Содержание

Часть1

Т Е О Р И Я

Часть1

ОСНОВЫ ЯЗЫКА ЛИСП

1.1 СИМВОЛЫ И СПИСКИ

Символы языка Коммон Лисп могут состоять из букв, цифр и некоторых других знаков, а именно

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

Символы могут состоять как из прописных, так и из строчных букв.

При желании в состав символов можно включать пробелы и другие зарезервированные специальные знаки. Для этого либо символ с двух сторон ограничивается вертикальной чертой (|, bar), либо перед каждым таким специальным знаком, входящим в символ, ставится обратная косая черта (\, backslash).

Числа являются константами.

Наряду с символами в Лиспе используются и числа, которые, как и символы, записываются при помощи ограниченной пробелами последовательности знаков. Числа все же не являются символами, так как число не может представлять иные лисповские объекты, кроме самого себя, или своего числового значения.

В Лиспе для различных целей используется различные типы чисел. Примеры чисел:

```
  746      ;  целое число
-3.14      ;  десятичное число
3.055E8    ;  число, представленное мантиссой и порядком
```

Числа отличаются от символов способом записи.

Логические значения T и NIL

Символы T и NIL имеют в Лиспе специальное назначение: T. обозначает логическое значение истина (true), а NIL - логическое значение ложь (false). (для обозначения логического значения ложь используется также специальный символ F (false).) Символом NIL обозначается также и пустой список. Символы T и NIL имеют всегда одно и то же фиксированное встроенное значение. Их нельзя использовать в качестве имен других лисповскнх объектов.

Константы и переменные

Числа и логические значения T и NIL являются константами (constant), остальные символы - переменными (variable), которые используются для обозначения других лисповских объектов.

Система Коммон Лисп содержит глобальные специальные переменные (global special/dynamic variable), имеющие изменяемые встроенные значения. Например, такими переменными являются символ PI, представляющий значение числа π, и символ *STANDARD-INPUT*, определяющий, откуда вводятся значения. В языке Лисп предусмотрена специальная директива (DEFCONSTANT), используемая для превращения любого символа в константу. Символ, определенный как константа, может обозначать лишь то значение, которое ему предписано таким определением.

Атомы - Символы + Числа

Символы и числа представляют собой те простейшие объекты Лиспа, из которых строятся остальные структуры. Поэтому их называют атомарными объектами или просто атомами (atom).

Построение списков из атомов и подсписков

Атомы и списки (list) - это основные типы данных языка Лисп. В естественном языке под списком понимают перечень. **Список** в Лиспе - это упорядоченная последовательность, элементами (element) которой являются атомы либо списки (подсписки). Списки заключаются в круглые скобки, элементы списка разделяются пробелами. Список всегда начинается с открывающей скобки и заканчивается закрывающей скобкой. Например, следующий список состоит из трех символов и одного подсписка, который в свою очередь состоит из двух атомов:

(a b (c d) e)

Таким образом, список – это многоуровневая или иерархическая структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии. Например, приведенные ниже выражения являются правильно составленными списками:

(+ 2 3) ; список из трех элементов

(((((первый) 2) третий) 4) 5) ; список из двух элементов

В следующих списках соответственно 5 и 4 элемента:

(Добрый день сказал бородатый мужчина)

(кот-37 (кличка Петя) (цвет ?) (хвост nil))

Пустой список = NIL

Список, в котором нет ни одного элемента, называется пустым списком и обозначается "()" или символом NIL. Пустой список - это не то же самое, что "ничего". Он выполняет ту же роль, что и нуль в арифметике. NIL может быть, например, элементом других списков:

NIL то же, что и ()

(NIL) список, состоящий из атома NIL

(()) то же, что н (NIL)

((())) то же, что н ((NIL))

(NIL ()) список из двух пустых списков

Атомы и списки называются символьными выражениями или s-выражениями (s-expression).

Различная интерпретация списков

Одним из основных отличий языка Лисп является запись в виде списков не только данных, но и программ (или функций). Например, список: (+ 2 3) можно интерпретировать в зависимости от окружения и использования либо как действие, дающее в результате число 5, либо как список, состоящий из трех элементов. Способ интерпретации определяется положением выражения и алгоритмом функционирования Лисп-системы.

1.2 ПОНЯТИЕ ФУНКЦИИ

Определение и вызов функции

Определение (definition) функции в языке Лисп позволяет задать произвольное вычислимое отображение. Например, функцию, вычисляющую сумму квадратов, можно определить с помощью сложения и умножения:

суммаквадратов:

аргументы: x, y

значение: x * x + y * y

Здесь x и y переменные, представляющие произвольные числа. Вызов (call) функции означает запуск, или применение (apply) определения функции к конкретным значениям аргументов. Например, применим функцию суммаквадратов к аргументам x = 2 и y = 3 :

суммаквадратов(2,3) > 13

Единообразная префиксная нотация

В языке Лисп как для вызова функции, так и для записи выражений принята единообразная **префиксная** форма записи, при которой как имя функции или действия, так и сами аргументы записываются внутри скобок:

(f x), (g x y), (h x (g y z)) и т.д.

Таким же образом записываются арифметические действия. Приведенные выше арифметические выражения в лисповской префиксной записи выглядели бы так:

(+ x y), (– x y), (* x (+ y z)) и т.д.

Единообразная и простая структура вызова функции удобна как для программиста, так и для вычисляющего значения выражений интерпретатора Лиспа. Вычисляя значения функций, не нужно осуществлять сложный синтаксический анализ выражений, так как сразу по первому символу текущего выражения система уже знает, с какой структурой имеет дело и как ее интерпретировать или как с ней обращаться.

Диалог с интерпретатором Лиспа

Лисп напоминает естественный язык и тем, что он обычно используется в диалоге (интерактивно) и в режиме интерпретации (interpretation). Интерпретатор Лиспа функционирует следующим образом. Когда пользователь заканчивает ввод какого-либо вызова функции или другого выражения, то интерпретатор вычисляет (evaluate) и выдает значение этого выражения. Если мы хотим, например, вычислить выражение (+ 2 3), то введем его в машину и сразу получим результат:

```
_(+ 2 3)   ;  пользователь вводит s-выражение
5          ;  интерпретатор вычисляет и выдает результат
```

Символ подчеркивания "_", показанный перед вводимым выражением, - это так называемое приглашение (prompt), при помощи которого интерпретатор дает знать, что он выполнил вычисление предыдущего выражения и ждет новое выражение :

```
_(– 3 2)   ;  пользователь вводит s-выражение
1          ;  интерпретатор вычисляет и выдает результат
_          ;     интерпретатор выдает на экран приглашение и ждет ввода
           ;     очередного выражения
```

Иерархия вызовов

В конструкцию вводимой функции могут, в свою очередь, входить функциональные подвыражения. Тогда аргументы вычисляемой функции заменяются на новые вычисления, ведущие к определению значений этих аргументов. Таким путем вычисления сводятся в конечном счете к вычислению выражений, значения которых, как, например констант, можно определить непосредственно, например:

(* (+ 1 2) (+ 3 4))

Вычисляя значения выражения, интерпретатор Лиспа сначала пытается слева направо вычислить значения аргументов внешнего вызова. Если первый аргумент является константой или иным объектом, значение которого можно определить

непосредственно, то вычисление аргументов вычисляемой функции можно продолжить. Если аргументом будет вложенный вызов функции или другая вычислимая форма, то перед ее вычислением определяются значения ее аргументов и т.д.

