении функций, обрабатывающих списки, мы часто использовали рекурсию. Это удобно: запись определения получается краткой и наглядной.

Однако при выполнении рекурсивной функции требуется дополнительная память, объем которой пропорционален глубине рекурсии и во многих реализациях вычисляется по формуле:

(M*P+K)*L байтов,

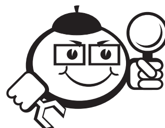
где K и P – некоторые фиксированные величины, M – число локальных переменных и формальных параметров функции, а L – глубина рекурсии. Из формулы вытекает, что при обработке длинных списков рекурсивными функциями может не хватить памяти.

Вместо рекурсии можно воспользоваться циклами. Объем памяти, используемый в цикле, без учета памяти, занимаемой значениями переменных, не зависит от числа итераций. Кроме того, циклические алгоритмы, как правило, выполняются быстрее рекурсивных.

В языке COMMON LISP и некоторых других современных диалектах для организации циклов используется функции LOOP, DO и DOTIMES.

***Функция* LOOP**

(LOOP форма1 форма2 .....). Предложение LOOP реализует бесконечный цикл, в котором формы вычисляются до тех пор, пока не встретится явный оператор завершения RETURN.


Пример 3.5

Применение LOOP *для численных итераций.*

Определим функцию ADD_INTEGERS, которая будет брать один аргумент N, являющийся положительным целым, и возвращать сумму всех чисел между 1 и этим числом: 1+2+3+4+ ... +N.

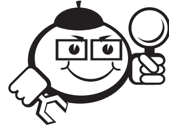
Например, при N=4. Функция должна возвратить 10:

```
(ADD_INTEGERS 4) ==> 10
```

Определение функции:

```
(DEFUN ADD_INTEGERS (LAST)
     (LET ((COUNT 1) (TOTAL 1))
          (LOOP (COND ((EQUAL COUNT LAST) (RETURN TOTAL)))
```

```
                 (SETQ COUNT (+ 1 COUNT))
                 (SETQ TOTAL (+ TOTAL COUNT))  )))
```



Пример 3.6

Рассмотрим еще один пример численной итерационной функции. Определим функцию `INT_MULTIPLY`, выполняющую умножение двух целых чисел через сложение. Таким образом, умножение `X` на `Y` выполняется путем сложения `X` с самим собой `Y` раз.

Например, `3*4` это `3 + 3 + 3 + 3 = 12`

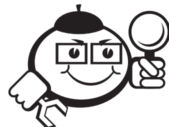
```
(DEFUN INT_MULTIPLY (X Y)
   (LET ((RESULT 0)(COUNT 0))
        (LOOP  (COND ((EQUAL COUNT Y) (RETURN RESULT)))
               (SETQ COUNT (+ 1 COUNT))
               (SETQ RESULT (+ RESULT X))  )))
```

Вызов функции:

```
(INT_MULTIPLY 3 4) ==> 12
```

Из приведенных примеров можно получить *общую форму для численной итерации*:

```
(DEFUN  < имя-функции > < список-параметров >
     (LET (< инициализация переменной индекса >
           < инициализация переменной результата >)
     (LOOP
          (COND < проверка индекса на выход > (RETURN резуль-
     тат))
          < изменение переменной счетчика >
          < другие действия в цикле, включая изменение переме-
     ной результата >
)))
```



Пример 3.7

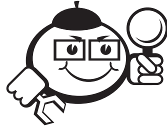
Определим функцию `FACTORIAL` с помощью общей формы для численной итерации:

```
(DEFUN FUCTORIAL ( NUM )
     (LET ((C 0)( P 1))
          (LOOP
               (COND (( EQUAL C NUM) (RETURN P)))
```

```
                (SETQ C (+ 1 C))
                (SETQ P (* C P ))  )))
```

Вызов функции:

```
(FACTORIAL 5) ==> 120
```



Пример 3.8

Определим функцию READ-SUM, использующую печать и ввод. Функция без аргументов читает серию чисел и возвращает сумму этих чисел, когда пользователь вводит не число.

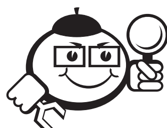
Для удобства ввода функция должна печатать "Enter the next number: " перед каждым вводом.

Определение функции:

```
(DEFUN READ-SUM ()
     (LET ((INPUT) (SUM 0))
          (LOOP (PRINC "Enter the next number: ")
                (SETQ INPUT (READ))
                (COND ((NOT (NUMBERP INPUT)) (RETURN SUM)))
                (SETQ SUM (+ INPUT SUM))  )))
```

Вызов функции:

```
(READ-SUM) ==>
     Enter the next number: 15
     Enter the next number: 30
     Enter the next number: 45
     Enter the next number: stop
     ==> 90
```



Пример 3.9

Применение LOOP *для итераций со списками.*

Необходимо создать функцию DOUBLE-LIST, принимающую список чисел и возвращающую новый список, в котором каждое число удвоено.

Описание функции:

```
(DEFUN DOUBLE-LIST ( LIS )
     (LET ((NEWL NIL))
          (LOOP
```

```
            (COND (( NULL LIS ) (RETURN NEWL)))
            (SETQ NEWL (APPEND NEWL (LIST (* 2 (CAR LIS)))))
            (SETQ LIS (CDR LIS ))  )))
```

Вызов функции:

```
(DOUBLE-LIST '(5 15 10 20)) ==> (10 30 20 40)
```

Посмотрим, как будет идти вычисление (табл. 3.1).

Таблица 3.1 – Процесс итерации для примера 3.9

	LIS	NEWL
Начальное состояние	(5 15 10 20)	()
Итерация 1	(15 10 20)	(10)
Итерация 2	(10 20)	(10 30)
Итерация 1	(20)	(10 30 20)
Итерация 4	()	(10 30 20 40)
Результат		(10 30 20 40)

. . . .

***Функция* DO**

Это самое общее циклическое предложение. Форма записи:

```
(DO ((v1 z1 step1) (v2 z2 step2) ...)
     (P s1 s2 ... sn)
     e1 e2 ... ek).
```

Функция работает следующим образом:

1. Вначале объявляются локальные переменные `v1...vn`, которым присваиваются начальные значения `z1...zn`. Переменным, для которых не заданы начальные значения, присваивается `NIL`.
2. Затем проверяется условие окончания цикла `P`, если оно выполняется, то вычисляются выражения `s1`, `s2`, `...`, `sn`. В качестве значения берется значение последнего выражения. На этом выполнение функции `DO` завершается.
3. Если условие `P` не выполняется, то вычисляются выражения `e1`, `e2`, `... ek`.
4. На следующем цикле переменным `vi` присваиваются одновременно новые значения, определяемые значениями шагов `stepi`, и все повторяется.

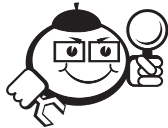
Так продолжается до тех пор, пока значение `P` не станет истинным.

Поскольку значения шагов присваиваются переменным одновременно, то при отсутствии побочных эффектов (что весьма желательно) порядок vi в списке локальных переменных не влияет на результат выполнения DO.

Например, выражение

```
(DO (( X 1 ( + 1 X))) (( > X 10) ( PRINT 'END)) ( PRINT X))
```

будет печатать последовательность чисел от 1 до 10, а в конце будет напечатано «END».



Пример 3.10

Напишем функцию LIST-ABS, которая берет список чисел и возвращает список абсолютных величин этих чисел.

Первый вариант функции запишем с использованием функции LOOP:

```
(DEFUN LIST-ABS (LIS)
      (LET ((NEWL NIL))
            (LOOP
                  (COND (( NULL LIS ) (RETURN (REVERSE NEWL))))
                  (SETQ NEWL (CONS (ABS (CAR LIS)) NEWL))
                  (SETQ LIS (CDR LIS ))  )))
```

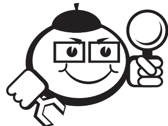
Вызов функции:

```
(LIST-ABS  '(-1 2 -4 5)) ==> (1 2 4 5)
```

Запишем теперь эту же функцию, используя функцию DO:

```
(DEFUN LIST-ABS (LIS)
      (DO ((OLDL LIS (CDR OLDL))
           (NEWL NIL (CONS (ABS (CAR OLDL)) NEWL)))
           ((NULL OLDL) (REVERSE NEWL))  ))
```

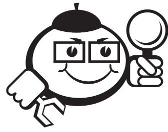
Отметим еще одну особенность выполнения DO: при отсутствии s1, s2, ..., sn значением DO будет NIL.



