Московский государственный университет
имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Е.И. Большакова, Н.В. Груздева

Основы программирования на языке Лисп

Учебное пособие

Москва – 2010

УДК
ББК

Рецензенты: доцент, к.ф.-м.н. В.В. Малышко
ст. научный сотрудник, к.ф.-м.н. Ю.С. Владимирова

Большакова Елена Игоревна,
Груздева Надежда Валерьевна

Основы программирования на языке Лисп: Учебное пособие. – М.: Издательский отдел факультета ВМК МГУ имени М.В.Ломоносова (лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2010 – 112 с.

В учебном пособии излагаются основные понятия и базовые механизмы функционального языка программирования Лисп, ориентированного на решение задач обработки символьных данных. Рассматриваются приемы рекурсивного программирования, в том числе с использованием функционалов. Подробно разбираются примеры лисп-программ. В пособие включено также описание заданий практикума по программированию на языке Лисп, проводимого для студентов факультета ВМК МГУ.

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

ISBN 978-5-89407-441-2

Содержание

Предисловие

Язык Лисп – один из старейших языков программирования и первый *функциональный язык*, получивший широкое распространение. Ядро языка было создано в 60-х годах прошлого века известным ученым Дж. Маккарти для решения задач обработки символьной информации [1, 2]. Основная структура данных языка Лисп – список, отсюда и название языка: *Lisp – List Processing*.

Пик популярности Лиспа пришёлся на 70-е и 80-е годы, когда он применялся как основной язык для научных исследований в области искусственного интеллекта. В эти годы было разработано несколько достаточно развитых диалектов языка, из которых наиболее известны InterLisp, MacLisp, Common Lisp. Одна из первых реализаций Лиспа в России была осуществлена для ЭВМ БЭСМ-6 [8]. На базе концепций Лиспа был создан ряд других функциональных языков, в частности – язык Planner [10] для задач искусственного интеллекта и язык Scheme [3].

Хотя язык Лисп существует и развивается давно, тем не менее, он не был стандартизован, и его диалекты различаются по ряду аспектов. Официальный стандарт разработан только для языка Common Lisp [4, 5, 16], возникшего в результате попытки объединить в одном языке средства нескольких диалектов Лиспа, а также языка Scheme. В настоящее время кроме нескольких реализаций языка Common Lisp используется диалект AutoLisp [12] известного пакета AutoCAD, а также достаточно старый, но удачный диалект MuLisp, частично описанный в [13].

Несмотря на то, что Лисп потерял главенствующую роль как язык программирования задач искусственного интеллекта, в силу ряда своих качеств он остается языком, удобным для изучения функционального программирования. Лисп – один из наиболее ярких представителей *функциональной парадигмы программирования* [14, 15]. Простота и ясность синтаксиса сочетаются в нём с мощностью языковых средств и естественностью его расширения. Одна из наиболее удивительных особенностей Лиспа – это единая синтаксическая форма записи программ и данных, что позволяет обрабатывать структуры данных как программы и модифицировать программы как данные.

В основу данного учебного пособия легли лекции по функциональному программированию на базе языка Лисп, неоднократно читавшиеся Е.И. Большаковой на факультете ВМК МГУ им. Ломоносова. Основное внимание в пособии уделяется базовым функциональным средствам языка Лисп, позволяющим изучить основополагающие принципы функциональной парадигмы. Поскольку непременной составляющей функционального программирования является использование *рекурсии*, в учебном пособии подробно разбираются приёмы построения рекурсивных программ, иллюстрируемые

многочисленными примерами. Рассматриваются также вопросы построения *функционалов* – функций, аргументами и значением которых являются функции.

Учебное пособие не ставит своей целью описание какого-то конкретного диалекта языка Лисп, однако в нём освещается ряд важных особенностей двух его известных диалектов – Common Lisp и MuLisp, в случае их принципиальных отличий. Существующие в современных диалектах Лиспа конструкции, нарушающие принципы строгой функциональности и внесённые в язык под давлением *императивной парадигмы программирования* (присваивания, блоки и др.), описываются лишь для полноты картины языка и сравнительно кратко.

В последнем разделе пособия приводятся задания практикума по программированию на языке Лисп, поддерживающие его изучение.

Авторы благодарят Н.В. Баеву за помощь в подготовке пособия.

1. Базовый Лисп

В данном разделе рассматриваются основы языка Лисп – в своём большинстве это функциональные средства, предложенные автором языка Дж. Маккарти и составившие чисто функциональный диалект Lisp 1.5 [2, 9]. Эти базовые средства представлены во всех современных диалектах языка Лисп.

1.1. Атомы и списки, функции

Обрабатываемые в языке Лисп данные можно разделить на скалярные (простые) и структурные (составные). К скалярным данным относятся ***атомы*** (атом – неделимое целое), а к структурным – ***списки***, или списочные структуры. В свою очередь атомы подразделяются на:

- *символьные*, т.е. идентификаторы (например: `BETA` , `A` , `SYM_1`);
- *числовые*, т.е. целые и вещественные числа (`-25`, `375.08` и др.).

Символьные атомы состоят из букв, цифр и других символов и отличаются от числовых атомов первым символом. Если атом начинается с цифры и знаков **`+`** и **`-`**, то он считается числом, которое должно быть записано по общепринятым правилам. При записи символьных атомов обычно можно использовать спецзнаки (*, = и др.), регистр же букв не важен: так, `B5` и `b5` – это два варианта записи одного и того же атома `B5`.

Символьные атомы используются как для записи обрабатываемых списков, так и в качестве имён переменных и функций.

Среди множества атомов базового Лиспа ***константами*** являются:

- все числовые атомы;
- символьные атомы **`T`** и **`NIL`**, обозначающие соответственно логические значения *истина* и *ложь*; а атом **`NIL`** также обозначает *пустой список*.

Ключевым понятием языка Лисп является ***список*** – рекурсивная структура, которая может быть описана следующими БНФ (металингвистическими формулами Бэкуса-Наура):

список ::= (*последовательность_элементов*) | *пустой список*

последовательность_элементов ::= *элемент* |

элемент ␣ *последовательность_элементов*

элемент ::= *атом* | *список*

пустой список ::= `()` | `NIL`

В приведённых БНФ знак ␣ обозначает пробел. Пробелы используются в качестве разделителей элементов списка. Другими разделителями элементов являются круглые скобки, при этом расположенные рядом со скобками пробелы могут быть опущены. Примеры списков языка Лисп:

```
(BETA 56 (9))                    ()
((C ()) NIL 81 (EE B C))         ((((T))))
```

Таким образом, лисповский список может содержать в качестве своих элементов атомы и другие, вложенные списки. В каждом списке может быть произвольное количество элементов. Количество уровней вложенности списков также не ограничивается. Первый уровень элементов списка, открывающийся внешней круглой скобкой, называется *верхним*. *Глубиной* списка считается максимальное количество вложенных пар скобок. Так, на верхнем уровне списка `(BETA 56 (9))` находятся три элемента: символьный атом `BETA`, число `56` и вложенный список `(9)`, а глубина этого списка равна 2.

Единственной константой составного типа является пустой список `()`, который может быть записан и как атом `NIL`, обе эти записи эквивалентны. Отметим, что уникальной особенностью этой константы является её принадлежность одновременно и к атомам, и к спискам.

Программа на языке Лисп записывается в виде последовательности атомов и списков, т.е. программа и данные имеют одинаковый синтаксис. Точнее, ***лисп-программа*** представляет собой последовательность форм, а *форма*, или *вычислимое выражение* – это атом или список, который можно вычислить и получить значение. Далее для указания связи форм с их значениями будем использовать запись вида *форма* => *значение* .

Вычисление программы реализует ***лисп-интерпретатор***, который считывает очередную входящую в программу форму, вычисляет её и выводит полученный результат (атом или список).

Важным является вопрос: какие атомы и списки вычислимы (т.е. являются формами). Охарактеризуем три самых распространённых случая вычислимых выражений: константа, переменная со значением и обращение к функции.

1) Любая константа представляет собой вычислимое выражение, значением которого является она сама. Например,

   для чисел: `72` => `72`

   для символьных атомов: `t` => `T`

   `()` => `NIL`

   `NIL` => `NIL`

   Заметим, что при выводе атомов лисп-интерпретатор использует обычно заглавные (прописные) буквы, а атом `NIL` применяется как каноническая запись пустого списка.

2) Символьный атом вычислим только тогда, когда он представляет собой имя переменной, точнее – имя формального параметра (аргумента) некоторой функции, и этот параметр имеет значение, например: `x` => `(B C D)`.

3) Список можно вычислить, если он представляет собой обращение к функции, или *функциональный вызов*: $(f\ e_1\ e_2\ \ldots\ e_n)$,

   где `f` – символьный атом, имя вызываемой функции,

   а $e_1, e_2, \ldots, e_n$ – аргументы этой функции, $n \geq 0$.

   `n` - число аргументов функции.

   В случае `n=0` имеем вызов функции без аргументов: `(f)`. Обычно $e_1, e_2, \ldots, e_n$ являются вычислимыми выражениями и вычисляются последовательно слева направо.

Таким образом, если в процессе работы лисп-интерпретатора требуется вычислить некоторый список, то первым элементом этого списка должно быть имя функции. Если это не так, лисп-интерпретатор сообщает об ошибке и прерывает вычисление текущего выражения программы.

Вместо общепринятой в математике записи обращений к функциям вида `f(x,y)` в Лиспе используется нотация, при которой имя функции стоит не перед аргументными скобками, а внутри них, вместе с аргументами, которые разделяются уже не запятыми, а пробелами: `(f x y)`. Именно за счёт этого достигается синтаксическое единообразие программы и данных.

Функции Лиспа обычно делятся на:

- *встроенные*, или стандартные, которые могут применяться без определения;
- *определяемые* пользователем в его программе.

Другое важное в Лиспе деление функций на классы учитывает количество и вычислимость аргументов функции. Различают:

- *обычные* функции (их большинство);
- *особые*, или специальные функции.

Обычная функция имеет строго фиксированное число аргументов, и при вычислении её значения интерпретатор сначала вычисляет значения её аргументов, а уже затем функция применяется к вычисленным значениям.

У *особой* функции нарушается хотя бы одно из двух указанных требований, т.е. у неё может быть произвольное количество аргументов и/или некоторые её аргументы могут не вычисляться.

Поскольку возможны вложенные функциональные обращения, к примеру: `(f1 (f2 x) (f4 (f3 y)))` ,
существенен порядок их вычисления. Вычисление начинается с самой внешней функции, в данном примере – с функции `f1`. Если она обычная, то сначала необходимо вычислить вложенные функциональные вызовы, задающие значения её аргументов – обращения к функциям `f2` и `f4`. В свою очередь для вычисления значения функции `f4` (если она обычная) требуется предварительно вычислить значение функции `f3`. Функциональные вызовы, находящиеся на одном уровне, вычисляются обычно слева направо: сначала `f2`, затем `f4`. Вычисление вложенных функциональных вызовов завершается в соответствии с их иерархией, т.е. в первую очередь вычисляется самое вложенное обращение, а в последнюю – самое внешнее. Особые функции могут нарушать этот стандартный порядок. Таким образом, если в приведённом примере все функции `f1`, `f2`, `f3` и `f4` – обычные, то они вычисляются в следующем порядке: `f2`, `f3`, `f4`, `f1`.

1.2. Базовый набор функций

Базовый набор включает 8 встроенных функций Лиспа. Первые три функции являются обычными и реализуют основные операции над списками: две функции для расщепления и одна для построения списка.

Функция **`car`** от одного аргумента возвращает первый элемент списка, являющегося значением её аргумента.

Функция **cdr** возвращает *хвост* списка, являющегося значением её единственного аргумента (хвостом, или остатком списка является список без своего первого элемента).

Функция **cons** от двух аргументов с обращением(cons e_1 e_2) строит новый список, первым элементом которого является значение первого аргумента e_1, а хвостом – значение второго аргумента e_2. По сути, эта функция включает заданный элемент (значение выражения e_1) в начало списка, являющегося значением e_2.

Приведём примеры работы этих функций. Будем считать, что переменная x имеет значение (GAMMA (15)), а переменная y имеет значение (TETA):

```
          x => (GAMMA (15))     y => (TETA)
Тогда     (car x) => GAMMA
          (cdr x) => ((15))
          (cdr y) => NIL
          (car (cdr x)) => (15)
          (car(car(cdr x))) => 15
          (cons x y) => ((GAMMA (15)) TETA)
          (cons y ()) => ((TETA))
```

Заметим, что функция-конструктор cons (сокращение от англ. *construct*) реализует обратное действие по отношению к car и cdr. Будем считать пока, что вторым аргументом cons должен быть список. Что же касается области определения функций car и cdr, то, несмотря на то, что в ряде диалектов Лиспа (например, в MuLisp) они работают и для атомов, будем считать, что значением их аргумента должен быть непустой список (это обеспечит их правильную работу во всех диалектах).

Имена функций car и cdr происходят из самой ранней реализации Лиспа, в которой использовались специальные регистры: адресный и декрементный, и имена были даны как сокращения названий *Contents of Address Register* и *Contents of Decrement Register*. Современные диалекты Лиспа обычно допускают для функций car и cdr мнемоничные синонимичные названия: first и rest соответственно.

Следующие две функции базового набора являются предикатами, т.е. функциями, вычисляющими логическое значение T или NIL в зависимости от наличия определённых свойств у своих аргументов. Обе эти функции – обычные.

Функция **atom** вырабатывает значение T, если значением её единственного аргумента является атом (числовой или символьный). В противном случае возвращается NIL.

Функция-предикат **eq** (сокращение от англ. *equal*) проверяет совпадение двух своих аргументов-атомов, вырабатывая значение T, когда:

1) значением одного из аргументов является атом, и одновременно
2) значения аргументов равны (идентичны).

В ином случае значением функции eq является NIL. Будем считать, что когда оба аргумента функции eq – списки, её результат неопределён. Кроме того, поскольку одно и то же число может быть записано разными способами (например, +6 и 6), функция eq корректно осуществляет сравнение только *символьных* атомов, а для проверки равенства чисел служит обычно другая встроенная функция.

Приведём примеры работы рассмотренных функций-предикатов, используя прежнее значение переменной x:

```
x => (GAMMA (15))
(atom NIL) => T
(atom T) => T
(atom x) => NIL
(atom ()) => T
(atom (cdr (cdr x))) => T
(eq (atom 5) T) => T
(eq (car x)(cdr x)) => NIL
```

Шестая функция базового набора – особая функция **quote**, которая в качестве своего значения выдаёт сам аргумент, не вычисляя его:

```
(quote e) => e
```

По сути, эта функция блокирует вычисление своего аргумента (англ. *quote* – цитировать, взять в кавычки). Необходимость в этом нередко возникает при использовании обычных встроенных функций (т.е. вычисляющих значения своих аргументов), чтобы задать аргументы в явном виде и избежать их вычисления, например:

```
(cons 5 (quote(A T)) ) => (5 A T).
```

Если убрать в этом примере обращение к quote, то второй аргумент функции cons будет рассматриваться как обращение к функции с именем A, и в случае, когда функция с таким именем не определена, лисп-интерпретатор выдаст сообщение об ошибке.

Отметим, что константы – числа и атомы T, NIL при их использовании в качестве аргументов обычных функций можно не квотировать, поскольку значением любой константы является она сама.

Функция quote используется часто, поэтому допускается упрощённый способ обращения к ней с помощью апострофа, маркирующего квотируемое выражение:

```
(quote (ATOM B)) => (ATOM B)
'(ATOM B) => (ATOM B)
(cons (quote(A (B))) NIL) => ((A (B)))
(cons '(A (B)) NIL) => ((A (B)))
(atom '(NIL)) => NIL
```

Следующая функция базового набора **`eval`** выполняет двойное вычисление своего аргумента. Эта функция является обычной, и первое вычисление аргумента выполняет так же, как и любая обычная функция. Полученное при этом выражение вычисляется ещё раз. Такое двойное вычисление может понадобиться либо для снятия блокировки вычислений (установленной функцией `quote`), либо же для вычисления сформированного в ходе первого вычисления нового функционального вызова. Первые три из приводимых ниже примеров иллюстрируют использование функции `eval` для снятия блокировки, а последний пример – вычисление динамически построенного функционального вызова:

```
(eval (quote (atom b))) => T
(eval (quote (quote quote))) => QUOTE
(eval '(car '(a b c))) => A
(eval (cons (quote car) (quote (x)))) => GAMMA
      └──────────────(car x)──────────┘
```

В последнем примере на нижней линии показан результат выполненного функцией `eval` первого вычисления своего аргумента.

Заметим однако, что

```
(quote (EVAL (ATOM B))) => (EVAL (ATOM B))
```

поскольку вычисление любого функционального вызова начинается с вычисления внешней функции `quote`, а эта функция не вычисляет свой аргумент, возвращая его в неизменном виде.

Последняя (по порядку, но не по значимости) функция базового набора **`cond`** (сокращение от англ. *condition* – условие) служит средством разветвления вычислений. В строго функциональном программировании вызов этой функции, как правило, имеет вид

$$(\mathtt{cond}\ (p_1\ e_1)\ (p_2\ e_2)\ \ldots\ (p_n\ e_n)),\ n \geq 1.$$

Обращение к функции `cond` называется ***условным выражением***, выражения $(p_i\ e_i)$ – ***ветвями*** условного выражения, а выражения-формы p_i – ***условиями*** ветвей.

Функция `cond` является особой, поскольку в условном выражении может быть произвольное количество ветвей, и не все формы e_i

вычисляются в общем случае. Вычисление условного выражения выполняется по следующим правилам:

I. последовательно вычисляются условия ветвей до тех пор, пока не встретится выражение-форма p_i, значение которой отлично от NIL;

II. вычисляется выражение e_i соответствующей ветви и его значение возвращается в качестве значения функции cond;

III. если все условия p_i имеют значение NIL, то значением условного выражения становится NIL.

Часто в качестве условия последней ветви p_n берут атом T, обозначающий логическое значение *истина* – тогда значение e_n становится значением условного выражения в случае, когда все значения предыдущих условий оказались равны NIL. Например:

```
(cond ((eq 'A T) 'are_equal)
      (T 'not_equal))              =>   NOT_EQUAL
```

Заметим, что в таких случаях вместо атома T может использоваться любое выражение, значение которого не равно NIL. По сути, работа функции cond опирается на расширенное понимание логического значения *истина*: истинным значением считается не только T, но и любое лисповское выражение, отличное от NIL.

В обращении к функции cond могут быть опущены некоторые выражения-формы e_i – в этом случае их роль выполняет соответствующее p_i, и если результат вычисления p_i будет отличен от NIL, то этот результат и будет возвращён в качестве значения всего условного выражения. Например:

```
(cond ((eq 'A1 'A2) 'are_equal)
      ('not_equal))                =>   NOT_EQUAL
```

Заметим, что каждую ветвь условного выражения принято записывать на отдельной строке, при этом часто используются пробельные отступы, показывающие вложенность функциональных вызовов.

Приведём пример вложенных условных выражений: записанная ниже форма с двумя переменными x и y вычисляет логическое значение – результат применения к значениям переменных операции *исключающего или* (сложение по модулю 2):

```
(cond (x (cond (y NIL)
               (T))
         (y))
```

Эта форму можно упростить так:

```
(cond (x (eq y NIL)) (y)) .
```

Задавая всевозможные комбинации логических значений T и NIL для переменных x и y (всего таких комбинаций – 4, NIL обозначает логическое значение *ложь*), получаем логические значения, например, для комбинации x=NIL и y= T:

```
(cond (NIL (eq T NIL))(T))  => T .
```

Следующие два выражения-формы с двумя переменными x и y выражают соответственно логические конъюнкцию и дизъюнкцию:

```
(cond (x y))
(cond (x) (y))
```

Эти условные выражения вычислимы не только при логических значениях переменных. В качестве значений x и y могут быть взяты любые лисповские выражения, к примеру, x=(H7(12)) и y=(T), и при этом могут получиться значения, отличные от атомов T и NIL. По сути, эти два выражения реализуют конъюнкцию и дизъюнкцию при расширенном понимании логического значения *истина*. Аналогичная ситуация имеет место и для формы, реализующей операцию сложения по модулю 2.

Такая трактовка логических значений, при которой истинным считается любое выражение (атом, список), отличное от NIL, является особенностью языка Лисп. Она позволяет считать произвольное лисповское выражение логическим, а любую лисповскую функцию – предикатом.

Рассмотренные примеры показывают гибкость и мощность функции cond, с помощью которой можно выразить все известные логические операции. Описанный набор из 8 встроенных лисповских функций вкупе со средствами определения новых функций является *алгоритмически полным*, т.е. позволяет записать любой алгоритм обработки списочных структур. Однако этот набор явно недостаточен для практического программирования.

1.3. Определение функций

Для определения новой функции необходимо задать выражение для вычисления её значения и указать, какие переменные этого выражения будут служить формальными параметрами функции. Кроме того, следует определить, в каком порядке должны рассматриваться формальные параметры при передаче им фактических значений. Для этих целей в базовом Лиспе используется конструкция, называемая ***лямбда-выражением*** или *определяющим выражением функции* (термин *лямбда-выражение* был заимствован из лямбда-исчисления – специального формализма для описания вычислимых функций). БНФ этой конструкции:

лямбда-выражение ::= (lambda *лямбда-список тело_функции*)

Лямбда-список представляет собой лисповский список из символьных атомов, рассматриваемых как имена *формальных параметров* функции. В частном случае этот список может быть пустым, что соответствует функции без аргументов.

Телом функции является некоторое лисповское выражение (форма), в которое в общем случае входят заданные в лямбда-списке формальные параметры. Тело функции служит для вычисления её значения.

Приведём пример лямбда-выражения с двумя параметрами `x` и `y`:

```
(lambda (x y) (cond (x) (y)))
```

и лямбда-выражения с одним параметром `x`:

```
(lambda (x) (cond (x) (y)))
```

Отличие этих выражений в том, что в последнем случае переменная `y` не внесена в число параметров функции.

По сути, лямбда-выражение – это средство параметризации вычисляемого выражения. В базовом Лиспе лямбда-выражение служит для описания обычной функции, причём *безымянной* (поскольку с лямбда-выражением не связывается никакого имени). Пример безымянной функции, вычисляющей дизъюнкцию двух своих аргументов (при расширенном понимании истинного значения):

```
(lambda (x y) (cond (x) (y)))
```

Отметим, что атом `lambda` не является именем встроенной функции и лямбда-выражение не является формой (его нельзя вычислить). Однако оно может быть использовано для вызова определённой в нём функции – для этого служит конструкция, называемая ***лямбда-вызовом***:

лямбда-вызов ::= (*лямбда-выражение последовательность_форм*)

К примеру, вычисление лямбда-вызова для приведённой выше безымянной функции-дизъюнкции:

```
((lambda(x y)(cond (x) (y))) 'A  3)  => A
```

В таком вызове функции вида $(\lambda\ p_1\ p_2\ \ldots\ p_k)$ лямбда-выражение λ, стоящее на обычном месте имени функции, непосредственно её определяет, а идущая за λ последовательность форм задаёт *фактические параметры* (аргументы) этой функции.

Итак, лямбда-вызов является формой следующей структуры:

$((\text{lambda}\ (x_1\ x_2\ \ldots\ x_k)\ e)\ p_1\ p_2\ \ldots\ p_k)$

где $k \geq 0$,

$p_1, p_2 \ldots p_k$ – произвольные выражения-формы (фактические параметры),

$x_1, x_2 \ldots x_k$ – символьные атомы (формальные параметры).

Опишем этапы вычисления этой формы при $k \geq 1$:

1) Вычисление аргументов: последовательно вычисляются фактические параметры p_1, p_2, … p_k лямбда-вызова, пусть их значения – v_1, v_2, …, v_k .
2) Связывание формальных параметров: формальные параметры x_1, x_2, … x_k попарно связываются (на время вычисления лямбда-вызова) соответственно со значениями v_1, v_2, …, v_k фактических параметров лямбда-вызова: $x_1=v_1$, $x_2=v_2$, …, $x_k=v_k$.
3) Вычисление тела: вычисляется форма `e` (тело функции), причём всюду, где необходимо вычислить x_i, в качестве его значения берётся v_i. Вычисленное таким образом значение формы `e` служит итоговым значением лямбда-вызова.

Важно, что формальные параметры функции локализованы в её теле и все связи $x_i=v_i$ носят временный характер: по окончании вычисления тела эти связи разрушаются, и если переменные x_i были ранее связаны с какими-либо другими значениями, то более ранние связи восстанавливаются.

В частном случае `k=0` вычисление лямбда-вызова без аргументов:

```
((lambda () e))
```

сводится к вычислению формы `e` и выдаче полученного результата в качестве значения этого лямбда-вызова.

Таким образом, лямбда-вызов является средством связывания формальных и фактических параметров на время вычисления тела функции. А лямбда-выражение и лямбда-вызов, взятые вместе – механизм определения и выполнения параметризованных вычислений. Отметим, что присутствие в этом механизме этапа 1 означает, что передача фактических параметров выполняется *по значению*, а значит, с помощью него может быть определена и вычислена только *обычная* функция.

Ясно, что безымянная функция, определённая в некотором лямбда-выражении и применённая в лямбда-вызове с некоторыми фактическими аргументами, уже не может быть использована в других лямбда-вызовах, для этого она должна быть повторно в них определена. Поэтому безымянные функции – это функции, вызываемые один раз, чаще всего они используются в *функционалах* (функциях, аргументом или значением которых является функция).

Для неоднократного применения функции (а также для построения рекурсивной функции) требуется средство её именования – для этого служит особая встроенная функция `defun`, обращение к которой обычно имеет вид:

`(defun` *имя_функции лямбда-список тело_функции* `)`

В качестве имени функции выступает символьный атом. Значением этой формы является имя определяемой функции:

(defun f (x_1 x_2 … x_k) e) => F, $k \geq 0$

Однако основное действие функции defun – связывание с именем f лямбда-выражения (lambda (x_1 … x_k) e), что означает определение новой функции с именем f и указанным определяющим выражением. В дальнейшем можно обращаться к этой функции с фактическими параметрами p_1, …, p_k : (f p_1 … p_k), что будет означать вычисление лямбда-вызова ((lambda (x_1 … x_k) e) p_1 … p_k),
т.е. имя функции при вычислении функционального вызова заменяется её определяющим выражением.

Приведём пример определения и использования функции List2, образующей список из значений двух своих аргументов:

```
(defun List2 (x y) (cons x (cons y ()))) => LIST2

(List2 '(A) 'B) => ((A) B)
(List2 'Y  'X) => (Y X)
(List2 T '(C (E))) => (T (C (E)))
```

Действие List2 сводится к вложенным вызовам функции cons, причём второй аргумент внутреннего вызова cons – пустой список, служащий основой формирования результирующего списка.

Ещё один пример – определение функции InsNew, вставляющей атом NEW после первого элемента своего аргумента-списка (на верхнем его уровне):

```
(defun InsNew(x)
    (cons(car x)(cons 'NEW (cdr x)))) => INSNEW

(InsNew '(F (T E ())C)) => (F NEW (T E NIL) C)
```

Эта функция фактически строит новый список, состоящий из первого элемента исходного списка, атома NEW и хвоста исходного списка.

Кроме рассмотренного способа обращения к функции defun обычно допускается и такое обращение:

(defun f (lambda (x_1 … x_k) e)) .

При этом способе (являющимся исторически первым) более явно видна связь имени f функции с её определяющим выражением. Однако гораздо чаще используется упрощённая запись, когда исключены внешние скобки лямбда-выражения и атом lambda.