Наконец, опустившись на самый нижний уровень, можно начать в качестве значений аргументов предыдущего уровня подставлять значения, даваемые вложенными вычислениями. Последним возвращается значение всего выражения. Как и в математике, в Лиспе в первую очередь вычисляются выражения, заключенные в скобки:

_(* (+ 1 2) (+ 3 4))
21 ; = (1 + 2) * (3 + 4)
_(* (+ 1 2) (* 1 (+ 2 3)))
; = (1 + 2) * (1 * (2 + 3))

В Лиспе можно задавать и произвольный порядок вычислений.

QUOTE блокирует вычисление выражения

В некоторых случаях не надо вычислять значение выражения, а нужно само выражение. Нас, например, может не интересовать значение функционального вызова (+ 2 3), равное 5, а мы хотим обрабатывать форму (+ 2 3) как список. Чтобы предотвратить вычисление значения выражения, нужно перед этим выражением поставить апостроф “ ' ” (quote). В Лиспе начинающееся с апострофа выражение соответствует заключению в кавычки в обыкновенном тексте, и в обоих случаях это означает, что цитату нужно использовать в том виде, как она есть. Например:

_' (+ 2 3)
(+ 2 3)
_' y
Y

Апостроф перед выражением - это сокращение формы QUOTE, записываемой в единообразной для Лиспа префиксной нотации:

'выражение - (QUOTE выражение)

QUOTE можно рассматривать как специальную функцию с одним аргументом, которая ничего с ним не делает, даже не вычисляет его значение, а возвращает в качестве значения вызова сам этот аргумент.

_(quote (+ 2 3)
(+ 2 3)
_(quote y)
У

Интерпретатор Лиспа, считывая начинающееся с апострофа выражение, автоматически преобразует его в соответствующий вызов функции QUOTE. Числа не надо предварять апострофом, так как интерпретатор считает, что число и его значение совпадают. Однако использование апострофа перед числом не запрещено. Перед другими константами (T и NIL) тоже не надо ставить апостроф, поскольку и они представляют самих себя. Однако в приведенных примерах на месте имени функции в функциональном вызове апостроф не использовался. Интерпретатор Коммон Лиспа не допускает использование в качестве первого элемента вызова функции вычисляемого выражения.

1.3 БАЗОВЫЕ ФУНКЦИИ

Основные функции обработки списков

В Лиспе для построения, разбора и анализа списков существуют простые базовые функции, которые можно сравнить с основными действиями в арифметических вычислениях или в теории чисел.

Базисными функциями обработки символьных выражений (атомов и списков) являются:

CAR, CDR, CONS, ATOM и EQ

Функции по принципу их использования можно разделить на функции разбора, создания и проверки:

ИСПОЛЬЗОВАНИЕ	ВЫЗОВ	РЕЗУЛЬТАТ
Разбор:	(CAR список)	s-выражение
	(CDR список)	список
Создание:	(CONS s-выражение список)	список
Проверка:	(ATOM s-выражение)	1 или NIL
	(EQ символ символ)	T или NIL

У функций CONS и EQ имеются два аргумента, у остальных примитивов - по одному. В качестве имен аргументов и результатов функций использовали названия типов, описывающих аргументы, на которых определена (т. е. имеет смысл) функция и вид возвращаемого функциями результата. S-выражение обозначает атом или список.

Функции ATOM и EQ являются базовыми предикатами. **Предикаты** (predicate) - это функции, которые проверяют выполнение некоторого условия и возвращают в качестве результата логическое значение T (в более общем виде, произвольное выражение, отличное от NIL) или NIL

Функция CAR возвращает в качестве значения головную часть списка

Первый элемент списка называется головой (head), а остаток списка, т.е. список без первого его элемента, называется хвостом списка (tail). С помощью селекторов CAR и CDR можно выделить из списка его голову и хвост.

Функция CAR (произносится "кар") возвращает в качестве значения первый элемент списка, т.е. головной элемент списка:

```
_(car '(первый второй третий) )      ;
   ПЕРВЫЙ
```

Функция CAR имеет смысл только для аргументов, являющихся списками, а следовательно, имеющих голову:

car. список > s-выражение

Для аргумента атома результат функции CAR неопределен, и вследствие этого появляется следующее сообщение об ошибке:

```
_(car 'кот)
Error: КОТ is not a list
```

(т.е. КОТ не является списком.).

Головной частью пустого списка считают для удобства NIL:

```
_(car nil)        ;  голова пустого списка -
NIL               ;  пустой список
_(car 'nil)       ;  знак ' можно опускать
NIL
```

_(car '(nil a)) ; голова списка NIL
NIL

Функция CDR возвращает в качестве значения хвостовую часть списка

Функция CDR (произносится "кудр") применима к спискам. Значением ее будет хвостовая часть списка, т.е. список, получаемый из исходного списка после удаления из него головного элемента:

cdr : список > список
_(cdr '(a b c))
(B C)
_(cdr '(a (b c)))
((B C))

Функция CDR не выделяет второй элемент списка, а берет весь остаток списка, т.е. хвост. Заметим, что хвост списка - тоже список, если только список не состоял из одного элемента. В последнем случае хвостом будет пустой список (), т.е. NIL:

_(cdr '(a))
NIL

Из соображений удобства значением функции CDR от пустого списка считается NIL:

_(cdr nil)
NIL

Так же как и CAR, функция CDR определена только для списков. Значение для атомов не определено, что может приводить к сообщению об ошибке

_(cdr 'кот)
Error: КОТ is not a list

Необычные имена функций CAR и CDR возникли по историческим причинам. Эти имена являются сокращениями от наименований регистров Contents of Address Register (CAR) и Contents of Decrement Register (CDR) вычислительной машины IBM 605. Автор Лиспа Джон Маккарти (США) реализовал первую Лисп-систему именно на этой машине и использовал регистры CAR и CDR для хранения головы и хвоста списка.

В Коммон Лиспе и на некоторых других диалектах Лиспа наряду с именами CAR и CDR используют и более наглядные имена FIRST и REST'.

_(first '(1 2 3 4))
1
_(first (rest '(1 2 3 4)))
  2
_first '(rest '(1 2 3 4)))
REST

(В некоторых системах используются имена HEAD и TAIL.)