Пример 3.11

Определение функции REVERSE можно записать следующим образом:

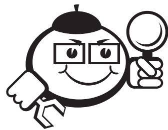
```
(DEFUN REVERSE (L)
      (DO ((X L (CDR X))
            (Y NIL (CONS (CAR X) Y)))
            ((NULL X) Y) ))
```



Пример 3.12

Приведем определение функции (EXP1 X), которая вычисляет значение e^x для простейшего случая, когда $abs(x)<1$, по ряду $1+\frac{x}{1!}+\frac{x^2}{2!}+...+\frac{x^n}{n!}+...$ с точностью 10^{-6}.

```
(DEFUN EXP1 (X)
   (DO ((I 1 (+ I 1))
        (U 1 (/ (* U X) I))
        (Z 0 (+ Z U)))
       ((< (ABS U) 0.000001) Z)))
```



Пример 3.13

Рассмотрим определение функции LIST_SORT, которая сортирует список, располагая его элементы по не убыванию. Исходная структура списка при этом разрушается. Вводятся вспомогательные функции S_LIST и MSORT. Функция S_LIST формирует упорядоченный список из элементов двух упорядоченных списков.

```
(DEFUN S_LIST (L1 L2)
   (COND ((ATOM L1) L2)
         ((ATOM L2) L1)
         (T (LET ((L NIL))
           (COND ((< (CAR L1) (CAR L2)) (SETQ L L1) (POP L1))
                 (T (SETQ L L2) (POP L2)) )
           (DO ((P L (CDR P)))
               ((OR (ATOM L1) (ATOM L2))
               (COND ((ATOM L1) (RPLACD P L2))
                     (T (RPLACD P L1)))
              L)
          (COND ((< (CAR L1) (CAR L2)) (RPLACD P L1) (POP L1))
                (T (RPLACD P L2) (POP L2))) ) ) )))
(DEFUN MSORT (X1 N)
   (COND ((<= N 1) X1)
         (T (DO (X2 (N2 (TRUNCATE N 2)) (K 1 (+ K 1))
                   (P X1 (CDR P)))
                ((= K N2) (SETQ X2 (CDR P))
                    (RPLACD P NIL)
```

```
                        (SETQ X1 (MSORT X1 K))
                        (SETQ X2 (MSORT X2 (- N K)) )
                        (MERGE X1 X2))
               ) )))
(DEFUN LISTSORT (L) (MSORT L (LENGTH L)))
```

....

DOTIMES

DOTIMES можно использовать вместо DO, если надо повторить вычисления заданное число раз.

Общая форма:

(DOTIMES (var num форма-return) (форма-тело)),

здесь var – переменная цикла,

num – форма определяющая число циклов,

форма-return – результат, который должен быть возвращен.

Прежде всего вычисляется num-форма, в результате получается целое число – N.

Затем var меняется от 0 до N (исключая N) и, соответственно, каждый раз вычисляется форма-тело.

Последним вычисляется форма-return.

Если форма-return отсутствует, возвращается nil.

Например,

```
(dotimes ( x 3 ) ( print x)) ==>
    0 - автоматически
    1
    2
    t
(let ((x nil))
    (dotimes (n 5 x)
        (setq x (cons n x))))
(4 3 2 1 0)
```

3.6 Массивы

Для хранения большого количества данных в Лиспе используются массивы. Массив – это набор переменных (ячеек) имеющих одно имя, но разные номера, обеспечивающие доступ к этим ячейкам.

В Лиспе поддерживаются только одномерные массивы-векторы.

Определение массива

Для определения массива заданной размерности используется функция MAKE-ARRAY. Обращение к функции:

```
(MAKE-ARRAY  N),
```

где N – размерность массива (количество ячеек в массиве).

Например, создадим массив DATA, содержащий 10 ячеек:

```
(setq data (make-array 10)) ==>
                         (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

При создании массива всем его элементам первоначально присваивается значение NIL.

Нумерация элементов массива начинается с 0.

Доступ к ячейке массива

Доступ к ячейкам массива производится с помощью функции AREF:

```
(AREF N I).
```

Функция имеет два аргумента – имя массива N и индекс I и возвращает значение ячейки с указанным индексом, например

```
(aref data 8) ==> NIL
```

Запись данных в массив

Поместить данные в массив можно, используя функцию SETF:

```
(SETF (AREF  N I) <значение>).
```

Функция AREF вызывает значение ячейки с номером I из массива с именем N, функция SETF помещает значение в эту ячейку:

```
(setf (aref data 2) 'dog) ==> dog
```

Создадим следующий массив testdata:

```
(setq testdata (make-array 4))
(setf (aref testdata 1) 'dog)
(setf (aref testdata 0) 18 )
(setf (aref testdata 2) '(a b) )
(setf (aref testdata 3) 0 )
```

В результате получим

```
testdata ==> #(18 dog (a b) 0)
```

Можно использовать эти данные для анализа, обработки, формирования новых списков и т. д. Например, из сформированного массива testdata можно сформировать новый список:

```
(cons (aref testdata 1) (list (aref testdata 3) (aref test-
data 2))) ==> (dog 0 (a b))
```

Можно использовать значения ячеек массива в качестве индекса для определения значения других ячеек массива:

```
(aref testdata (aref testdata 3)) ==> 18
```

В приведенном примере `(aref testdata 3)` возвращает 0, а вызов `(aref testdata 0)` вернет 18.

Обработка массивов

Так как доступ к элементам массива производится по номерам, то удобно использовать численные итерации и рекурсии.

Рассмотрим функцию, которая берет два аргумента: имя массива и его длину – и возвращает все значения, помещенные в список.

```
(defun array-list (arnam len)
     (do (( i 0 ( + 1 i)) (result nil (cons (aref arnam i)
result)))
          ((equal i len) (reverse result)) ))
```

Вызов функции:

```
(array-list testdata 4) ==> ( 18 dog (a b) 0)
```

Длина массива

Функция `ARRAY-LENGTH` возвращает длину массива:

`(array-length Y)`. Здесь `Y` – имя рассматриваемого массива.

Например, `(array-length testdata)` вернет 4.

3.7 Свойства символов

В Лиспе с символом можно связывать не только значение, но и информацию, называемую списком свойств (property list).

Списки свойств широко используются для представления смысловой информации, когда с атомом необходимо связать список определенных свойств и связанных с ними значений. Существуют специальные функции языка, позволяющие выполнять различные действия с такими списками.

Например, рассмотрим следующую информацию (табл. 3.2)

Таблица 3.2 – Информация о женщине с именем Лена

Свойство	Значение
Возраст	28
Профессия	Юрист
Зарплата	90
Дети	Петя Юра Ира

Данную информацию можно представить в виде списка свойств, который будет выглядеть следующим образом:

```
(возраст 28 профессия юрист зарплата 90 дети (Петя Юра Ира))
```

В общем виде список свойств выглядит следующим образом:

```
(P1 X1 P2 X2 … PN XN),
```

где `Pi` – `i`-е свойство,

`Xi` – значение `i`-го свойства.

При этом имя свойства `Pi` может быть задано только нечисловым атомом, в то время как его значение `Xi` может быть как атомом, так и списком.

Свойств у атома может быть много, но у каждого свойства только одно значение.

Присвоение свойства

Чтобы задать свойство, необходимо использовать обобщенную функцию присвоения `SETF`:

```
(SETF ( GET <символ> <свойство>) <значение>).
```

При внесении нового свойства, оно помещается в начало списка свойств.

Определим список свойств из нашего примера:

```
(SETF (GET `Лена `возраст) 28)
(SETF (GET `Лена `профессия) `юрист)
(SETF (GET `Лена `зарплата) 90)
(SETF (GET `Лена `дети) `(Петя Юра Ира))
```

Заметим, что если присвоение свойств происходило именно в таком порядке, то с атомом Лена будет связан следующий список свойств:

```
(дети ( Петя Юра Ира) зарплата 90 профессия юрист возраст 28)
```

Чтение свойства

Сначала свойство задается, а затем извлекается. Узнать свойство атома можно, используя функцию:

```
(GET <символ> <свойство>).
```

Функция `GET` возвращает значение указанного свойства. Например:

```
(GET `Лена `возраст) ==> 28
(GET `Лена `дети) ==> ( Петя Юра Ира)
(GET `Лена `зарплата) ==> 90
```

Замена свойства

Замена значения свойства производится повторным присвоением.

Например,

```
(SETF (GET `Лена `возраст) 29)
```

Если возникает необходимость замены текущего значения новым, используя при этом текущее, можно поступить следующим образом:

```
(SETF (GET `Лена `возраст) (+ 1 (GET `Лена `возраст))
```

Удаление свойства

Удаление свойства и его значения производится функцией

```
(REMPROP <символ> <свойство>).
```

Например, если из вышеприведенного списка свойств необходимо удалить свойство «дети», запишем:

```
(REMPROP `Лена  `дети)
```

Просмотр информации о списке свойств

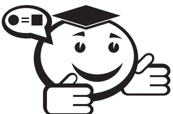
```
(SYMBOL-PLIST <символ>).
```

Функция `SYMBOL-PLIST` даст информацию о списке свойств