Во многих диалектах Лиспа есть встроенная особая функция `let`, обращение к которой является ничем иным, как другой формой записи лямбда-вызова:

$$(\text{let } ((x_1\ p_1)(x_2\ p_2)\ldots(x_n\ p_n))\ e) \equiv ((\text{lambda } (x_1\ \ldots\ x_n)\ e)\ p_1\ \ldots\ p_n)\ .$$

В конструкции `let` формальные и фактические параметры сгруппированы попарно и помещены в начало, а её вычисление происходит таким же образом, как и вычисление соответствующего ей лямбда-вызова. Например:

```
(let ((x 'A) (y 3)) (cond (x) (y)))    => A
```

Чаще всего эта конструкция используется для сокращения вычислений в случаях, когда некоторое выражение надо вычислить и использовать несколько раз. К примеру, следующее условное выражение:

```
(cond ((atom (F1 x))(F1 x))
      ((eq x (F2 y))(F3 (F2 y)))
      (T (F3 x) ))
```

в котором дважды вычисляется `(F1 x)` и `(F2 y)`, можно переписать с использованием `let`, исключив повторные вычисления:

```
(let ((x1 (F1 x))(y2 (F2 y)))
     (cond ((atom x1) x1)
           ((eq x y2) (F3 y2))
           (T (F3 x) )))
```

Подытожим возможные случаи вычислимости лисповского списка. Лисповский список является функциональным вызовом только в двух случаях:

- ✓ Первый случай – обращение к функции по имени: $(f\ e_1\ \ldots\ e_k)$, $k \geq 0$; этот случай включает обращение к функции `let`.
- ✓ Второй случай – лямбда-вызов, или обращение к безымянной функции: $(\lambda\ e_1\ \ldots\ e_k)$, где λ – лямбда-выражение, $k \geq 0$.

Подчеркнём, что первый элемент списка-вызова функции не вычисляется лисп-интерпретатором, он представляет собой либо явно заданное имя функции, либо явно заданное определяющее выражение функции.

1.4. Встроенные функции

Для практического программирования на Лиспе требуется более широкий набор встроенных функций. Различные диалекты Лиспа включают разное количество встроенных функций – от нескольких десятков до нескольких сотен. Однако количество не столь важно,

поскольку большинство из них может быть запрограммировано на основе описанного выше базового набора лисп-функций и функции defun.

Опишем ряд встроенных функций Лиспа, существующих в большинстве диалектов и широко применяемых как вспомогательные при программировании более сложных задач обработки списков. В случае, когда эти функции могут быть легко запрограммированы на основе базового набора, мы приводим их определения.

Описываемые функции сгруппированы по своему назначению. Большинство из них являются обычными, поэтому вид функции явно оговаривается только для особых функций.

Функции обработки списков

К этой группе относятся 28 обычных функций от одного аргумента caar, cadr, cdar, cddr, caaar, caadr, …, cdddar, cddddr, действие которых эквивалентно определённой суперпозиции функций car и cdr. Эти суперпозиции позволяют, в частности, выбирать некоторый элемент с нужного уровня списка-аргумента. Например, функция caddr выбирает из списка-аргумента третий элемент верхнего уровня, а функция caadr – первый элемент списка, стоящего вторым на верхнем уровне:

```
(caddr '(P (C E) B ())) => B,
(caadr '(P (C E) B ())) => C.
```

Функция cddr выдаёт список без двух первых элементов верхнего уровня:

```
(cddr '(P (C E) B ())) => (B NIL)
```

Их определения:

```
(defun caddr (x) (car (cdr (cdr x))))
(defun caadr (x) (car (car (cdr x))))
(defun cddr (x) (cdr (cdr x)))
```

Укажем общий вид имени функции-суперпозиции: c{[a|d]}r , где фигурные скобки обозначают повторение, а в квадратных скобках указаны повторяющиеся буквы. Между буквами c и r имени может быть не более четырёх букв a и d. Последовательность a и d в имени функции соответствует порядку следования car и cdr в эквивалентной суперпозиции. Предполагается, что список-аргумент у всех функций-суперпозиций содержит необходимое число элементов.

В группу функций обработки списков входит также часто используемая функция-конструктор list, составляющая список из значений своих аргументов. Эта функция относится к особым, поскольку у неё может быть произвольное число аргументов, но при этом все аргументы вычисляются. Например:

```
(list '(NUM K) () 'C) => ((NUM K) NIL C)
```

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

Отметим, что в отличие от функции `cons`, результат функции `list` симметричен относительно аргументов:

```
(list '(A) '(B)) => ((A)(B)),
```
но `(cons '(A) '(B)) => ((A) B).`

Для любого фиксированного количества аргументов определить функцию `list` на основе базового набора и функции `defun` легко (см. пример функции `List2` из предыдущего раздела), однако для организации произвольного количества аргументов необходимы специальные средства определения особых функций.

Арифметические функции

Эти функции выполняют операции сложения, вычитания, умножения и деления чисел и обозначаются общепринятым образом: `+ - * /`. Операции `+` и `*` обычно определены для произвольного количества аргументов (от одного и более). Операции `–` и `/` определены для двух и одного аргумента, в последнем случае они означают соответственно изменение знака числа-аргумента на противоположный и вычисление обратной величины. Например:

```
(+ 12 -67 34) => -21
(* 1 2 3 4) => 24
(- -15 -3) => -12
(- -56) => 56
(/ -12 3) => -4
```

Заметим, что в случае операции деления результат может быть различен в разных диалектах Лиспа:

`(/ -12 5) => -2.4` в MuLisp и `-12/5` в Common Lisp,
`(/ 4) => 0.25` в MuLisp и `1/4` в Common Lisp,

т.е. когда результат деления целых чисел не является целым, в Common Lisp он записывается как рациональное число.

Часто полезны встроенные в MuLisp функции `add1` и `sub1` прибавления и вычитания единицы:

```
(add1 23) => 24  , (add1 -15) => -14
(sub1 23) => 22  , (sub1 -15) => -16
```

Эти функции можно определить следующим образом:

```
(defun add1 (x) (+ x 1))
(defun sub1 (x) (- x 1)) .
```

Арифметические предикаты

Все эти функции имеют два вычисляемых аргумента, значениями которых должны быть числа. Предикаты вырабатывают логическое значение T или NIL, в зависимости от выполнения проверяемого условия: **=** (равенство чисел), **/=** (неравенство), **<** (меньше), **>** (больше), **<=** (не больше), **>=** (не меньше). Например:

```
(= 1/4 0.25) => T
(/= 1/4 (- 0.3 0.05)) => NIL
(< 7 (+ 12 3)) => T
(<= 7 (+ -12 3)) => NIL
```

Арифметический предикат evenp от одного числового аргумента возвращает T, если значение его аргумента является чётным числом, и NIL в противном случае. Примеры его использования:

```
(evenp 25) => NIL
(evenp (+ -15 -3)) => T
```

Отметим, что для сравнения атомов-чисел применяется функция **=**, а не eq (поскольку запись одного числа может варьироваться). В случае, когда заранее неизвестно, какими будут сравниваемые атомы – символьными или числовыми, удобна функция-предикат **eql**, встроенная в MuLisp. Её можно определить следующим образом:

```
(defun eql (x y)(cond ((numberp x)
                       (cond ((numberp y)(= x y))
                             (T NIL)))
                (T (eq x y))))
```

Предикаты типа

Одна из самых часто используемых функций в Лиспе – предикат null, который вырабатывает атом T, если значение его аргумента – NIL, и атом NIL в противном случае. Приведём определение и пример использования этого предиката:

```
(defun null (x) (eq x NIL))
(null '(M)) => NIL
(null (cdr '(M))) => T
```

Функция-предикат listp выдаёт значение T, если значением её аргумента является список, и NIL в противном случае.

```
(listp 'A )  => NIL
(listp (car '((A)) )) => T
(listp (caar '((A)))) => NIL
```

Отметим, что в случае пустого списка `listp` вырабатывает значение `T`:

```
(listp (cdr '((a)))) => T
```

Функция `listp` может быть определена на основе базового набора встроенных лисп-функций:

```
(defun listp (x)(cond ((eq x NIL))
                      ((atom x) NIL)
                      (T)))
```

Предикат `numberp` вырабатывает `T`, если значение его аргумента – числовой атом, и `NIL` в противном случае. Парный к нему встроенный предикат `symbolp` выдаёт `T`, если значением его аргумента является символьный атом, и `NIL` в противном случае:

```
(numberp (car '(12 A S))) => T
(numberp (cadr '(12 A S))) => NIL
(symbolp (car '(12 A S))) => NIL
(symbolp (cadr '(12 A S))) => T
(symbolp '(12 A S)) => NIL
(numberp '(12 A S)) => NIL
(symbolp ()) => T
```

Логические функции

К логическим функциям-предикатам относят логическое отрицание `not`, конъюнкцию `and` и дизъюнкцию `or`. Первая из этих функций является обычной, а другие две – особыми, поскольку допускают произвольное количество аргументов, которые не всегда вычисляются.

Логическое отрицание `not` вырабатывает соответственно:

`(not NIL)` => `T` и
`(not T)` => `NIL`

и может быть определено функцией

```
(defun not (x) (eq x NIL)) .
```

Фактически действие этой функции эквивалентно действию функции `null`, работающей не только с логическими значениями `T` и `NIL`, но и с произвольными лисповскими выражениями. Поэтому, например:

`(not '(B ()))` => `NIL`

Тем самым, определение функции `not` соответствует лисповскому расширенному пониманию логического значения *истина*. Заметим, что два разных имени `not` и `null` для фактически одной функции удобны для её использования в разных контекстах.

Две другие встроенные логические функции также используют расширенное понимание истинного значения.

Вызов функции and, реализующей конъюнкцию, имеет вид

(and e_1 e_2 … e_n), $n \geq 0$.

При вычислении этого функционального обращения последовательно слева направо вычисляются аргументы функции e_i – до тех пор, пока не встретится значение, равное NIL. В этом случае вычисление прерывается и значение функции равно NIL. Если же были вычислены все значения e_i и оказалось, что все они отличны от NIL, то результирующим значением функции and будет значение последнего выражения e_n .

Вызов функции-дизъюнкции имеет вид (or e_1 e_2 … e_n), $n \geq 0$. При выполнении вызова последовательно вычисляются аргументы e_i (слева направо) – до тех пор, пока не встретится значение e_i, отличное от NIL. В этом случае вычисление прерывается и значение функции равно значению этого e_i. Если же вычислены значения всех аргументов e_i, и оказалось, что они равны NIL, то результирующее значение функции равно NIL.

При n=0 значения функций: (and)=>T, (or)=>NIL.

Приведём примеры вычислений этих функций:

```
(and (atom T)(= 1 23)(eq 'A 'B)) => NIL
(and (< 12 56) (atom 'Z) '(A S)) => (A S)
(or (eq 'A 'B) (atom 'Y) ()) => T
(or (eq 'A 'B) 'Y '(T R)) => Y
(or (atom '(Y V)) () (eq 'A 'B)) => NIL
```

Таким образом, значение функции and и or не обязательно равно T или NIL, а может быть произвольным атомом или списочным выражением.

1.5. Рекурсивные функции

Рекурсия – мощный механизм вычислений, поддерживаемый интерпретатором языка Лисп. Примем за рабочее следующее определение: функция называется *рекурсивной,* если в её теле содержится хотя бы одно обращение к ней самой. Рассмотрим несколько задач на программирование рекурсивных функций, используя для этого в основном базовый набор встроенных функций Лиспа.

В качестве первой задачи определим функцию Length (встроена в MulLisp и CommonLisp), вычисляющую длину своего аргумента-списка, т.е. количество элементов на верхнем уровне:

```
(Length '(A (5 6) D)) => 3.
```

Ясно, что для этого необходимо организовать проход по верхнему уровню списка-аргумента с одновременным подсчётом числа его элементов. Из функций базового набора для реализации прохода подходят лишь функции `car` и `cdr`. В рассматриваемой задаче достаточно использовать функцию `cdr`, вырабатывающую хвост списка – её последовательное применение и будет означать движение по верхнему уровню списка. Ясно, что применять `cdr` следует до тех пор, пока список не закончится, т.е. не станет пустым. Для подсчета длины будем использовать следующее правило: длина списка равна длине его хвоста, увеличенной на `1`. Очевидно, что длина пустого списка равна нулю. Таким образом, получаем функцию, телом которой служит условное выражение с двумя ветвями:

```
(defun Length (L)
          (cond ((null L) 0)
                (T (+ 1 (Length (cdr L))) )))
```

Вторая ветвь функции содержит рекурсивный вызов, а первая – условие выхода из рекурсии (равенство аргумента функции пустому списку). Ветви рекурсивной функции, не содержащие рекурсивных вызовов и определяющие условия завершения рекурсии, называются *элементарными*. Обычно элементарные ветви предшествуют ветвям с рекурсивными вызовами.

В данном решении задачи рекурсивные вычисления обязательно закончатся, поскольку с каждым рекурсивным вызовом функции `Length` список-аргумент становится на один элемент короче, и по исчерпании элементов списка проработает первая, элементарная ветвь функции.

Для понимания хода вычисления рекурсивной функции полезно трассировать её вычисление для конкретных значений аргументов. Лисп-интерпретатор для выполнения рекурсивных вызовов использует *стек*, в котором запоминаются ещё необработанные вызовы (отложенные вычисления). Ниже схематично показано изменение содержимого стека в ходе вычисления вызова `Length` с аргументом `(A (5 6) D)`:

```
(Length: L=(A (5 6) D) )   => 3
    → (+ 1 (Length: L=((5 6) D) ))   => 3 ↑
        → (+ 1 (Length: L=(D) ))     => 2 ↑
            → (+ 1 (Length: L=() ) ) => 1 ↑
                  └──────=>0──────┘
```

Первоначально в стеке находится исходное обращение к функции `Length`, с помощью знака = указано значение её формального параметра `L`. Затем в стек поочередно заносятся функциональные вызовы, получаемые в результате работы второй ветви функции `Length`. Каждое

записываемое в стек выражение – это вызов функции сложения с аргументом-вызовом Length при новом значении её параметра L. В нашем изображении стека эти вызовы записываются на очередной строке, и стоящая слева от них стрелка → означает занесение их в стек. Для формального параметра L в каждом вызове Length указывается его локальное значение.

Стек растёт до тех пор, пока в него не попадёт выражение, содержащее вызов Length с аргументом, равным NIL. Это выражение можно сразу вычислить (проработает элементарная ветвь функции, вычисляющая длину пустого списка) и убрать его из стека. Вычисленное значение используется для вычисления предыдущего записанного в стек выражения. Так последовательно, в обратном порядке выполняются отложенные в стек функциональные вызовы, и вычисленные при этом значения передаются для вычисления других отложенных в стек выражений. В нашем схематическом изображении эта передача показывается стрелкой ↑, стоящей справа от вычисленных значений. Этот процесс заканчивается, когда стек пуст и вычислено значение исходного обращения к функции Length.

Отметим, что при заполнении стека новые локальные связи переменной-параметра L отменяют на время старые, но старые связи восстанавливаются лисп-интерпретатором тогда, когда вычисления возвращаются к соответствующему функциональному вызову.

В качестве следующей задачи запрограммируем функцию-предикат Member от двух аргументов A и L: (Member A L). Функция проверяет, входит ли аргумент-атом A в заданный список L на верхнем его уровне, и вырабатывает по результатам этой проверки значение T или NIL:

```
(Member 'Q '(E (C) Q B)) => T
(Member 'Q '(E (C) B)) => NIL
```

Оба аргумента функции вычисляются. Будем предполагать, что значение аргумента A всегда есть символьный атом, в то же время список L может содержать на верхнем уровне неатомарные элементы.

Как и в предыдущей задаче, требуется последовательный просмотр списка, однако теперь необходимо ещё сравнение очередного элемента списка с заданным атомом. Проход по списку опять можно организовать с помощью функции cdr, тогда очередной элемент списка может быть получен обращением к функции car. Сравнение этого элемента со значением аргумента A реализует функция eq (она требует обязательной атомарности только одного своего аргумента). Поскольку возможны два разных исхода сравнения, необходимы две ветви функции: если найден атом A, то следует завершить вычисление функции со значением T, если

же результат сравнения отрицателен, то необходимо рекурсивно продолжать поиск атома A (в хвосте списка L). Если же в процессе рекурсивных вызовов список L стал пуст, это означает, что искомого атома нет в этом списке, и значение функции должно быть равно NIL.

Получающееся таким образом определение функции:

```
(defun Member (A L)
   (cond ((null L) NIL)
         ((eq A (car L)) T)
         (T (Member A (cdr L)) )))
```

Эта функция имеет три ветви, из них две первые – элементарные, порядок всех ветвей строго определён. Первая ветвь работает в том случае, когда список L пуст, и не может быть переставлена со второй, так как функция car должна иметь непустой аргумент-список. Завершаемость рекурсивных вычислений гарантируется тем, что рекурсивный вызов происходит с более простым аргументом.

Схематично покажем процесс рекурсивных вычислений в стеке:

```
(Member: A=Q, L=(E (C) Q B) )          => T
    → (Member: A=Q, L=((C) Q B) )   => T ↑
        → (Member: A=Q, L=(Q B)) => T ↑
```

Заметим, что во многих реализациях Лиспа (в том числе в MuLisp и CommonLisp) есть встроенная функция member, отличающаяся от запрограммированной нами тем, что если атом-значение параметра A встречается в списке L, то функция в качестве своего значения выдаёт часть списка, начинающуюся с этого атома:

```
(member 'Q  '(E (C) Q B)) => (Q B).
```

Такая особенность позволяет использовать member и как предикат, и как функцию поиска нужного подсписка. Для её реализации во второй ветви приведённого выше определения функции надо заменить атом T на переменную L.

Ещё одна задача – построение рекурсивной функции Append от двух аргументов-списков L1 и L2. Функция соединяет (сливает) элементы верхнего уровня обоих списков в один список, например:

```
(Append '(Q R T) '(K M)) => (Q R T K M)
(Append '(B (A)) '((C C) () D))
                      => (B (A) (C C) NIL D)
```

Действие Append иногда называют конкатенацией списков. В результате должен быть построен новый список, на верхнем уровне которого находятся сначала элементы верхнего уровня первого списка, а

затем – элементы верхнего уровня второго списка, причём в той же последовательности, что и в исходных списках. Поскольку в базовом наборе функций Лиспа есть только одна встроенная функция cons, формирующая новый список, то для решения нашей задачи придётся последовательно её использовать, каждый раз отщепляя очередной элемент от одного из списков-аргументов (например, от первого) и присоединяя его к другому при помощи cons. Этот процесс надо закончить, когда первый список-аргумент станет пустым (при этом значением функции будет второй список).

Определённая таким образом функция имеет две ветви:

```
(defun Append(L1 L2)
  (cond ((null L1) L2)
        (T (cons (car L1)(Append (cdr L1) L2)))))
```

Ключевым при программировании рекурсивной ветви является правильный порядок вызовов функций, согласно следующей идее: результат слияния двух списков есть результат присоединения первого элемента первого списка-аргумента к списку, получающемуся в результате рекурсивного слияния хвоста первого списка со вторым списком.

Рассмотрим вычисление вызова функции Append в стеке:

```
(Append: L1=(Q R T), L2= (K M) )      => (Q R T K M)
 →(cons 'Q(Append: L1=(R T), L2=(K M)))=>(Q R T K M)↑
   →(cons 'R(Append: L1=(T), L2=(K M) )) =>(R T K M)↑
     →(cons 'T(Append: L1=(),L2=(K M)))  =>(T K M)↑
              └──────────=>(K M)──────────┘
```

Видно, что реальное присоединение элементов первого списка ко второму списку откладывается до тех пор, пока не проработает элементарная ветвь функции. Тогда будет вычислено выражение, положенное в стек последним (в нём подчеркнуто вычисленное по элементарной ветви обращение к Append и показано полученное значение), и начнётся поочередное вычисление предыдущих отложенных функциональных вызовов. Результирующий список (Q R T K M) последовательно строится, начиная с присоединения T (последнего элемента первого списка-аргумента) к началу второго списка.

Если же поменять в рекурсивной ветви определения функции Append порядок вызовов функций cons и Append, то получим функцию RevAppend:

```
(defun RevAppend(L1 L2)
  (cond ((null L1) L2)
        (T (RevAppend (cdr L1)
                      (cons (car L1) L2)) )))
```

Эта функция при соединении списков будет менять порядок следования элементов первого списка на противоположный:

```
(RevAppend '(Q R T) '(K M)) => (T R Q K M).
```

В этом можно убедиться, проследив вычисление функции в стеке:

```
(RevAppend: L1=(Q R T), L2= (K M))    => (T R Q K M)
 →(RevAppend: L1=(R T), L2=(Q K M))  => (T R Q K M)↑
  →(RevAppend: L1=(T), L2=(R Q K M)) => (T R Q K M)↑
   →(RevAppend: L1=(), L2=(T R Q K M))=>(T R Q K M)↑
```

Рассмотренная функция `Append` является встроенной в диалектах Common Lisp и MuLisp.

1.6. S-выражения

Основной лисповской структурой данных является так называемое ***S-выражение***, обобщающее понятие списка и атома:

S-выражение ::= *атом* | (*S-выражение* ␣ . ␣ *S-выражение*)

Как и ранее, знак ␣ в приведённой БНФ обозначает пробел.

В простейшем случае лисповское выражение – это атом, а в более сложном – списочная структура, или ***точечная пара***, соединяющая два S-выражения. Как и введённое ранее понятие лисповского списка, понятие S-выражения определяется рекурсивно и допускает произвольное количество вложений точечных пар друг в друга, например:

```
((T . K) . NIL)
((A . B) .(C . (A . E )))
```

В общем случае разделяющая точка в точечной паре должна быть выделена пробелами слева и справа, чтобы не слиться со стоящим слева или справа атомом (т.к. обычно точка допускается в именах атомов). Если же слева или справа от точки стоит скобка, то соответствующий пробел может быть опущен.

Для S-выражений функции базового набора `car`, `cdr` и `cons` определены следующим образом. Пусть значением формы `E` является S-выражение $(v_1 . v_2)$, а v_1 и v_2 – некоторые S-выражения:

$E \Rightarrow (v_1 . v_2)$,

тогда (car E) $\Rightarrow v_1$, (cdr E) $\Rightarrow v_2$.

Если S-выражения e_1 и e_2 имеют значения v_1 и v_2 соответственно:

$e_1 \Rightarrow v_1$, $e_2 \Rightarrow v_2$,

то $(\text{cons } e_1\ e_2) \Rightarrow (v_1 . v_2)$.

Таким образом, аргументами функции cons могут быть произвольные лисповские выражения, в том числе – атомы.

Поскольку S-выражение – обобщение понятий атома и списка, все лисповские списки можно записать как S-выражения, используя *точечную запись*. Соответствие двух форм записи – *точечной* и обычной *списочной* – для одного и того же лисповского объекта можно установить на основе определения функции cons, в частности:

```
                (cons 'A NIL) => (A . NIL) ,
в то же время   (cons 'A NIL) => (A) ,
поэтому         (A) ≡ (A . NIL);
аналогично:     (cons 'A(cons 'B NIL)) => (A . (B . NIL))
и               (cons 'A (cons 'B NIL)) => (A B),
поэтому         (A B) ≡ (A . (B . NIL)).
```

Обобщая, получаем:

$(v_1 . (v_2 . (v_3 . \ldots (v_k . \text{NIL}) \ldots))) \equiv (v_1\ v_2\ v_3 \ldots v_k)$

где v_1, v_2, … v_k могут быть произвольными S-выражениями.

Заметим, что второй аргумент функции cons может быть произвольным S-выражением, в том числе атомом, отличным от NIL:

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

Тем самым второе выражение v_{k+1} в самой вложенной точечной паре S-выражения может оказаться произвольным атомом, и в этом случае эта точечная пара не может быть преобразована в списочную форму:

$(v_1 . (v_2 . (v_3 . \ldots (v_k . v_{k+1}) \ldots)))$

$\equiv (v_1\ v_2\ v_3 \ldots v_k . v_{k+1})$

Сформулируем правила перевода списочных выражений из точечной формы записи в обычную списочную:

1) Точка, за которой непосредственно следует открывающая скобка, может быть опущена вместе с этой открывающей скобкой и соответствующей закрывающей скобкой.
2) Точка, за которой непосредственно следует атом NIL, может быть опущена вместе с NIL.

Эти правила не всегда позволяют полностью избавиться от точечных пар: если в правой части некоторой точечной пары S-выражения стоит атом, отличный от NIL, то эта точечная пара остаётся.

В то же время обратные правила преобразования из списочной формы в точечную позволяют записать в точечных обозначениях любой список. Для этого:

1) Каждый пробел, разделяющий элементы списка верхнего уровня, заменяется точкой, за которой ставится открывающая скобка; соответствующая закрывающая скобка вставляется непосредственно перед скобкой, закрывающей верхний уровень списка.
2) Последний элемент `v` списка заменяется точечной парой вида`(v . NIL)`.
3) К подспискам (т.е. не атомарным элементам) преобразуемого списка применяются эти же правила преобразования.

Приведём примеры точечной и списочной записи для одних и тех же списочных выражений:

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

Как мы видим, списочная форма записи более простая, в то же время точечная форма – более общая, она позволяет записывать списочные структуры, которые нельзя выразить без точечных пар.

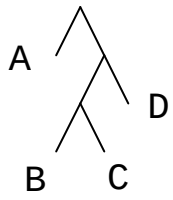
Отметим, что хотя S-выражения – списочные структуры более общего вида, чем списки, их рекурсивное определение (одна БНФ) проще и короче, чем рассмотренное нами ранее определение лисповского списка. Как мы увидим далее, более простое определение позволяет в ряде случаев строить более простые рекурсивные функции.

Ещё одно преимущество точечной записи – она упрощает понимание действия базовых функций обработки списков – пары селекторов `cdr`, `car` и конструктора `cons`. Действие функций `car` и `cdr` для S-выражения становится симметричным. Симметричен также результат функции `cons` по отношению к двум своим аргументам. Указанные три базовые функции взаимно обратны, что можно записать как

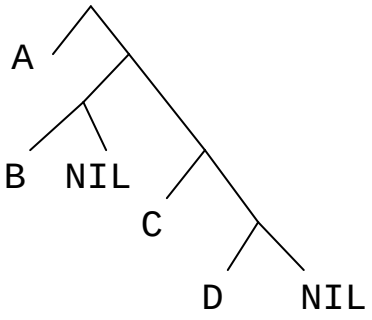
```
(cons (car S)(cdr S)) ≡ S
(car (cons S1 S2)) ≡ S1
(cdr (cons S1 S2)) ≡ S2
```

Произвольное S-выражение можно изобразить в виде бинарного дерева, у которого листья – атомы этого выражения, а поддеревья – точечные пары. Поэтому считают, что *логической структурой* S-выражения является бинарное дерево. Примеры таких деревьев показаны на Рис. 1.

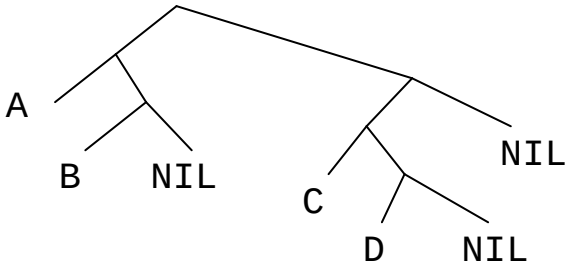
(A .((B . C). D))



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



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



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

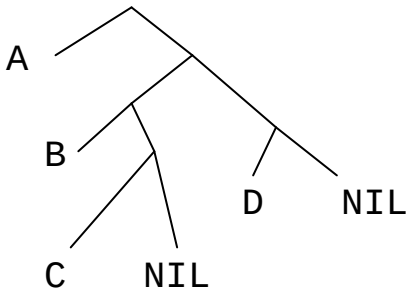


Рис.1. Древовидная структура S-выражений

1.7. Внутреннее представление S-выражений

Понятие S-выражения возникло в связи с определённым способом представления этих выражений в памяти компьютера. Рассмотрим поочерёдно внутреннее представление атомов и списочных структур.

Лисп-интерпретатор в ходе своей работы поддерживает специальную *таблицу символьных атомов* (*таблицу символов*), в которой хранится информация обо всех атомах, встретившихся в тексте интерпретируемой программы или используемых в качестве обрабатываемых данных. При обработке очередного символьного атома интерпретатор проверяет, занесён ли он в таблицу. Если атом уже есть в таблице, интерпретатор извлекает нужную информацию о нём из таблицы. Если же атом отсутствует в таблице (т.е. новый), соответствующая информация о нём записывается в таблицу.