Функция CONS включает новый элемент в начало списка

Функция CONS (construct) строит новый список из переданных ей в качестве аргументов головы и хвоста:

(CONS голова хвост)

Функция добавляет новое выражение в список в качестве первого элемента:

_(cons 'a '(b c))
(A B C)

```
_(cons '(a b) '(c d))
((A B) C D)
_(cons (+ 1 2)  '(+ 4))        ;   первый аргумент
(3+4)                          ;   без апострофа, поэтому
                               ;   берется его значение
_(cons '(+1 2)  '(+4))         ;   первый аргумент
((+ 1 2) + 4)                  ;   не вычисляется
```

У начинающих при манипуляциях с пустым списком очень легко возникают смысловые ошибки. Проследите за тем, что происходит:

```
_(cons '(a b c) nil)
((ABC))                        ;  одноэлементный список
_(cons nil '(a b c))
(NIL A B C)                    ;  NIL - элемент списка
_(cons nil nil)
(NIL)                          ;  не то же самое,  что NIL
```

Для того чтобы можно было включить первый аргумент функции CONS в качестве первого элемента значения второго аргумента этой функции, второй аргумент должен быть списком. Значением функции CONS всегда будет список:

cons: s-выражение * список > список

(Позже мы увидим, что вторым аргументом может быть и атом, но в таком случае результатом будет более общее символьное выражение, так называемая точечная пара (dotted pair).)

Связь между функциями CAR, CDR и CONS

Селекторы CAR и CDR являются обратными для конструктора CONS. Список, разбитый с помощью функций CAR и CDR на голову и хвост, можно восстановить с помощью функции CONS, эту связь между функциями CAR, CDR и CONS можно выразить следующим образом:

```
_(cons (car '(это просто список)
        (cdr '(это просто список) ) )
(ЭТО ПРОСТО СПИСОК)
```

Предикат проверяет наличие некоторого свойства

Чтобы осуществлять допустимые действия со списками и избежать ошибочных ситуаций, необходимы, кроме селектирующих и конструирующих функций, средства опознавания выражений. Функции, решающие эту задачу, в Лиспе называются предикатами (predicate).

ATOM и EQ являются базовыми предикатами Лиспа. С их помощью и используя другие базовые функции, можно задать более сложные предикаты, которые будут проверять наличие более сложных свойств.

Предикат ATOM проверяет, является ли аргумент атомом

При работе с выражениями необходимо иметь возможность проверить, является ли выражение атомом или списком. Это может потребоваться, например, перед применением функций CAR и CDR, так как эти функции определены лишь для аргументов, являющихся списками. Базовый предикат ATOM используется для идентифицирования лисповских объектов, являющихся атомами:

(ATOM s-выражение)

Значением вызова ATOM будет T, если аргументом является атом, и NIL - в противном случае.

```
_(atom  'x)
 T                              ; x - это атом
_(atom '(Я программирую - следовательно существую))
NIL
_(atom (cdr '(a b c)))
NIL                     ; ATOM от списка (B C) - это NIL
```

С помощью предиката ATOM можно убедиться, что пустой список NIL, или (), является атомом:

```
_(atom nil)
T
_(atom ())              ;  () - то же самое,  что и NIL
T
_(atom '())             ;  Пустой список с апострофом
T                       ;  все равно есть NIL
_(atom '(nil))          ; Аргумент - одноэлементный
NIL                     ; список
_(atom (atom (+ 2 3)))    ;  Логическое
T                               ;  значение T является атомом
```

В последнем примере результатом сложения является число, а результатом внутреннего вызова предиката ATOM - значение T, которое, в свою очередь, тоже является атомом.

EQ проверяет тождественность двух символов

Предикат EQ сравнивает два символа и возвращает значение T, если они идентичны, в противном случае – NIL

```
_(eq 'x 'кот)
NIL
_(eq 'кот (car '(кот пес)))
T
_(eq O nil)
T
_(eq t 't)
T
_(eq t (atom 'мышь))
T
```

Предикат EQ накладывает на аргументы определенные требования. С его помощью можно сравнивать только символы или константы T и NIL, и результатом будет значение T лишь в том случае, когда аргументы совпадают. Для проверки чисел в Лиспе EQ не используется.

Так как EQ определен лишь для символов, то, сравнивая два выражения, прежде всего надо определить, являются ли они атомами (ATOM).

```
_(eq '(a b c)  '(a b c))
NIL
```

_(eq 3.14 3.14)
NIL

Если хотя бы один из аргументов является списком, то предикат EQ нельзя использовать для логического сравнения. При сравнении чисел проблемы возникают с числами различных типов. Например, числа 3.000000, 3 и О.ЗЕ! логически представляют одно и то же число, но записываются внешне неодинаково. Для различных видов и степеней равенства в Лиспе наряду с EQ используются и другие предикаты, которые мы сейчас рассмотрим.

EQL сравнивает числа одинаковых типов

Предикат EQL работает так же, как EQ, но дополнительно позволяет сравнивать однотипные числа (и элементы строк).

```
_(eql 3 3)    ;     сравниваются два целых
T             ;     числа
_(eql 3.0 3.0)  ;   сравниваются два
T             ;     вещественных числа
_(eql 3 3.0)  ;     не годится для разных
NIL           ;     типов
```

Предикат EQL, как правило, используется во многих встроенных функциях, осуществляющих более сложные операции сравнения. Его использование для сравнения списков - это часто встречающаяся ошибка.

Предикат = сравнивает числа различных типов Сложности, возникающие при сравнении чисел, легко преодолимы с помощью предиката =, значением которого является Т в случае равенства чисел независимо от их типов и внешнего вида записи:

_(= 3 3.0)
T
_(= 3.00 0.3e1)
T

Обеспечить применение предиката к числовым аргументам может предикат NUMBERP, который истинен для чисел:

_(numberp 3e-34)
T
_(numberp t)
NIL

EQUAL проверяет идентичность записей

Обобщением EQL является предикат EQUAL Он работает как EQL, но, кроме того, проверяет одинаковость двух списков:

_(equal 'x 'x)
T
_(equal '(x y z) '(x y 2»
T
_(equal '(a b c) (cons 'a '(b c)))
T
_(equal '(nil) '((nil)))
NIL

Принцип работы предиката EQUAL состоит в следующем: если внешняя структура двух лисповских объектов одинакова, то эти объекты между собой равны в смысле EQUAL Предикат EQUAL также применим к числам и к другим типам данных (например, к строкам), но он не подходит для сравнения разнотипных чисел, так как их внешние представления различаются:

```
_(equal 3.00  3)      ;  различное внешнее
NIL                   ;  представление
_(= 3.00  3)
T
```

EQUALP проверяет наиболее общее логическое равенство

Наиболее общим предикатом, проверяющим в Лиспе наличие логического равенства, является EQUALP, с помощью которого можно сравнивать произвольные объекты, будь то числа различных типов, выражения или другие объекты. Этот предикат может потребоваться, когда нет уверенности в типе сравниваемых объектов или в корректности использования других предикатов сравнения.

Недостатком универсальных предикатов и функций типа EQUALP является то, что их применение требует от системы несколько большего объема вычислений, чем использование специализированных предикатов и функций.