```
(SYMBOL-PLIST `Лена) ==> (дети ( Петя Юра Ира) зарплата 90
                               профессия юрист возраст 28)
```

3.8 Ассоциативные списки

Структура ассоциативных списков



*Ассоциативный **список**, или просто **а-список (a-list)**, есть основанная на списках и точечных парах структура данных, описывающая связи наборов данных и ключевых полей, для работы с которой существуют готовые функции.*

Ассоциативные списки могут быть следующих двух видов:

`((key1.obj1) (key2.obj2) … (keyN.objN))` или

`((key1 obj1) (key2 obj2) … (keyN objN))`,

где `keyi` – `i`-е ключевое поле,

`obji` – набор данных `i`-го ключевого поля.

Первый вид ассоциативных списков имеет место в тех случаях, когда наборы данных, связанные с ключами, являются атомарными объектами.

Ассоциативные списки применяются при работе с динамическими базами данных в оперативной памяти. Ассоциативный список можно рассматривать как отображение множества ключей на множество соответствующих им объектов.

Создание ассоциативного списка

Ассоциативный список формируется с помощью встроенной функции PAIRLIS. Формат вызова:

```
(PAIRLIS Keys Objects A_list).
```

Keys – список ключей, Objects – список соответствующих им объектов, A_list – а-список, в начало которого добавляются новые пары «ключ – объект». При вызове в качестве значения A_list либо задается NIL, либо предполагается, что A_list был сформирован ранее. Например:

```
(pairlis `(a b c) `(1 2 3) ())
     ==> ((c . 3) (b . 2) (a . 1))
```

Поиск элементов в ассоциативном списке

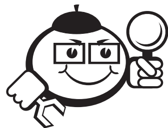
Ассоциативный список можно рассматривать как отображение множества ключей на множество соответствующих им объектов. Конкретные данные можно получить по значению ключа с помощью функции:

```
(ASSOC  Key  A_list).
```

В качестве значения функция возвращает пару «ключ – объект».

Пример поиска:

```
(setq X (pairlis `(a b c) `(1 2 3) ()))
(assoc `b X)
     ==> (b . 2)
```



Пример 3.14

Совместное использование списка свойств и ассоциативного списка.

Создадим список свойств о Лене без указания ее зарплаты, но с указанием занимаемой должности:

```
(setf (get `lena `age) 28)
(setf (get `lena `profes) `юрист)
(setf (get `lena `children) `(ira jura petya))
(symbol-plist `lena) ==>
     (CHILDREN (IRA JURA PETYA) PROFES юрист AGE 28)
```

Пусть у нас имеется информация о зарплатах по штатному расписанию на предприятии, где работает Лена, представленная в виде ассоциативного списка:

```
(setq штаты (pairlis `(бухгалтер юрист менеджер)  `(70 90 80) ()))  ==> ((менеджер . 80) (юрист . 90) (бухгалтер . 70))
```

Если мы захотим узнать, какая у Лены зарплата, мы получим:

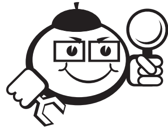
```
(cdr (assoc (get `lena `profes) штаты))
                              ==> 90
```

При изменении штатного расписания или окладов вопрос о зарплате останется прежним!

. . . .

Модификация ассоциативных списков

При необходимости мы можем модифицировать ассоциативный список, работая с ним так же, как и с обычным списком, используя функции CAR, CDR и CONS.



Пример 3.15

Построим функцию, позволяющую изменять значение заданного ключа. Обозначим:

x – имя ассоциативного списка,

k – ключ,

n – новое значение ключа.

```
(defun new_assoc (x k n)
   (cond ((null x) nil)
         ((eq (caar x) k) (cons (cons k n) (cdr x)))
         (t (cons (car x) (new_assoc (cdr x) k n)))))
```

Вернемся к примеру 3.14. Изменим штатное расписание, установив юристу новую зарплату:

```
(setq штаты (new_assoc штаты `юрист 110)) ==>
          ((менеджер . 80) (юрист . 110) (бухгалтер . 70))
```

Теперь, если мы захотим узнать, какая у Лены зарплата, мы получим:

```
(cdr (assoc (get `lena `profes) штаты)) ==> 110
```

. . . .

3.9 Работа с файлами

3.9.1 Определение входных и выходных потоков

При вводе и выводе информации в Лиспе используется понятие потоков – *stream*. Для потока определены ИМЯ, операции открытия *open*, операции закрытия *clouse*, направления *output* и *input*.

***Функция* OPEN**

(OPEN F [:DIRECTION :OUTPUT]). Обычная функция. Аргумент F должен быть строкой. Строка определяет имя файла. Если присутствует только первый параметр, то функция выдает в качестве значения локальный файл (значение используемое функциями ввода-вывода) ввода, связанный с внешним

файлом F. Если присутствуют оба параметра (второй ключевой – :DIRECTION :OUTPUT), то функция выдает локальный файл вывода, связанный файлом F. При этом функция открывает внешний файл с именем F.

Локальные файлы можно присвоить переменным. Имена переменных, значениями которых являются локальные файлы ввода, можно указывать в качестве параметров в обращениях к функциям READ, READ-CHAR и READ-LINE. Локальные файлы ввода могут быть дополнительными аргументами функций PRINT, PRINC, PRIN1 и TERPRI. В таких случаях в качестве файла ввода или вывода будет использоваться внешний файл, связанный с данным локальным файлом. Если в указанных функциях вместо локального файла указан идентификатор T, то соответствующая функция ввода или вывода будет работать с терминалом. Отметим, что при отсутствии такого параметра вывод будет осуществляться на стандартное устройство вывода, которое не обязательно является терминалом.

Для того чтобы при выполнении функции ввода определить, все ли содержимое файла прочитано, следует указать еще два параметра: NIL и некоторое значение, которое будет выдано функцией ввода после того, как все содержимое файла будет прочитано.

Например, откроем файл "sesame" для записи:

```
(setq our-output-stream (open "sesame" :direction :output))
```

Зададим

```
(setq s 'e)
```

Можно вывести это значение в файл:

```
(princ s our-output-stream)
```

Можно занести в файл список:

```
(print '(a b c d) our-output-stream)
```

Чтобы правильно закрыть поток, необходимо в конец поместить

```
(terpri our-output-stream)
```

Затем файл закрывается:

```
(close our-output-stream)
```

Можно посмотреть информацию в файле. Для этого откроем файл для чтения:

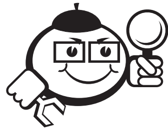
```
(setq our-input-stream (open "sesame" :direction :input))
```

Прочитаем информацию:

```
(read our-input-stream)
```

Закроем файл:

```
(close our-input-stream)
```



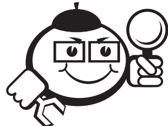
Пример 3.16

Выполнение следующего выражения приводит к выдаче выражений из файла с именем Q.LSP на терминал.

```
(LET ((Y (OPEN "Q.LSP")))
     (DO ((X (READ Y NIL '=EOF=) (READ Y NIL '=EOF=)))
     ((EQ X '=EOF=) NIL)
     (PRINT X T))
```

***Функция* CLOSE**

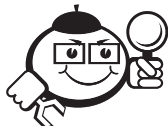
(CLOSE P). Обычная функция. Аргумент P должен быть локальным файлом. Функция закрывает внешний файл, соответствующий данному локальному файлу. После выполнения этой функции возобновить работу с внешним файлом можно только после того, как он будет открыт функцией OPEN. Рекомендуется после завершения работы с файлами закрывать их.



Пример 3.17

Рассмотрим определение функции EXEC, которая выполняет программу из файла, имя которого определяется строкой, являющейся аргументом функции. Считается, что среди выражений файла нет пустых списков.

```
(DEFUN EXEC (FF)
     (LET ((YY (OPEN FF)))
          (DO ((XX  (READ YY NIL NIL) (READ YY NIL NIL)))
              ((NULL XX) (CLOSE YY))
          (EVAL XX))))
```



Пример 3.18

Определим функцию (COPY A B), которая копирует содержимое текстового файла A в файл B.

```
(DEFUN COPY (A B)
     (SETQ A (OPEN A))
     (SETQ B (OPEN B :DIRECTION :OUTPUT))
     (DO ((X (READ-LINE A NIL NIL) (READ-LINE A NIL NIL)))
         ((NULL X) (CLOSE A) (CLOSE B))
         (PRINC X B) (TERPRI B)))
```

3.9.2 Чтение символов из файла

Предположим, что в файле хранится символьная информация, необходимая нам для обработки. Причем нас интересует каждый символ в файле. До сих пор мы могли вводить только атомы, числа и списки. Сформируем файл.

Пусть

```
(setq  s  "---+++")
(setq  p  "+++---")
```

Определим поток вывода:

```
(setq our-output-stream (open "picture.spl" :direction
:output))
```

`(princ s our-output-stream)` ; записываем первую строку

`(terpri our-output-stream)` ; заканчиваем ее

`(princ p our-output-stream)` ; записываем вторую строку

`(terpri our-output-stream)` ; заканчиваем файл

Теперь файл закрывается:

```
(close our-output-stream)
```

В файле теперь находится следующая информация:

```
---+++
+++---
```

Для чтения символов из файла будем использовать функцию

(READ-CHAR <входной поток>).

Данная функция позволяет читать печатные символы (CHAR) из файла. В качестве значения получается десятичное представление кода символа. Используем эту функцию для посимвольного ввода информации из файла для ее последующего анализа. Определим