В общем случае для каждого символьного атома в таблице хранится:

- его *внешнее имя* (т.е. строка, представляющая его при вводе/выводе);
- его *значение как аргумента функции* – фактическое значение, которое с ним связано при его использовании в роли формального параметра некоторой функции;
- его *функциональное значение* – определяющее выражение функции, при его использовании в качестве имени этой функции;
- *список свойств атома* (который будет рассмотрен в дальнейшем).

Перечисленные пункты можно считать разными видами значений одного атома. За исключением внешнего имени, эти значения могут отсутствовать. В общем же случае атом имеет несколько разных значений, и они независимы друг от друга. Это означает, что один и тот же символьный атом может быть использован и как имя функции, и как имя формального параметра. Нужная его интерпретация обеспечивается исходя из контекста его применения.

Например, можно определить функцию, один из формальных параметров которой совпадает с именем встроенной функции `car`:

```
(defun Func (z car)
   (cond  (z NIL)
          (T (car car))))          => FUNC
```

В этом определении атом `car` объявлен как формальный параметр функции `Func` и используется в её теле дважды: как имя функции и как её аргумент. При выполнении вызова функции `Func`:

```
(Func (cdr '(G)) '(G)) => G
```

с её параметрами `z` и `car` будут связаны соответственно значения `NIL` и `(G)`. Далее при вычислении тела функции для атома `car` сначала будет взято его определяющее выражение (т.к. он встречается в позиции имени функции), а затем – его значение как параметра (т.к. он употреблён в позиции аргумента функции).

Уточним, что в таблице символьных атомов для каждого атома хранятся не сами его значения (рассмотренных выше видов), а указатели (ссылки) на внутренние представления этих значений. В ряде диалектов Лиспа есть встроенные функции, использующие эти указатели, чтобы для заданного аргумента-атома выдать в качестве результата его разные значения: значение атома как параметра функции, функциональное значение, список свойств этого атома (в Common Lisp это соответственно функции `Symbol-value`, `Symbol-function`, `Symbol-plist`).

Внутренним представлением символьного атома фактически является указатель, ссылка на соответствующий элемент таблицы атомов. Уникальность всех значений, связанных с атомом, обеспечивается тем, что каждый известный лисп-интерпретатору атом хранится в таблице только один раз.

При представлении в памяти компьютера списочных структур каждой точечной паре `(A . B)` соответствует *списочная ячейка*, состоящая из двух полей. Эти поля, называемые соответственно *`car`-поле* и *`cdr`-поле*, содержат указатели на лисповские объекты `A` и `B` (атомы или списочные пары) – см. Рис. 2. Адрес `C` самой списочной ячейки служит указателем на эту точечную пару.

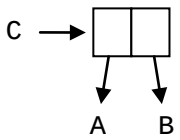


Рис. 2. Списочная ячейка

Внутренним представлением S-выражения, состоящего в общем случае из нескольких точечных пар, является совокупность взаимосвязанных списочных ячеек. Адрес *заглавной* (первой) списочной ячейки, соответствующей самой внешней точечной паре этого S-выражения, служит указателем на всю списочную структуру.

На Рис. 3 показано внутреннее представление следующих лисповских выражений:

```
(A (B C) D) ≡ (A .((B .(C . NIL)).(D . NIL)))
```
и `(A .((B . C). D))`.

При этом как указатели списочных структур используются адреса `x` и `y`: `x` ссылается на `(A (B C) D)`, а `y` – на `(A .((B . C). D))`. В каждой структуре число списочных ячеек равно числу точечных пар представляемого выражения.

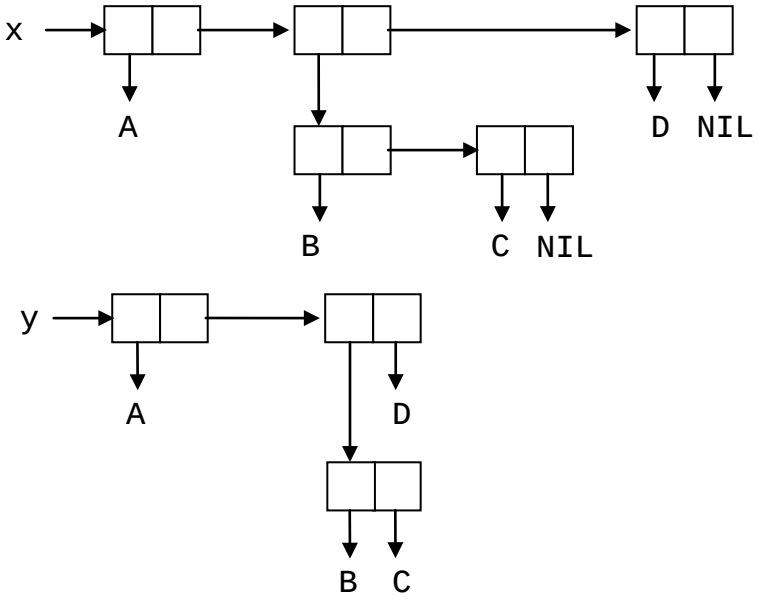


Рис 3. Внутреннее представление списочных структур

Реально в памяти компьютера списочной ячейке может соответствовать либо одна ячейка памяти, либо несколько ячеек.

Действие встроенных функций базового набора `car`, `cdr` и `cons` легко описывается в терминах внутреннего представления S-выражений:

- `car` выбирает содержимое (указатель) из `car`-поля заглавной списочной ячейки структуры, полученной в результате вычисления аргумента этой функции;
- `cdr` выбирает содержимое (указатель) из `cdr`-поля заглавной списочной ячейки структуры, являющейся значением аргумента функции;
- `cons` создаёт новую списочную ячейку, помещает в поля этой ячейки указатели лисповских объектов, являющихся значениями её аргументов, и возвращает в качестве своего значения указатель на созданную ячейку.

Рассмотрим, какие структуры могут образовать списочные ячейки в памяти компьютера. В общем случае из полей списочной ячейки указатели ведут к другим списочным ячейкам или к элементу таблицы атомов. На

списочную ячейку могут указывать car- и cdr-поля некоторых других списочных ячеек. Также возможны ссылки на списочные ячейки из таблицы атомов – для тех атомов, которые используются как формальные параметры некоторой функции и в текущий момент имеют значения. На Рис. 4 показана конфигурация списочных ячеек, возникающая после вычисления выражения

```
(append '(Q(R)()) (cons X (cons 'P X)))
```

если значением X было выражение (A . B).

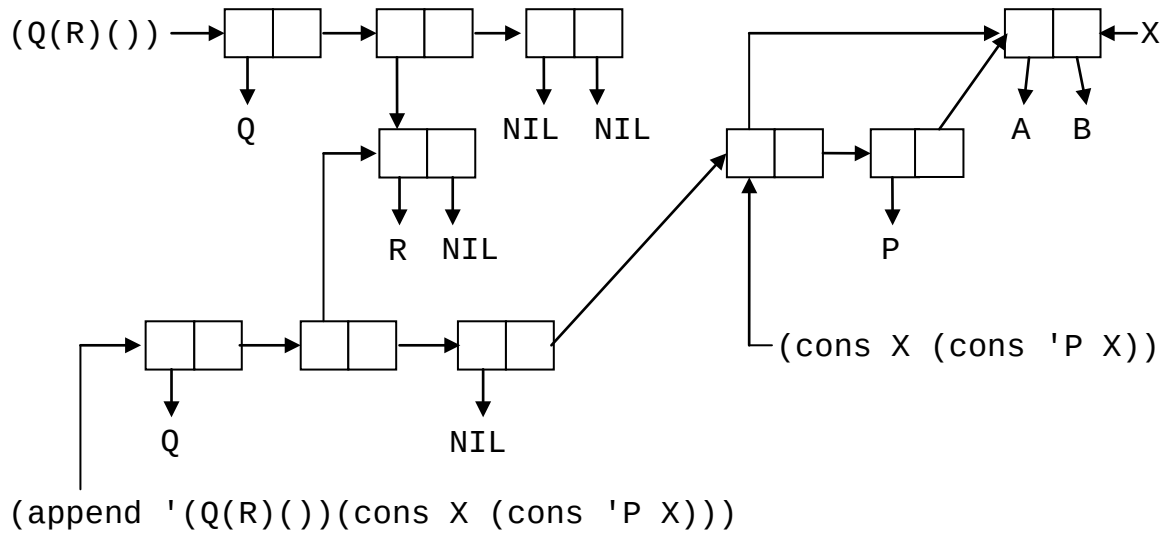


Рис. 4. Списочные структуры в памяти

При вычислении первого аргумента функции append будет создана списочная структура из четырёх ячеек, а при вычислении второго её аргумента – структура из трёх ячеек. При вычислении же тела этой функции будет создана копия верхнего уровня первого списка-аргумента, к которой и будет подсоединена вторая списочная структура (напомним, функция append при своей работе использует cons для подсоединения элементов первого списка-аргумента ко второму списку-аргументу).

Особенность вычислений с использованием встроенных функций car, cdr, cons и функций, определённых на основе базового набора, состоит в том, что при этом в памяти компьютера лишь создаются новые списочные ячейки и структуры на основе уже существующих, ранее же созданные структуры никогда не изменяются. Единственное, что может произойти с уже существующей списочной структурой – потеря указателя на неё. Например, после вычисления вызова функции append (см. Рис. 4) пропадает указатель на первый её аргумент – структуру (Q(R)()), и три списочные ячейки этой структуры становятся недоступны.

Неизменяемость уже созданных списочных структур представляет такое важное свойство функционального программирования, как отсутствие побочных эффектов вычислений [14, 15].

Изучая списочные структуры, показанные на Рис. 4, можно также заметить, что стрелки (указатели) могут сходиться на одной списочной ячейке, но не могут образовать замкнутый цикл. Это справедливо для всех списочных выражений, построенных на основе базового набора лисп-функций. Таким образом, *физической* (внутренней) *структурой* S-выражения является ациклический ориентированный граф.

Внутреннее представление S-выражений проясняет вопрос о способах их сравнения на равенство. Поскольку любой символьный атом единственным образом представлен в таблице символов, то для проверки равенства двух атомов достаточно сравнить их адреса в таблице. Именно так реализована встроенная функция `eq`, проверяющая равенство двух атомов. Другими словами, физическое равенство атомов означает и логическое их равенство.

Однако для списочных структур логическое и физическое равенство не одно и то же, поскольку логически одинаковые списки могут быть представлены различными списочными ячейками памяти. Подобная ситуация возникает, например, в результате работы функции `Copy`, копирующей в памяти заданное списочное выражение:

```
(defun Copy (x)
    (cond ((atom x) x)
          (T(cons (Copy (car x))(Copy (cdr x)) )) ))
```

Поскольку копирование элементов происходит на всех уровнях списочного выражения `x`, в результате в памяти создаётся его дубликат, причём указатель на этот дубликат отличен от указателя на исходное списочное выражение.

Таким образом, логическое равенство двух списков не может быть проверено путём сравнения указателей на списочные ячейки в памяти, необходима последовательная проверка содержимого соответствующих списочных ячеек сравниваемых списков. Такую проверку реализует встроенная функция-предикат `equal` с двумя вычисляемыми аргументами. Она выполняет сравнение на равенство двух произвольных S-выражений и вырабатывает соответственно `T` или `NIL`.

Для работы лисп-интерпретатора традиционно выделяются две, как правило, независимые области памяти:

- под таблицу символьных атомов;
- под списочные ячейки (*списочная память*).

В процессе работы часть списочной памяти уже использована интерпретатором под списочные ячейки, остальная часть называется *свободной списочной памятью*, и из неё берётся новая списочная ячейка при вызове функции `cons`. Было показано, что в ходе вычислений часть списочных ячеек перестаёт использоваться и ссылки на них могут теряться. Если не уничтожать ненужные списочные ячейки (а они могут занимать существенную часть памяти), рано или поздно область свободной списочной памяти будет исчерпана, что обнаружится при очередном обращении к функции `cons`. Для решения указанной проблемы лисп-интерпретаторы обычно используют специальную утилиту *сборки мусора* (т.е. списочных ячеек, которые недоступны на текущий момент вычислений) и подсоединения их к области свободной списочной памяти. Эта утилита вызывается либо по исчерпании свободной списочной памяти, либо периодически в ходе работы интерпретатора.

Во многих диалектах языка Лисп есть так называемые *структурно-разрушающие* встроенные функции (`rploca`, `rplocd`, `nconc` и др.), изменяющие содержимое уже существующих списочных ячеек. Как результат, уже созданные списочные структуры могут изменяться (разрушаться), а в графовой физической структуре списочных выражений могут возникать циклы. Из-за указанных побочных эффектов эти функции не относятся к строго функциональному программированию (и в данном пособии не рассматриваются). Их применение несёт в себе определенную опасность, хотя за счёт их побочных эффектов можно получить в ряде случаев более эффективные программы.

1.8. Лисп-программа

Явная особенность рассмотренного нами функционального ядра языка Лисп (и особенность функционального программирования вообще) состоит в том, что все операции над данными оформляются и записываются как функции, которые имеют значение, даже если их основное предназначение – осуществление некоторого побочного эффекта (например, определение новой функции). Более важно, однако, другое – подавляющее число функций побочного эффекта как раз не имеет. Именно это и представляет собой одну из главных отличительных особенностей функционального программирования. Кроме неё можно указать и другие особенности, связанные с характером использования переменных и управления последовательностью операций [14, 15].

В строго функциональном языке переменные скорее подобны математическим переменным – они употребляются, чтобы дать имена значениям аргументов некоторой функции, и связаны с этими значениями на протяжении всего вычисления тела этой функции. Обычный в

традиционных императивных языках программирования (Паскаль, С и др.) оператор присваивания переменной значения отсутствует в строго функциональных языках, и переменная может получить значение, только будучи формальным параметром некоторой функции, во время вычисления её вызова. По окончании вычисления вызова функции эта переменная-параметр автоматически теряет полученное значение.

В функциональном программировании нужная последовательность операций достигается вызовом функций в определённом порядке, т.е. суперпозицией функций. Передача данных между функциями выполняется через их аргументы и возвращаемые результаты (значения). Вместо традиционных условных операторов и циклов применяется соответственно условное выражение (`cond`) и более мощное средство – рекурсия.

Программа на функциональном языке является ничем иным, как набором запрограммированных функций. Еще одна особенность функциональной программы связана с тем, что в строго функциональном языке отсутствует понятие глобальной переменной и константы. Все переменные локальны, поскольку представляют собой формальные параметры функций с областью действия – телом функции.

Типичная лисп-программа включает:

- определения новых функций на базе встроенных функций и других функций, определённых в этой программе;
- вызовы этих новых функций для конкретных значений их аргументов.

Как правило, в начале программы ставятся определения функций, а затем – их вызовы.

В тексте лисп-программы можно использовать *комментарии*. Комментарии обычно начинаются точкой с запятой (`;`), при этом вся часть строки, следующая за этим символом, считается комментарием.

В качестве примера приведём программу, в которой определяются основные операции над комплексными числами. Сами комплексные числа представляются как двухэлементные списки чисел вида

(*действительная_часть мнимая_часть*),

например: список `(1.2 -5)` представляет число `1.2-5i`.

Сначала в программе определяются функции-операции над комплексными числами – сложения, умножения, вычитания, деления, сравнения на равенство. Затем эти функции используются для вычисления нескольких выражений, в которых используются комплексные числа. В программе даётся комментарий к каждой определяемой функции и вычисляемым затем выражениям.

```
; Программа для работы с комплексными числами
; комплексное число - список вида
; (действительная_часть мнимая_часть)

; функция сложения двух комплексных чисел
(defun AddCom(Com1 Com2)
   (cons (+ (car Com1)(car Com2))
         (cons (+ (cadr Com1)(cadr Com2)) NIL)))

; функция вычитания двух комплексных чисел
(defun SubCom(Com1 Com2)
   (cons (- (car Com1)(car Com2))
         (cons (- (cadr Com1)(cadr Com2)) NIL)))

; функция сравнения двух комплексных чисел
(defun EqCom(Com1 Com2)
   (and (= (car Com1)(car Com2))
        (= (cadr Com1)(cadr Com2))))

; функция умножения двух комплексных чисел
(defun MultCom(Com1 Com2)
   (cons (- (* (car Com1)(car Com2))
            (* (cadr Com1)(cadr Com2)))
      (cons (+ (* (car Com1)(cadr Com2))
               (* (cadr Com1)(car Com2)))  NIL)))

; функция деления комплексных чисел
(defun DivCom(Com1 Com2)
  (let ((z (+ (* (car Com2)(car Com2))
              (* (cadr Com2)(cadr Com2))) ))
    (cons (/ (+ (* (car Com1)(car Com2))
                (* (cadr Com1)(cadr Com2)))
             z)
      (cons (/ (- (* (cadr Com1)(car Com2))
                  (* (car Com1)(cadr Com2)))
               z)
            NIL ))))

; вычисление суммы чисел 2+3i и 1.5-8i
(AddCom '(2 3) '(1.5 -8))

; вычисление произведения чисел 3-i и 3+i
(MultCom '(3 -1) '(3 1))
```

Хотя входящие в эту программу вычисляемые выражения (определения функций и их вызовы) могут быть последовательно введены пользователем с клавиатуры, обычно лисп-программа хранится в текстовом файле (имеющем в диалекте MuLisp расширение `.lsp`). Лисп-интерпретатор загружает этот файл, последовательно вычисляя входящие в него лисповские выражения. По окончании обработки всего файла интерпретатор переходит к вычислению выражений, вводимых с клавиатуры.

Рассмотренный пример программы демонстрирует лёгкость и естественность расширения языка Лисп новыми операциями. За счёт этого в рамках одного диалекта Лисп можно реализовать конструкции другого диалекта. Например, в MuLisp файл `common.lsp` содержит определения ряда встроенных функций диалекта Common Lisp (отсутствующих в MuLisp), и после загрузки его лисп-интерпретатором появляется возможность использования этих функций.

В языке Лисп уже установленная связь имени и определяющего выражения функции реально используется только в момент вызова функции и вычисления её значения, согласно так называемой концепции *позднего связывания* [11]. Поэтому порядок определения функций в программе может быть произвольным, и возможны случаи, когда функция используется в определении другой функции, но сама определяется позднее. Главное, к моменту своего реального вычисления функция должна быть определена. В приведённом примере лисп-программы вызовы функций `AddCom` и `MultCom` (для вычисления суммы и произведения двух конкретных комплексных чисел) стоят после определения всех функций, но каждый из этих вызовов может быть поставлен раньше, сразу после определения соответствующей функции.

Для переменных позднее связывание означает динамическую типизацию и динамическое связывание со значением [11, 15].

В языке Лисп переменные не связываются предварительно с какими-то типами данных, тип данных любой переменной определяется во время исполнения программы – в тот момент, когда эта переменная получает значение определенного типа. Необходимая проверка этого типа тоже осуществляется динамически. *Динамическая типизация* даёт дополнительную гибкость использования переменных в лисп-программах, однако неизбежной платой за это является трата времени на проверку нужного типа на этапе исполнения.

Динамическое связывание, или динамический доступ к значениям переменных (применялось во всех ранних диалектах Лиспа, а ныне – в MuLisp) означает, что значение переменной берётся из ближайшей объемлющей вычисляемой функции, где есть переменная с таким именем.

Указанная стратегия связывания упрощает создание интерпретатора функционального языка. Однако она может приводить к нежелательным эффектам в случаях, когда тела вычисляемых функций содержат не только её параметры, но и другие переменные, значения которых необходимо брать из внешнего окружения. Простейший пример такой функции:

```
(lambda (x) (cond (x) (y)))
```

Нежелательные эффекты возникают в случаях конфликта имён таких переменных (в данном примере – переменная `y`); более подробно этот вопрос будет рассмотрен в разделе про функционалы. Заметим, что в текущем разделе во всех определениях лисп-функций в телах этих функций для вычисления их значения использовались только значения их формальных параметров.

Для исключения возможных проблем многие диалекты Лиспа применяют *статическое связывание*. В частности, в Common Lisp по умолчанию действует статическое (лексическое) связывание, но при необходимости для некоторых переменных можно установить динамическое связывание. В целом, статическое связывание позволяет проводить более эффективную компиляцию функциональных программ, однако проигрывает динамическому по времени исполнения программы. Различные диалекты и реализации языка Лисп отличаются наиболее значительно именно стратегией связывания.

Хотя в строго функциональном программировании на Лиспе глобальные переменные отсутствуют, есть некоторое исключение. К глобальным можно отнести связи, установленные для имён функций (а также связь атома с его списком свойств). Эти связи доступны в любой момент вычислений и могут быть использованы для организации в программе констант. К примеру, можно ввести и применять затем функцию без аргументов, выдающую в качестве своего значения нужную константу:

```
(defun PI () 3.14)
```

Такую функцию имеет смысл помещать в начало лисп-программы, аналогично тому, как в традиционных языках программирования определения констант идут в начале программы или блока.

2. Рекурсивное программирование

Рекурсия естественно возникает при обработке данных, имеющих рекурсивную внутреннюю структуру, в том числе – при обработке списков [6]. Использование рекурсии в таких случаях не только упрощает написание программ, но и делает их более понятными.

В данном разделе на примерах решения задач рассматриваются типичные приёмы построения рекурсивных программ и обсуждаются виды используемой при этом рекурсии.

2.1. Простая рекурсия

Будем говорить, что рекурсия *простая*, если рекурсивный вызов функции встречается не более одного раза в любой ветви условного выражения, служащего телом функции. Простая рекурсия уже использовалась нами при программировании функций `Length`, `Member`, `Append`, `RevAppend`.

В качестве ещё одного примера простой рекурсии приведём определение функции `Reverse` (обычно это встроенная функция). Она переворачивает свой список-аргумент, т.е. меняет порядок его элементов верхнего уровня на противоположный, например:

```
(Reverse '(A (B D) C)) => (C (B D) A).
```

```
(defun Reverse (L)
    (cond ((null L) NIL)
          (T (append (Reverse (cdr L))
                     (cons (car L) NIL) )) ))
```

В этом определении реализована следующая идея рекурсивного построения требуемого списка: он получается из перевёрнутого списка-хвоста исходного списка (т.е. списка без первого элемента верхнего уровня) присоединением к нему справа первого элемента. Для присоединения используется функция `append`, а не `cons`, т.к. последняя может добавлять элементы только в начало списка. Поскольку оба аргумента `append` должны быть списками, в качестве второго аргумента берётся одноэлементный список из первого элемента исходного списка.

Покажем ход вычисления функции `Reverse`. Теперь для иллюстрации динамики вычислений вместо схематичного изображения стека с отложенными функциональными вызовами будем просто по шагам переписывать вычисляемое выражение, последовательно заменяя в нём вычисленное на данном шаге подвыражение (функциональный вызов) на его значение. На каждом шаге вычисляется один функциональный вызов.

Номер шага вычисления указывается слева от выражения, полученного на этом шаге вычисления, а внутри выражения подчёркивается функциональный вызов, подлежащий вычислению на следующем шаге.

```
0) (Reverse: L=(A (B D) C) )
1) (append (Reverse: L=((B D) C)) '(A))
2) (append (append (Reverse: L=(C)) '((B D))) '(A))
3) (append (append (append (Reverse: L=()) '(C))
                   '((B D))) '(A))
4) (append (append (append () '(C)) '((B D))) '(A))
5) (append (append '(C) '((B D))) '(A))
6) (append '(C (B D)) '(A))
   => (C (B D) A)
```

Видно, что при движении по исходному списку от начала в конец (реализуемого с помощью функций `car` и `cdr`), очередные элементы исходного списка сохраняются (в виде одноэлементных списков) во втором аргументе отложенных рекурсивных вызовов функций `append`. Выполнение же этих отложенных вызовов и реальное построение перевёрнутого списка начинается, когда пройдены все элементы верхнего уровня исходного списка (т.е. список стал пуст). При этом подсоединение сохранённых элементов к результирующему списку происходит в порядке, противоположном их следованию в исходном списке.

Сложность рекурсивных вычислений можно оценивать такой характеристикой, как *глубина рекурсии* – количество сохранённых и обработанных в стеке вложенных вызовов рекурсивной функции. В рассмотренном примере функции `Reverse` глубина рекурсии равна длине реверсируемого списка.

Заметим, что в традиционных языках программирования (Паскаль, С и др.) для программирования повторяющихся вычислений используются циклы. Известно, что рекурсия – более мощное средство, чем любая циклическая конструкция, в то же время циклы требуют меньшего объёма памяти и, как правило, выполняются быстрее. Объём памяти, требуемой для выполнения цикла, можно оценить некоторой константой, не зависящей от числа шагов цикла (итераций). Для рекурсии необходимый объем памяти (стека) линейно зависит от глубины рекурсии.

В целом, рекурсия – более затратный по памяти и времени механизм (из-за расходов, связанных с организацией стека). Тем не менее, есть класс рекурсивных функций, вычисление которых можно выполнять без стека – это функции, представляющие так называемую *хвостовую рекурсию*. Современные лисп-интерпретаторы и компиляторы распознают случаи хвостовой рекурсии и оптимизируют вычисление таких функций.

Ключевая идея *оптимизации хвостовой рекурсии* – выполнение рекурсивной функции так, как если бы она была циклической (итерационной), т.е. её вычисление в строго фиксированном участке памяти, при этом рекурсивная функция как бы вызывает сама себя без стека.

Из приведённых нами примеров простой рекурсии к хвостовой рекурсии относятся только функции `Member` и `RevAppend` . Рекурсия считается хвостовой, если в теле функции рекурсивный её вызов не встречается в качестве аргумента никакой другой функции. В то же время при вычислении аргументов рекурсивного вызова допускаются обращения к другим функциям. В теле рассмотренной выше функции `Reverse` рекурсивный вызов стоял внутри обращения к `append`, являясь первым её аргументом, поэтому `Reverse` нельзя отнести к хвостовой рекурсии.

Ещё один пример хвостовой рекурсии – функция `FirstAtom`, выдающая первый слева атом заданного списочного выражения, например:

```
(FirstAtom '((((P C) D) V)) => P
```

Её определение:

```
(defun FirstAtom(L)
    (cond((null L) NIL) ;случай пустого списка
         ((atom(car L))(car L));первый элемент - атом
         (T(FirstAtom(car L))))) ;просмотр вглубь
```

В отличие от функции `Reverse`, выполняющей рекурсивный просмотр верхнего уровня исходного списка, называемый иногда рекурсией вширь, функция `FirstAtom` осуществляет просмотр списочного выражения вглубь, выбирая для дальнейшего рассмотрения его первый элемент (т.е. двигаясь всё время по левой ветви бинарного дерева, представляющего структуру этого выражения). Просмотр заканчивается, когда первый элемент оказался атомом, он и выдаётся в качестве результата.

Простая рекурсия используется также при программировании задач, требующих вложенных просмотров списков (вложенных циклов просмотра). Определим функцию-предикат `(Sublist L1 L2)` с двумя аргументами – одноуровневыми списками атомов. Функция `Sublist` выдаёт `T` в случаях, когда `L1` является подсписком `L2` (на верхнем уровне), например:

```
(SubList '(A B) '(Q E A S A B R)) => T
(SubList '(A B) '(Q B E A S A R)) => NIL
```

Для решения этой задачи кроме рекурсивной `SubList` потребуется вспомогательная рекурсивная функция `SubL`.

```
; функция SubList ищет в L2 подсписок L1
(defun SubList(L1 L2)
 (cond((null L2) NIL)   ; в L2 нет L1
      ((SubL L1 L2))    ; вызов SubL для проверки
      (T (SubList L1 (cdr L2))) ) ;сдвиг по L2

;вспомогательная функция SubL определяет,совпадает ли
; L1 с началом списка L2
(defun SubL(L1 L2)
 (cond((null L1)T);закончено сравнение L1 и начала L2
      ((null L2) NIL)  ;список L2 кончился раньше L1
      ((eql (car L1)(car L2)) ;сравнение 1-х элементов
        (SubL (cdr L1)(cdr L2)))));и других элементов
```

Функция SubList выполняет рекурсивный просмотр списка L2 для поиска в нём подсписка L1. Если начало списка L2 совпадает с L1 – для проверки этого вызывается функция SubL (на второй ветви cond), то L1 является подсписком L2. Если же результат этой проверки отрицателен, то происходит сдвиг по списку L2 для поиска нового возможного начала подсписка. Функция SubList реализует внешний цикл просмотра L2, а SubL – вложенный цикл одновременного просмотра и сравнения элементов списка L1 с элементами списка L2.