Другие примитивы

NULL проверяет на пустой список

Встроенная функция NULL проверяет, является ли аргумент пустым списком:

```
_(null  '())
T
_(null (cddr '(a b c)))
NIL
_(null NIL)
T
_(null T)
NIL
```

Из последних двух примеров видно, что NULL работает как логическое отрицание, у которого в Лиспе есть и свой, принадлежащий логическим функциям, предикат (NOT x):

```
_(not (null nil))
NIL
```

Функции NULL и NOT можно выразить через EQ:

(NULL x) - (EQ NIL x)

Вложенные вызовы CAR и CDR можно записывать в сокращенном виде. Комбинируя селекторы CAR и CDR, можно выделить произвольный элемент списка. Например:

```
_(cdr ( cdr ( car '((a b c) (d e) (f) ) ) ) )
(C)
```

Комбинации вызовов CAR и CDR образуют уходящие в глубину списка обращения, и в Лиспе используется для этого более короткая запись: желаемую комбинацию вызовов CAR и CDR можно записать в виде одного вызова функции:

(C...R список)

Вместо многоточия записывается нужная комбинация из букв A (для функции CAR) и D (для функции CDR):

(cadr x) > (car (cdr x))

(cddar x) > (cdr (cdr (car x)))

Например:

_(cadr '(программировать на Лиспе просто?))

НА

_(caddr '((a b c) (d e) (f))

(F)

_(cadar (cons '(a b) nil))

B

Для функций CAR, CADR, CADDR, CADDDR и т.д. в Лиспе используются и более наглядные имена FIRST, SECOND, THIRD, FOURTH и т.д. Можно воспользоваться и наиболее общей функцией NTH, выделяющей n-й элемент списка:

(NTH n список)

_(nth 2 '(1 2 3)) ; индексы начинаются с

3 ; нуля

_(third (cons 1 (cons 2 (cons 3 nil))))

3

_(fourth '(1 2 3))

NIL

Последний элемент списка можно выделить с помощью функции LAST:

(LAST x)

LIST создает список из элементов

Другой часто используемой встроенной функцией является:

(LIST x1 x2 x3 …),

которая возвращает в качестве своего значения список из значений аргументов. Количество аргументов функции LIST произвольно:

_(list 1 2)

(1 2)

_(list 'a 'b (+ 1 2))

(A B 3)

_(list 'a '(b c) 'd)

(A (B C) D)

_(list (list 'a) 'b NIL)

((A) B NIL)

_(list NIL)

(NIL)

Построение списков нетрудно свести к вложенным вызовам функции CONS, причем вторым аргументом последнего вызова является NIL, служащий основой для наращивания списка:

_(cons 'c NIL) ; <=> (list 'c)

(c)

_(cons 'b ; <=> (list 'b 'c)

```
            (cons 'c NIL))
(B C)
_(cons 'a                ;  <=> (list 'a 'b 'c)
            (cons 'b (cons 'c NIL)))
(A B C)
```

В дальнейшем станет ясно, как, в частности, функцию LIST описать через комбинацию базовых функций.

1.4 ИМЯ И ЗНАЧЕНИЕ СИМВОЛА

Значением константы является сама константа

Атомы могут использоваться для обозначения каких-нибудь значений. Как константы они обозначают самих себя. Если мы введем константу, то интерпретатор в качестве результата выдаст саму эту константу:

```
_t        ;  значением T является его имя
T
_'t            ;  апостроф излишен
T
_nil
NIL
_3.14
3.14
```

Символ может обозначать произвольное выражение

Символы можно использовать как переменные. В этом случае они могут обозначать некоторые выражения. У символов изначально нет какого-нибудь значения, как у констант. Если, например, введем символ ФУНКЦИИ, то мы получим сообщение об ошибке:

```
_функции       ;  у символа нет значения
Error:  Unbound atom ФУНКЦИИ
```

интерпретатор здесь не может вычислить у значение символа, поскольку его у него нет.

SET вычисляет имя и связывает его

При помощи функции SET символу можно присвоить (set) или связать (bind) с ним некоторое значение. Если, например, мы хотим, чтобы символ ФУНКЦИИ обозначал базовые функции Лиспа, то введем:

```
(set 'функции '(car cdr cons atom eq))
(CAR CDR CONS ATOM EQ)
```

Теперь между символом ФУНКЦИИ и значением (CAR CDR CONS ATOM EQ) образована связь (binding), которая действительна до окончания работы, если, конечно, этому имени функцией SET не будет присвоено новое значение. После присваивания интерпретатор уже может вычислить значение символа ФУНКЦИИ:

```
_функции
(CAR CDR CONS ATOM EQ)
```

SET вычисляет оба аргумента. Если перед первым аргументом нет апострофа, то с помощью функции SET можно присвоить значение имени, которое получается путем вычисления. Например, вызов

```
_(set (car Функции) '(взбрести в голову))
```

(ВЗБРЕСТИ В ГОЛОВУ)

присваивает переменной CAR выражение (ВЗБРЕСТИ В ГОЛОВУ), так как вызов (CAR ФУНКЦИИ) возвращает в качестве значения символ CAR, который и используется как фактический аргумент вызова функции SET:

```
_(car функции)                ;  первый аргумент
 CAR                          ;  предыдущего вызова
_CAR                          ;  присвоенное
(ВЗБРЕСТИ В ГОЛОВУ)           ;  функцией SET
_функции                      ;  значение
(CAR CDR CONS ATOM EQ)
```

На значение символа можно сослаться, записав его без апострофа. Значение имени никак не проявится до тех пор, пока оно не примет участия в вычислениях. Значения символов определяются с помощью специальной функции SYMBOL-VALUE, которая возвращает в качестве своего значения значение символа, являющегося ее аргументом.

```
_(symbol-value (car функции))
(ВЗБРЕСТИ В ГОЛОВУ)
```

SETQ связывает имя, не вычисляя его

Связать символ с его значением можно с помощью функции SETQ. Эта функция отличается от SET тем, что она вычисляет только свой второй аргумент. Об автоматическом блокировании вычисления первого аргумента напоминает буква Q (quote) в имени функции. Например:

```
(setq функции '(car cdr cons atom eq))
(CAR CDR CONS ATOM EQ)
```

При использовании функции SETQ отпадает надобность в знаке апострофа перед первым аргументом. (В Интерлиспе есть функция SETQQ, которая блокирует вычисление обоих аргументов.)

Проверить, связан ли атом, можно с помощью предиката BOUNDP, который истинен, когда атом имеет какое-нибудь значение:

```
_(boundp 'беззначения)
NIL
_(boundp 'функции)
T
_(boundp 't)          ;  константа всегда связана
T
```

SETF - обобщенная функция присваивания

В Коммон Лиспе значение символа сохраняется в ячейке памяти (storage location), связанной с самим символом. Под ячейками памяти при этом понимаются поля списочной ячейки, которую мы рассмотрим ниже, элементы массива и другие структуры, содержащие данные. Так же как на значения символов можно сослаться через их имена, так и на ячейки памяти можно ссылаться через вызов функции SYMBOL-VALUE и в общем случае другими способами, зависящими от типа данных.

Для присваивания, т.е. занесения значения в ячейку памяти, существует обобщенная функция обновления данных SETF, которая записывает в ячейку памяти новое значение:

```
(SETF ячейка-памяти значение)
```

Через функцию SETF можно представить описанные нами ранее функции SET и SETQ:

(setq x y) - (setf x y)
(set x y) - (setf (symbol-value x) y)
_(setf список '(a b c))
(A B C)
_список
(A B C)

Переменная СПИСОК без апострофа указывает на ячейку памяти, куда помещается в качестве значения список (A B C).