```
(setq our-input-stream (open "picture.spl" :direction
:input))
```

Для чтения символа используем

```
(read-char our-input-stream)
```

Будем получать последовательность значений:

```
43
43
43
45
45
45
```

10 и т. д.

Для восстановления содержимого файла применяется перекодировка:

```
(setq x (read-char our-input-stream)
```

Содержимое x можно показать:

```
(cond (( = x 43) (prin1'+))
     (( = x 45) (prin1 '-))
     (( = x 10 ) (terpri)))
```

Можно представить информацию без искажений, если использовать цикл:

```
(loop (progn  (setq x  (read-char our-input-stream) )
      (cond (( = x 43) (prin1'+))
          (( = x 45) (prin1 '-))
          (( = x 10 ) (terpri))))))
```

После вывода имеем:

```
---+++
+++---
```

Закрытие входного потока:

```
(close our-input-stream)
```

Для поиска конца файла можно анализировать ошибку чтения, но лучше знать длину файла до начала работы с ним.

Контрольные вопросы по главе 3

1. Какова область действия локальных переменных?
2. Как определяется доступ к глобальным переменным?
3. Для чего используются разрушающие функции?
4. Что такое функционал?
5. Чем различаются применяющий и отображающий функционалы?
6. Чем отличаются циклические предложения от рекурсии?
7. Какие массивы поддерживаются в Лиспе?
8. Для чего используются списки свойств?
9. Чем отличается ассоциативный список от обычного?
10. Какие функции используются для работы с файлами?

4 Модели представления знаний

4.1 Фреймы

4.1.1 Понятие и состав фрейма

Фреймовая модель представления знаний была предложена М. Минским в 1979 г. как структура знаний для восприятия пространственных сцен [6].

В психологии и философии известно понятие абстрактного образа. Например, слово «комната» вызывает у слушающих образ комнаты: «жилое помещение с четырьмя стенами, полом, потолком, окнами и дверью, площадью 6-60 м^2». Из этого описания ничего нельзя убрать (например, убрав окна, получим уже чулан, а не комнату), но в нем есть «дырки», или «слоты», – это незаполненные значения некоторых атрибутов – количество окон, цвет стен, высота потолка, покрытие пола и др.

В теории такой абстрактный образ называется фреймом.

Фреймом называется также и формализованная модель для отображения образа.

В качестве идентификатора фрейму присваивается имя фрейма. Это имя должно быть единственным во всей фреймовой системе.

Модель фрейма является достаточно универсальной, поскольку позволяет отобразить все многообразие знаний о мире [6]:

- через фреймы-структуры, для обозначения объектов и понятий (заем, залог, вексель);
- через фреймы-роли (менеджер, кассир, клиент);
- через фреймы-сценарии (банкротство, собрание акционеров, празднование именин);
- через фреймы-ситуации (тревога, авария, рабочий режим) и др.

Фрейм имеет определенную внутреннюю структуру, состоящую из множества элементов, называемых *слотами*, которым также присваиваются имена. Каждый слот в свою очередь представляется определенной структурой данных. В значение слота подставляется конкретная информация, относящаяся к объекту, описываемому этим фреймом.

Структуру фрейма можно представить так:

```
ИМЯ ФРЕЙМА:
(имя 1-го слота: значение 1-го слота),
```

```
(имя 2-го слота: значение 2-го слота),
- - - -
(имя N-го слота: значение N-го слота).
```

Состав слота:

- имя слота – это идентификатор, присваиваемый слоту. Слот должен иметь уникальное имя во фрейме, к которому он принадлежит. Обычно не несет никакой смысловой нагрузки и является лишь идентификатором данного слота, но в некоторых случаях оно может иметь специфический смысл;
- указатель наследования (для фреймовых моделей иерархического типа) показывает, какую информацию об атрибутах слотов во фрейме верхнего уровня наследуют слоты с такими же именами во фрейме нижнего уровня;
- указатель атрибутов – указатель типа данных слота: указатель, целое, вещественное, присоединенная процедура, текст, список и другие;
- значение слота – значение, соответствующее типу данных слота и удовлетворяющее условиям наследования;
- демон – процедура, автоматически запускаемая при выполнении некоторого условия. Демоны запускаются при обращении к конкретному слоту фреймовой модели. Например, демон запускается, если в момент обращения к слоту его значение не было установлено, при подстановке в слот значения, при стирании значения слота.

Важнейшим свойством теории фреймов является наследование свойств.

Во фреймах наследование происходит по АКО-связям (A-Kind-Of = это). Слот АКО указывает на фрейм более высокого уровня иерархии, откуда неявно наследуются, то есть переносятся, значения аналогичных слотов.

Значением слота может быть практически что угодно:

- числа;
- формулы;
- тексты на естественном языке или программы;
- правила вывода;
- ссылки на другие слоты данного фрейма или других фреймов;
- набор слотов более низкого уровня, что позволяет реализовывать во фреймовых представлениях «принцип матрешки».

Связи между фреймами задаются значениями специального слота с именем «Связь».

Виды фреймов [5]:

- фрейм-экземпляр – конкретная реализация фрейма, описывающая текущее состояние в предметной области;
- фрейм-образец – шаблон для описания объектов или допустимых ситуаций предметной области;
- фрейм-класс – фрейм верхнего уровня для представления совокупности фреймов-образцов.

Фреймы образуют иерархию. Иерархия во фреймовых моделях порождает единую многоуровневую структуру, описывающую либо объект, если слоты описывают только свойства объекта, либо ситуацию или процесс, если отдельные слоты являются именами процедур, присоединенных к фрейму и вызываемых при его актуализации.

Характеристики фрейма в иерархической структуре

Состав фреймов и слотов в каждой конкретной фреймовой модели может быть разным, однако в рамках одной системы целесообразно единое представление для устранения лишнего усложнения:

- имя фрейма – символ. Имя фрейма должно быть уникальным в данной фреймовой системе;
- положение в иерархической структуре задается указателями на родительский фрейм и список дочерних фреймов;
- информация, относящаяся к фрейму содержится в слотах. Каждый слот может представляться как атомом, так и списочной структурой, первый элемент которой выполняет функцию ключа (соответствует имени слота);
- присоединенные процедуры – служебные программы, являющиеся значениями слотов и запускаемые по сообщениям из других фреймов (аналоги методов в ООП).

Примечание. Фрейм наследует все свойства предка!

4.1.2 Пример построения фреймовой структуры на Лиспе

Рассмотрим в качестве примера фреймовой структуры классификацию собак, представленную на рисунке 4.1.

Для описания приведенной классификации определим следующую структуру фрейма (табл. 4.1).

Рис. 4.1 – Классификация пород собак

Таблица 4.1 – Структура фрейма «Собака»

Свойство	**Значение**
`frm_name`	Имя фрейма
`father`	Имя родительского фрейма
`info`	Информация (слоты)
`child_list`	Список дочерних фреймов

По сути, в этой структуре отражена иерархия объектов через указание предка (отца) и потомков (детей) для каждого объекта. Частная информация для каждого фрейма хранится в слоте `info`.

Для описания фреймов в Лиспе будем использовать списки свойств.

Опишем положение фрейма в структуре.

Функция **put_frm** описывает положение фрейма в структуре:

```
(defun put_frm (frame father slots_info children)
 (setf (get frame `frm_name) frame)
 (setf (get frame `father) father)
 (setf (get frame `info) slots_info)
 (setf (get frame `child_list) children)
 (setf (get father `child_list)
     (put_to_list  frame (get  father `child_list))) ;связь с
     отцом
 (cond ((not (null children))
     (put_frm_child frame children))) ;создание дочерних фрей-
мов
 )
```

Функция **put_to_list** включает объект в список, если он там отсутствует:

```
(defun put_to_list (X Y)
 (cond ((not (member X Y)) (cons X Y))
      (t Y)))
```

Функция **put_frm_child** обрабатывает список дочерних фреймов (предварительно создаются фреймы с пустой информацией по слотам, указывается только связь с отцовским фреймом):

```
(defun put_frm_child (frame children)
 (setf (get (car children) `frm_name) (car children))
 (setf (get (car children) `father) frame)
 (setf (get (car children) `info) nil)
 (setf (get (car children) `child_list) nil)
 (cond ((not (null (cdr children))) (put_frm_child frame (cdr
children)))))
```

Теперь определим функцию добавления фрейма в существующую иерархию:

```
Аргументы: fr_struct - фреймовая структура
           new_fr - имя добавляемого фрейма
           slots_info - слоты добавляемого фрейма
           children - список дочерних фреймов добавляемого
фрейма
           father - имя родительского фрейма
(defun add_frm (fr_struct new_fr slots_info children father)
 (cond ((and (not (equal (get fr_struct `frm_name) father))
```