2.2. Косвенная рекурсия

Все рассмотренные нами рекурсивные функции относятся к *прямой рекурсии*, при которой в теле функции встречается один или несколько рекурсивных её вызовов. Более сложным случаем является неявная, *косвенная рекурсия*, при которой рекурсивный вызов не записан в определении функции, но возникает во время её вычисления.

В качестве примера рассмотрим лисп-программу для перевода традиционной (инфиксной) записи логического выражения в лисповскую (префиксную). Программа должна преобразовать, к примеру, выражение (X & (Y V TRUE)) в выражение (AND X (OR Y T)).

Синтаксис логических выражений можно описать следующими БНФ:

логическое выражение ::= *логический_операнд* & *логический_операнд* | *логический_операнд* V *логический_операнд*

логический_операнд ::= TRUE | FALSE | *логическая_переменная* | (*логическое_выражение*)

Для упрощения программы будем считать, что в исходной логической формуле логические переменные, константы и операции выделены

пробелами, а сама формула заключена в скобки (это позволяет обрабатывать её как лисповский список).

Будем строить нужное преобразование логической формулы на основе приведённых БНФ. Определим для этого две лисповские функции, преобразующие к нужному виду логическое выражение и логический операнд соответственно. Ветви каждой функции соответствуют альтернативам БНФ, описывающим обрабатываемые конструкции.

```
; функция перевода логического выражения
(defun TranExpr (X)
   (cond((eq (cadr X) '&) ; распознавание конъюнкции
           (list 'AND (TranOp (car X))
                      (TranOp (caddr X))))
         (T (list 'OR(TranOp(car X));случай дизъюнкции
                    (TranOp (caddr X)) )) ))

; функция перевода логического операнда
(defun TranOp (X)
  (cond((eq X 'TRUE) T)    ; перевод константы TRUE
       ((eq X 'FALSE) NIL) ; перевод константы FALSE
       ((atom X) X)  ;случай логической переменной
       (T(TranExpr X))) ;случай скобочного выражения
```

Функция `TranExpr` имеет две ветви для преобразования выражений с операцией конъюнкции и операцией дизъюнкции соответственно – при этом строится список-обращение к нужной операции с преобразованными операндами. Функция `TranOp`, преобразуя логические константы и скобочные выражения, в то же время не изменяет переменные.

Чтобы инициировать работу этих функций, необходимо обратиться к `Tranop`, например:

```
(TranOp '(X & (Y V TRUE))) =>(AND X (OR Y T))
```

Полученное выражение является лисповской формой и может быть вычислено в случаях, когда переменные `X` и `Y` имеют значения. Воспользуемся для этого функцией `eval`, пусть значения `X=T, Y=NIL`:

```
(let ((X T)(Y NIL))
     (eval(TranOp '(X & (Y V TRUE))))) => T .
```

При первом вычислении функцией `eval` своего аргумента произойдёт преобразование логической формулы из традиционной записи в лисповскую, а при втором вычислении – вычисление логического значения преобразованной формулы.

Итак, лисп-программа состоит из определений двух функций `TranOp` и `TranExpr`, а также обращения к функции `TranOp`,

запускающего преобразование логического выражения. Проиллюстрируем процесс преобразования на примере выражения (X & (Y V TRUE)), используя вышеописанную технику переписывания:

```
 0) (TranOp: X=(X & (Y V TRUE)))
 1) (TranExpr: X=(X & (Y V TRUE)))
 2) (list 'AND (TranOp: X=X)(TranOp: X=(Y V TRUE)))
 3) (list 'AND 'X (TranOp: X=(Y V TRUE)))
 4) (list 'AND 'X (TranExpr: X=(Y V TRUE)))
 5) (list 'AND 'X (list 'OR (TranOp: X=Y)
                            (TranOp: X=TRUE)))
 6) (list 'AND 'X (list 'OR 'Y (TranOp: X=TRUE)))
 7) (list 'AND 'X (list 'OR 'Y 'T))
 8) (list 'AND 'X '(OR Y T))
 => (AND X (OR Y T))
```

Видно, что функции TranOp и TranExpr реализуют рекурсивную обработку логического выражения, однако ни та, ни другая не соответствует определению прямой рекурсии, поскольку в их телах отсутствует вызов их самих. Значит, необходимо уточнить определение рекурсивной функции с учетом динамики вычислений.

Функция называется *рекурсивной*, если в процессе вычисления её тела может возникнуть обращение к ней самой. Это определение уже охватывает и случаи косвенной рекурсии, подобные рассмотренным функциям TranOp и TranExpr: первая функция в процессе вычислений обращается ко второй, а она, в свою очередь – к первой. Вообще говоря, может быть более сложная цепочка вызовов, включающая большее количество взаимосвязанных функций.

При косвенной рекурсии вызывающие друг друга функции часто называют *взаимно рекурсивными*.

2.3. Параллельная рекурсия

Параллельная рекурсия относится к прямой рекурсии, но представляет более сложный (чем простая рекурсия) её вид. Она обычно возникает, когда необходим анализ и обработка всех уровней некоторого списочного выражения, как в ранее приведённой функции Copy.

Рассмотрим этот вид рекурсии на примере построения функции-предиката MemberS от двух аргументов A и L. Функция определяет, содержится ли атом, являющийся значением аргумента A, в списочном выражении, являющемся значением L, хотя бы на одном из его уровней, и вырабатывает соответственно T или NIL:

```
(MemberS 'S '(A (X K) M (Z (D S)) V)) => T
(MemberS 'Y '(A (X K) M (Z (D S)) V)) => NIL
```

Эта функция обобщает действие ранее рассмотренной функции Member, проверяющей вхождение только на верхнем уровне списочного выражения.

Необходимый рекурсивный просмотр списка проще организовать на основе определения S-выражения. Согласно нему, следует рассмотреть два случая для S-выражения – атом и точечную пару. В случае атома его надо проверить на равенство значению A, а в случае точечной пары – продолжать поиск в выражениях, являющихся левой и правой частью этой пары (они получаются соответственно функциями car и cdr). Таким образом, просмотр исходного списочного выражения проводится до получения очередного его атома (лист бинарного дерева, представляющего обрабатываемое S-выражение). Следует также учесть, что атом-значение A входит в точечную пару, если он входит либо в левую, либо в правую её часть (в левое или правое поддерево). Соответствующее определение:

```
(defun MemberS (A L)
 (cond((atom L)(eql A L)) ; дошли до атома
      (T(or(MemberS A(car L));поиск в левом поддереве
           (MemberS A(cdr L));поиск в правом поддереве
)) ))
```

Отметим, что структура тела этой функции соответствует рекурсивной структуре обрабатываемого S-выражения: две её ветви соответствуют атому и точечной паре, причём во второй ветви находятся сразу два рекурсивных вызова (аргументы функции or). Говорят, что имеет место *параллельная рекурсия*, если в теле определяемой функции содержится вызов некоторой функции, не менее двух аргументов которой являются рекурсивными вызовами определяемой функции. В таких случаях говорят также, что рекурсия проводится и вширь, и вглубь списочного выражения.

Покажем ход вычисления функции MemberS, используя технику переписывания:

```
0) (MemberS: A=S, L=(A (Z (D S)) V))
1) (or (MemberS: A=S, L=A)
       (MemberS: A=S, L=((Z(D S))V)))
2) (or NIL (MemberS: A=S, L=((Z(D S))V))
3) (MemberS: A=S, L=((Z(D S))V)
4) (or (MemberS: A=S,L=(Z(D S)))(MemberS: A=S, L=(V)))
5) (or (or (Members: A=S, L=Z)
           (Members: A=S, L=((D S))))
       (MemberS: A=S, L=(V)))
```

```
6) (or (or NIL (MemberS: A=S, L=((D S))))
       (MemberS: A=S, L=(V)))
7) (or (MemberS: A=S, L=((D S))))(MemberS: A=S, L=(V))
8) (or (or(MemberS: A=S, L=(D S))(MemberS: A=S, L=()))
       (MemberS: A=S, L=(V)))
9) (or (or(or(MemberS: A=S, L=D)(MemberS: A=S, L=(S)))
          (MemberS: A=S, L=()))
       (MemberS: A=S, L=(V)))
10)(or (or(or NIL (MemberS: A=S, L=(S)))
           (MemberS: A=S, L=()))
       (Member: A=S, L=(v)))
11)(or (or(MemberS: A=S,L=(S))) (MemberS: A=S, L=()))
        (MemberS: A=S, L=(V)))
12)(or (or(or(MemberS: A=S, L=S)(MemberS: A=S, L=()))
          (MemberS: A=S, L=())) (MemberS: A=S, L=(V)))
13)(or (or(or T (MemberS: A=S, L=()))
           (MemberS: A=S, L=()))(MemberS: A=S, L=(V)))
14)(or (or T (MemberS: A=S, L=()))
       (MemberS: A=S, L=(V)))
15)(or T (MemberS: A=S, L=(V)))
 => T
```

В рассмотренной задаче можно было бы обойтись без параллельной рекурсии, если поместить проверку поддеревьев на разные ветви функции:

```
(defun MemberS (A L)
    (cond ((atom L)(eql A L))
          ((MemberS A (car L)) )
          ((MemberS A (cdr L)) )))
```

Возможен и другой вариант решения этой задачи – на основе рекурсивного определения лисповского списка, а не S-выражения:

```
(defun MemberSL (A L)
    (cond ((null L) NIL) ;случай пустого списка
          ((atom (car L)) ;анализ 1-го элемента L
            (cond ((eql (car L) A) T)
                  (T (MemberSL A (cdr L)) )))
          ((MemberSL A (car L))
          ((MemberSL A (cdr L)) )) ))
```

В этом варианте на второй ветви тела функции выполняется анализ первого элемента списка L, и поэтому нужна дополнительная (первая) ветвь, на которой проверяется непустота списка L. В итоге получается более длинное и более сложное для понимания определение.

Ещё один пример параллельной рекурсии, которую можно преобразовать в простую – это функция Equal, проверяющая равенство двух произвольных S-выражений и выдающая соответственно T или NIL:

```
(Equal '(A (S) D) '(A (S) D)) => T
(Equal '(A . B) '(A . B)) => T
(Equal '(R T) '(Q R T)) => NIL
```

Её определение с учётом структуры S-выражений:

```
(defun Equal (X Y)
   (cond ((atom X)(eql X Y))
         ((atom Y) NIL)
         (T (and (Equal (car X)(car Y))
                 (Equal (cdr X)(cdr Y)) )))
```

Основная идея проверки – одновременный проход по списочным выражениям X и Y вместе со сравнением соответствующих их частей. На ветвях функции последовательно рассматриваются все возможные случаи (X – атом; X не атом, а Y – атом; X и Y – не атомы). Как и в функции MemberS, для сравнения атомов используется встроенная функция eql, реализующая сравнение как символьных, так и числовых атомов. Заметим, что можно упростить последнюю ветвь cond, записав в качестве условия этой ветви первый аргумент функции and, а в качестве вычисляемого выражения – второй аргумент.

Рассмотрим теперь задачу, в которой нельзя обойтись без параллельной рекурсии. Запрограммируем функцию Number, подсчитывающую общее количество атомов в произвольном списочном выражении:

```
(Number '((A (X K) M ()(Z (D S)) V)) => 8
```

Для программирования подсчёта атомов на базе определения S-выражения желательно сформулировать действие этой функции в терминах бинарного дерева, представляющего обрабатываемое списочное выражение. Ясно, что должен вестись подсчёт листьев этого дерева, однако остаётся вопрос, учитывать ли при этом атомы NIL (они могут встречаться как в левых, так и в правых листьях дерева). Для простоты будем считать, что функция не будет учитывать атомы NIL. Поэтому в определении функции Number кроме двух ветвей, соответствующих двум случаям S-выражения (атом и точечная пара), появляется дополнительная, первая ветвь, соответствующая атомам NIL.

Таким образом, вторая ветвь тела функции учитывает атомы, отличные от NIL, а на рекурсивной ветви суммируется число атомов в левом и правом поддереве:

```
(defun Number (X)
    (cond ((null X) 0)
          ((atom X) 1)
          (T (+ (Number (car X))
                (Number (cdr X)) )) ))
```

Покажем процесс вычисления вызова функции `Number`:

```
0)  (Number: X=(A (X K) M ()) )
1)  (+(Number: X=A)(Number: X=((X K) M ()) ) )
2)  (+ 1 (Number: X=((X K) M ()) ) )
3)  (+ 1 (+ (Number: X=(X K) ) (Number: X=(M ()) )))
4)  (+ 1 (+ (+ (Number: X=X)(Number: X=(K) ))
            (Number: X=(M ()) )))
5)  (+ 1(+(+ 1 (Number: X=(K) ))(Number: X=(M ()) )))
6)  (+ 1 (+ (+ 1 (+ (Number: X=K)(Number: X=() )))
            (Number: X=(M ()) )))
7)  (+ 1 (+ (+ 1 (+ 1 (Number: X=() )))
            (Number: X=(M ()) )))
8)  (+ 1 (+ (+ 1 (+ 1 0))(Number: X=(M ()) )))
9)  (+ 1 (+ (+ 1 1)(Number: X=(M ()) )))
10) (+ 1 (+ 2 (Number: X=(M ()) )))
11) (+ 1 (+ 2 (+ (Number: X=M)(Number: X=(()) ))))
12) (+ 1 (+ 2 (+ 1 (Number: X=(()) ))))
13) (+ 1 (+ 2 (+ 1 (+ (Number: X=() )
                       (Number: X=() )))))
14) (+ 1 (+ 2 (+ 1 (+ 0 (Number: X=() )))))
15) (+ 1 (+ 2 (+ 1 (+ 0 0))))
16) (+ 1 (+ 2 (+ 1 0)))
17) (+ 1 (+ 2 1))
18) (+ 1 3)
    => 4
```

Так же, как и в случае с функцией `MemberS`, данное решение, основанное на рекурсивном определении S-выражения, проще и понятнее решения, полученного на основе определения лисповского списка, в котором отдельно анализируются случаи с первым элементом списка:

```
(defun NumberL (X)
  (cond ((null X) 0)
        ((null (car X)) (NumberL (cdr X)))
        ((atom (car X)) (+ 1 (NumberL (cdr X))))
        (T (+ (NumberL (car X))
              (NumberL (cdr X)) )) ))
```

Таким образом, при решении задач, требующих полного просмотра списочного выражения, опора на рекурсивное определение S-выражения приводит обычно к более простым и понятным функциям.

2.4. Накапливающий параметр

Типичным приёмом функционального программирования является введение и использование функции с дополнительным аргументом, выполняющим роль *накапливающего параметра*.

Рассмотрим задачу одновременного подсчёта суммы и произведения нескольких чисел. Эти числа задаются как элементы входного списка, а результатом вычисления должна быть списочная пара `(S . P)`, состоящая из суммы `S` этих чисел и их произведения `P`. Например:

```
(SumPr '(12 4 3 10)) => (29 . 1440)
```

Одним из решений этой задачи является функция `SumPr1`:

```
(defun SumPr1 (X)
  (cond((null X) (cons 0 1))
       (T(cons(+ (car X) (car (SumPr1 (cdr X))))
              (* (car X) (cdr (SumPr1 (cdr X)))))))))
```

На элементарной ветви этой функции строится точечная пара из суммы и произведения, соответствующих пустому списку чисел. Во второй ветви для пересчёта этой точечной пары используется параллельная рекурсия. Очевидна неэффективность этого решения: дважды вычисляется выражение `(car X)`, а также дважды вычисляется одно и то же рекурсивное обращение к `SumPr1` – такое дублирование рекурсивных вызовов функции даёт экспоненциальный эффект, увеличивая общее число рекурсивных вызовов с `n` (длина исходного списка) до 2^n.

Дублирования можно избежать, если сохранять промежуточные результаты вычисления – для этого необходимо завести дополнительную функцию или же использовать эквивалентную, но более удобную на практике конструкцию `let`, которая вводит локальные переменные для вычисляемых выражений. В нашем случае в `let` нужны две переменные: `Y` – для значения `(car X)`, и `Z` – для вычисленного рекурсивного обращения. В итоге задача решается простой рекурсией (вместо параллельной):

```
(defun SumPr2 (X)
   (cond ((null X) (cons 0 1))
         (T (let ((Y (car X))(Z (SumPr2 (cdr X)))
              (cons (+ Y (car Z))(* Y (cdr Z))) ))))
```

При этом для обработки списка длины n потребуется n рекурсивных обращений, а также n+1 обращений к функции cons для создания точечных пар, соединяющих сумму и произведение чисел, и n расщеплений этих пар функциями car и cdr.

Можно ещё сократить вычисления, если строить итоговую точечную пару только один раз, по завершении подсчёта суммы и произведения, а их значения накапливать в специальных параметрах-аргументах. Для этого потребуется вспомогательная функция от трёх аргументов, назовем её Accum. Эта функция выполняет рекурсивный процесс прохода по входному списку чисел X с соответствующим пересчётом суммы и произведения. Задача же основной функции SumPr3 – вызвать Accum, передав ей начальные значения её параметров:

```
(defun SumPr3 (X) (Accum X 0 1))
(defun Accum (X S P)
    (cond((null X) (cons S P));итоговая точечная пара
         (T (Accum (cdr X) (+ S (car X)) ;пересчёт
                           (* P (car X)) )) ))
```

Аргументы S и P рекурсивной функции Accum сохраняют промежуточные результаты вычислений, постепенно накапливая сумму и произведение чисел. Тем самым, используя два накапливающих параметра, удаётся избежать излишних рекурсивных вызовов и ненужных соединений и расщеплений точечных пар. Поскольку в Accum применяется хвостовая рекурсия, вычисления могут быть дополнительно оптимизированы лисп-интерпретатором.

В качестве следующего примера использования накапливающего параметра вновь рассмотрим функцию Reverse, переворачивающую свой список-аргумент:

```
(Reverse '(A (B D) C)) => (C (B D) A).
```

Предложенное ранее определение опиралось на функцию append:

```
(defun Reverse (X)
    (cond ((null X) NIL)
          (T (append (Reverse (cdr X))
                     (cons (car X) NIL) )) ))
```

Оценим вычислительную сложность этого решения, учитывая число вызовов функции cons (это более затратная операция по сравнению с car и cdr). Если n – длина исходного списка, то требуется n вызовов функции append. В свою очередь append вызывает себя рекурсивно m раз (где m – длина её первого аргумента-списка), копируя элементы своего первого аргумента с помощью операции cons:

```
(defun append(L1 L2)
   (cond ((null L1) L2)
         (T (cons (car L1)(append (cdr L1) L2)) )))
```

После каждого рекурсивного вызова функции Reverse длина аргумента для append уменьшается; имеем таким образом n вызовов append с длиной списка m, равной соответственно n-1, n-2, …, 1, 0. Общее число вызовов cons: $n+(n-1)+(n-2)+\ \ldots+1+0 = n(n+1)/2 = O(n^2)$.

Получающуюся квадратичную зависимость от длины n реверсируемого списка хотелось бы сократить до O(n), и этого можно достичь при введении накапливающего параметра.

Пусть в ходе вычислений накапливающий параметр Res сохраняет промежуточный результат – часть перевёрнутого списка. На каждом шаге рекурсии к Res будет подсоединяться слева очередной отщепленный от списка элемент, и по окончании обработки всех элементов в Res будет сформирован весь реверсированный список. Такое отщепление элементов списка X и их подсоединение к накапливающему параметру Res реализует вспомогательная рекурсивная функция Rev от двух аргументов: исходного списка и накапливающего параметра. Тогда задача основной функции ReverseA сводится к вызову Rev с начальным значением накапливающего параметра, равным пустому списку:

```
(defun ReverseA (X) (Rev X NIL)
(defun Rev (X Res) ; Res - накапливающий параметр
    (cond ((null X) Res) ;выдача перевёрнутого списка
          (T (Rev (cdr X)(cons (car X) Res)))))
```

Отметим, что как и в прошлом примере с SumPr3, вспомогательная функция Rev использует хвостовую рекурсию.

Покажем на примере ход реверсирования списка, используя технику переписывания вычисляемого выражения:

```
0) (ReverseA: X=(A (B D) C))
1) (Rev: X=(A (B D) C), Res=())
2) (Rev: X=((B D) C), Res=(cons 'A ()))
3) (Rev: X=((B D) C), Res=(A))
4) (Rev: X=(C), Res=(cons '(B D) '(A)))
5) (Rev: X=(C), Res=((B D) A))
6) (Rev: X=(), Res=(cons 'C '((B D) A)))
7) (Rev: X=(), Res=(C (B D) A))
  => (C (B D) A)
```

В отличие от функции Reverse, в данном случае перевёрнутый список начинает формироваться сразу (в параметре Res), сформированная его

часть передаётся на следующий шаг рекурсии. Количество обращений к функции cons равно длине n списка X. Таким образом, использование накапливающего параметра уменьшает сложность Reverse от квадратичной до линейной.

Рассмотренный приём с накапливающим параметром может быть использован для построения функции ReverseAll, реверсирующей заданный список на всех его уровнях, например:

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

Укажем сначала решение без накапливающего параметра, с использованием функции append и параллельной рекурсии:

```
(defun ReverseAll (X)
   (cond ((atom X) X)
         (T(append (ReverseAll (cdr X))
                   (cons (ReverseAll (car X)) NIL)))))
```

Попутно заметим, что в последней строке можно переставить местами обращения к функциям cons и ReverseAll.

Очевидно, более эффективно решение с накапливающим параметром, которое реализуют две взаимосвязанные функции:

```
(defun RevDepth (X)
        (cond ((atom X) X)
              (T (RevBroad X NIL)) ))
(defun RevBroad (Y Res)
   (cond ((null Y) Res)
         (T (RevBroad (cdr Y)
               (cons (RevDepth (car Y)) Res)))))
```

Обработка исходного списка и его подсписков идёт в двух направлениях – вглубь функцией RevDepth, а вширь – RevBroad. Эти две функции взаимно рекурсивны, а в RevBroad дополнительно использована простая рекурсия. В теле функции RevDepth для обработки неатомарных X при вызове RevBroad заводится накапливающий параметр (первоначально равный пустому списку). В теле же функции RevBroad в этот накапливающий параметр Res собираются (в обратном порядке) обработанные (реверсированные) элементы верхнего уровня списка Y.

Покажем это на примере:

```
0) (RevDepth: X=(A (B D) C))
1) (RevBroad: Y=(A (B D) C), Res=NIL)
2) (RevBroad: Y=((B D) C),
              Res=(cons (RevDepth: X=A) ()))
```

```
3) (RevBroad: Y=((B D) C), Res=(cons 'A ()))
4) (RevBroad: Y=((B D) C), Res=(A))
5) (RevBroad: Y=(C),
            Res=(cons (RevDepth: X=(B D)) '(A)))
6) (RevBroad: Y=(C),
       Res=(cons (RevBroad: Y=(B D), Res=()) '(A)))
7) (RevBroad: Y=(C),
      Res=(cons (RevBroad: Y=(D),
                   Res=(cons (RevDepth: X=B) ()))
                '(A)))
8) (RevBroad: Y=(C),
      Res=(cons (RevBroad: Y=(D), Res=(cons 'B ()))
                '(A)))
9) (RevBroad: Y=(C),
      Res=(cons (RevBroad: Y=(D), Res=(B))
                '(A)))
10)(RevBroad: Y=(C),
      Res=(cons (RevBroad: Y=(),
                   Res=(cons (RevDepth: X=D) '(B)))
                '(A)))
11)(RevBroad: Y=(C),
     Res=(cons (RevBroad: Y=(), Res=(cons 'D '(B)))
               '(A)))
12)(RevBroad: Y=(C),
     Res=(cons (RevBroad: Y=(), Res=(D B)) '(A)))
13)(RevBroad: Y=(C), Res=(cons '(D B) '(A)))
14)(RevBroad: Y=(C), Res=((D B) A))
15)(RevBroad: Y=(), Res=(cons (RevDepth: X=C)
                              '((D B) A)))
16)(RevBroad: Y=(), Res=(cons 'C '((D B) A)))
17)(RevBroad: Y=(), Res=(C (D B) A))
  => (C (D B) A)
```

2.5. Рекурсия более высокого порядка

Рассмотрим задачу *выравнивания* списка произвольной длины и глубины, т.е. удаления в нём всех внутренних скобок, без изменения состава и порядка следования входящих в список атомов:

```
(Flatten '((P (Q)) R (T))) => (P Q R T).
```

Результатом функции `Flatten` является одноуровневый список атомов.

Решение этой задачи параллельной рекурсией можно получить, опираясь на рекурсивное определение S-выражения:

```
(defun Flatten (X)
    (cond ((null X) NIL)
          ((atom X) (cons X ()))
          (T (append (Flatten (car X))
                     (Flatten (cdr X)) )) ))
```

Отметим, что при объединении результатов рекурсивных вызовов используется функция `append`, а не `cons`, поскольку необходимо соединить в один список два списка атомов, уже выровненных в результате рекурсивных обращений. Чтобы это было возможно, оба аргумента функции `append` должны быть списками, а значит, результатом `Flatten` всегда должен быть список. Поэтому во второй ветви функции, при обработке атома, он заключается в скобки. В то же время не должен заключаться в скобки пустой список (атом `NIL`) – для этого служит первая ветвь функции. Тем самым функция не сохраняет в результирующем списке атомы `NIL`:

```
(Flatten '((P NIL (Q)) R ()(T))) => (P Q R T).
```

Продемонстрируем ход вычислений функции `Flatten`:

```
0) (Flatten: X=((A(B))C))
1) (append (Flatten: X=(A(B)))(Flatten: X=(C)))
2) (append (append (Flatten: X=A)(Flatten: X=((B))))
           (Flatten: X=(C)))
3) (append (append '(A) (Flatten: X=((B))))
           (Flatten: X=(C)))
4) (append (append '(A)
                   (append (Flatten: X=(B))
                           (Flatten: X=())))
           (Flatten: X=(C)))
5) (append (append '(A)
                   (append (Flatten: X=(B))
                           (Flatten: X=())))
           (Flatten: X=(C)))
6) (append (append '(A)
                   (append (append (Flatten: X=B)
                                   (Flatten: X=()))
                           (Flatten: X=())))
           (Flatten: X=(C)))
```

```
7)  (append (append '(A)
                    (append (append '(B)
                                    (Flatten: X=()))
                            (Flatten: X=())))
            (Flatten: X=(C)))
8)  (append (append '(A)
                    (append (append '(B) NIL)
                            (Flatten: X=())))
            (Flatten: X=(C)))
9)  (append (append '(A)
                    (append '(B) (Flatten: X=())))
            (Flatten: X=(C)))
10) (append (append '(A) (append '(B) NIL))
            (Flatten: X=(C)))
11) (append (append '(A) '(B)) (Flatten: X=(C)))
12) (append '(A B) (Flatten: X=(C)))
13) (append '(A B) (append (Flatten: X=C)
                           (Flatten: X=())))
14) (append '(A B) (append '(C) (Flatten: X=())))
15) (append '(A B) (append '(C) NIL))
16) (append '(A B) '(C))
      => (A B C)
```

Поскольку рассмотренная функция использует для выравнивания высоко затратную функцию `append`, построим решение с накапливающим параметром. Оно включает две функции: основная функция `FlattenA` вызывает вспомогательную функцию `Flat`, устанавливая для неё начальное значение накапливающего параметра. `Flat` реализует собственно рекурсивный процесс выравнивания:

```
(defun FlattenA (X) (Flat X nil))
(defun Flat (Y Res) ) ;вспомогательная функция
   (cond((null Y)Res);Res - накапливающий параметр
        ((atom Y)(cons Y Res))
        (T (Flat (car Y) (Flat (cdr Y) Res)) )))
```

В ходе рекурсии `Flat` в параметре `Res` постепенно накапливает результат, рассматривая три возможных случая. Если `Y` – пустой список, то `Res` не меняется. Если аргумент `Y` – атом, отличный от `NIL`, то он непосредственно заносится в `Res`. Если же `Y` – неатомарное выражение (точечная пара), его можно рассматривать как бинарное дерево, и сначала рекурсивно выравнивается правое поддерево (полученное операцией `cdr`), при этом используется параметр `Res`: `(Flat (cdr Y) Res)`.