Побочный эффект псевдофункции

Функции SET, SETQ и SETF отличаются от других рассмотренных функций тем, что помимо того, что они имеют значение, они обладают и побочным эффектом. Эффект функции состоит в образовании связи между символом и его значением, а значением функции является связываемое значение. Символ остается связанным с определенным значением до тех пор, пока это значение не изменят.

В Лиспе все действия возвращают некоторое значение. Значение имеется даже у таких действий, основное предназначение которых заключается в осуществлении побочного эффекта, таких, например, как связывание символа или вывод результатов на печать. Функции, обладающие побочным эффектом, в Лиспе называют псевдофункциями. Мы будем все же как для функций, так и для псевдофункций использовать понятие функции, если только нет особой надобности подчеркнуть наличие побочного эффекта.

Вызов псевдофункции, например оператор передачи управления (а это тоже вызов), с точки зрения использования его значения может стоять на месте аргумента другой функции. (В языках программирования, основанных на операторном подходе, это обычно невозможно.)

```
_(list (+ (setq a 3) 4) a)
(7 3)
_a
3                                   ;     аргументы
_(list b (setq b 3))                ;     вычисляются
Error:  Unbound atom B              ;     слева направо
```

Вызов интерпретатора EVAL вычисляет значение значения

Интерпретатор Лиспа называется EVAL, и его можно из программы. При обычном программировании вызывать интерпретатор не надо, так как этот вызов неявно присутствует в диалоге программиста с Лисп-системой. Лишний вызов интерпретатора может, например, снять эффект блокировки вычисления или позволяет найти значение значения выражения, т.е. осуществить двойное вычисление:

```
_(quote (+ 2 3))
(+ 2 3)
_(eval (quote (+ 2 3)))
5
```

Здесь интерпретатор вычисляет (evaluate) значение первого выражения (QUOTE (+ 2 3)), получая в результате (+ 2 3). Далее вызов EVAL позволяет вновь запустить интерпретатор, и результатом будетзначенне выражения (+ 2 3), т.е. 5. Исходное выражение обрабатывается в два этапа. Приведем еще несколько примеров:

```
_(setq x '(a b c))
(A B C)
_' x
X
_(eval 'x)
(A B C)
_(eval x)                          ;  значением X является
Error: Undefined function A    ;  (ABC)
_(eval (quote (quote (a b c))))
(A B C)
_(quote (eval x))
(EVAL x)
```

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

EVAL- это универсальная функция Лиспа, которая может вычислить любое правильно составленное лисповское выражение. EVAL определяет семантику (semantics) лисповских форм, т.е. определяет, какие символы и формы совместно с чем и что означают и какие выражения имеют значение.

Основной цикл: READ-EVAL-PRINT

Диалог с интерпретатором Лиспа на самом верхнем, командном уровне (top level), можно описать простым циклом:

```
(print '_)                ;   вывод приглашения
(setq e (read))           ;   ввод выражения(setq v (eval e))          ;   вычисление его значения
(print v)                 ;   вывод результата
(print '_)                ;   повторение цикла
```

Интерпретатор Лиспа отвечает на заданный ему вопрос и ждет новое задание.

Некоторые системы на командном уровне работают в режиме, который называют EVALQUOTE. В этом режиме имя функции можно выносить за открывающую скобку, как это обычно принято при записи математических функций. Кроме того, как это видно и из названия, автоматически блокируется вычисление аргументов. Например:

```
+(2 3) > 5
cons(a (b c)) > (a b c)
```

Хотя при этом можно избежать использования знака апострофа, но в то же время на командном уровне пропадает простая возможность задать вычисляемые аргументы функции. Для аргументов, требующих вычисления, можно либо использовать способ отмены блокировки вычислений, либо в самой функции специально использовать вызов интерпретатора (EVAL).

1.5 ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

Лямбда-выражение изображает параметризованные вычисления

Определение функций и их вычисление в Лиспе основано на лямбда-исчислении (lambda calculus) Черча, предлагающем для этого точный и простой формализм. Лямбда-выражение, позаимствованное Лиспом из лямбда-исчисления является важным механизмом в практическом программировании. Подробнее мы вернемся к этому позже при рассмотрении функционалов.

В лямбда-исчислении Черча функция записывается в следующем виде:

lambda(xl,x2,.. .,xn).fn

В Лиспе лямбда-выражение имеет вид:

(LAMBDA (xl x2 ... xп) fn)

Символ LAMBDA означает, что мы имеем дело с определением функции. Символы x_i являются формальными параметрами (formal parameter) определения, которые именуют аргументы в описывающем вычисления теле (body) функции fn. Входящий в X состав формы список, образованный из параметров, называют лямбда-списком (lambda list).

Телом функции является произвольная форма, значение которой может вычислить интерпретйтор Лиспа, например: константа связанный со значением символ или КОМПОЗИЦИЯ из вызовов функций. Функцию, вычисляющую сумму квадратов двух чисел можно, например, определить следующим лямбда-выражением:

(lambda (x y) (+ (* x x) (* y y)))

Формальность параметров означает, что их можно заменить на любые другие символы, и это не отразится на вычислениях, определяемых функцией. Именно это и скрывается за лямбда-нотацией. С ее помощью возможно различать понятия определения и вызова функции. Например, функцию LIST для двух аргументов можно определить любым из двух последующих лямбда-выражений:

(lambda (x y) (cons x (cons y nil)))

(lambda (кот пес) (cons кот (cons пес nil)))

Здесь значение вызова функции CONS, являющегося телом лямбда-выражения, зависит от значений связей (другими словами, от значений переменных).

Лямбда-вызов соответствует вызову функции

Лямбда-выражение - это определение вычислений и параметров функции в чистом виде без фактических параметров, или аргументов (actual parameter). Для того, чтобы применить такую функцию к неко-торым аргументам, нужно в вызове функции поставить лямбда-определение на место имени функции:

(лямбда-выражение al a2 ... an)

Здесь a_i - формы, задающие фактические параметры, которые вычисляются как обычно. Например, действие сложения для чисел 2 и 3

_(+ 2 3) 5

можно записать с использованием вызова лямбда-выражения:

```
_((lambda (x y)
   (+ x y))          ;  лямбда-определение
        23)            ;  аргументы
5                      ;  результат
```

Следующий вызов строит список из аргументов A и B:

```
_((lambda (x y)  (cons x (cons y nil))) 'a 'b)
(A B)
```

Такую форму вызова называют лямбда-вызовом.

Вычисление лямбда-вызова, или лямбда-преобразование

Вычисление лямбда-вызова, или применение лямбда-выражения к фактическим параметрам, производится в два этапа. Сначала вычисляются значения фактических параметров и соответствующие формальные параметры связываются с полученными значениями. Этот этап называется связыванием параметров (spreading). На следующем этапе с учетом новых связей вычисляется форма, являющаяся телом лямбда-выражения, и полученное значение возвращается в качестве значения лямбда-вызова. Формальным параметрам после окончания вычисления возвращаются те связи, которые у них, возможно, были перед вычислением лямбда-вызова. Весь этот процесс называют лямбда-преобразованием (lambda conversion).