```
        (null (get  fr_struct `child_list)))
           nil) ;возвращает NIL, если некуда вставить!
 ((equal (get fr_struct `frm_name) father) ; если нашли отца -
      (put_frm new_fr father slots_info children)) ; вставляем
 (t            ;иначе спускаемся ниже - в доч. фреймы и снова
               ;проверяем - не отец ли это нов. фр
      (add_frm_child (get fr_struct `child_list)
                          new_fr slots_info children father))))
```

Функция обработки списка дочерних фреймов при добавлении нового фрейма в существующую иерархию:

```
(defun add_frm_child (children new_fr slots child_newfr father)
  (cond ((null children) nil)
     ((not (add_frm (car children) new_fr slots child_newfr father))
          (add_frm_child (cdr children) new_fr slots child_newfr father))
     (t  (add_frm (car children) new_fr slots child_newfr father))))
```

Построим теперь нашу фреймовую структуру «Собака» с использованием написанных функций:

Имя фреймовой структуры – dog.

Создание фрейма «Собака»:

```
(put_frm `dog nil `(it_is_a_dog) `(short-haired long-haired))
```

Создание фреймов «Короткошерстная» и «Длинношерстная»:

```
(add_frm `dog `shot-haired `(short-haired) `(english_bulldog beagle great_dane american_foxhound) `dog)
(add_frm `dog `long-haired `(long-haired) `(cocker_spanial irish_setter collie st_bernard) `dog)
```

Для короткошерстных пород:

```
(add_frm `dog `english_buldog
         `(height_under_22_inches    low-set_tail
           good_natural_personality)
     nil `short-haired)
(add_frm `dog `beagle
         `(height_under_22_inches    longer_ears
           good_natural_personality)
     nil `short-haired)
(add_frm `dog `american_foxhound
         `(height_under_30_inches
           longer_ears    good_natural_personality)
     nil `short-haired)
```

```
(add_frm `dog `great_dane
        `(low-set_tail    longer_ears
          weight_over_100_f      good_natural_personality)
    nil `short-haired)
```

Для длинношерстных пород:

```
(add_frm `dog `cocker_spaniel
        `(height_under_22_inches
          low-set_tail   longer_ears
          good_natural_personality)
    nil `long-haired)
(add_frm `dog `irish_setter
        `(height_under_30_inches   longer_ears)
    nil `long-haired)
(add_frm `dog `collie
        `(height_under_30_inches   low-set_tail
          good_natural_personality)
    nil `long-haired)
(add_frm `dog `st_bernard
        `(low-set_tail   good_natural_personality
           weight_over_100_f)
    nil `long-haired)
```

Рассмотрим теперь поиск объекта с заданными свойствами в созданной фреймовой структуре (функция **what_is_it**):

```
Аргументы: fr_struct - фреймовая структура
           descr - описание объекта в виде перечня свойств
(defun what_is_it (fr_struct  descr)
   (cond   ((and   (null   descr)   (null   (get   fr_struct
`child_list)))
              (get fr_struct `frm_name))  ;нашли!
         ((equal (get fr_struct `info) descr)
              (get fr_struct `frm_name));нашли!
         ((not (null (compare_slots (get fr_struct `info) de-
    scr))) ;спускаемся
              (search_childs (get fr_struct `child_list)
                   (compare_slots(get  fr_struct  `info)  de-
              scr)))))
```

Функция **search_childs** просмотра информации дочерних фреймов:

```
(defun search_childs (fr_l descr)
   (cond ((null fr_l) nil)
     ((equal (what_is_it (car fr_l) descr) nil) (search_childs
   (cdr fr_l) descr))
     (t (what_is_it (car fr_l) descr))))
```

Функция **compare_slots** определяет, является ли список inf_fr подсписком списка inf_search:

```
(defun compare_slots (inf_fr inf_search)
   (cond ((null inf_fr) inf_search)
         ((equal (car inf_fr)(car inf_search))
              (compare_slots (cdr inf_fr)(cdr inf_search)))
         (t nil)))
```

Вывод экспертного заключения

В предложенной реализации вывода экспертного заключения перечень свойств искомого объекта соответствует утвердительным ответам на вопросы экспертной системы, касающиеся наличия у объекта свойств, представленных в слотах фреймовой структуры.

```
(what_is_it  `dog  `(it_is_a_dog
                     long-haired
                     height_under_30_inches
                     low-set_tail
                     good_natural_personality))
```

В качестве ответа будет выдано название породы собаки, которую характеризуют представленные в списке свойства, то есть **COLLIE**.

Давайте теперь попробуем модернизировать нашу фреймовую структуру. Добавим к существующим породам собак экземпляры класса – конкретных собак, относящихся к данной породе.

Кроме этого, к свойствам пород добавим функциональное свойство «экстерьер», которое в дальнейшем опишем в виде функции и будем использовать для определения качества экстерьера рассматриваемой собаки.

Для простоты возьмем только две породы из рассмотренных ранее и по два экземпляра к каждой породе (рис. 4.2).

Для удобства дальнейшей работы и анализа содержимого слотов построим структуру слота INFO следующим образом (для конкретного фрейма некоторые свойства могут отсутствовать) (табл. 4.2).

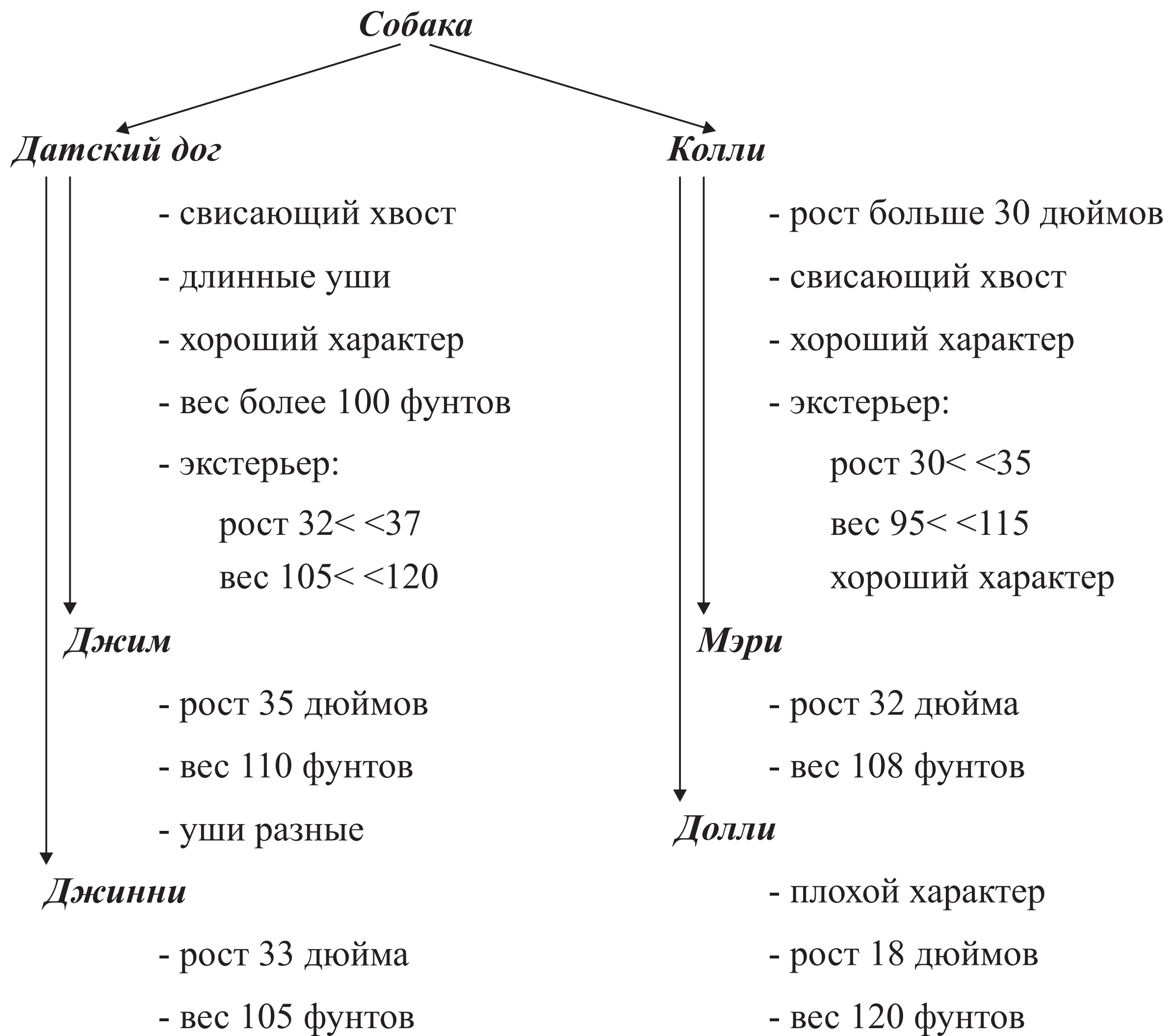


Рис. 4.2 – Расширенная классификация собак

Таблица 4.2 – Структура слота `INFO`

Свойство	Значение	Тип данных
wool	Длина шерсти	Симв. атом
height	Рост	Число/список
weight	Вес	Число/список
character	Характер	Симв. атом
tail	Хвост	Симв. атом
ears	Уши	Симв. атом
exterior	Проверка параметров	Функция

Добавление фрейма в существующую иерархию (вариант 2).