Полученный результат (выровненное правое поддерево) используется как накапливающий параметр для выравнивания левого поддерева (полученного операцией `car`): `(Flat (car Y)(Flat(cdr Y) Res))`. Именно такой порядок рекурсивных вызовов функции `Flat` обеспечивает сохранение исходного порядка атомов в выровненном списке.

В данном случае один рекурсивный вызов стоит внутри другого рекурсивного вызова, демонстрируя ещё один, более сложный вид рекурсии – *рекурсию более высокого порядка*. Точнее, `Flat` – это рекурсия первого порядка, поскольку порядок отсчитывается по числу вложенных рекурсивных вызовов.

Покажем на примере вычисление функции `FlattenA`:

```
0) (FlattenA: X=((A(B))C))
1) (Flat: Y=((A(B))C), Res=())
2) (Flat: Y=(A(B)), Res=(Flat: Y=(C), Res=()) )
3) (Flat: Y=(A(B)), Res=(Flat: y=C,
                             Res=(Flat: Y=(), Res=())))
4) (Flat: Y=(A(B)), Res=(Flat: Y=C, Res=()) )
5) (Flat: Y=(A(B)), Res=(C) )
6) (Flat: Y=A, Res=(Flat: Y=((B)), Res=(C)) )
7) (Flat: Y=A, Res=(Flat: Y=(B),
                          Res=(Flat: Y=(), Res=(C)) ))
8) (Flat: Y=A, Res=(Flat: Y=(B), Res=(C)) )
9) (Flat: Y=A, Res=(Flat: Y=B,
                          Res=(Flat: y=(), Res=(C))))
10)(Flat: Y=A, Res=(Flat: Y=B, Res=(C)) ))
11)(Flat: Y=A, Res=(B C))
  => (A B C)
```

Заметим, что по окончании обработки очередного подсписка и возврате на предыдущий уровень вложенности списка туда передаётся список-параметр `Res`, в котором уже накоплены атомы обработанных подсписков.

В отличие от функции `Flatten`, общее количество обращений к операции `cons` при работе функции `FlattenA` минимально, т.е. равно числу атомов обработанного списочного выражения. При этом для обработки одного и того же выражения потребовалось меньшее число шагов вычисления (11 против 16) и меньшее количество вложенных функциональных вызовов (3 против 5).

В качестве ещё одного примера рекурсии более высокого порядка приведём определение функции `Collect`, собирающей все различные атомы в заданном списочном выражении:

```
(collect '(A(S A D)((Z(S))X C))) => (A D Z S X C)
```

Результирующий список – список атомов без повторений, т.е. множество.

Как и в задаче выравнивания списка, `Collect` использует вспомогательную функцию с накапливающим параметром `Res`, но добавление атома в `Res` происходит только в том случае, когда этого атома там нет:

```
(defun Collect (X) (Col X nil))
(defun Col (Y Res) ;вспомогательная функция
    (cond ((null Y) Res)
          ((atom Y) (cond ((member Y Res)Res)
                          (T(cons Y Res))))
          (T (Col (car Y) (Col (cdr Y) Res)) ) ))
```

2.6. Задачи на программирование рекурсии

Простая рекурсия

1. Составить функцию `(RemoveLast L)`, удаляющую из списка последний элемент. Например:
   `(RemoveLast '(A (S D) E (Q))) => (A (S D) E)`
2. Определить функцию-предикат `(OneLevel L)`, которая проверяет, является ли список-аргумент одноуровневым списком:
   `(OneLevel '(A B C)) => T,`
   `(OneLevel '((A) B C)) => NIL`
3. Запрограммировать функцию `(Bubl N A)` с двумя вычисляемыми аргументами – числом `N` и атомом `A`. Функция строит список глубины `N`; на самом глубоком уровне элементом списка является `A`, а на любом другом уровне список состоит из одного элемента. Например: `(Bubl 3 5)=>(((5)))`.
4. Определить функцию `(LastAtom L)`, выбирающую последний от начала списка (невзирая на скобки) атом списка:
   `(LastAtom '(((5)A))) => A`
5. Составить функцию `(Delete L X)`, удаляющую из списка `L` на его верхнем уровне
   а) первое вхождение значения `X`;
   б) все вхождения значения `X`.
6. Написать функцию `(Remove2 L)`, удаляющую из списка каждый второй элемент верхнего уровня:
   `(Remove2 '(A B C D E)) => (A C E).`
7. Составить функцию`(Pair L)`, которая разбивает элементы списка `L` на точечные пары, например:
   `(Pair '(A B C D E)) => ((A . B)(C . D)(E))`

8. Определить функцию(Mix1 L1 L2), которая образует новый список, чередуя элементы заданных:
   (Mix1 '(A B C) '(Z X)) => (A Z B X C)
9. Определить функцию(Mix2 L1 L2), которая образует список точечных пар элементов, взятых последовательно из заданных списков L1 и L2, например:
   (Mix2 '(A B C) '(Z X))=> ((A . Z)(B . X)(C))
10. Написать функцию (Elem N L), которая выдаёт N-тый элемент верхнего уровня списка L. Если длина списка меньше N, то функция возвращает NIL.
11. Составить функцию (Position X L), возвращающую порядковый номер значения X в списке L, либо 0, если выражение X не встречается в списке на верхнем уровне.
12. Запрограммировать функцию (RevBr L), которая переворачивает свой аргумент-список атомов и разбивает его на уровни; количество уровней равно количеству элементов исходного списка:
    (RevBr '(A B C)) => (((C) B) A)
13. Определить функцию (RightBr L), которая преобразует свой аргумент – список атомов, разбивая его на уровни. Количество уровней равно количеству атомов, на самом глубоком уровне находится последний атом исходного списка:
    (RightBr '(A B C)) => (A (B (C)))
14. Определить функцию (LeftBr L), которая делает преобразование исходного списка атомов, подобное RightBr, но на самом глубоком уровне находится первый атом исходного списка:
    (LeftBr '(A B C)) => (((A) B) C)
15. Составить функцию (RightBrOut L), являющуюся обратной к функции RigthBr: (RightOut '(A(B(C)))) => (A B C)
16. Составить функцию (LeftBrOut L), обратную к функции LeftBr:
    (LeftOut '(((A)B)C)) => (A B C)
17. Написать функцию (Fact N) с одним аргументом – натуральным числом N. Функция строит списочное выражение, являющееся записью произведения натуральных чисел от 1 до N:
    (Fact 4)=>(1 * 2 * 3 * 4) или (4 * 3 * 2 * 1).
18. Определить функцию (MakeSet L), которая преобразует свой аргумент L – одноуровневый список атомов во множество, исключая в нём повторяющиеся элементы.

19. Запрограммировать основные операции с множествами: пересечение, объединение, разность множеств. Множества представляются как списки атомов без повторений. Составить также функции-предикаты для проверки равенства множеств и вхождения одного множества в другое.

Параллельная рекурсия и рекурсия высшего порядка

1. Определить функцию `(Depth L)`, вычисляющую глубину списка `L`, т.е. максимальное количество уровней в нём. Например:
```
(Depth '(((A (5) 8) B (K))(G (C)))) => 4
```
2. Составить функцию `(Subst A L E)`, заменяющую в произвольном списочном выражении `L` на всех его уровнях все вхождения атома `A` на выражение `E`. Например:
```
(Subst 'Q '(Q (B (Q)) C ((Q) 8))) '(A Z)) =>
             ((A Z) (B ((A Z))) C (((A Z)) 8)))
```
3. Определить функцию-предикат `(OnlyZ L)`, которая вырабатывает `T` в том случае, если в списке `L` на всех его уровнях встречается только атом `Z`, иначе вырабатывает `NIL`. Например:
```
(OnlyZ '((Z (Z())Z)()Z) )=> T,
(OnlyZ '((Z (Z())8)()Z) )=> NIL.
```
4. Написать функцию `(Trans N S)`, которая упрощает структуру списочного выражения `S`, заменяя в нём все списочные элементы, находящиеся на уровне `N` (`N>=1`), на атом “#”. Например:
```
(Trans 2 '(((A(5)8)B(K))(G(C)))) => ((# B #)(G #))
```
5. Определить функцию`(Level N S)`, которая строит список из элементов списочного выражения `S`, находящихся на уровне `N` (`N>=1`), например:
```
(Level 2 '(((A (5) 8) B) 7 (G ((())))))
                        => ((A (5) 8) B G (NIL))
```
6. Составить функцию `(Deep S)`, вырабатывающую атом списочного выражения `S`, который находится на наиболее глубоком уровне (если таких атомов несколько, выбирается любой). Например:
```
(Deep '(A (B (A)) C ((D) 8))) => A
```
7. Определить функцию `(Freq S)`, которая для каждого атома, входящего в списочное выражение `S`, вычисляет частоту его вхождения в `S` и выдает список пар вида: *атом – его частота*, например:
```
(Freq '(A (B (A)) C ((A) 8)))
                  => ((A 3)(B 1)(C 1)(8 1))
```

3. Функционалы

Функционалы не усиливают вычислительную мощность языка Лисп, они усиливают его выразительные возможности, позволяя в сжатой форме описать повторяющиеся вычисления [15]. Некоторые задачи очень естественно решаются с помощью функционалов, приводя к компактным, и в то же время мощным программам.

3.1. Понятие функционала

До сих пор мы рассматривали лисповские функции, в качестве аргументов которых выступали обрабатываемые данные – атомы и списочные выражения. В языке Лисп данные и функции имеют одинаковое синтаксическое представление, поэтому в нём достаточно естественно функция может быть аргументом или результатом вычисления другой функции. Последняя при этом называется *функционалом*, или *функцией высших порядков*, а функции, используемые как ее аргументы – *функциональными аргументами*.

Функции высших порядков возникают довольно часто при решении разных классов задач, к примеру, при вычислении определённого интеграла, когда параметрами решаемой задачи являются пределы интегрирования и подынтегральная функция.

Рассмотрим пример определения функционала. Сначала составим функцию `Add1List`, работающую с одноуровневым списком чисел и увеличивающую каждое из чисел списка на единицу:

```
(defun Add1List(X)
  (cond ((null X) NIL)
        (T (cons (add1 (car X))(Add1List (cdr X))))))
```

Пример работы этой функции:

```
(Add1List '(5 17 0 23)) => (6 18 1 24)
```

Функция `Add1List` последовательно просматривает исходный список-аргумент и, увеличивая (с помощью функции `add1`) на `1` каждый элемент-число, строит результирующий список. По такой схеме рекурсивной обработки элементов верхнего уровня списка можно составить и другие лисповские функции. Например, для получения функции, уменьшающей на `1` каждое из чисел-элементов исходного списка, в записи функции `Add1List` потребуется лишь заменить функцию `add1` на `sub1`, уменьшающую свой аргумент на единицу, а также изменить имя функции, взяв, например, `Sub1List` вместо `Add1List`:

```
(defun Sub1List (X)
  (cond ((null X)    NIL)
        (T (cons(sub1 (car X))(Sub1List (cdr X))))))
```

Попробуем определить вместо двух одну, более общую функцию `FList`, которая применяла бы нужную операцию ко всем элементам исходного списка чисел и составляла бы из полученных значений список-результат. Эта функция будет иметь два аргумента – применяемую операцию `F` и исходный список `X`, т.е. будет функционалом. Определим `FList` как обычную функцию с вычисляемыми аргументами, поэтому при ее вызове с нужными нам операциями последние должны квотироваться: `(FList 'add1 X)` и `(FList 'sub1 X)`. Эти два функциональных вызова должны быть эквивалентны соответственно вызовам функций `(Add1List X)` и `(Sub1List X)`, в частности:

```
(FList 'add1 '(5 17 0 3)) => (6 18 1 4)
(FList 'sub1 '(5 17 0 3)) => (4 16 -1 2)
```

Кажущееся очевидным определение `FList`, получающееся из `Add1List` заменой конкретной операции на формальный параметр `F`:

```
(defun FList (F X)
   (cond ((null X) NIL)
         (T (cons (F (car X)) (FList F (cdr X))))))
```

всё же неверно, и в этом легко убедиться, попробовав вычислить `(FList 'add1 X)` или `(FList 'sub1 X)`.

Причина ошибки связана с особенностью вычислений функциональных вызовов в языке Лисп. Любой вычислимый в Лиспе список должен иметь в качестве своего первого элемента либо символьный атом – имя функции (встроенной или уже определённой), либо лямбда-выражение, определяющее некоторую безымянную функцию. Иными словами, первый элемент вычисляемого списка (функционального вызова) никогда не вычисляется, он должен быть задан явно. В случае, когда первый элемент – это атом (имя функции), лисп-интерпретатор использует связанное с этим символьным атомом определяющее выражение функции.

Поэтому при вычислении тела функции `FList`, точнее, при вычислении выражения `(F (car X))` символ `F` будет трактоваться как имя функции, а такая функция не была определена. Чтобы исправить ошибку, необходимо при вычислении этого выражения использовать вместо символа `F` его значение как фактического параметра. Применяя встроенную функцию `eval`, это можно сделать, например, так:

```
(defun FList (F X)
  (cond ((null X) NIL)
```

```
(T (cons (eval (list F (list 'quote (car X))))
         (FList F (cdr X))))))
```

При вычислении первого аргумента функции `cons` сначала формируется список-обращение к нужной функции `F` (за счёт первого вычисления функцией `eval` своего аргумента), а затем сформированное обращение вычисляется (за счёт второго вычисления функцией `eval` своего аргумента). Чтобы при этом аргумент функции `F` не вычислялся дважды, в ходе первого этапа вычислений строится форма для его последующего квотирования:

`(list 'quote(car X)) => (quote` *`первый_элемент_X`*`)`.

Функции `add1` и `sub1` являются встроенными в диалекте MuLisp, но их нет в Common Lisp. Однако и в этом диалекте легко применить созданный функционал, используя вместо имени функции её определяющее выражение:

```
(FList '(lambda(x)(+ x 1)) '(5 17 0 3)) => (6 18 1 4)
(FList '(lambda(x)(- x 1)) '(5 17 0 3)) =>(4 16 -1 2)
```

При вычислении первого аргумента этих вызовов будет получено лямбда-выражение, которое будет использовано в теле функционала `FList` вместо формального параметра `F`.

Таким образом, функциональный аргумент функционала `FList` вычисляется, и его значением должно быть либо имя функции от одного аргумента (вычисляемого), либо её определяющее выражение:

```
(FList 'list '(5 17 0 3)) => ((5)(17)(0)(3))
(FList '(lambda(X)(cons X NIL)) '(5 17 0 3))
                                  => ((5)(17)(0)(3))
```

Рассмотренная нами функция `FList` встроена во многие диалекты Лиспа (в том числе – в MuLisp и Common Lisp) под именем `mapcar`. В большинстве диалектов есть целый набор встроенных функционалов, облегчающих программирование, включая создание и применение новых функционалов.

3.2. Встроенные функционалы

Встроенные функционалы обычно включают группу применяющих и группу отображающих функционалов.

Применяющие функционалы позволяют применить свой функциональный аргумент (функцию) к заданным её аргументам.

Функционал `apply` является обычной функцией с двумя вычисляемыми аргументами, обращение к ней имеет вид: `(apply F L)`,

где F – функциональный аргумент и L – список: L =>(p_1 … p_n), $n \geq 0$, рассматриваемый как список фактических параметров для F. Значение функционала – результат применения F к этим фактическим параметрам, например:

```
(apply '* '(2 5)) => 10
(apply 'cons '(2 (5))) => (2 5)
(apply 'list '(A B)) => (A B)
(apply 'apply '(* (2 5))) => 10
```

Отметим, что в приведённых примерах квотирование функционального аргумента является обязательным, поскольку функционал apply вычисляет значения всех своих аргументов. Это вычисление действительно необходимо в случаях, когда функциональный аргумент не может быть задан непосредственно, поскольку не известен заранее и определяется в ходе вычислений.

Функционал funcall – особая функция с вычисляемыми аргументами, обращение к ней: (funcall F e_1 … e_n), $n \geq 0$. Её действие аналогично apply, отличие состоит в том, что аргументы применяемой функции F задаются не списком, а по отдельности. Например:

```
(funcall '* 2 5) => 10
(funcall 'cdr '((B) A)) => (A)
(funcall 'list '(A B)) => ((A B))
(funcall 'apply 'list '(A B)) => (A B)
```

Поскольку функционал apply – обычная функция, он может быть реализован на основе базового набора встроенных функций Лиспа и функции defun по схеме:

```
(apply F E) ≡ (eval (list F '(car E) '(cadr E) … ))
```

Его определение:

```
(defun apply(F L)(eval (cons F (Qu L))))
(defun Qu(L) ;вспомогательная функция квотирования
  (cond ((null L) NIL)
        (T (cons (list 'quote (car L))
                 (Qu (cdr L))))))
```

В этом определении используется Qu – вспомогательная функция квотирования (блокировки вычисления) всех аргументов функции F, т.к. их число заранее неизвестно.

В отличие от apply, функционал funcall может иметь произвольное количество аргументов (это особая функция), и для его реализации по схеме:

```
(funcall F e1 … en) ≡ (eval (list F 'e1 'e2 … 'en))
```

требуются уже дополнительные средства определения особых функций, которые будут рассмотрены в дальнейшем.

В группу *отображающих функционалов* входят функции `mapcar` и `maplist`. Их имена имеют префикс `map` (*mapping* – отображение), поскольку их действие – отображение списка-аргумента в список-результат за счёт применения заданной функции к элементам исходного списка.

Обращение к `mapcar` обычно имеет вид: `(mapcar F L)`, при этом оба аргумента вычисляются. В этом случае функционал последовательно применяет свой функциональный аргумент `F` (функцию от одного аргумента) к элементам списка `L`, которые извлекаются из `L` функцией-селектором `car` (поэтому имя `car` входит в имя функционала), и возвращает список из полученных значений. Схематично результат может быть записан как

`((F (car L)) (F (cadr L)) (F (caddr L)) …).`

Покажем на примерах действие этого функционала (напомним, функция `length` вычисляет количество элементов на верхнем уровне своего списка-аргумента):

```
(mapcar 'length '((A B) (C) (D E ()))) => (2 1 3)
(mapcar 'list '(A S D F)) => ((A)(S)(D)(F))
```

Описанный функционал `mapcar` может быть легко запрограммирован на основе функционала `funcall`:

```
(defun mapcar (F L)
   (cond ((null L) NIL)
         (T (cons (funcall F (car L))
                  (mapcar F (cdr L))))))
```

Другой отображающий функционал `maplist` с обращением `(maplist F L)` последовательно применяет свой функциональный аргумент `F` к списку-аргументу `L` и его хвостовым частям (полученным из `L` путём отбрасывания первого элемента, первых двух элементов и т.д.) и возвращает список из вычисленных значений. Как и `mapcar`, функционал `maplist` является обычной функцией, вычисляющей свои аргументы. Схематично действие этого функционала можно записать как

`((F l) (F (cdr L)) (F (cddr L)) …)`

Приведём примеры его использования:

```
(maplist 'reverse '(X Y Z))  => ((Z Y X) (Z Y) (Z))
(maplist 'list '(A S D F))
               => (((A S D F))((S D F))((D F))((F)))
```

Определение функционала maplist на основе funcall:

```
(defun maplist (F L)
    (cond ((null L) NIL)
          (T (cons (funcall F L)
                   (maplist F (cdr L))))))
```

Без использования funcall этот функционал можно определить так:

```
(defun maplist (F L)
   (cond ((null L) NIL)
         (T (cons (eval (cons F
                     (cons (cons 'quote L)NIL)))
                  (maplist F (cdr L))))))
```

В общем случае функционалы mapcar и maplist допускают функциональный аргумент с числом параметров $N \geq 1$. Вызов функционалов имеет вид: (mapcar FN L_1 L_2 … L_N) и (maplist FN L_1 L_2 … L_N)

где FN – функция от N аргументов. Функционал mapcar применяет эту функцию к первым элементам списков L_1 … L_N, затем ко вторым элементам и т.д. Функционал maplist применяет FN к спискам L_1…L_N, затем к (cdr L_1)…(cdr L_N), далее к (cddr L_1)…(cddr L_N) и т.д. Из полученных при этом значений составляется список, который является значением функционалов. Например:

```
(mapcar 'cons '(1 2 3) '((A)(S)(D))) =>
                              ((1 A)(2 S)(3 D))
(mapcar 'list '(A S D) '(1 2 3) '(Z X C)) =>
                              ((A 1 Z)(S 2 X)(D 3 C))
(maplist 'cons '(1 2 3) '(A S D)) =>
                   (((1 2 3) A S D)((2 3) S D)((3) D))
(maplist 'list '(A S D) '(1 2 3) '(Z X C)) =>
(((A S D)(1 2 3)(Z X C))((S D)(2 3)(X C))((D)(3)(C)))
```

3.3. Замыкание функционального аргумента

Рассмотрим проблему, которая может возникнуть, если в качестве функционального аргумента берётся функция, в теле которой есть переменные, не входящие в число её формальных параметров. Такой функцией является, к примеру, функция с определяющим выражением

(lambda (X) (cons L X)).

Именно такие переменные могут составлять проблему при вычислениях.

Переменная называется *свободной* для некоторой функции, если она используется в теле этой функции, но не входит в число её формальных

параметров (аргументов), и называется *связанной* в противном случае. Одна и та же переменная может быть одновременно связанной для одной функции, и свободной – для другой. Например, переменная L связана для функции Ex, но свободна для безымянной функции, используемой в теле Ex как функциональный аргумент:

```
(defun Ex (L Y)(funcall '(lambda (X)(cons L X)) Y))
```

Действие Ex: `(Ex '(1 2) '(A S)) => ((1 2) A S)` .

Рассмотрим подробнее вычисление функциональных аргументов со свободными переменными. Будем использовать данное ранее определение функционала mapcar:

```
(defun mapcar (F L)
   (cond ((null L) NIL)
         (T (cons (funcall F (car L))
                  (mapcar F (cdr L))))))
```

а также определение функции MapCons, которая использует mapcar для того, чтобы образовать двухэлементные списки из заданного элемента L и элементов заданного списка Z:

```
(defun MapCons (L Z)
       (mapcar '(lambda (X) (list L X)) Z))
```

Согласно этим определениям функций, при вызове MapCons ожидается результат: `(MapCons 7 '(A(B)C)) => ((7 A) (7 (B)) (7 C))`
Однако реально в результате вычислений будет получено:

```
(MapCons 7 '(A (B) C))
            => (((A (B) C) A) ((B) C) (B)) ((C) C))
```

Причина этого связана с переменной L, которая является связанной в теле mapcar и MapCons, но свободной в теле безымянной функции, являющейся фактическим аргументом mapcar. Ясно, что значение свободной переменной лямбда-выражения должно определяться в некотором внешнем выражении, причём это внешнее выражение может фиксироваться либо статически (по записи определений функций) либо динамически (при их вычислении).

Согласно принципу динамического связывания, при вычислении тела функции значение переменной L должно быть взято из ближайшей объемлющей вычисляемой функции. Поясним полученный результат MapCons с помощью *контекста вычисления*, под которым будем понимать набор зафиксированных связей вида *имя-переменной* = *значение*, например: {x=3; y=(U V); z=(NIL)}. При динамическом связывании имён с их значениями контекст формируется в ходе вычисления функциональной программы по следующим правилам:

1. При вызове функции (F p_1 … p_k) , $k \geq 1$ каждый её фактический параметр p_i при необходимости вычисляется в контексте вызова, после чего контекст пополняется связями вида $x_i=v_i$ или $x_i=p_i$ (x_i – формальный параметр функции F, а v_i – значение p_i). Вычисляемость параметров зависит от вида функции (обычная или особая, не вычисляющая свои аргументы).
2. Если при пополнении вычислительного контекста имена x_i формальных параметров уже встречаются в контексте вызова (т.е возникает коллизия имён), то старые связи вытесняются новыми (обычное решение проблемы коллизии имён в программировании: локальные связи замещают уже существующие).
3. По окончании вычисления тела функции из текущего вычислительного контекста удаляются введённые связи вида $x_i=v_i$ или $x_i=p_i$, т.е. восстанавливается контекст вызова.

Вернёмся к примеру вычисления функции MapCons. При её вызове устанавливается контекст

```
MapCons: {L = 7; Z =(A (B) C) },
```

а при вызове в теле MapCons вложенного mapcar добавляются связи и новая связь для L вытесняет более старую:

```
mapcar: {F = (lambda (X)(cons X L));
          L =(A (B) C); Z =(A (B) C)}.
```

Поэтому при вычислении в теле mapcar обращения к функции F и будет взято последнее, новое значение L, а не прежнее 7, и в итоге получится результат (((A (B) C) A) ((B) C) (B)) ((C) C)).

Таким образом, проблема заключается в том, что по правилам вычисления будет взято последнее значение L – по месту вызова функции F, т.е. динамически, а подразумевалось (при определении функций) брать статически, по месту записи лямбда-выражения (функционального аргумента mapcar). Плохо то, что значение этой безымянной функции может меняться в зависимости от вычислительного контекста, в котором она используется, что противоречит идее строгой функциональности.

Контекст определения функции часто называют статическим, в отличие от формируемого динамически контекста вычисления. Рассмотренная проблема определения контекста при вычислении свободных переменных функционального аргумента получила название *funarg-проблемы*, или *проблемы функционального аргумента*. Она возникает, когда сочетаются два условия:

- ✓ В качестве функционального аргумента используется функция со свободными переменными.

✓ В контексте определения функции и в контексте вызова этой функции используются одинаковые имена для обозначения двух разных лисповских выражений, т.е. возникает конфликт имён.

Существуют два возможных пути решения этой проблемы:

1) Исключить конфликты имён путём использования для свободных переменных функций особых экзотических имён, которые не могут встретиться в других функциях (`*****1` и т.п.). Тем самым решение проблемы фактически возлагается на программиста, который должен отслеживать использование имён в программе.
2) Применить так называемое *замыкание функционального аргумента* [15, с. 112]. Можно считать, что замыкание – это лямбда-выражение (функциональный аргумент) вместе с некоторой частью контекста его определения – той, что фиксирует связи его свободных переменных. Вычисление замыкания отложено на момент вызова функционального аргумента.

В диалекте MuLisp (а также в ранних диалектах Лиспа) замыкание отсутствует, поэтому возможен только первый из приведённых путей решения проблемы функционального аргумента.

В диалекте Common Lisp для замыкания функционального аргумента встроена специальная форма `(function F)`, где `F` – определяющее выражение функции. Эту форму часто называют *функциональной блокировкой*, поскольку она аналогична по действию функции `quote`, но не просто квотирует аргумент, а как бы *замыкает* значения используемых в функциональном аргументе `F` свободных переменных, фиксируя их значения из контекста его определения. Функциональную блокировку можно записывать короче, с помощью двух знаков `#'`:

`(function F)` ≡ `#'F`

Для получения правильного вычисления `MapCons` на базе Common Lisp необходимо заменить в её определении при записи функционального аргумента функционала `mapcar` простую блокировку (`quote` или `'`) на функциональную (`function` или `#'`):

```
(defun MapCons (L Z)
       (mapcar (function(lambda (X) (list L X))) Z))
```

Тогда при вычислении `mapcar` всегда для `L` будет браться в качестве значения фактический аргумент функции `MapCons`. В нашем конкретном случае будет установлен следующий контекст для `mapcar` (замыкание показано квадратными скобками):

```
mapcar: {F = [(lambda (X)(cons X L)); {L=7}];
            L = (A (B) C) ; Z =(A (B) C)}
```

В итоге: `(MapCons 7 '(A (B) C))` => `((7 A)(7 (B))(7 C))`.