Объединение лямбда-вызовов

Лямбда-вызовы можно свободно объединять между собой и другими формами. Вложенные лямбда-вызовы можно ставить как на место тела лямбда-выражения, так и на место фактических параметров. Например, в следующем вызове тело лямбда-выражения содержит вложенный лямбда-вызов:

```
_((lambda (y)                    ;   у лямда-вызова
   ((lambda(x)                   ;   тело вновь
     (list y x))                 ;    лямда-вызов
      'ВНУТРЕННИЙ))
           'ВНЕШНИЙ)
(ВНЕШНИЙ  ВНУТРЕННИЙ)
```

В приведенном ниже примере лямбда-вызов является аргументом другого вызова:

```
_((lambda (x)                ;   лямбда-вызов
  (list 'ВТОРОЙ x))          ;   у которого
 ((lambda (y)                ;   аргументом является
  (list y))                  ;   новый лямбда-вызов
      'ПЕРВЫЙ))
    (ВТОРОЙ (ПЕРВЫЙ))
```

Обратите внимание, что лямбда-выражение без аргументов (фактических параметров) представляет собой лишь определение, но не форму, которую можно вычислить. Само по себе оно интерпретатором не воспринимается:

```
_(lambda (x y)  (cons x (cons y nil)))
Error:  Undefined function LAMBDA
```

(Функция LAMBDA не определена.- Ред.)

Лямбда-выражение - функция без имени

Лямбда-выражение является как чисто абстрактным механизмом для определения и описания вычислений, дающим точный формализм для параметризации вычислений при помощи переменных и изображения вычислений, так и механизмом для связывания формальных и фактических параметров на время выполнения вычислений. Лямбда-

выражение - это безымянная функция, которая пропадает тотчас после вычисления значения формы. Ее трудно использовать снова, так как нельзя вызвать по имени, хотя ранее выражение было доступно как списочный объект. Однако используются и безымянные функции, например при передаче функции в качестве аргумента другой функции или при формировании функции в результате вычислений, другими словами, при синтезе программ.

DEFUN дает имя описанию функции

Лямбда-выражение соответствует используемому в других языках определению процедуры или функции, а лямбда-вызов - вызову процедуры или функции. Различие состоит в том, что для программиста лямбда-выражение - это лишь механизм, и оно не содержит имени или чего-либо подобного, позволяющего сослаться на это выражение из других вызовов. Записывать вызовы функций полностью с помощью лямбда-вызовов не разумно, поскольку очень скоро выражения в вызове пришлось бы повторять, хотя разные вызовы одной функции отличаются лишь в части фактических параметров. Проблема разрешимапутем именования лямбда-выражений и использования в вызове лишь имени.

Дать имя и определить новую функцию можно с помощью функции DEFUN (define function). Ее действие с абстрактной точки зрения аналогично именованию данных (SET и другие). DEFUN вызывается так:

(DEFUN имя лямбда-список тело)

Что можно представить себе как сокращение записи (DEFUN имя лямбда-выражение) из которой для удобства исключены внешние скобки лямбда-выражения и сам атом LAMBDA. Например:

```
(defun listl (lambda (x y) (cons x (cons y nil))))
>
(defun listl   (x y) (cons x (cons y nil)))
```

DEFUN соединяет символ с лямбда-выражением, и символ начинает представлять (именовать) определенные этим лямбда-выражением вычисления. Значением этой формы является имя новой функции:

```
_(defun listl (x y)          ;  определение
   (cons x (cons y nil)))
LIST
_(listl  'a 'b)              ;  вызов
(A B)
```

Приведем еще несколько примеров:

```
_(defun lambdap (выражение)      ;  проверяет
   (eq (car выражение)   'lambda))
     LAMBDAP                          ;  на лямбда-выражение
_(lambdap '(list1 'a 'b))
NIL
_(defun проценты (часть чего)     ; вычисляет
   (* (/ часть чего) 100))        ; % части
ПРОЦЕНТЫ
_(проценты 4 20)
20
```

SYMBOL-FUNCTION выдает определение функции

Ранее был рассмотрен предикат BOLJNDP, проверяющий наличие у символа значения. Соответственно предикат FBOUNDP проверяет, связано ли с символом определение функции:

```
_(fboundp 'listl)
T
```

Значение символа можно было получить при помощи функции SYMBOL-VALUE. Аналогично функция SYMBOL-FUNCTION дает определение функции, связанное с символом:

```
_(symbol-function 'listl)
(LAMBDA (X Y)   (CONS X (CONS Y NIL)))
```

Поскольку определение функции задается списком, а он всегда доступен программе, то можно исследовать работу функций и даже время от времени модифицировать ее, изменяя определения (например, в задачах обучения). В традиционных языках программирования, предполагающих трансляцию, это было бы невозможно. Символ может одновременно именовать некоторое значение и функцию, и эти возможности не мешают друг другу. Позиция символа в выражении определяет его интерпретацию:

```
_(setq listl 'a)
A
_(listl listl 'b)
(A B)
```

В некоторых системах (например, Franz Lisp) определение функции может храниться как значение символа, а не распологаться в специально отведенном месте в памяти. В Ком мои Лиспе последняя возможность отсутствует:

```
_(setq 'список
   (lambda (x y)  (cons x (cons y nil))))
   (LAMBDA (X У)   (CONS X (CONS Y NIL))))
_(список 'a 'b)      ;  не работает в
                     ;  Коммон Лиспе
Error:  Undefined function СПИСОК
```

Значение символа в таких системах интерпретируется как определение функции лишь тогда, когда с символом не связано определение функции.

В некоторых системах функцию можно задавать произвольным вычислимым выражением. В таких случаях говорят, что функции - это "полноправные граждане" (first class citizen).

Задание параметров в лямбда-списке

Рассмотренной ранее DEFUN-формы вполне достаточно для изучения Лиспа. Однако DEFUN-форма содержит, кроме этого, очень богатый механизм ключевых слов (lambda-list keyword), с помощью которых аргументы вызова функции можно при желании трактовать по-разному".

С помощью ключевых слов в лямбда-списке можно выделить:

- необязательные (optional) аргументы,
- параметр, связываемый с хвостом списка аргументов изменяющейся длины,

- ключевые (key) параметры,
- вспомогательные (auxiliary) переменные функции.

Аргументу и вспомогательной переменной можно присвоить значение по умолчанию (default). Ключевые слова начинаются со знака &, и их записывают перед соответствующими параметрами в лямбда-списке. Действие ключевого слова распространяется до следующего ключевого слова. Приведем список наиболее важных ключевых слов и типов параметров или вспомогательных переменных, обозначаемых ими:

Ключевое слово	**Значение**
&OPTIONAL	Необязательные параметры
&REST	Переменное количество параметров
&KEY	Ключевые параметры
&ALJX	Вспомогательные переменные

Параметры, перечисленные в лямбда-списке до первого ключевого слова, являются обязательными. До настоящего момента мы определяли функции без использования ключевых слов, т.е. фактически использовали лишь обязательные параметры.