```
(defun put_frm (frame father slots_info)
 (setf (get frame `frm_name) frame)
```

```
 (setf (get frame `father) father)
 (setf (get frame `info) slots_info)
 (setf (get frame `child_list) ())) ;первоначально дочерний
список пуст

(defun add_frm (fr_str new_fr father info)
 (cond ((and (not (equal (get fr_str `frm_name) father))
        (null(get fr_str `child_list)))
                  (print "предок не найден") nil)
     ((equal (get fr_str `frm_name) father)
                  (put_frm new_fr father info)
        ;добавляемый фрейм включается в список детей к отцу
                  (setf (get father `child_list)
                  (cons new_fr (get father `child_list))))
       (t  (add_frm_child (get fr_str `child_list) new_fr fa-
ther info))))

(defun add_frm_child (children new_fr father slots)
 (cond ((null children) (print "отец не найден") nil)
     ((not (add_frm (car children) new_fr father slots))
     (add_frm_child (cdr children) new_fr father slots))
     (t (add_frm (car children) new_fr father slots))))
```

Определим слот `info` через ассоциативные списки. Теперь мы можем написать функцию, которая будет добавлять новый фрейм во фреймовую структуру. Для ввода новых данных организуем диалог с пользователем.

Учитывая, что слот `info` может содержать не все свойства, а только некоторые из них, для ввода информации используем цикл `loop`.

```
(defun made_frm (y)
  (print "введите имя нового фрейма")
 (setq name (read))
 (print "введите отца")  (setq father (read)) (setq y nil)
(loop (print "для ввода информации по фрейму выберите необхо-
димый параметр (введите цифру):
1 шерсть
2 рост
3 вес
4 характер
5 хвост
6 уши
7 функция для определения экстерьера
8 окончить ввод")
```

```
  (setq x (read))
  (cond ((eq x 1)  (setq y (pairlis `(wool) (list (read)) y)))
    ((eq x 2)  (setq y (pairlis `(height) (list (read)) y)))
    ((eq x 3)  (setq y (pairlis `(weight) (list (read)) y)))
    ((eq  x  4)   (setq  y  (pairlis  `(character)  (list  (read))
y)))
    ((eq x 5)  (setq y (pairlis `(tail) (list (read)) y)))
    ((eq x 6)  (setq y (pairlis `(ears) (list (read)) y)))
    ((eq x 7)  (setq y (pairlis `(exterior) (list (read)) y)))
    ((eq x 8)  (return y))))
(add_frm father name father y))
```

Построим нашу фреймовую структуру:

```
(put_frm `dog () ())
(add_frm `dog `short_haired `dog `((wool . short)))
(add_frm `dog `long_haired `dog `((wool . long)))

(add_frm    `dog    `dog_dane    `short_haired    `((tail    .
свисающий)(ears    .    long)(character    .    good)(weight    .    (>
100))(exterior . ex1)))
(add_frm  `dog  `djim  `dog_dane  `((height  .  35)(weight  .
110)(ears . different)))
(add_frm  `dog  `djinny  `dog_dane  `((height  .  33)(weight  .
105)))
```

Для проверки экстерьера собаки построим функцию **ex1**. Если выполняются все условия экстерьера ОДНОВРЕМЕННО, то экстерьер ГОДЕН:

```
(defun ex1 (frame)
(cond ((and (>= (cdr(assoc `height (get frame `info)))32)
            (<=  (cdr(assoc `height (get frame `info)))37)
            (>= (cdr(assoc `weight (get frame `info))) 105)
            (<= (cdr(assoc `weight (get frame `info)))120)
            (cond ((member_as `character (get frame `info))
                    (eq  (cdr(assoc  `character  (get  frame  `in-
               fo)))  `good))
                         (t t))
            (cond ((member_as `tail (get frame `info))
                    (eq  (cdr(assoc  `tail  (get  frame  `info)))
               `свисающий))
                         (t t))
            (cond ((member_as `ears (get frame `info))
                    (eq  (cdr(assoc  `ears  (get  frame  `info)))
               `long))
                         (t t))
```

```
         )
             (print "экстерьер годен"))
        (t (print "экстерьер не годен"))))
```

Функция member_as используется для определения, содержится ли заданное свойство в списке свойств конкретной собаки:

```
(defun member_as (x y)
  (cond ((null y) nil)
     ((eq x (caar y))t)
     (t (member_as x (cdr y)))))
```

Мы можем проверить наших собак на годность экстерьера:

```
(defun proverka (frame y1)
   (cond ((null frame) (print "не найдена функция для опреде-
ления экстерьера"))
         ((member_as `exterior (get frame `info))
              (funcall (cdr(assoc `exterior (get frame `in-
         fo))) y1))
         (t (proverka (get frame `father) y1))))
```

Так, для собаки Джинни получаем:

```
CL-USER 18 : 8 > (proverka `djinny `djinny)
"экстерьер годен"
```

Для собаки Джим получим:

```
CL-USER 19 : 8 > (proverka `djim `djim)
"экстерьер не годен"
==> Этого и следовало ожидать, так как у Джима «разные уши»,
то проверку на экстерьер он не проходит.
```

4.2 Семантические сети

4.2.1 Представление знаний с помощью семантических сетей

Для представления знаний в искусственном интеллекте используются различные модели. Одной из часто применяемых моделей является конструкция, названная семантической сетью. Семантический подход к построению систем искусственного интеллекта находит применение в системах понимания естественного языка, в вопросно-ответных системах, в различных предметно-ориентированных системах.

Семантические сети широко используются в экспертных системах в качестве языка представления знаний, в системах распознавания речи и понимания естественного языка.

Термин «семантическая» означает смысловая, а семантика – это наука, устанавливающая отношения между символами и объектами, которые они обозначают, то есть наука, определяющая смысл знаков.

В самом общем случае *семантическая сеть* представляет собой информационную модель предметной области и имеет вид графа, вершины которого соответствуют *объектам* предметной области, а дуги – *отношениям* между ними [4].

По сути, вершины в семантической сети отражают понятия базы знаний. В качестве объектов могут выступать понятия, события, свойства, процессы. Объекты могут быть как абстрактными (любовь, дружба), так и конкретными (огурец, машина, Маша).

Дуги могут быть определены разными методами, зависящими от вида представляемых знаний. Обычно дуги, используемые для представления иерархии, включают дуги типа «множество», «подмножество», «элемент». Семантические сети, применяемые для описания естественных языков, используют дуги типа «агент», «объект», «реципиент».

Классификация семантических сетей

Для всех семантических сетей справедливо разделение по арности и количеству типов отношений [4].

По *количеству типов отношений* сети могут быть *однородными* и *неоднородными*:

- *однородные сети* обладают только одним типом отношений;
- в *неоднородных сетях* количество типов отношений больше двух. Классические иллюстрации данной модели репрезентации знаний демонстрируют именно такие сети. Неоднородные сети представляют, с одной стороны, больший интерес для практических целей, но, с другой стороны, большую сложность для исследования. Неоднородные сети можно представлять как переплетение древовидных многослойных структур. Примером такой сети может быть семантическая сеть Википедии.

По *арности* семантические сети подразделяются:

- на *сети с бинарными отношениями* – сети, связывающие ровно два понятия. Бинарные отношения очень просты и удобно изображаются на графе в виде стрелки между двумя объектами. Кроме того, они играют исключительную роль в математике;

- *N-арные сети* – сети, имеющие отношения, связывающие более двух понятий. При этом возникает сложность – как изобразить подобную связь на графе, чтобы не запутаться. Концептуальные графы снимают это затруднение, представляя каждое отношение в виде отдельного узла.

По *назначению* семантические сети подразделяются:

- на *сети для решения конкретных* задач, например тех, которые решают системы искусственного интеллекта;
- *семантические сети отраслевого масштаба*. Такая сеть должна служить базой для создания конкретных систем, не претендуя на всеобщее значение;
- *глобальные семантические сети*. Теоретически такая сеть должна существовать, поскольку все в мире взаимосвязано. Возможно, когда-нибудь такой сетью станет Всемирная паутина.

Отношения в семантических сетях

Отношения – это связи типа «это», «имеет частью», «принадлежит», «любит». Количество типов отношений в семантической сети определяется ее создателем, исходя из конкретных целей. В реальном мире их число стремится к бесконечности. Каждое отношение является, по сути, предикатом, простым или составным. Скорость работы с базой знаний зависит от того, насколько эффективно реализованы программы обработки нужных отношений.

Наиболее часто возникает потребность в описании отношений между элементами, множествами и частями объектов.

Отношение между объектом и множеством, обозначающим, что объект принадлежит этому множеству, называется *отношением классификации* (*ISA*). Говорят, что множество (класс) классифицирует свои экземпляры («Шарик является собакой» = Шарик является объектом типа «собака»). Иногда это отношение именуют также MemberOf, InstanceOf или подобным образом. Связь ISA предполагает, что свойства объекта наследуются от множества.

Обратное к ISA отношение используется для обозначения примеров, поэтому так и называется – «Example», или, по-русски, «Пример».

Иерархические отношения образуют древовидную структуру, в связи с чем часто используются отношения «*A Kind Of*» и «*HasPart*»:

- *отношение между надмножеством и подмножеством* называется AKO – «A Kind Of», «разновидность» («собака является животным» =

тип с именем собака является подтипом типа «животные»). Элемент подмножества называется гипонимом (собака), а надмножества – гиперонимом (животное), а само отношение называется отношением гипонимии. Альтернативные названия – «SubsetOf» и «Подмножество». Это отношение определяет, что каждый элемент первого множества входит и во второе (выполняется ISA для каждого элемента). Также оно определяет логическую связь между самими подмножествами: первое не больше второго, свойства первого множества наследуются вторым. Отношение АКО (род – вид) часто используется для навигации в информационном пространстве;

- *отношение меронимии* (*HasPart*) описывает связь частей и целого. Объект, как правило, состоит из нескольких частей или элементов. Например, компьютер состоит из системного блока, монитора, клавиатуры, мыши и т. д. В этом случае свойства первого множества не наследуются вторым. Мероним и холоним – противоположные понятия.

  Мероним – объект, являющийся частью для другого (двигатель – мероним автомобиля).

  Холоним – объект, который включает в себя другое (например, у дома есть крыша; дом – холоним крыши; компьютер – холоним монитора).

В семантических сетях также часто используются следующие отношения:

- функциональные связи (определяемые обычно глаголами «производит», «влияет»...);
- количественные (больше, меньше, равно...);
- пространственные (далеко от, близко от, за, под, над...);
- временные (раньше, позже, в течение...);
- атрибутивные связи (иметь свойство, иметь значение...);
- логические связи (и, или, не) и др.

Рассмотрим некоторые примеры представления знаний с помощью семантических сетей. На рисунке 4.3 представлена семантическая сеть для понятия «корабль». Здесь рассматриваются отношения типа: «Куин Мэри» является океанским лайнером», «Каждый океанский лайнер является кораблем» и т. п.

На следующем рисунке 4.4 приведены примеры еще двух простых семантических сетей. Одна из них описывает понятие «помидор», а другая описывает факт «Маша укрепила стул клеем».

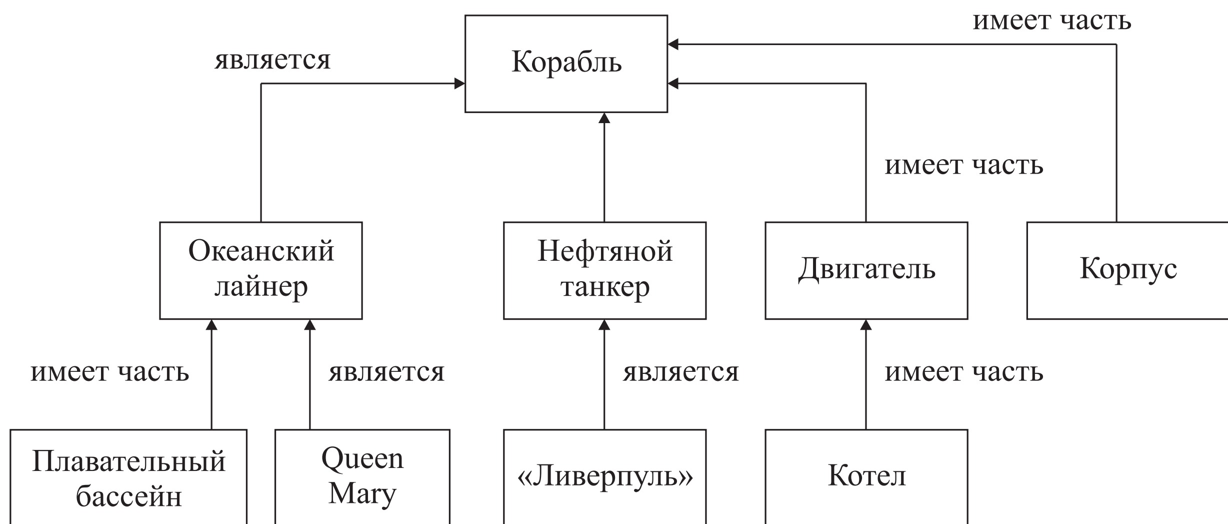


Рис. 4.3 – Семантическая сеть для понятия «корабль»

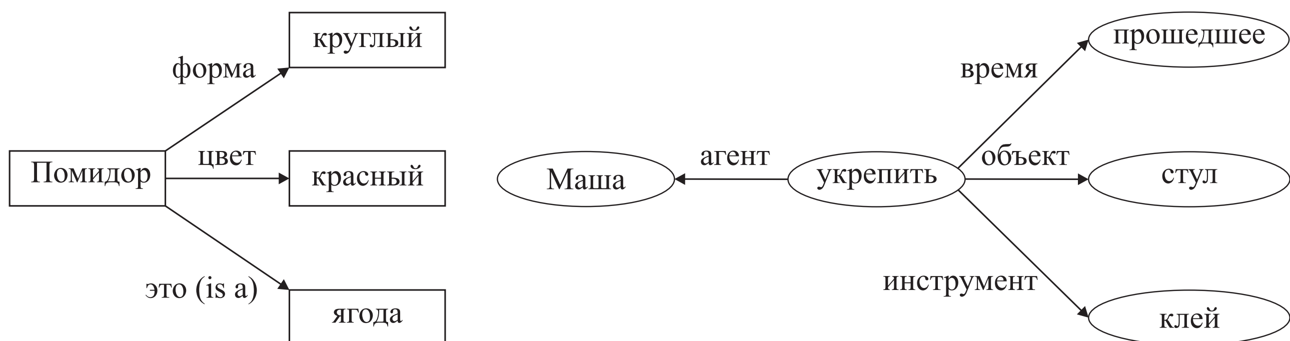


Рис. 4.4 – Примеры семантических сетей

В разных вариациях семантических сетей для отображения понятий используются различные геометрические примитивы: прямоугольники, овалы, прямоугольники со скругленными углами и т. п.

Проблема поиска решения в семантической сети сводится к задаче поиска фрагмента сети, соответствующего поставленному запросу. Например, вопрос «Какого цвета помидор?» можно графически представить в виде подсети (рис. 4.5).

Рис. 4.5 – Представление вопроса в виде подсети

Наложение подсети вопроса на сеть, описывающую предметную область, дает ответ – «красный».

Проблема поиска решения в базе знаний типа семантической сети сводится к задаче поиска фрагмента сети, соответствующего некоторой подсети, соответствующей поставленному вопросу.

4.2.2 Пример построения семантической сети с помощью Лиспа

В качестве иллюстрации рассмотрим следующий пример семантической сети (рис. 4.6). В сети описана птица канарейка, которая является животным, умеет летать, дышать, петь и т. п.

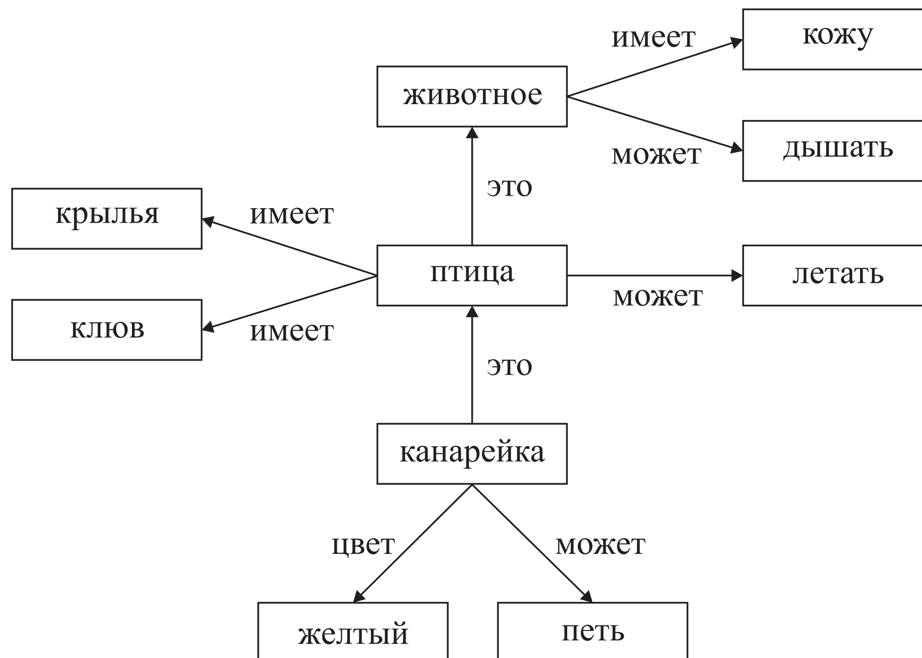


Рис. 4.6 – Семантическая сеть «канарейка»

Описание объекта в узле сети

Функция `LINK` описывает объект в сети, связывая атрибуты и их значения с помощью ассоциативного списка. Введем следующие обозначения:

`Y` – узел семантической сети

`attrs` – список атрибутов

`vals` – список значений

Функция будет выглядеть следующим образом:

```
(defun link (Y attrs vals)
 (cond ((and (not (atom attrs)) (not (atom vals))
                 (equal (length attrs)(length vals)))
       (setf (get Y `info) (pairlis attrs vals ())))
       (t nil)))
```

Для описания канарейки будем использовать следующие отношения:

- `is-a` – это (отношение классификации);
- `may` – может (функциональная связь);
- `have` – имеет (атрибутивная связь);
- `color` – цвет (атрибутивная связь).

Тогда рассматриваемая сеть будет строиться следующим образом:

```
(link `animal `(may have) `(breathe skeen))
(link `bird `(is-a may have) `(animal fly (wings beak)))
(link `canary `(is-a color may) `(bird yellow sing))
```

Вывод в семантической сети определяется с помощью использующих ее процедур. Наиболее типичный способ вывода основан на сопоставлении частей сетевой структуры, которое может быть определено рекурсивно. При этом вывод, как правило, происходит в три этапа:

- выделение составных частей запроса (лексем) и определение их семантической ориентации по словарю;
- построение семантической сети запроса;
- сопоставление семантических сетей запроса и области знаний.

Построение семантической сети запроса можно заменить поиском отношений между выделенными из запроса концептуальными объектами по семантической сети области знаний, поскольку сеть запроса строится на основе сети области знаний.

Поиск в семантической сети.

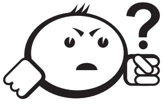
`Relation` – рассматриваемое отношение между выделенными понятиями `conc1` и `conc2`

```
(defun search1 (conc1 conc2 relation)
 (cond ((equal conc2 (cdr (assoc relation (get conc1 `in-
fo)))) t)
        ((and (not(atom(cdr (assoc relation (get conc1 `in-
        fo)))))
             (member conc2 (cdr (assoc relation (get conc1
        `info))))) t)
        ((null (assoc `is-a (get conc1 `info))) nil)
        (t
             (search1 (cdr(assoc `is-a (get conc1 `info)))
        conc2 relation))))
```

Пример вывода для вопроса:

```
«Может ли канарейка дышать?»
(seach1 `canary `breathe `may)
```

Результат: `T` (да).



Контрольные вопросы по главе 4

1. Что такое фрейм?
2. Что может содержать слот?
3. Какие типы фреймов Вы знаете?
4. Как можно представить слоты в Лиспе?
5. Что такое семантическая сеть?
6. Какие типы семантических сетей Вы знаете?

Заключение

В данном учебном пособии рассмотрены основы функционального программирования на примере языка Лисп и его принципиальное отличие от алгоритмических языков. Основными методами программирования являются суперпозиция функций и рекурсия, что позволяет писать более короткие и простые программы.

При описании языка автор попытался как можно полнее раскрыть его возможности и особенности на многочисленных примерах. Тем не менее, данный язык имеет намного больше возможностей и средств, в том числе и для моделирования работы интеллектуальных систем, чем это было представлено в учебном пособии. Задачей автора было показать подход построения моделей интеллектуальных систем средствами, отличными от алгоритмических языков, а также преимущества и удобства такого подхода.

Автор надеется, что изложенный в пособии материал позволит читателю применять свои знания на практике.

Литература

1. Кубенский А. А. Функциональное программирование : учеб. пособие [Электронный ресурс] / А. А. Кубенский. – СПб. : СПбНИУ ИТМО, 2010. – 251 с. – Режим доступа: http://e.lanbook.com/books/element.php?pl1_id=40771 (дата обращения: 03.08.2016).
2. Роганова Н. А. Функциональное программирование : учеб. пособие для вузов / Н. А. Роганова ; Федеральное агентство по образованию, Московский государственный индустриальный университет. – М. : МГИУ, 2007. – 214 с.
3. Зюзьков В. М. Функциональное программирование : учеб. пособие / В. М. Зюзьков ; Федеральное агентство по образованию, Томский государственный университет систем управления и радиоэлектроники, Кафедра автоматизированных систем управления. – Томск : ТМЦДО, 2005. – 140 с.
4. Осуга С. Обработка знаний / С. Осуга. – М. : Мир, 1989. – 293 с.
5. Уэно Х. Представление и использование знаний / Х. Уэно, М. Исидзука. – М., 1989. – 220 с.
6. Минский М. Фреймы для представления знаний : пер. с англ. / М. Минский. – М. : Энергия, 1979. – 152 с.

Глоссарий

S-выражение – это основная структура данных в Лиспе, которая может быть представлено либо атомом, либо списком.

Аргумент – входной параметр функции. Аргументом обычной функции является значение фактического параметра, а аргументом особой функции – сам фактический параметр.

Ассоциативный список, или *а-список (a-list)*, – основанная на списках и точечных парах структура данных, описывающая связи наборов данных и ключевых полей. Ассоциативные списки применяются при работе с динамическими базами данных в оперативной памяти. Ассоциативный список можно рассматривать как отображение множества ключей на множество соответствующих им объектов.

Атомы – это простейшие объекты Лиспа, из которых строятся остальные структуры. Атомы бывают двух типов: символьные и числовые. Символьный атом рассматривается как неделимое целое.

Вычислимые выражения – выражения, имеющие значения. Значения вычислимых выражений могут быть получены в процессе вычислений.

Локальные переменные – переменные, которые имеют значение только в теле функции, где они были объявлены. После завершения работы функции локальные переменные теряют свои значения.

Лямбда-выражение – анонимная (безымянная) функция, которая используется для явного задания определения функции в обращении к функции.

Массив – это набор переменных (ячеек), имеющих одно имя, но разные номера, обеспечивающие доступ к этим ячейкам.

Обычные функции – функции, при выполнении которых вначале вычисляются фактические параметры, а затем выполняются действия, предписанные данными функциями.

Особые функции – функции, при выполнении которых вычисляются лишь некоторые фактические параметры либо параметры вовсе не вычисляются.

Отношение классификации (ISA) – отношение между объектом и множеством, обозначающим, что объект принадлежит этому множеству.

Отображающий функционал – функционал, который применяет функциональный аргумент к элементам списка, в результате чего строится новый список: исходный список отображается на результирующий.

Предикаты – функции, проверяющие наличие некоторых свойств аргументов. Значением предиката является «Истина» или «Ложь».

Применяющий функционал – функционал, который применяет функциональный аргумент к остальным аргументам.

Пустой список – список, не имеющий ни одного элемента. В Лиспе пустой список является одновременно и атомом, и списком.

Разрушающая функция – функция, которая изменяет структуру списков.

Рекурсивная ветвь – ветвь вычислений в рекурсивной функции, содержащая вызов этой функции с упрощенным аргументом.

Рекурсивная функция – функция, во время выполнения которой может встретиться обращение к этой функции. Если в теле функции находится обращение к ней самой, то такая рекурсия называется прямой.

Рекурсивный объект – объект, который содержит сам себя или определен с помощью самого себя.

Свободная переменная – переменная в теле функции, не входящая в число ее формальных параметров.

Семантическая сеть (смысловая сеть) – модель предметной области, представленная в виде графа, вершинами которого являются понятия, а дуги (ребра) – отношения между ними.

Слот – поименованный элемент фреймовой структуры. Каждый слот в свою очередь представляется определенной структурой данных. В значение слота подставляется конкретная информация, относящаяся к объекту, описываемому этим фреймом.

Список – это многоуровневая или иерархическая структура данных, в которой открывающиеся и закрывающиеся скобки находятся в строгом соответствии. Элементами списка могут быть как атомы, так и другие списки.

Список свойств – список специального вида, содержащий в качестве элементов попарно имена и значения свойств. Имя свойства может быть задано только нечисловым атомом, значение свойства может быть задано как атомом, так и списком. Существуют специальные функции языка, позволяющие выполнять различные действия с такими списками.

Строка – это последовательность знаков, заключенная в кавычки. Строка является атомом, но не может быть переменной. Как и у числа, значением строки является сама строка

Терминальная ветвь – завершающая ветвь вычислений в рекурсивной функции. Терминальная ветвь необходима для окончания вызова функции: воз-

вращает результат, который является базой для вычисления результатов рекурсивных вызовов.

Фрейм (англ. *frame* – рамка, каркас) – структура данных для представления некоторого концептуального объекта. Информация, относящаяся к фрейму, содержится в составляющих его слотах.

Фреймовая модель представления знаний – структура знаний для восприятия пространственных сцен

Функционал – функция, аргументами которой являются имена других функций, применяемых функционалом для обработки данных.

Функциональное программирование – программирование с помощью функций в математическом их понимании. Основано на следующей идее: в результате каждого действия возникает значение, которое может быть аргументом следующего действия.

Функциональный аргумент – аргумент, значением которого является функция.