Особенностью диалекта Common Lisp является необходимость применения формы `function` для лямбда-выражений при любом их использовании в качестве функционального аргумента встроенных функционалов, даже когда в них нет свободных переменных. Поэтому в таких случаях вызов встроенных функционалов в этих двух диалектах различается:

```
; вызов встроенных функционалов в MuLisp
(apply '(lambda (x) (+ 2 x)) '(3)) => 5
(funcall '(lambda (x) (+ 2 x)) 3) => 5
(mapcar '(lambda(y)(* y 2)) '(1 2 3)) => (2 4 6)
(maplist '(lambda(v)v)'(X Y Z)) => ((X Y Z)(Y Z)(Z))

; вызов встроенных функционалов в Common Lisp
(apply #'(lambda (x) (+ 2 x)) '(3)) => 5
(funcall #'(lambda (x) (+ 2 x)) 3) => 5
(mapcar #'(lambda(y)(* y 2)) '(1 2 3)) => (2 4 6)
(maplist #'(lambda(v)v)'(X Y Z)) => ((X Y Z)(Y Z)(Z))
```

Как уже отмечалось ранее, в Common Lisp для переменных программы по умолчанию действует статическое (лексическое) связывание, согласно которому значения переменных в ходе вычислений определяются исходя из контекста их определения. В то же время в этом диалекте есть средство (`defvar` *`имя_переменной`*), устанавливающее для заданной переменной динамическое связывание.

3.4. Применение функционалов

Рассмотрим примеры определения и использования различных функционалов. При их программировании в случае необходимости будем применять функциональную блокировку `function` (`#'`) и указывать различие решений в диалектах MuLisp и в Common Lisp.

Первый пример – функционал `SubstIf` с тремя аргументами: `(SubstIf Z F S)`, который на всех уровнях списочного выражения `S` (S-выражения) заменяет на значение `Z` те элементы выражения, для которых одноместный предикат `F` принимает истинное значение. В качестве результата `SubstIf` выдаёт модифицированное таким образом списочное выражение, например:

```
(SubstIf '* 'null '(A (B ()) ())) => (A (B *) *)
(SubstIf '(NUM) 'numberp '(6 (C 8 A) (7) B))
         => ((NUM) (C (NUM) A) ((NUM)) B))
```

Определение этого функционала с использованием `funcall`:

```
(defun SubstIf(Z F S)
   (cond ((null S) NIL)
         ((funcall F (car S))
             (cons Z (SubstIf Z F (cdr S))))
         ((atom (car S))
             (cons (car S)(SubstIf Z F (cdr S))))
         (T (cons (SubstIf Z F (car S))
                  (SubstIf Z F (cdr S))))  ))
```

Первая ветвь функции служит для завершения рекурсии, вторая – для проверки нужного свойства (предикат F) первого элемента обрабатываемого списочного выражения и его замене в случае положительной проверки. После этой замены рекурсивно продолжается обработка остальных элементов выражения, также как и на третьей ветви, в случае атомарности первого элемента. Параллельная рекурсия в последней ветви обеспечивает обработку списочного выражения вширь и вглубь в случае отрицательных проверок предыдущих ветвей.

Рассмотрим теперь довольно мощный функционал reduce, встроенный в ряд диалектов языка Лисп и позволяющий выразить большое число сложных операций обработки списков. Он реализует *редукцию*, или *свёртку* заданного списка, которая может трактоваться по-разному.

В диалекте MuLisp функционал reduce выполняет следующее преобразование исходного списка L=>(e_1 e_2 … e_n) с использованием значения A и бинарной операции-функции F:

(reduce F L A) ≡ (F(…(F(F A e_1) e_2))…e_n)

Например: (reduce 'list '(1 2 3) 0) => (((0 1) 2) 3)
(reduce '+ '(1 2 3 4) 0) => 10 ;сумма чисел
(reduce '* '(1 2 3 4) 1) =>24 ;их произведение

Определим этот функционал на основе функционала funcall:

```
(defun ReduceL (F L A)
   (cond ((null L) A)
         (T (ReduceL F (cdr L)
                     (funcall F A (car L))))))
```

и покажем его применение в задаче слияния двух упорядоченных списков чисел. Пусть есть функция Insert от двух аргументов:

```
(defun Insert (N L)
  (cond((null L)(cons N NIL));вставка в конец списка
       ((<= N (car L))(cons N L)); условие вставки
       (T (cons (car L) ;место вставки ищется дальше
                (Insert N (cdr L) ) )) ))
```

которая вставляет заданное число N в упорядоченный по неубыванию список чисел L с сохранением упорядоченности:

```
(Insert 5 '(1 3 8 10)) => (1 3 5 8 10)
```

Эту функцию можно использовать вместе с ReduceL для компактного программирования функции Merge слияния двух упорядоченных списков чисел в один упорядоченный список:

```
(defun Merge(X Y) (ReduceL 'Insert X Y))
(Merge '(1 3 5 8) '(2 6 8 10)) => (1 2 3 5 6 8 8 10)
```

Другой пример использования функционала ReduceL – поиск максимума в списке чисел на базе функции Max вычисления максимума двух чисел:

```
(defun Max(X Y) (cond((> X Y) X) (T Y)))
(defun MaxList(L) (ReduceL 'Max (cdr L) (car L)))
(MaxList '(-6 4 -82 2 94 -3)) => 94
```

В ряде случаев полезен другой вариант редукции (свёртки) списка, при котором (reduce F L A) для списка L=>(e_1 … e_n), выражения A и бинарной функции F означает

(reduce F L A) ≡ (F e_1 (F e_2 … (F e_n A) …)).

Например: (reduce 'list '(1 2 3) 0) => (1 (2 (3 0)))

Соответствующее определение функционала:

```
(defun ReduceR (F L A)
   (cond ((null L) A)
         (T (funcall F (car L)
                       (ReduceR F (cdr L) A)))))
```

В этом определении выражение A фигурирует уже как второй операнд редуцирующей операции F, а в качестве её первого операнда берутся очередные элементы списка L, начиная с конца списка. На основе такой редукции легко определить многие функции, в частности:

```
(ReduceR 'cons X ()) ≡ (copy X)
(ReduceR 'cons X Y) ≡ (append X Y)
```

Также как и ReduceL, функционал ReduceR применим для слияния двух упорядоченных списков чисел и поиска максимума в списке чисел (отличие – в направлении обработки списка L):

```
(defun Merge(X Y) (ReduceR 'Insert X Y))
(defun MaxList(L) (ReduceR 'Max (cdr L) (car L)))
```

Рассмотренные варианты (ReduceL и ReduceR) функционала reduce часто называются *левой* и *правой свёрткой*. Отметим, что в случае

коммутативности и ассоциативности бинарной операции F их действие одинаково: (ReduceL F L A)= (ReduceR F L A).

В диалекте Common Lisp реализованный функционал reduce может выполнять оба варианта редукции за счёт использования так называемых *ключевых параметров*. Первый ключевой параметр initial-value задаёт значение A, а второй параметр from-end используется для указания направления свёртки:

(ReduceL F L A) ≡ (reduce F L :initial-value A),
(ReduceR F L A) ≡
(reduce F L :initial-value A :from-end T).

Примеры использования этого функционала в Common Lisp:

```
(reduce 'list '(1 2 3) :initial-value 0)
                              => (((0 1) 2) 3)
(reduce 'list '(1 2 3) :initial-value 0 :from-end T)
                              => (1 (2 (3 0)))
```

Покажем теперь применение функционалов на примере задачи построения декартова произведения двух множеств, представленных одноуровневыми списками атомов:

```
(Decart '(A B C) '(D E))
                  => ((A D)(A E)(B D)(B E)(C D)(C E))
```

Будем записывать решения этой задачи для диалекта Common Lisp, решения же для MuLisp получаются заменой функциональной блокировки #' на обычную блокировку ', а также исключением в вызове reduce имени ключевого параметра :initial-value.

Сначала составим функцию Decart1 с использованием обычной техники рекурсивного программирования: эта функция будет выполнять проход по элементам первого списка, обращаясь с очередным его элементом-атомом к вспомогательной функции Dec для формирования всех кортежей-пар с этим элементом.

```
(defun Decart1 (M1 M2)
    (cond ((null M1) NIL)
          (T (append (Dec (car M1) M2)
                     (Decart1 (cdr M1) M2) )) ))
(defun Dec (A M) ;вспомогательная функция
    (cond ((null M) NIL)
          (T (cons (list A (car M))
                   (Dec A (cdr M)))) ))
```

Функция Dec формирует декартово произведение одноэлементного множества (заданного атомом-значением параметра A) и множества M, например: (Dec 'C '(D E)) => ((C D)(C E)).

Обе приведённые функции сами по себе рекурсивны, в дополнение Decart1 вызывает Dec, а она выполняет проход по списку-множеству M2 (это внутренний цикл по отношению к внешнему циклу прохода по списку M1). Этот проход можно реализовать без вспомогательной функции, используя функционал mapcar. Получим таким образом более короткое и простое решение:

```
(defun Decart2 (M1 M2)
  (cond((null M1) NIL)
       (T (append(mapcar
                   #'(lambda(Z)(list(car M1)Z))
                   M2) ; M2 - второй аргумент mapcar
                 (Decart2 (cdr M1) M2) )) ))
```

На первый взгляд кажется, что и проход по элементам списка M1 (внешний цикл) можно реализовать аналогичным образом на базе mapcar:

```
(defun Decart3 (M1 M2)
   (mapcar  #'(lambda (Y)
                 (mapcar #'(lambda (Z) (list Y Z))
                         M2 ))
            M1 ))
```

Однако это не совсем верно, поскольку в итоговом списке-произведении получаются лишние скобки:

```
(Decart3 '(A B C)'(D E)) =>
            (((A D)(A E))((B D)(B E))((C D)(C E)))
```

Причина ошибки в том, что в функциях Decart_1 и Decart_2 для соединения промежуточных списков-результатов (декартовых произведений элементов-атомов из M1 со вторым множеством M2) применялась функция append, а функционал mapcar использует для их соединения функцию cons.

Один из возможных способов исправления этой ошибки – ликвидировать возникшие лишние скобки, обратившись дополнительно к какой-либо функции, например, к встроенному функционалу reduce (пригоден любой из двух вариантов редукции):

```
(defun Decart4 (M1 M2)
      (reduce 'append (Decart3 M1 M2)
                      :initial-value NIL))
```

Однако более удачным представляется решение, в котором при слиянии списков, полученных внутренним mapcar, лишние скобки просто не образуются. Для этого потребуется определить новый функционал MapAppend – аналог mapcar, который соединяет результаты применения своего аргумента-функции F к элементам списка L с помощью функции append:

```
(defun MapAppend (F L)
    (cond ((null L) NIL)
          (T (append (funcall F (car L))
                     (MapAppend F (cdr L) )) )) )
```

Пример обращения к этому функционалу:

```
(MapAppend #'(lambda (Y)(list Y 'W)) '(5 9 3))
                                      => (5 W 9 W 3 W)
```

Заменяя теперь в функции Decart3 внешнее обращение к mapcar на MapAppend, получаем короткое и верное решение исходной задачи, демонстрирующее выразительную мощность функционалов:

```
(defun Decart5 (M1 M2)
   (Mapappend #'(lambda (Y)
                  (mapcar #'(lambda (Z)(list Y Z))
                          M2 ))
              M1 ))
```

На основе аналогичных рассуждений построим функцию формирования всех подмножеств заданного множества (булеан заданного множества) с помощью встроенных функционалов. Назовём эту функцию Subsets. Как и в предыдущей задаче построения декартова произведения множеств, исходное множество задаётся как одноэлементный список атомов, а результирующий список является уже списком списков-подмножеств. Порядок элементов в исходном и в итоговом списке не важен, важно однако, что в обоих списках нет повторяющихся элементов. Пример вызова функции Subsets и её определение для диалекта Common Lisp:

```
(Subsets '(A S D)) =>
        (NIL (A) (S) (S A) (D) (D A) (D S) (D S A))
;определение Subsets для Common Lisp
(defun Subsets(M)
  (reduce #'(lambda (X Y)
              (append X
                 (mapcar #'(lambda(Z)(cons Y Z)) X)))
          M :initial-value '(NIL)  ))
```

Идея решения заключается в последовательном рассмотрении элементов исходного списка. При этом на каждом шаге для очередного элемента на основе уже построенных подмножеств строится группа новых подмножеств, получающаяся вставкой этого элемента в каждое из имеющихся подмножеств. Полученная группа новых подмножеств объединяется функцией `append` с подмножествами предыдущего шага. Таким образом, количество построенных подмножеств после каждого шага увеличивается вдвое, а общее количество подмножеств заданного множества – 2^N, где N – количество элементов-атомов в этом множестве. На первом шаге множество построенных подмножеств состоит из единственного пустого множества, обозначаемого `NIL` (третий аргумент функционала `reduce` – `(NIL)`).

Как и для функций `Decart` из предыдущей задачи, для получения определения `Subsets` в диалекте MuLisp необходима замена в приведённом определении функциональной блокировки `#'` на обычную блокировку `'`, а также исключение в вызове функционала `reduce` имени ключевого параметра `:initial-value`.

В заключение, для дополнительной иллюстрации отличий в программировании функционалов в диалектах MuLisp и Common Lisp рассмотрим вновь задачу определения функционала `mapcar` – теперь уже на основе функционала `maplist`. В Common Lisp решение этой задачи получается достаточно просто:

```
;определение Mapcar на основе Maplist для Common Lisp
(defun Mapcar(F L)
       (maplist #'(lambda(X)(funcall F (car X))) L))
```

В MuLisp функциональная блокировка отсутствует, а заменить её на простое квотирование в данном случае нельзя, т.к. в лямбда-выражении вместо формального параметра `F` необходимо брать его конкретные значения. Поэтому в решении для MuLisp нужное лямбда-выражение строится в ходе вычислений:

```
;определение Mapcar на основе Maplist для MuLisp
(defun Mapcar(F L)
   (maplist (list 'lambda '(X) (list F '(car X))) L))
```

3.5. Функционалы с функциональным значением

Аналогично тому, как функция может быть аргументом другой функции, она может быть и результатом (значением) функции, т.е. функция высшего порядка может иметь *функциональное значение*. Следующий пример показывает определение такой функции:

```
; определение функции Increment для языка Common Lisp
(defun Increment (N)  #'(lambda (Z) (+ Z N)) ))
```

Значением функционала `Increment` является определяющее выражение безымянной функции (лямбда-выражение) от одного аргумента, которая добавляет к своему аргументу некоторое значение `N` (это свободная переменная в теле возвращаемой функции).

В данном определении функционала `Increment` использована функциональная блокировка `function` (или `#'`), которая кроме квотирования лямбда-выражения выполняет также связывание свободной для этого лямбда-выражения переменной `N` со значением аргумента функции `Increment`, что позволяет затем использовать такое замкнутое лямбда-выражение в качестве функционального аргумента других функционалов.

Определение функционала `Increment` в диалекте MuLisp, без средств встроенного замыкания, опять сложнее:

```
; определение функции Increment для языка MuLisp
(defun Increment (N)
    (list 'lambda '(Z) (list '+ 'Z N)))
```

Это определение основано на следующей идее: чтобы вернуть функцию, увеличивающую свой аргумент на заданное число, необходимо построить и вернуть лямбда-выражение, в котором вместо `N` уже будет подставлено заданное в качестве аргумента функции `Increment` значение.

Различие двух приведённых определений функции `Increment` обусловлено разными видами связывания переменных с их значениями, реализованными в диалектах MuLisp (динамическое связывание) и Common Lisp (статическое связывание), а также наличием в Common Lisp функциональной блокировки – встроенной формы `function` (или `#'`). Последнее существенно при программировании именно функционалов с функциональным значением, поэтому далее для таких функционалов даём два определения – для MuLisp и для Common Lisp, а примеры возвращаемых значений этих функционалов приводим только для MuLisp.

При разных `N` функционал `Increment` в MuLisp выдаёт в качестве своего значения разные лямбда-выражения – безымянные функции с разными приращениями своего аргумента, например:

```
(Increment 2)  =>(LAMBDA (Z) (+ Z 2))
(Increment 10) =>(LAMBDA (Z) (+ Z 10))
```

В обоих диалектах функционал `Increment` можно использовать в других функционалах:

```
(mapcar (Increment 2) '(1 8 15)) => (3 10 17)
(funcall (Increment 10) 5) => 15
```

Приведём теперь определение функции Twice, также с функциональным значением:

```
; определение функции Twice для языка Common Lisp
(defun Twice (F)
   (function (lambda (X) (funcall F (funcall F X)))))
; определение функции Twice для языка MuLisp
(defun Twice (F)
      (list 'lambda '(X)
          (list 'funcall (list 'quote F)
              (list 'funcall (list 'quote F) 'X))))
```

У функции Twice не только функциональное значение, но и функциональный аргумент. Результат её вычисления – определяющее выражение функции, дважды применяющей заданную функцию F от одного аргумента. Например:

```
(Twice 'list) =>
        (LAMBDA (X)(FUNCALL (QUOTE LIST)
                        (FUNCALL (QUOTE LIST) X)))
(mapcar(Twice 'list)'(5 7 8)) => (((5)) ((7)) ((8)))
(funcall (Twice (Increment 3)) 5)) => 11
```

Ещё один функционал Composition с функциональными аргументами и значением выполняет композицию F•G двух заданных функций (порядок применения F и G соответствует общепринятому):

```
; определение функции Composition для Common Lisp
(defun Composition (F G)
   (function (lambda (X)(funcall G (funcall F X)))) )
; определение функции Composition для языка MuLisp
 (defun Composition (F G)
      (list 'lambda '(X)
          (list 'funcall (list 'quote G)
                (list 'funcall (list 'quote F) 'X))))
```

Примеры применения этого функционала:

```
(funcall (Composition (Increment 5) 'list) 7) => (12)
(Composition (Increment 5) 'list) =>
    (LAMBDA (X)(FUNCALL (QUOTE LIST)
        (FUNCALL (QUOTE (LAMBDA (Z)(+ Z 5))) X)))
;возвращаемое лямбда-выражение приведено для MuLisp
```

4. Дополнительные возможности

Описанное в предыдущих разделах функциональное подмножество языка Лисп достаточно для программирования многих задач обработки символьных данных. Рассматриваемые в данном разделе средства расширяют возможности языка и могут изучаться по мере необходимости.

4.1. Список свойств атома

Как уже отмечалось ранее, с любым символьным атомом (идентификатором) в Лиспе могут быть связаны следующие значения: его внешнее имя, его значение как параметра некоторой функции, функциональное значение и *список свойств атома*. Список свойств появился уже в ранних диалектах Лиспа как удобное средство хранения информации, доступной в любой момент вычислений, и назывался также *ассоциативным списком*.

Свойство понимается как пара *имя_свойства – значение_свойства*. Именем свойства может быть произвольный символьный атом, а значением – произвольное лисповское выражение. Например, атом `APPLE` может иметь свойство `COLOUR` со значением `(RED YELLOW GREEN)` и свойство `TASTE` со значением `ACID`.

Список свойств атома может быть пустым или содержать произвольное количество свойств, при этом свойства никак не влияют друг на друга. Главная особенность списка свойств в том, что он не зависит от вычислительного контекста, содержащиеся в нём свойства доступны в произвольный момент вычислений.

Для работы со списком свойств обычно используются следующие встроенные функции (все они есть в диалекте MuLisp).

Функция `get` с обращением `(get` *`a name`*`)` вычисляет значения своих аргументов и возвращает значение свойства с именем *`name`* у атома *`a`*, либо `NIL`, если у этого атома нет свойства с таким именем. Например:

```
(get ' APPLE 'COLOUR) => (RED YELLOW GREEN)
(get ' APPLE 'TASTE) => ACID
(get ' APPLE 'SIZE) => NIL
```

Заметим, что нецелесообразно в качестве значения некоторого свойства использовать пустой список, поскольку тогда неразличимы два принципиально разных случая – отсутствие свойства и наличие этого свойства со значением `NIL`.

Функция put с вычисляемыми аргументами и обращением (put *a name p*) выдаёт в качестве результата значение p, но основное её действие – добавление к списку свойств атома *a* нового свойства с именем *name* и значением *p*. Если свойство с заданным именем уже есть, то старое значение заменяется на новое, к примеру:

```
(put 'APPLE 'TASTE 'SWEET) => SWEET
(get 'APPLE 'TASTE) => SWEET
```

Функция remprop с обращением (remprop *a name*) служит для удаления свойства *name* у атома *a*, оба аргумента функции вычисляются. В диалекте MuLisp эта функция возвращает в качестве результата значение свойства *name* у атома *a*. Если свойства с заданным именем у этого атома нет, то значение функции равно NIL. Например:

```
(remprop 'APPLE 'TASTE) => SWEET
(get 'APPLE 'TASTE) => NIL
```

В Common LISP в случае успешного удаления свойства атома функция remprop выдает T. В этом диалекте не реализована функция put, поскольку её действие легко осуществить при помощи встроенной функции присваивания setf. Вместо обращения к put используется конструкция вида (setf (get *a name*) *p*). Но для удобства можно определить и функцию put:

```
(defun put (a name p)(setf (get a name) p))
```

Отметим, что функции put и remprop имеют побочный эффект – они изменяют внутреннее представление списка свойств. Это отступление от принципов чисто функционального программирования тем не менее полезно при решении практических задач.

В качестве примера использования списка свойств рассмотрим задачу проверки правильности выражения языка Паскаль, которое представляет собой суперпозицию стандартных функций от одного аргумента, например: SIN(PRED(TRUNC(LN(4.5))))).

В суперпозиции могут использоваться следующие функции [7] : ABS (абсолютное значение числа), SQR (квадрат), SIN (синус), COS (косинус), EXP (экспонента), LN (логарифм), SQRT (квадратный корень), TRUNC (целая часть числа), ROUND (округление до целого), SUCC (следующее целое) и PRED (предыдущее целое). Аргументами ABS и SQR могут быть как целые, так и вещественные числа, а тип возвращаемого результата совпадает с типом аргумента. Функции синуса, косинуса, экспоненты, логарифма и квадратного корня также в качестве аргумента могут иметь как целое, так и вещественное число, в то время как их результат – вещественное число. Функции TRUNC и ROUND преобразуют

вещественное число в целое, но если в качестве аргумента получают целое, то просто возвращают его в качестве результата. Функции SUCC и PRED предназначены для работы с целыми числами, поэтому и аргумент, и результат этих функций являются целыми числами.

Необходимо проверить правильность типов аргументов функций, входящих в заданное выражение, и в случае правильного выражения определить тип его значения. Аргументом самой внутренней функции суперпозиции является целое или вещественное число.

Информацию о допустимых типах аргумента каждой стандартной функции Паскаля и типе ее результата (INT, REAL) будем хранить в виде списка свойств атома, служащего именем функции. Свойство ArgType хранит список возможных типов аргумента функции, а свойство FunType – тип результата функции. В случаях, когда тип значения функции совпадает с типом её аргумента (для ABS и SQR), будем обозначать его атомом =TypeArg.

В приведённой ниже программе для языка MuLisp сначала в списки свойств атомов-имён паскалевских функций заносится необходимая информация о типах их аргументов и значений. Затем определяется функция EvalType, проверяющая правильность заданного выражения-суперпозиции паскалевских функций и в случае правильности вычисляющая тип этого выражения. Для упрощения программы считаем, что на вход функции EvalType подаётся выражение, заключённое в круглые скобки, к примеру: (ABS(SUCC(-2))). В случае ошибки в типе аргумента функции, входящей в суперпозицию, EvalType возвращает атом ERROR. Например:

```
(EvalType '(ABS(SUCC(-2)))) => INT
(EvalType '(SIN(PRED(TRUNC(LN(4.5)))))) => REAL
(EvalType '(SUCC(SIN(ABS(-5))))) => ERROR
```

```
;MuLisp-программа проверки типов аргументов выражения
; ввод типов аргументов и типов результатов
; функций в соответствующие списки свойств
(put 'ABS 'ArgType '(INT REAL))
(put 'ABS 'FunType '=TypeArg)
(put 'SQR 'ArgType '(INT REAL))
(put 'SQR 'FunType '=TypeArg)
(put 'SIN 'ArgType '(INT REAL))
(put 'SIN 'FunType 'REAL)
(put 'COS 'ArgType '(INT REAL))
(put 'COS 'FunType 'REAL)
(put 'EXP 'ArgType '(INT REAL))
```

```
(put 'EXP 'FunType 'REAL)
(put 'LN 'ArgType '(INT REAL))
(put 'LN 'FunType 'REAL)
(put 'SQRT 'ArgType '(INT REAL))
(put 'SQRT 'FunType 'REAL)
(put 'TRUNC 'ArgType '(INT REAL))
(put 'TRUNC 'FunType 'INT)
(put 'ROUND 'ArgType '(INT REAL))
(put 'ROUND 'FunType 'INT)
(put 'SUCC 'ArgType '(INT))
(put 'SUCC 'FunType 'INT)
(put 'PRED 'ArgType '(INT))
(put 'PRED 'FunType 'INT)
; вычисление типа выражения-суперпозиции функций
(defun EvalType (L)
  (cond ((null(cdr L))(MyType(car L)))  ;тип числа
        ((eq (get (car L) 'FunType) '=TypeArg)
            (EvalType (cadr L)) )
        ((member (EvalType (cadr L))
                 (get (car L) 'ArgType) )
             (get (car L) 'FunType))
        (T ERROR) ))
;вычисление типа числа(самого внутреннего аргумента)
(defun MyType(A)
   (cond ((member '\. (unpack A)) REAL)
         (T INT) ))
```

Функция MyType возвращает тип числа, служащего аргументом самой вложенной функции суперпозиции: если в запись числа входит точка, то его тип – REAL, иначе тип числа – INT. При этом используется встроенная функция MuLisp unpack (описывается в следующем разделе).

При вычислении типа всего выражения-суперпозиции поочерёдно рассматриваются следующие случаи. Если аргумент функции EvalType – список из одного элемента (первая ветвь EvalType), то этот элемент является аргументом самого вложенного функционального обращения, т.е. числом, и его тип вычисляет функция MyType. В противном случае аргумент EvalType представляет собой обращение к паскалевской функции, и с помощью get из списка свойств атома-имени этой функции извлекается тип её значения (вторая ветвь EvalType). Если этот тип – =TypeArg, рекурсивно вычисляется и возвращается тип аргумента данной функции. В ином случае (третья ветвь EvalType) вычисляется тип

аргумента анализируемой функции и проверяется, входит ли он в список-значение свойства `ArgType` этой функции: если входит, то `EvalType` возвращает тип результата данной функции (значение свойства `FunType`), иначе (четвертая ветвь) возвращает атом `ERROR`.

Заметим, что использование списка свойств позволило организовать программу так, что расширение набора функций, входящих в проверяемое выражение-суперпозицию, как и расширение возможных типов их значений (например, добавление логического типа) возможно без изменения функции `EvalType`, рекурсивно проверяющей паскалевское выражение и вычисляющей его тип.

4.2. Функции ввода, вывода и работы с символами

Для ввода данных в ходе вычислений используется встроенная лисповская функция `read` без аргументов, её вызов имеет вид `(read)`. Когда лисп-интерпретатор начинает обрабатывать вызов этой функции, вычисления приостанавливаются до тех пор, пока не будет полностью введено лисповское выражение, т.е. атом или списочное выражение, сбалансированное по круглым скобкам. Введённое выражение переводится во внутреннее представление и возвращается в качестве результата вызова `read`, например:

при вводе атома `asd`: `(read) => ASD`
при вводе списка `(G(56)dee)`: `(read) => (G (56) DEE)`
при вводе точечной пары `(ci . B)`: `(read) => (CI . B)`

Перевод строчных (малых) букв в соответствующие им заглавные (большие) происходит в ходе преобразования введенного выражения во внутреннее представление, когда выполняется построение внутренних имён символьных атомов.

Внутренним именем атома может быть произвольная последовательность символов (литер), включая буквы обоих регистров, цифры, спецзнаки, а также пробел, точка, круглые скобки и другие символы, играющие в Лиспе особую роль. Именно внутренние имена атомов служат для их идентификации при работе лисп-интерпретатора. Допущение во внутренних именах атомов любых символов даёт возможность интерпретатору обрабатывать произвольные тексты.

Соответственно, в языке Лисп есть возможность считывать функциями ввода, а также записывать в лисп-программе атомы, в состав которых входят *особые символы*. Кроме пробела и круглых скобок к особым символам относятся апостроф `'`, обратный слэш `\`, знак вертикальной черты `|`, кавычки `"`, точка с запятой `;` , любая цифра, если

она является первым символом имени, а также строчные (малые) буквы, если их регистр существенен.