Значения объявленных при помощи &OPTIONAL необязательных параметров можно в вызове не указывать. В этом случае они связываются со значением NIL или со значением выражения по умолчанию (init form), если таковое имеется. Например, у следующей функции есть обязательный параметр X и необязательный Y со значением по умолчанию (+ X 2):

```
_(defun fn (x &optional (y (+ x 2)))
   (list ж y))
FN
_(fn 2)                                  ; вычисляется значение по
(2 4)                                ; умолчании Y=X+2
_(fn 2 5)                                ; умолчание не используется
(2 5)
```

Естественно, ключевые слова можно использовать и в лямбда-выражениях:

```
_((lambda (x (&optional (y (+ x 2)))
   (list x y)) 2 5)
(2 5)
```

Параметр, указанный после ключевого слова &REST, связывается со списком несвязанных параметров, указанных в вызове. Таким функциям можно передавать переменное количество параметров. Например:

```
_(defun fn (x &optional y &rest x)
(list x y z))
FN
_(fn 'a)
(A NIL NIL)
_(fn 'a 'b 'c 'd)
(A B (C D))
```

Обратите внимание, что &REST-параметр связывается со значениями последних аргументов, поскольку отсутствует QUOTE.

Фактические параметры, соответствующие формальным параметрам, обозначенным ключевым словом &KEY, можно задавать в вызове при помощи символьных ключей. Ключом является имя формального параметра, перед которым поставлено двоеточие, например ":X". Соответствующий фактический параметр будет следовать в вызове функции за ключом и отделятся от него пробелом. Достоинством ключевых параметров является то, что их можно перечислять в вызове, не зная их порядок в определении функций или лямбда-выражении. Например, у следующей функции параметры X, Y и Z являются ключевыми:

```
_(defun fn (&key z y (z 3))
  (list x y z))
FN
_(fn :y 2 :x 1)
(1 2 3)
```

Параметр Z можно было не задавать, так как ключевые параметры являются необязательными.

При помощи ключевых параметров можно определять функции, которые в зависимости от используемой при вызове комбинации параметров запускаются с различными их значениями.

Функции вычисляют все аргументы

В Коммон Лиспе нет механизма, с помощью которого можно было бы обозначать параметры, не требующие вычисления. Блокирующие вычисление аргумента функции и формы, такие как QUOTE, SETQ и DEFUN, определяются через механизм макросов или макрооператоров, который мы рассмотрим позже.

Многозначные функции

В Коммон Лиспе можно определить и многозначные функции (multiple valued functions), которые возвращают множество значений. Этот механизм более удобен, чем возврат значений через глобальную переменную или через построение списка результатов. Для выдачи и принятия многокомпонентных значений используются специальные формы. Ограничимся в этом случае ссылкой на оригинальное руководство по языку.

При вычислении NLAMBDA аргументы не вычисляются

Функции, которые трактуют свои аргументы так же, как QUOTE и DEFUN, часто называют функциями типа FEXPR. Если вычислять значения аргументов не надо, то такую функцию нужно определять с помощью NLAMBDA-выражения (NO-spread lambda), имеющего ту же структуру, что лямбда-выражение:

(NLAMBDA параметры тело-функции)

Определяемые лямбда-выражением функции типа EXPR(*) и определяемые NLAMBDA-выражением функции типа FEXPR(*) могут получать фиксированное или неопределенное число параметров в зависимости от того, определяются ли параметры через список атомов или одним атомом. Как ранее было сказано, в Коммон Лиспе нельзя определить функцию, не вычисляющую значения аргументов (однако можно использовать так называемые макросы).

1.6 ПЕРЕДАЧА ПАРАМЕТРОВ И ОБЛАСТЬ ИХ ДЕЙСТВИЯ

В языках программирования в основном используются два способа передачи параметров - это передача параметров по значению (call by value) и по ссылке (call by reference). При передаче параметров по значению формальный параметр связывается с тем же значением, что и значение фактического параметра. Изменения значения формального параметра во время вычисления функции никак не отражаются назначении фактического параметра. С помощью параметров, передаваемых по значению, информацию можно передавать только внутрь процедур, но не обратно из них. При передаче параметров по ссылке изменения значений формальных параметров видны извне и можно возвращать данные из процедуры с помощью присваивания значений формальным параметрам.

В Лиспе используется передача параметров по значению

Передача параметров в Лиспе осуществляется в основном по значению. Параметры в Лиспе используются преимущественно лишь для передачи данных в функцию, а результаты возвращаются как значение функции, а не как значения параметров, передаваемых по ссылке. (В Лиспе все-таки можно с помощью псевдофункций, меняющих структуры, использовать формальные параметры таким же образом, как это происходит при передаче параметров по ссылке. Побочные эффекты их использования отражаются на значениях всех переменных, ссылающихся на некоторый элемент данных. Мы вернемся к этим псевдофункциям чуть позже. Кроме того, механизм возврата значений из многозначных функций также напоминает передачу параметров по ссылке.)

Статические переменные локальны

Формальные параметры функции в Коммон Лиспе называют лексическими или статическими переменны-ми (lexical/static variable). Связи статической переменной действительны только в пределах той формы, в которой они определены. Они перестают действовать в функциях, вызываемых во время вычисления, но текстуально описанных вне данной формы. Изменение значений переменных не влияет на одноименные переменные вызовов более внешнего уровня. Статические переменные представляют собой лишь формальные имена других лисповских объектов. После вычисления функции, созданные на это время связи формальных параметров ликвидируются и происходит возврат к тому состоянию, которое было до вызова функции. Например:

```
_(defun не-меняет (x)    ;  X статическая
   (setq x 'новое))
НЕ-МЕНЯЕТ
_(setq x 'старое)
СТАРОЕ
_(не-меняет 'новое)          ;  статическое
НОВОЕ                        ;  значение X изменяется
_x                           ;  первоначальная связь
СТАРОЕ                  ;  не меняется
```

Свободные переменные меняют свое значение

Возникшие в результате побочного эффекта изменения значений свободных переменных (tree variable), т.е. используемых в функции, но не входящих в число ее формальных параметров, остаются ъ силе после окончания выполнения функции.

Определим далее функцию ИЗМЕНИТЬ, в которой переменная X свободна. Ее значение будет меняться:

```
_(defun изменить ()
    (setq x 'новое))        ;  X свободная
ИЗМЕНИТЬ
_(изменить)
НОВОЕ
_x                          ;  значение свободной
НОВОЕ                       ;  переменной изменилось
```

Динамическая и статическая область действия

Под вычислительным окружением или контекстом (evaluation environment) будем понимать совокупность действующих связей переменных с их значениями. Связи формальных параметров вызова со значениями аргументов действительны (по умолчанию) только в пределах текста определения функции. Будем говорить, что область действия (scope) или видимость переменных статическая.

В Коммон Лиспе существует однако возможность использования динамических, или специальных (dynamic/special variable), переменных. Это обычно достигается при помощи директивы

```
(DEFVAR переменная
 &OPTIONAL начальное значение)
```

Значение переменной, объявленной специальной, определяется динамически во время вычисления, а не в зависимости от контекста места ее определения, как для статических переменных. Будем говорить, что временем действия (extent) связи динамического формального параметра является все время вычисления вызова, в котором возникла эта связь.