Возможны два варианта записи атома, содержащего особые символы: либо перед каждым особым символом поставить знак \ , либо заключить все его символы в знаки вертикальной черты |. Например, для задания атома с внутренним именем `2A(80c` следует записать `\2A\(80\c` (как особые указаны: первая цифра, открывающая круглая скобка и строчная буква `c`) или `|2A(80c|`. При записи `\2A\(80c` (буква `c` записывается без \) внутреннее имя будет другим: `2A(80C`.

Кроме внутреннего имени каждый используемый в лисп-программе атом имеет *внешнее имя* (это одно из указывавшихся ранее четырёх разных значений, связанных с атомом). Если атом содержит особые символы, то для получения внешнего имени внутреннее имя записывается в знаках вертикальной черты, в ином случае внешнее имя совпадает с внутренним. Внешнее имя используется при вводе атома, к примеру:

при вводе атома `\2A\(80c:` `(read) => |2A(80C|`
при вводе атома `|2A(80c|:` `(read) => |2A(80c|.`
при вводе атома `a\(s\)\d:` `(read) => |A(S)d|.`

Всё сказанное равно относится к диалектам MuLisp и Common Lisp, за исключением атомов из одного особого символа, внешнее имя которых в MuLisp записывается со знаком \, например:

при вводе атома `\1` в MuLisp: `(read) => \1`
при вводе атома `\1` в Common Lisp: `(read) => |1|`

Подчеркнём, что обе эти записи `\1` и `|1|` обозначают одно и то же внутреннее имя символьного атома, состоящего из цифры `1`, и не имеют ничего общего с числом `1`:

```
(symbolp '|1|) => T
(symbolp 1)  => NIL
(eq '|1| '\1) => T
(numberp '\1) => NIL
```

Важно, что атомы с особыми символами, несмотря на их специфическую запись, тем не менее являются такими же атомами, как и все остальные: они заносятся в таблицу атомов, могут быть именами функций или переменных, с ними может быть связан список свойств.

Особенность описанной лисповской функции `read` – считывание атомов и выражений, сбалансированных по скобкам. Поскольку в ряде случаев требуется посимвольный ввод обрабатываемых данных, в Лиспе для этого встроена функция ввода `read-char`. Обращение к ней имеет вид `(read-char)`, а её значение зависит от диалекта языка Лисп.

В MuLisp значением функции `read-char` является внешнее имя символа, считанного из входного потока, например:

при вводе символа `1`: `(read-char) => \1`
при вводе символа `(`: `(read-char) => \(`
при вводе символа `A`: `(read-char) => A`
при вводе символа `a`: `(read-char) => \a`

В Common Lisp введённый символ преобразуется в константу символьного типа (этот диалект поддерживает несколько типов данных, включая символы, строки, списки). Символ-константа изображается как последовательность трёх знаков: `#`, `\` и сам введённый символ, например:

при вводе символа `(`: `(read-char) => #\(`
при вводе символа `A`: `(read-char) => #\A`
при вводе символа `a`: `(read-char) => #\a`

Символьные константы не являются атомами, тем не менее они могут сравниваться на равенство встроенными функциями `eq` и `eql` (для корректной работы `eql` оба её аргумента должны быть одного типа).

Рассмотрим теперь способы ввода и дальнейшей обработки текста, содержащего особые символы (в частности, математических формул, в которых есть круглые скобки). Для этого можно использовать либо функцию посимвольного ввода `read-char`, либо функцию `read`, позволяющую ввести весь текст за одно обращение к ней.

В Common Lisp для такого ввода функцией `read` необходимо заключить вводимый текст в кавычки, сделав из него строку, например: `"(a+d)/c-12"`. Введённую строку символов можно преобразовать в список символов с помощью встроенной функции `coerce`. Эта функция имеет два вычисляемых аргумента, значением первого должно быть преобразуемое выражение, а значением второго – тип, к которому оно должно быть преобразовано (в частности: `list`, `string`). В нашем случае функцию `coerce` имеет смысл использовать для преобразования строки в список символов-констант, и для обратного преобразования такого списка в строку:

```
(coerce "(a+d)/c-12" 'list)
        => (#\( #\a #\+ #\d #\) #\/ #\c #\- #\1 #\2)
(coerce '(#\a #\* #\( #\b #\+ #\c #\) ) 'string))
        => "a*(b+c)"
```

После преобразования функцией `coerce` строки в список символов этот список уже может быть проанализирован с помощью известных встроенных функций, включая функцию `eq`.

В диалекте MuLisp для единоразового ввода текста с помощью функции `read` необходимо заключить его либо в кавычки, либо в символы вертикальной черты. При этом введённый текст будет сохранен во внутреннем представлении как символьный атом. Для дальнейшей обработки этого атома может быть использована встроенная в MuLisp функция `unpack` (распаковка) с одним вычисляемым аргументом.

Функция `unpack` преобразует значение своего аргумента – символьный или числовой атом – в список односимвольных атомов, например:

```
(unpack 'ASD123)  => (A S D \1 \2 \3)
(unpack '|(a*b)+C|) => ( \( \a * \b \) + C)
(unpack '458)  =>  (\4 \5 \8)
```

Обратное преобразование выполняет встроенная в MuLisp функция `pack` (упаковка) с одним вычисляемым аргументом.

Функция `pack` преобразует заданный список атомов (не обязательно символьных и односимвольных) в один символьный атом путём конкатенации входящих в их запись символов, например:

```
(pack '(AS 127 D 34)) => AS127D34
(pack '(127 AS D 34)) => |127ASD34|
(pack '(as -12.56 df 34) => AS-12.56DF34
(pack '(a |sd| 56 \8t)) => |Asd568T|
```

Заметим, что в исходном списке могут быть числовые атомы, при конкатенации функцией `pack` берутся символы их записи.

Часто в процессе работы программы (в частности, при её отладке) требуется производить выдачу каких-либо сообщений и промежуточных результатов вычислений. Для этого служат встроенные функции вывода лисповских выражений `print`, `prin1`, `princ` и `terpri`.

Функция `print` с обращением `(print e)` вычисляет свой аргумент – выражение `e` и выполняет вывод полученного значения вместе с переводом строки. В качестве результата `PRINT` возвращает значение своего аргумента, например:

```
(print (cdr '(a (s) d)) => ((S) D)
```

кроме того, как побочный результат функции этот список-результат будет выведен на печать или в выходной поток.

В диалекте MuLisp `print` сначала выводит значение выражения `e` с текущей позиции строки, после чего осуществляет перевод строки. В CommonLisp сначала происходит перевод строки, а затем выводится значение аргумента `e` и символ пробела.

Функция `prin1` от одного аргумента `(prin1 e)` вычисляет свой аргумент `e` и выводит полученный результат с текущей позиции строки. Значением функции является вычисленное значение `e`. Отличие этой функции от `print` - вывод выполняется без перевода строки и без дополнительных пробелов.

Функция `princ` отличается от функции `prin1` только тем, что для вывода символьных атомов использует не внешние имена атомов, а их внутренние имена, например,

```
(princ '(|as| 23 \(sd\))) =>(|as| 23 |(SD)|)
```

но выведено будет выражение `(as 23 (SD))`.

Для перевода строки при выводе служит функция `terpri` без аргументов, её значением является атом `NIL`.

Ещё одна встроенная функция `load` служит для загрузки и вычисления файла с лисп-программой. Вызов функции имеет вид:

```
(load "имя_файла_с_расширением") .
```

Указанный в вызове текстовый файл должен состоять из последовательности вычислимых лисповских выражений (форм). При выполнении вызова `load` из этого файла последовательно считываются и вычисляются лисповские выражения. Если в процессе ввода и вычисления очередного выражения обнаружена ошибка, то выводится диагностическое сообщение о ней, а ввод и вычисление следующих выражений из файла прекращается. В случае успешного вычисления всех выражений из заданного программного файла выводится сообщение об этом, а результатом работы функции `load` является атом `T`.

Функция `load` обычно применяется для загрузки определений новых функций, в частности, после загрузки в системе MuLisp файла `common.lsp` появляется возможность использования ряда встроенных функций диалекта CommonLisp.

4.3. Общий вид функций cond, defun, let

До сих пор мы записывали лисповские особые функции `cond`, `defun`, `let` в их простой, хотя и часто используемой форме, применяемой при программировании строго функциональных программ, в которых исключены побочные эффекты вычислений. При использовании функций с побочными эффектами (в частности, при использовании описанных выше встроенных функций ввода/вывода и функций работы со списком свойств атомов) может оказаться полезной запись управляющих конструкций `cond`, `defun`, `let` в их более общей форме.

Общий вид условного выражения:

```
(cond (p1 e11 e12 … e1m1)
      (p2 e21 e22 … e2m2)
          …
      (pn en1 en2 … en mn))      mi ≥ 0, n ≥ 1
```

Вычисление условного выражения общего вида выполняется по следующим правилам:

I. последовательно вычисляются условия p_1, p_2, … p_n ветвей выражения до тех пор, пока не встретится выражение p_i, значение которого отлично от NIL;

II. последовательно вычисляются выражения-формы e_{i1} e_{i2} … e_{imi} соответствующей ветви, и значение последнего выражения e_{imi} возвращается в качестве значения функции cond;

III. если все условия p_i имеют значение NIL, то значением условного выражения становится NIL.

Как и ранее, ветвь условного выражения может иметь вид (p_i), когда $m_i=0$. Тогда если значение $p_i \neq$ NIL, значением условного выражения cond становится значение p_i.

В случае, когда $p_i \neq$ NIL и $m_i \geq 2$, т.е. ветвь cond содержит более одного выражения e_i, эти выражения вычисляются последовательно, и результатом cond служит значение последнего из них e_{imi}. Таким образом, в дальнейших вычислениях может быть использовано только значение последнего выражения, и при строго функциональном программировании случай $m_i \geq 2$ обычно не возникает, т.к. значения предшествующих e_{imi} выражений пропадают.

Использование более одного выражения e_i на ветви cond имеет смысл тогда, когда вычисление предшествующих e_{imi} выражений даёт побочные эффекты, как при вызове функций ввода и вывода, изменении списка свойств атома, а также определении новой функции с помощью defun. К примеру:

```
(cond ((< X 5)(print "Значение x меньше пяти") X)
      ((= X 10)(print "Значение x равно 10") X)
      (T(print "Значение x больше пяти, но не 10")X))
```

Значением этого условного выражения всегда будет значение переменной X, но при этом на печать будет выведена одна из трёх строк, в зависимости от текущего значения X.

Во многих диалектах, в том числе в MuLisp и Common Lisp, допускается запись нескольких выражений в теле функции при её определении. Общий вид обращения к defun:

(defun f (x_1 … x_k) e_1 … e_n) , $n \geq 0$

Если тело функции пусто, то значение функции равно NIL. В ином случае последовательно вычисляются входящие в тело функции выражения e_i, и в качестве значения функции берётся значение последнего выражения e_n.

Опять же, включение в тело функции более одного выражения имеет смысл тогда, когда их вычисление имеет побочный эффект, например:

```
(defun Plus5()(princ "Введите целое число: ")
              (put 'N 'Value (+ 5 (read)))
              (terpri)
              (princ "Число, большее на 5: ")
              (print (get 'N 'Value))  )
```

При вызове этой функции без аргументов (Plus5) у пользователя запрашивается число, и после его ввода оно увеличивается на 5 и сохраняется в списке свойств атома N. После вывода поясняющей строки из списка свойств достаётся и выводится увеличенное число, и оно становится итоговым значением функции Plus5.

Аналогично, в теле лямбда-выражения, являющегося, по сути, определением безымянной функции, допускается произвольное число выражений:

(lambda (m_1 m_2 … m_k) e_1 e_2 … e_n) $n \geq 0$, $k \geq 0$.

При соответствующем лямбда-вызове

((lambda (m_1 m_2 … m_k) e_1 e_2 … e_n) p_1 p_2 … p_k)

после связывания формальных параметров m_1 m_2 … m_k происходит последовательное вычисление выражений-форм e_1 … e_n, и в качестве результата лямбда-вызова берётся значение последней формы e_n . Опять же, в случае пустого тела функции её значение равно NIL.

В лисповской конструкции let, позволяющей вводить локальные переменные и являющейся другой формой записи лямбда-вызова, также допускается любое количество вычисляемых выражений. Общий вид let:

(let ((m_1 p_1)(m_2 p_2)…(m_k p_k)) e_1 e_2 … e_n) $n \geq 0$, $k \geq 1$

Эта форма эквивалентна лямбда-вызову:

((lambda (m_1 m_2 … m_k) e_1 e_2 … e_n) p_1 p_2 … p_k)

и позволяет завести и связать локальные переменные m_1 m_2 … m_k со значениями p_1 p_2 … p_k на время вычисления форм e_1 e_2 … e_k. Значением обращения к let является значение последней формы e_n (или NIL в случае n=0). Отметим, что сначала вычисляются значения фактических параметров (форм p_1 p_2 … p_n), а затем происходит

одновременное связывание полученных значений с формальными параметрами m_1 m_2 … m_k.

Опишем ещё одну полезную конструкцию `let*`, которая аналогична `let`, но вычисление форм p_i и связывание локальных переменных m_i происходит строго последовательно. Сначала вычисляется p_1, и полученное значение связывается с m_1, затем вычисляется p_2, и его значение связывается с m_2, и т.д. На момент вычисления формы p_i переменные m_1 … m_{i-1} уже связаны и их значения могут быть использованы для вычисления выражения p_i. Приведём пример использования этой конструкции:

```
(let* ((X 12) (Y (* X 2)) (Z (+ X Y)) )
      (print X) (print Y) (print Z) )
```

В результате вычисления этой формы будут выведены последовательно числа `12`, `24`, `36`, а число `36` станет значением формы.

Подчеркнём, что использование конструкций `defun` и `let` в их общей форме предполагает некоторые побочные эффекты при вычислении входящих в них выражений e_1 e_2 … e_n, возможно за исключением последнего e_n, служащего для определения результирующего значения. По этой причине в функциональных программах обычно используется не более одного выражения `e` в конструкции `let` и не более одного выражения в теле функции при её определении функцией `defun`.

4.4. Определение особых функций

До сих пор для определения новых функций нами использовалась функция `defun` в основном с обращением

`(defun F (`x_1 … x_k`) e)`

или, что то же самое `(defun F (lambda (`x_1 … x_k`) e))`

При необходимости можно использовать более общий вид:

`(defun F (`x_1 … x_k`) `e_1 … e_n `), n ≥ 0`

или `(defun F (lambda (`x_1 … x_k`) `e_1 … e_n`)), n ≥ 0 .`

При этом происходит связывание имени `F` с определяющим выражением обычной функции, которая имеет фиксированное количество аргументов, причем каждый из них вычисляется перед выполнением тела функции. Хотя в большинстве случаев этого достаточно для решения практических задач, иногда всё же удобно ввести и затем применять функцию, имеющую произвольное число аргументов либо не вычисляющую аргументы (например, чтобы при обращении к функции их не квотировать).

Для определения новых особых функций в диалекте MuLisp допускается более общая форма обращения к `defun`:

$$\texttt{(defun F (}\left[\begin{array}{l}\texttt{lambda}\\ \texttt{nlambda}\end{array}\right]\left[\begin{array}{c}\texttt{X}\\ \texttt{(}x_1\ldots x_k\texttt{)}\end{array}\right]\ e_1\ \ldots\ e_n\texttt{))}\ .$$

В этой записи: `F` – имя функции (символьный атом);

`X`, x_1, x_2 … x_k – формальные параметры (символьные атомы), $k \geq 0$;

e_1 … e_n – тело функции (лисповские формы), $n \geq 0$.

Квадратные скобки обозначают выбор одного из двух альтернативных вариантов. Таким образом, имеем 4 возможные комбинации:

I. `(defun F (lambda (`x_1 … x_k`)` e_1 … e_n`))` или сокращённо `(defun F (`x_1 … x_k`)` e_1 … e_n`)` – это определение обычной функции `F` (см. предыдущий раздел и раздел 1.3).

II. `(defun F (lambda X` e_1 … e_n`))` или сокращённо: `(defun F X` e_1 … e_n`)` – определение функции `F` с произвольным количеством вычисляемых аргументов. К определённой таким образом функции `F` можно обращаться с любым числом фактических параметров (аргументов). Все они перед выполнением тела функции вычисляются, и из их значений составляется список, который связывается с формальным параметром `X` (т.е. становится его значением). Таким образом, в процессе вычисления вызова функции `(F` $p_1 \ldots p_k$`)` выполняются следующие этапы:

1) вычисление фактических параметров: $p_1 \Rightarrow v_1$, … , $p_k \Rightarrow v_k$;
2) связывание формального параметра: `X =` `(`v_1 … v_k`)`;
3) вычисление тела e_1 … e_n при установленном значении (связи) `X`.

III. `(defun F (nlambda (`x_1 … x_k`)` e_1 … e_n`))` – определение функции `F` с фиксированным количеством аргументов, и они не вычисляются. При вызове функции `(F` $p_1 \ldots p_k$`)` её формальные параметры x_1, … , x_k попарно связываются с фактическими p_1, … , p_k в том виде, как они заданы в обращении. Поскольку вычисление значений фактических параметров не производится, в итоге остается 2 этапа выполнения функционального вызова:

1) связывание формальных параметров: $x_1 = p_1$, … , $x_k = p_k$;
2) вычисление тела e_1 … e_n при зафиксированных связях $x_i = p_i$; при этом всюду, где необходимо вычислить x_i, в качестве его значения используется p_i.

IV. `(defun F (nlambda X` e_1 … e_n`))` – определение функции `F` с произвольным количеством невычисляемых аргументов. При вызове этой функции `(F` $p_1 \ldots p_k$`)` может быть задано любое количество

фактических параметров-аргументов, и все они не вычисляются. Из них (в том виде, как они заданы) составляется список, который связывается с формальным параметром X. Таким образом, этапы выполнения функционального вызова:

1) Связывание формального параметра X = (p_1 … p_k);
2) вычисление тела e_1 … e_n при установленном значении (связи) X.

Во всех рассмотренных случаях в качестве значения выполненного функционального вызова при n>0 берётся вычисленное значение e_n; а при n=0 значение функции равно NIL.

Приведём примеры определений нескольких особых функций (все они являются встроенными).

```
; определение в MuLisp функции list
(defun list(lambda X X))
```

Функция list вычисляет свои аргументы (т.к. используется lambda), их количество произвольно, поэтому с формальным параметром X связывается список вычисленных аргументов, этот список (значение X) и возвращается в качестве результата list.

```
; определение в MuLisp функции quote
(defun quote(nlambda (X) X))
```

Функция quote имеет один аргумент X, и он не вычисляется (используется nlambda); в качестве результата возвращается значение X, т.е. выражение-аргумент в его исходном виде.

```
; определение в MuLisp функции or
(defun or(nlambda Y
            (cond ((null Y)NIL)
                  ((eval (car Y)))
                  (T (eval (cons 'or (cdr Y)))))))
```

Функция or не всегда вычисляет все свои аргументы, поэтому используется nlambda. Функция имеет произвольное число аргументов, и формальный параметр Y связывается со списком невычисленных её аргументов. Если список Y пуст, значение функции равно NIL. Иначе вычисляется значение первого аргумента or (первого элемента Y), и если оно отлично от NIL, то возвращается в качестве результата вычисления вызова or. В противном случае (когда значение первого аргумента or равно NIL) строится и вычисляется рекурсивный вызов функции or с оставшимися аргументами.

```
; определение в MuLisp функции and
(defun and(nlambda Z
            (cond ((null Z)T)
                  ((null (cdr Z))(eval (car Z)))
                  ((eval (car Z))
                     (eval (cons 'and (cdr Z)))))))
```

Функция and определяется аналогично функции or, поскольку также имеет произвольное количество аргументов и не всегда их вычисляет. Отличие состоит в возвращаемом ею результате и условии продолжения рекурсии. Если список аргументов Z пуст, в качестве значения функции and возвращается T, а если в списке Z один элемент – возвращается его значение. В случае, когда в списке Z больше двух элементов, вычисляется значение первого, и если оно отлично от NIL, то строится и вычисляется рекурсивный вызов функции and с оставшимися аргументами.

```
; определение в MuLisp функционала funcall
(defun funcall(nlambda L
          (eval (cons (eval (car L))(cdr L)))))
```

Напомним, функционал funcall имеет произвольное количество вычисляемых аргументов, но не менее одного (значение первого его аргумента – имя применяемой функции). При определении funcall реализована следующая идея: при вызове функционала аргументы не вычисляются и из них образуется список L, на основе которого строится и вычисляется нужный функциональный вызов. В этом функциональном вызове первым элементом берётся имя применяемой функции – значение первого элемента списка L, вычисленное вызовом внутренней eval. Остальные элементы функционального вызова – это аргументы применяемой функции, которые будут вычислены самой этой функцией. Для вычисления построенного функционального вызова служит внешнее обращение к функции eval.

В диалекте Common Lisp рассмотренные возможности определения особых функций (nlambda и список аргументов X) отсутствуют. В то же время есть другие средства, позволяющие при помощи функции defun определить новую функцию с произвольным количеством вычисляемых аргументов.

Список формальных параметров определяемой функции в Common Lisp может содержать так называемые *ключевые слова*, которые начинаются со знака &, записываются перед нужными параметрами списка и позволяют трактовать эти параметры определённым образом.

Ключевое слово &rest используется для описания функции с произвольным количеством вычисляемых аргументов. Формальный параметр, записанный в списке параметров после этого ключевого слова, будет связан со списком фактических параметров, указанных в вызове функции и несвязанных с другими формальными параметрами.

Например, функция list описывается следующим образом:

```
; определение в Common Lisp функции list
    (defun list (&rest LS) LS) .
```

При вызове этой функции с фактическими параметрами (их количество неограниченно), они будут вычислены, и из их значений будет составлен список LS, который и будет выдан в качестве значения функции.

По сути, в Common LISP обращение к функции defun вида

$$(\texttt{defun F (\&rest X)}\ e_1 \ \ldots\ e_n)\ ,\ n \geq 1$$

эквивалентно рассмотренному случаю II определения особой функции в MuLisp: $(\texttt{defun F X}\ e_1 \ \ldots\ e_n)$.

Покажем определение в Common LISP ещё одной функции с произвольным количеством вычисляемых аргументов.

```
; определение в Common Lisp функции funcall
(defun funcall(F &rest L)(eval (cons F (Qu L))))
(defun Qu(L) ;вспомогательная функция квотирования
   (cond ((null L) NIL)
         (T (cons (list 'quote (car L))
                  (Qu (cdr L))))))
```

Для определения в диалекте Common Lisp функций, не вычисляющих свои аргументы, необходим механизм макросов.

4.5. Макросредства

Чтобы построить и затем вычислить построенное выражение, можно использовать базовую функцию eval (примеры этого были даны ранее). Однако проще и естественнее это сделать с помощью определяемых пользователем макросов, или *макрофункций*. Заметим, то макросредства не встроены в MuLisp, однако возможность их определения и использования появляется после загрузки файла common.lsp.

Определение макроса похоже на определение функции:

$$(\texttt{defmacro}\ M\ (x_1 \ \ldots\ x_k)\ e_1 \ \ldots\ e_n)\ ,\ n \geq 0$$

где M – имя макроса,

$x_1 \ \ldots\ x_k$ – его параметры, $k \geq 0$,

а $e_1 \ \ldots\ e_n$ – тело.

Однако вычисление макровызова (M p_1 … p_k) с фактическими параметрами p_1 … p_k отлично от вычисления функционального вызова. Опишем этапы вычисления макровызова для случая n=1, когда в теле макроса стоит одно выражение:

(defmacro M (x_1 … x_k) e)

1) Связывание: формальные параметры макроса x_1 … x_k попарно связываются с фактическими параметрами макровызова p_1 … p_k, т.е. $x_1 = p_1, \ldots, x_k = p_k$.
2) Расширение (раскрытие): вычисляется форма e (тело макроса), при этом всюду в качестве значений форм x_1 … x_k берутся связанные с ними значения p_1 … p_k . В результате этого вычисления строится новая форма (вычислимое лисповское выражение), и связи формальных параметров x_1 … x_k с фактическими разрываются.
3) Вычисление: построенная на предыдущем этапе форма вычисляется, и результат её вычисления становится значением макровызова.

Отличие вызова макроса от вызова обычной функции состоит, во-первых, в том, что фактические параметры макроса не вычисляются. Во-вторых, вычисление тела макроса происходит как бы дважды, в два этапа: сначала его вычисление с использованием локальных связей x_i с p_i и получение *расширенной формы* (она называется так, потому что, как правило, больше и сложнее исходной формы), затем вычисление полученной формы уже вне контекста макровызова. Таким образом, содержащее макровызов выражение эквивалентно выражению, в котором макровызов заменён на его расширенную форму.

Укажем также различие макросов и функций, строящих и вычисляющих выражение с помощью базовой функции eval. Оба вычисления, выполняемые функцией eval, происходят в контексте вызова функции, вычисление же расширенной формы макровызова происходит уже вне контекста его вызова.

Макросы, так же как и функции, могут быть рекурсивными. В таком случае на этапе макрорасширения может быть получена форма, содержащая макровызовы. Тогда на этапе вычисления расширенной формы содержащиеся в ней макровызовы будут вычисляться по тем же (описанным выше) правилам.

Макросы – удобное средство расширения языка Лисп, поскольку с их помощью несложно определить новые синтаксические конструкции. Приведём пример определения макрофункции (If C E1 E2), встроенной в MuLisp и Common Lisp, которая вычисляет значение

выражения E1, если значение выражения C отлично от NIL, в ином случае она вычисляет значение E2:

```
(defmacro If (C E1 E2)
    (list 'cond (list C E1)(list T E2)) )
```

Этот макрос строит и вычисляет условное выражение cond, в котором в качестве условия первой ветви берётся выражение C (первый аргумент If), а выражения E1 и E2 (второй и третий аргумент If) размещаются соответственно на первой и второй ветви cond.

К примеру, для макровызова (If(numberp K)(+ K 10) K) на этапе макрорасширения будет построена конструкция (cond ((numberp K)(+ K 10)(T K)), а на этапе её вычисления в случае K=5 будет получено значение 15.

Поскольку для построения вычислимой формы в макросах обычно используется довольно большое количество вложенных друг в друга вызовов функций cons, list и append, определение макросов часто становится громоздким и не очень понятным. Для упрощения описания макросов в Лиспе применяется специальное средство блокировки вычислений, которое называют *обратной блокировкой*. В отличие от обычной блокировки вычислений функцией quote (или ') обратная блокировка вычислений записывается с помощью апострофа, наклонённого в другую сторону: ` (этот знак иногда называется обратной кавычкой от англ. *back quote*).

Отличие выражений, к которым применена обратная блокировка, от просто заквотированных выражений в том, что в них можно локально отменять блокировку вычислений, т.е. осуществлять вычисление некоторых подвыражений. Отмена блокировки помечается запятой (,), которая ставится перед каждым вычисляемым подвыражением. Каждое помеченное запятой подвыражение на этапе макрорасширения вычисляется и заменяется на его значение. Такое использование запятой называют обычно *замещающей отменой блокировки*.

Запятую можно использовать также вместе со знаком @, тогда вместо подвыражения, помеченного двумя знаками ,@, на этапе макрорасширения подставляется его значение без внешних скобок. Такую отмену блокировки называют *присоединяющей*.

Приведём примеры. Пусть переменные имеют следующие значения:

```
x=>A     y=>(A S D)     z=>(F (C (V)) G (7))
```

тогда

```
`(z x (,x 5) ,y)  =>  (Z X (A 5)(A S D))
`((1 2) ,x 3 4 ,@y (5 6) ,@z 7)  =>
          ((1 2) A 3 4 A S D (5 6) F (C (V)) G (7) 7)
```

```
`(,x (z ,(cadr z))(,(caddr y) ,@(cdr z)))  =>
                  (A (Z (C (V)))(D (C (V)) G (7)))
```

Использование обратной блокировки и её отмены (замещающей и присоединяющей) позволяет компактно и более понятно записать макроопределение. К примеру, определение макроса `If` можно записать так: `(defmacro If (C E1 E2) `(cond (,C ,E1)(T ,E2)))`.

Обратную блокировку вычислений допускается использовать не только при определении макросов, но и при определении функций. Например, данное для диалекта MuLisp в разделе 3.5 определение функционала `Twice` с функциональным значением (он возвращает функцию, которая дважды применяет его функциональный аргумент) можно существенно упростить:

```
; определение Twice при помощи обратной блокировки,
; требует предварительной загрузки файла common.lsp
(defun Twice (F)
     `(lambda (X)(funcall (quote ,F)
                          (funcall (quote ,F) X))))
```

Приведём ещё один пример, демонстрирующий изменение вычислительного контекста при обработке макровызова. Пусть описан макрос `ASD`:

```
(defmacro ASD (X Y)
  `(cons (list ,(car X) ,@(cdr X)) (list Y ,@Y)))
```

Вычисление выражения

```
(let ((X 12) (Y '(A B)) (Z '(1 2 3)))
     (ASD (X Y)(X Y Z)))
```

будет происходить следующим образом:

1. При вызове конструкции `let` вычислительный контекст будет пополнен связями `X=12`, `Y =(A B)`, `Z =(1 2 3)` , после чего будет вызван макрос `ASD`: `(ASD (X Y)(X Y Z))`.
2. На этапе макрорасширения этого вызова формальные параметры макроса (`X` и `Y`) будут связаны с фактическими параметрами вызова: `X=(X Y)`, `Y= (X Y Z)`, и при вычислении тела макроса вместо выражения `,(car X)` будет подставлено `X`, вместо выражения `,@(cdr X)` подставится `Y`, а вместо выражения `,@Y` – последовательность `X Y Z`. Таким образом будет построено выражение `(cons (list X Y)(list Y X Y Z))`, после чего связи между фактическими и формальными параметрами макроса разрываются.

3. Восстанавливаются прежние значения переменных X и Y: X=12, Y =(A B), и в этом контексте будет вычислено полученное на этапе макрорасширения выражение. Результатом вычисления макровызова, а также всей конструкции let будет выражение ((12 (A B))(A B) 12 (A B)(1 2 3)).

Макросы часто используются для определения особых функций, которые не вычисляют или не всегда вычисляют свои аргументы. В качестве такого примера приведём рекурсивные макроопределения встроенных функций or и and. В предыдущем разделе их определения были даны для MuLisp с помощью конструкции nlambda. Поскольку в Common Lisp эта конструкция отсутствует, указанные функции в этом диалекте могут быть описаны только в виде макросов:

```
; макроопределение в Common Lisp функции or
    (defmacro or(&rest X)
      (cond ((null X) NIL)
            ((eval (car X)) (car X))
            (T (cons 'or (cdr X)))))
; макроопределение в Common Lisp функции and
    (defmacro and(&rest X)
       (cond ((null X) T)
             ((null (cdr X)) (car X))
             ((eval (car X)) (cons 'and (cdr X))) ))
```

Заметим, что для обработки произвольного количества аргументов в определениях этих макрофункций используется ключевое слово &rest, и поэтому с формальным параметром X связывается список невычисленных аргументов функции (т.е. фактических параметров). В результате макрорасширения по второй ветви функции будет выдан первый невычисленный аргумент, который на следующем этапе обработки макровызова будет вычислен. Аналогично, при макрорасширении по третьей ветви будет сформирован рекурсивный вызов or или and (аргументами функции служат элементы списка X, за исключением первого), и на следующем этапе этот вызов будет вычисляться – при этом опять будут повторены описанные этапы обработки макровызова. Рекурсия завершится, когда сработает первая или вторая ветвь макрофункции, т.е. когда список X станет пуст или когда значение первого его элемента отлично от NIL.

Рассмотрим, к примеру, последовательность этапов вычисления макровызова (or (eq 'A 'B)(+ 2 3) NIL (atom 'C)):

1) В результате макрорасширения этого вызова (проработает третья ветвь cond) будет получено выражение
```
(or (+ 2 3) NIL (atom 'C)).
```
2) Поскольку полученная форма сама является макровызовом, она в результате макрорасширения (сработает вторая ветвь cond) будет заменена на выражение (+ 2 3).
3) Это выражение будет вычислено (на этапе вычисления макрорасширения), и его значение (число 5) будет возвращено как результат вычисления исходного макровызова.

4.6. Циклы, блоки и присваивания

Во многих диалектах Лиспа имеются функции-аналоги широкоупотребительных операторов традиционных императивных языков программирования: операторов присваивания, цикла, перехода и др. Особенность этих функций в том, что в основном они опираются на побочный эффект вычислений, а значит, их использование не вписывается в рамки строго функционального программирования.

Кратко опишем некоторые из этих функций.

Для присваивания переменной значения используется особая встроенная функция setq с обращением (setq *имя_переменной* e). Первый аргумент функции setq не вычисляется, он должен быть именем переменной (символьным атомом). Второй аргумент функции – лисповское выражение-форма e, оно вычисляется, и его значение присваивается указанной переменной. Значением функции setq является вычисленное значение второго аргумента.

Функция set отличается от setq тем, что вычисляет оба своих аргумента, т.е. является обычной. Значением первого аргумента должно быть имя переменной, которой и присваивается значение второго аргумента.

Обе описанные функции есть в диалектах MuLisp и Common Lisp. Примеры использования функций присваивания:
```
(setq Var '(12 8 67.5))
(set (cadr '(A S D)) (mapcar 'add1 Var))
```
В Common Lisp для программирования повторяющихся вычислений применяется конструкция-цикл do, обращение к ней имеет вид:
```
(do (v1 v2 … vm)
    (p t1 t2 … tк)
        e1 e2  … en )        m≥0, k≥0, n≥0
```
где (*v*₁ *v*₂ … *v*ₘ) – список переменных цикла (может быть пуст);

(p t_1 t_2 … $t_к$) задаёт условие p окончания цикла и выражения t_i, вычисляемые при выходе из цикла;
e_1 e_2 … e_n – тело цикла.

Более точно, v_i – это либо имя переменной, либо список вида
(*имя_переменной начальное_значение* [*следующее_значение*]),
в котором в качестве начального и следующего значения могут браться любые вычислимые лисповские выражения (формы); квадратные скобки указывают на необязательность соответствующего выражения.

Вычисление функции do начинается с одновременного присваивания переменным цикла начальных значений; если начальное значение не задано, то переменной присваивается NIL. Затем вычисляется условие окончания p, и если его значение отлично от NIL, то последовательно вычисляются формы t_1 ... t_k, и значение последней их них возвращается как значение обращения к do. В ином случае последовательно вычисляются выражения тела цикла e_1 e_2 ... e_n, и начинается выполняться следующий шаг цикла.

На каждом следующем шаге цикла сначала переменным цикла одновременно присваиваются значения форм *следующее_значение*, вычисляемых в текущем контексте; если же для некоторой переменной эта форма не задана, то её значение не меняется. Затем вновь проверяется условие окончания p и так далее. Цикл заканчивается, когда значение p станет неравным NIL.

Заметим, что в случае k=0 значением функции do будет NIL.

В качестве примера цикла do приведём определение функции ReverseDo, меняющей порядок элементов верхнего уровня списка на противоположный.

```
(defun ReverseDo1 (L)
    (do ((Rez NIL)) ;переменная цикла
        ((null L) Rez) ; условие окончания
        (setq Rez (cons (car L) Rez))
        (setq L (cdr L))))
```

Эту функцию можно определить иначе:

```
(defun ReverseDo (L)
  (do ((Lst L (cdr Lst));нач.установка и пересчет Lst
       (Rez NIL (cons(car Lst) Rez)) ) ;пересчёт Rez
      ((null Lst) Rez))) ;условие окончания
```

Во втором варианте определения функции ReverseDo тело цикла пусто, пересчёт значений переменных цикла происходит при переходе к следующему его шагу.

Ещё одна встроенная в Common Lisp функция для программирования циклов – это функция do*. Она отличается от do тем, что вычисление и присваивание значений переменным цикла происходит последовательно: сначала вычисляется и присваивается значение первой переменной, затем второй и т.д. Тем самым в выражении для вычисления значения очередной переменной можно использовать уже установленные значения предшествующих переменных.

В диалекте MuLisp функции do и do* не являются встроенными, их можно использовать только после загрузки файла common.lsp.

Встроенной функцией MuLisp для программирования циклов является функция loop с обращением

(loop s_1 … s_n) , n≥1.

Она последовательно вычисляет выражения s_1 … s_n, переходя после вычисления s_n к s_1, пока не сработает условие выхода из цикла. Выражение s_i может быть обычной лисповской формой, либо выражением вида (p e_1 e_2 … e_k), k≥0, где p – условие выхода из цикла.

По сути, s_i в последнем случае – это ветвь условного выражения cond, и при её вычислении сначала вычисляется условие p. Если значение p равно NIL, то далее продолжается последовательное вычисление выражений, стоящих после s_i. Если же значение p отлично от NIL, то вычисляются формы e_1, e_2, … e_k и выполнение цикла завершается, а значением функции loop становится значение последнего выражения e_k.

Заметим, что среди s_1 … s_n может быть несколько таких условных выражений для выхода из цикла loop.

В качестве примера применения loop приведём очередное определение функции Reverse:

```
(defun ReverseLoop (L)
   (let ((Res NIL)) ; локальная переменная
        (loop ((null L) Res) ;условие окончания цикла
              (setq Res (cons (car L) Res)
              (setq L (cdr L))   )))
```

Во многих диалектах Лиспа есть следующие блочные конструкции (в MuLisp встроенными являются только prog1 и progN):

(prog1 e_1 e_2 … e_N) , N≥1
(prog2 e_1 e_2 … e_N) , N≥2
(progN e_1 e_2 … e_N) , N≥1

Все эти функции осуществляют последовательное вычисление форм e_1 e_2 … e_N, а в качестве результата своего вычисления функция prog1 возвращает значение e_1, prog2 – значение e_2, а progN – значение e_N.

Ещё одна блочная конструкция prog, встроенная в Common Lisp, служит для программирования в операторном стиле с использованием передачи управления. Структура этой формы prog следующая:

(prog (v_1 … v_k) e_1 e_2 … e_n) , $k \geq 0$, $n \geq 0$.

В начале блока задаётся список локальных переменных v_1 … v_k (он может быть пустым), все переменные инициализируются значением NIL. Любое выражение e_i является либо вычислимой формой, либо *меткой*. В качестве метки может использоваться символьный атом или целое число.

Выражения e_1 e_2 … e_n блока вычисляются последовательно, при этом метки игнорируются. Такое последовательное вычисление может быть нарушено функциями-операторами go и return, которые могут встретиться среди выражений блока.

При вычислении особой функции (go *метка*) происходит переход на заданную метку (аргумент функции go не вычисляется), после которого будет вычисляться выражение блока, следующее за этой меткой.

Функция return служит для выхода из блока. При вычислении обращения к ней (return e) вычисляется выражение-форма e, далее выполнение блока прекращается, и в качестве значения функции prog возвращается значение e.

Если же во время вычисления блока оператор return не встретился, значением функции prog после вычисления последнего выражения e_n станет значение NIL.

В качестве примера использования prog приведём ещё одно определение функции Reverse:

```
(defun ReverseProg (L)
   (prog (Res)
        C    ;метка перехода
        (cond((null L)(return Res))) ;выход из блока
        (setq Res (cons (car L) Res))
        (setq L (cdr L))
        (go C)  )) ; переход по метке
```

5. Задания практикума

Задания практикума были разработаны в поддержку теоретического изучения языка Лисп и предполагают составление лисп-программ решения типичных задач анализа и обработки символьных данных.

Отождествление рефал-выражений

Реализовать алгоритм отождествления выражений языка Рефал в виде лисповской функции (Match P L), где L – произвольный список, P – список-образец, в состав которого кроме обычных атомов входят атомы-переменные вида SX, WX, EX и VX . Буква E, W, V или S обозначает тип переменной (множество ее допустимых значений). Буква X играет роль имени переменной; в качестве имени может быть взята любая латинская буква или десятичная цифра.

Функция Match производит *сопоставление* (*отождествление*) списка L с образцом P по правилам языка Рефал, рассматривая лисповские скобки как структурные рефальские скобки. Функция вырабатывает значение T или NIL в зависимости от успешности сопоставления. В случае удачного сопоставления переменные образца получают соответствующие значения: значением переменной вида SX может быть лисповский атом, а значением переменной вида WX – лисповское выражение (атом или список). Переменная вида EX или VX может быть сопоставлена соответственно произвольной или непустой последовательности лисп-выражений, при этом её значением является список из этих выражений.

Пример успешного сопоставления:

(Match '(WA (C1 SB) EX) '(DO (C1 2) 5 ON)) ⇒ T,

при этом: WA => DO, SB => 2, EX => (5 ON).

Набор тестов для функции Match должен включать тесты, демонстрирующие её применение для распознавания структуры алгебраических выражений.

Преобразование алгебраического выражения

Преобразовать в приведённый полином заданное алгебраическое выражение, в котором используются операции сложения, вычитания, умножения, возведения в степень, целые числа, однобуквенные переменные, а также круглые скобки, задающие нужный порядок вычисления операций.

Полином представляет собой сумму или разность нескольких одночленов. Одночленом может быть целое число, а также произведение целого числа и нескольких множителей – целых положительных степеней переменных (степень, равная единице, не записывается). Знак

произведения в записи полинома опускается. Например, полиномом является запись X+XY-15Y↑3+2.

Полином называется *приведённым*, если в нём нет подобных одночленов, а в каждом его одночлене любая из переменных не встречается более одного раза. Приведённым является полином X+XY-15Y↑3+2, полиномы же 46ZYZ и X+X-7+3XY+2 не являются приведёнными.

В упрощённом решении этой задачи при записи исходного алгебраического выражения знаки арифметических операций, числа и переменные разделяются пробелами, а само выражение заключается в круглые скобки (чтобы сделать его лисповским списком и тем самым облегчить его ввод и обработку). В полном решении задачи на вход программы подаётся текст алгебраического выражения, которое обычно не содержит объемлющих скобок и в котором знаки операций, имена переменных и числа могут быть записаны слитно.

Суммирование рациональных выражений

Реализовать символьное вычисление суммы двух заданных рациональных выражений. Полученное рациональное выражение должно состоять только из приведённых многочленов.

Рациональное выражение представляет собой дробь, в числителе и знаменателе которой стоят полиномы от одной однобуквенной переменной. Полиномы являются в свою очередь суммой или разностью нескольких одночленов. Одночленом может быть целое число, а также произведение целого числа и целой положительной степени переменной (степень, равная единице, не записывается). Знак операции умножения в записи одночлена опускается. Например, полиномом является запись X-15X↑3+2. Указанный полином не содержит подобных одночленов, такие полиномы называются *приведёнными*.

В упрощённом решении задачи при записи исходных рациональных выражений можно заключить их в скобки, а знаки арифметических операций, числа и переменные разделять пробелами (чтобы упростить ввод и обработку выражений). Например, выражения можно задать так: ((7 + 15 - 3 X) / X)+(X /(5 X ↑ 8)), и в результате будет вычислено выражение
(110 X ↑ 8 – 15 X ↑ 9 + X ↑ 2) / (5 X ↑ 9) , или с учётом упрощения: (110 X ↑ 6 – 15 X ↑ 7 + 1) / (5 X ↑ 7)

В полном решении задачи на вход программы поступают тексты рациональных выражений без лишних скобок, а числа, буквы переменных и знаки операций могут не разделяться пробелом, например: X/5X↑8.

Преобразование формулы алгебры логики в ДНФ

Преобразовать в *сокращенную дизъюнктивную нормальную форму* произвольную формулу алгебры логики, в которой используются логические константы, однобуквенные переменные и операции логического отрицания, конъюнкции, дизъюнкции и импликации. Результирующая дизъюнктивная нормальная форма должна быть *правильной*, т.е. каждая её элементарная конъюнкция не должна содержать повторных вхождений переменных, и все конъюнкции должны быть различными. Полученная формула не должна содержать избыточных скобок. Например, формула $(\neg A \vee D \vee FALSE) \supset \neg(B \& \neg C \vee D)$ преобразуется к виду $A \& \neg D \vee \neg B \& \neg D \vee C \& \neg D$.

В упрощённом решении задачи при записи исходной логической формулы знаки логических операций, константы и переменные разделяются пробелами (если только между ними не стоит другой разделитель – круглая скобка), а сама формула заключается в объемлющие скобки (чтобы сделать её лисповским списком и упростить её ввод и преобразование), например: $(A \supset (B \& \neg C \vee D))$

В полном решении задачи на вход программы поступает текст формулы, который обычно не содержит объемлющих скобок и в котором все символы могут быть записаны слитно.

Построение по ДНФ равносильной ей КНФ

По заданной дизъюнктивной нормальной форме некоторой булевской функции построить ее *совершенную конъюнктивную нормальную форму*, а также *сокращенную конъюнктивную нормальную форму*. Как исходная дизъюнктивная нормальная форма, так и результирующие конъюнктивные нормальные формы предполагаются *правильными*, т.е. все переменные, встречающиеся в каждой элементарной конъюнкции/дизъюнкции, различны, а также нет одинаковых элементарных конъюнкций/дизъюнкций. В *совершенной* КНФ каждая элементарная дизъюнкция содержит вхождения всех переменных булевской функции, в *сокращённой* форме это не требуется.

В исходной ДНФ используются только однобуквенные переменные. Построенные КНФ не должны содержать избыточных скобок. Например, для ДНФ $A \vee B \& \neg C$ получается сокращённая КНФ $(A \vee B) \& (A \vee \neg C)$ и совершенная КНФ $(A \vee B \vee C) \& (A \vee B \vee \neg C) \& (A \vee \neg B \vee \neg C)$.

В упрощённом решении задачи при записи исходной ДНФ знаки логических операций и переменные разделяются пробелами, а сама ДНФ заключается в объемлющие скобки (чтобы сделать её списком и упростить её ввод и обработку). В полном решении задачи на вход программы

подаётся запись ДНФ без объемлющих скобок, в которой символы переменных и знаки операций могут быть записаны слитно.

Сколемизация формулы исчисления предикатов

Преобразовать в *предварённую нормальную форму* правильную замкнутую формулу исчисления предикатов первого порядка, в которой используются переменные, предикаты, кванторы общности и существования, а также логические операции (конъюнкция, дизъюнкция, импликация, отрицание). Переменные и предикаты исходной формулы предикатов имеют однобуквенные имена.

Преобразование включает следующие этапы:

1) исключение операции импликации;
2) уменьшение области действия операции отрицания;
3) исключение кванторов существования путем *сколемизации*, т.е. введения функций Сколема;
4) вынесение кванторов общности в начало формулы.

Получаемая в результате этих преобразований формула включает *префикс* – цепочку кванторов общности и *основу* (матрицу), не содержащую кванторы. В этой формуле должны быть удалены все избыточные (не влияющие на порядок выполнения операций) скобки.

Например, формула $\forall x(P(x)\supset\neg(\exists y(Q(x,y)\supset P(y))))$ преобразуется к виду: $\forall x(\neg P(x)\vee Q(x,g(x)) \ \& \ \neg P(g(x)))$, где $g(x)$ – функция Сколема.

Предполагается, что в исходной формуле исчисления предикатов каждый квантор общности и существования связывает переменную с уникальным именем – поэтому для преобразования в предварённую нормальную форму нет необходимости производить переименование переменных с одним именем, но связываемых разными кванторами. Учесть также, что в формуле могут быть опущены незначащие скобки, т.е. скобки, не изменяющие порядка выполнения операций.

В упрощённом решении задачи при записи исходной формулы предикатов знаки логических операций, кванторов, имена переменных и предикатов разделяются пробелами, а сама формула заключается в объемлющие скобки (чтобы сделать её лисповским выражением и тем самым облегчить её ввод и преобразование). В полном решении задачи на вход программы подаётся запись формулы, в которой нет объемлющих скобок, а все символы имён переменных, предикатов, кванторов и операций могут быть записаны слитно.

Построение конечного автомата по грамматике

По заданному тексту *регулярной* (леволинейной или праволинейной) формальной грамматики построить соответствующий конечный автомат для распознавания предложений языка, порождаемого этой грамматикой.

Грамматика задаётся как конечный набор правил вида `T = aN|b`, альтернативы в правых частях правил не могут быть пустыми. Нетерминальные символы грамматики записываются заглавными (большими) латинскими буквами, а терминальные – строчными (маленькими). Начальный символ грамматики обозначается буквой `H` (для праволинейной) или `S` (для леволинейной грамматики).

Построенный автомат представляет собой ориентированный и помеченный граф. Вершины графа соответствуют состояниям автомата и помечены нетерминальными символами грамматики; в множество вершин входит начальное состояние `H` и заключительное состояние `S`. Рёбра графа соответствуют переходам между состояниями автомата и помечены терминальными символами грамматики. Граф записывается в виде списка входящих в него рёбер, каждое ребро представлено трехэлементным списком вида (*метка_вершины метка_ребра метка_вершины*).

В упрощённом решении задачи при записи исходной грамматики можно заключить в круглые скобки каждое её правило и составить из них лисповский список, в котором все символы грамматики разделены пробелами, а знак `|` и строчные буквы записаны как особые символы языка Лисп. Например:

```
((H = \a N \| \b N) (N = \c N \| \d)).
```

В полном решении задачи на вход программы подаётся текст грамматики без разделяющих пробелов, при этом правила грамматики разделены точкой с запятой, к примеру: `H=aN|bN;N=cN|d`. В случае недетерминированности полученного по грамматике автомата необходимо построить эквивалентный ему детерминированный автомат (детерминированный автомат, распознающий тот же самый язык).

Построение регулярной грамматики по конечному автомату

По заданному конечному автомату восстановить соответствующую *регулярную* леволинейную (или праволинейную) формальную грамматику, включающую алфавиты (множества) терминальных и нетерминальных символов и набор правил грамматики, а также начальный символ грамматики: `H` (для праволинейной) или `S` (для леволинейной).

Конечный автомат задан как ориентированный граф, вершины которого соответствуют состояниям автомата и помечены нетерминальными символами грамматики. В множество вершин входит

начальное состояние H и заключительное S. Рёбра графа соответствуют переходам между состояниями автомата и помечены терминальными символами грамматики. Граф записан как лисповский список входящих в него рёбер, каждое ребро представлено трёхэлементным списком вида (*метка_вершины метка_ребра метка_вершины*).

В записи восстановленной по конечному автомату грамматики нетерминальные символы грамматики записываются заглавными (большими) латинскими буквами, а терминальные – строчными (маленькими). Правая и левая часть каждого правила разделяются знаком =, а сами правила – знаком ; , в полученном тексте нет пробелов, например: B=aN|bN;N=cN|d.

Исходный конечный автомат может быть как детерминированным, так и недетерминированным. В полном решении задачи в случае недетерминированности автомата необходимо построить, кроме грамматики, эквивалентный ему детерминированный автомат (детерминированный автомат, распознающий тот же самый язык).

Исследование свойств графа

Для заданного графа провести исследование трёх его свойств, выбрав по одному свойству из трёх следующих групп:

1) – Связность графа: в случае связности графа необходимо найти его каркас, в ином случае – все его компоненты связности;
   – Древесность графа: является ли граф деревом или лесом, в последнем случае найти составляющие его деревья;
2) – Двудольность графа; если граф двудольный, то необходимо определить, является ли он полным двудольным графом;
   – Существование в графе мостов и точек сочленения, в случае существования найти их количество;
3) – Является ли граф эйлеровым, и если да, то найти эйлеров цикл;
   – Является ли он гамильтоновым, и если да, то найти гамильтонов цикл.

Каркас графа есть наименьшее по числу рёбер входящее в него дерево, сохраняющее связность между всеми его вершинами. Следует учесть, что в общем случае каркас в графе находится неоднозначно.

Дерево есть связный граф, не имеющий циклов. Совокупность несвязанных деревьев есть *лес*.

Граф называется *двудольным* (*биграфом*), если множество его вершин допускает разбиение на два непересекающихся подмножества (доли), причём каждое ребро графа соединяет вершины из разных долей. Двудольный граф есть *полный двудольный* граф, если каждая вершина одной доли соединена ребром с каждой вершиной другой доли. Граф

двудолен тогда и только тогда, когда все простые циклы в нём имеют чётную длину.

Мостом называется ребро, удаление которого увеличивает число компонентов связности графа. *Точка сочленения* – вершина, удаление которой ведёт к увеличению количества компонентов связности;

Замкнутый путь (цикл) в графе называется *эйлеровым*, если он проходит через каждое ребро графа; при этом граф с таким циклом называется *эйлеровым*. Цикл в графе называется *гамильтоновым*, если он содержит все вершины графа ровно по одному разу; граф с таким циклом называется *гамильтоновым*.

Исследуемый граф записывается как лисповский список входящих в него рёбер, а каждое ребро – как двухэлементный список из меток его вершин. В качестве меток вершин используются буквенные идентификаторы или целые числа.

Поиск путей по карте дорог

Задана карта железнодорожных и шоссейных дорог между несколькими городами, все дороги между городами являются двусторонними. Не для всех возможных пар рассматриваемых городов существует железнодорожная или шоссейная дорога между ними, в то же время между некоторыми городами возможно наличие обоих типов дорог. Известно, что дорожное сообщение позволяет добраться из произвольного города в любой другой город.

Для двух заданных городов найти все связующие их пути без циклов, длина которых не более чем в полтора раза превышает минимальный по длине путь между указанными городами, и из этих путей выбрать путь с минимальным количеством пересадок с одного вида транспорта на другой.

В качестве результата вывести найденный минимальный по длине путь и путь с минимальным числом пересадок, указав в нём пересадки. Если таких путей несколько, то последовательно показать их все, упорядочив по длине пути. Если такого пути не существует, предложить любой другой приемлемый путь из одного города в другой.

Карта дорог задана в виде помеченного связного *мультиграфа*, вершины которого соответствуют городам и помечены названиями городов, а рёбра графа соответствуют дорогам между городами. Каждое ребро графа помечено меткой типа дороги (железнодорожная, шоссейная), а также целым числом – длиной дороги между соответствующими городами. Граф записан как список входящих в него рёбер, причём каждое ребро представлено четырёхэлементным списком, который включает названия городов, соединенных этим ребром (дорогой), а также метку типа дороги и её длину (расстояние между городами).

6. Литература

1. McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 1960, Vol. 3, N 4, p. 184-195.
2. McCarthy J. Lisp 1.5 Programmer's Manual. MIT Press, Cambridge, Massachusetts, 1963.
3. Revised Report on the Algorithmic Language Scheme. Rees J., Clinger W., (Eds). SIGPLAN Notices 21(12), December 1986.
4. Steele, G.L. Common Lisp – the Language (Second Edition). Digital Press, Hanover, Massachusetts, 1990.
5. Common Lisp Hyper Spec – http://www.lispworks.com/documentation/HyperSpec/Front/index.htm
6. Баррон Д. Рекурсивные методы в программировании : Пер. с англ. – М: Мир, 1974.
7. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка : Пер. с англ. – М: Финансы и статистика, 1982.
8. Лавров С.С., Силагадзе Г.С. Автоматическая обработка данных. Язык лисп и его реализация. – М.: Наука, 1978.
9. Маурер У. Введение в программирование на языке лисп: Пер. с англ. – М.: Мир, 1976.
10. Пильщиков В.Н. Язык Плэнер. – М: Наука, 1983.
11. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация: Пер. с англ. – СПб: Питер, 2002.
12. Руководство пользователя по языку AutoLisp – http://www.codenet.ru/progr/alisp/
13. Семенов М.Ю. Язык лисп для персональных ЭВМ. – М: Изд-во Моск. ун-та, 1989.
14. Филд А., Харрисон П. Функциональное программирование: Пер. с англ. – М.: Мир, 1993.
15. Хендерсон П. Функциональное программирование. Применение и реализация: Пер. с англ. – М.: Мир, 1983.
16. Хювёнен Э., Сеппянен Й. Мир Лиспа. Том 1. Введение в язык Лисп и функциональное программирование: Пер. с англ. – М: Мир, 1990.

Приложение: Встроенные функции языка Лисп