Если переменная свободна и является формальным параметром какого-нибудь охватывающего вызова, то значения статической и динамической переменных вычисляются по-разному. Если переменная является статической (как, например, по умолчанию в Коммон Лиспе), то ее использование в качестве формального параметра в более внешней, предшествующей, форме не влияет на значение свободной переменной. У переменной либо не будет значения, что приведет к ошибке, либо ее значение определяется в соответствии с ее глобальным значением, присвоенным на самом внешнем уровне функцией SETQ.

Если переменная при помощи DEFVAR объявлена динамической, то связь, установленная в более внешнем вызове, остается в силе для всех вложенных контекстов, возникающих во время вычисления (при условии, что эта переменная снова не связывается). Например:

```
_(setq x 100)               ;  глобальное значение X
100
_(defun первая (x)     ;  статическая X
  (вторая 2))
ПЕРВАЯ
_(defun вторая (y)
```

```
   (list x y))              ;  X свободна
ВТОРАЯ
_(первая 1)                 ;  свободная нединамическая
                            ;  переменная
_(100 2)                    ;  значением X является
                            ;  глобальное значение X
_(defvar x 100)             ;  начиная с этого X
X                           ; момента X динамическая переменная
_(первая 1)                 ; X определяется динамически
(1 2)                       ;  по последней связи
```

Если в функции не используется ни одной свободной переменной, то вычисления в обоих случаях производятся совершенно одинаково. Именно таким функциям надо отдавать предпочтение во время обучения языку Лисп.

Одно имя может обозначать разные переменные

Интересные различия возникают в использовании статических и глобальных (динамических) переменных, когда один и тот же символ является и фактическим, и формальным параметром:

```
_(setq x … )            ; глобальная X
…
_(defun fn (x) … )      ; статическая X
FN
_(fn 'x)                ; статическая X связывается
…                       ; с глобальной X
```

В случае, подобном приведенному, X в функции FN может быть именем как статической, так и глобальной (или динамической) переменной. В этом случае нужно различать, какая переменная что означает.

В начале главы мы описали функцию НЕ-МЕНЯЕТ, которая изменяла значение статической переменной X, и поэтому глобальное значение X сохранялось. В функции SET-DYN будет меняться динамическое значение X, так как значением первого аргумента вызова SET, т.е. статического X, будет динамическое X из вызова функции. Предположим, что X не объявлена динамической через DEFVAR:

```
_(setq x 'старое)
СТАРОЕ
_(defun set-dyn (x)
   (set x 'новое))          ; меняет динамическую X
SET-DYN
_(set-dyn 'x)
НОВОЕ
_X
```

НОВОЕ

Все участвующие в вычислениях переменные, как и первый фактический параметр вызова SET в предыдущем примере, являются динамическими. Сослаться на статическую переменную можно лишь в форме, не вычисляющей своих аргументов, такой как SETQ. Статические переменные не могут выступать как лисповские объекты сами по себе или в составе более сложных структур, поскольку они используются лишь как формальные имена, при помощи которых записываются вычисления. Следующая функция SET-DYN работает точно так же, как и предыдущая, так как значение первого аргумента вызова функции SET ссылается не на статическую переменную X, а на динамическую:

```
_(defun set-dyn (x)
(set (car '(x y))   'новое))
```

Функция QUOTE также возвращает в качестве значения лишь динамическую переменную, что видно из следующего примера:

```
_(defun проба-eval (x)  (eval 'x))
ПРОБА-EVAL
_(setq x 'старое)                ;  динамическая связь X
СТАРОЕ
_(проба-eval  'новое)            ;  аргументом EVAL
    СТАРОЕ                          ;  является не статическая,
                                 ;  а динамическая X
```

В некоторых Лисп-системах переменные по умолчанию являются не статическими, а динамическими. В такихсистемах в приведенных выше примерах резуль-таты будут другие. Если, например, переменная X была бы определена как динамическая специ-альная переменная, то функция SET-DYN не присвоила бы ей нового глобального значения. Значение можно было бы изменить лишь в динамическом контексте вызова SET-DYN, уничтожаемом после завершения вызова.

Современная тенденция развития языка ведет к статическим Лисп-системам. Фактически разница между динамическими и статическими переменными позволяет использовать во вложенных функциях динамическое значение переменной внешнего уровня, несмотря на то что встречались статические переменные с тем же именем. К ошибкам, возникающим из подобных конфликтов имен, мы вернемся подробнее в связи с функциональными вычислениями, работой в вычислительном контексте момента определения и в связи с макросами.

Статические вычисления позволяют сделать более хорошие трансляторы. С другой стороны, динамический интерпретатор Лиспа реализовать легче, чем статический. В некоторых системах интерпретируемые функции вычисляются динамически, аоттранслирован-ные функции вычисляются по статическим правилам. В результате одна и та же программа может работать по-разному в режиме интерпретации и в оттранслированном виде!

1.7 ВЫЧИСЛЕНИЕ В ЛИСПЕ

Программа состоит из форм и функций

Под формой (form) понимается такое символьное выражение, значение которого может быть найдено, интерпретатором. Ранее мы уже использовали наиболее простые формы языка: константы, переменные, лямбда-вызовы, вызовы функций и их сочетания. Кроме них были рассмотрены некоторые специальные формы, такие как QUOTE и SETQ, трактующие свои аргументы иначе, чем обычные функции. Лямбда-выражение без фактических параметров не является формой.

Вычислимые выражения можно разделить на три группы:

1. Самоопределенные (self-evaluating) формы. Эти формы, подобно константам, являются лисповскми объектами, представляющими лишь самих себя. Это такие формы, как числа и специальные константы T и NIL, а также знаки, строки и битовые векторы, рассматриваемые далее в главе, посвященной типам данных. Ключи, начинающиеся с двоеточия и определяемые через ключевое слово &KEY в лямбда-списке, также являются самоопределенными формами.

2. Символы, которые используются в качестве переменных.

3. Формы в виде списочной структуры, которыми являются:

a) Вызовы функций и лямбда-вызовы.

b) Специальные формы (special form), в число которых входят SETQ, QUOTE и многие описанные в этой главе формы, предназначенные для управления вычислением и контекстом.

c) Макровызовы (будут рассмотрены позже).

У каждой формы свой синтаксис и семантика, основанные, однако, на едином способе записи и интерпретации.

Управляющие структуры Лиспа являются формами

В распространенных процедурных языках наряду с основными действиями есть специальные управляющие механизмы разветвления вычислений и организации циклов. В Паскале, например, используются структуры IF THEN ELSE, WHILE DO, CASE и другие.

Управляющие структуры Лиспа (мы будем для них использовать термин предложение (clause)) выглядят внешне как вызовы функций. Предложения будут записываться в виде скобочных выражений, первый элемент которых действует как имя управляющей структуры, а остальные элементы - как "аргументы". Результатом вычисления, так же как у функции, является значение, т.е. управляющие структуры представляют собой формы. Однако предложения не являются вызовами функций, и разные предложения используют аргументы поразному.

Наиболее важные с точки зрения программирования синтаксические формы можно на основе их использования разделить на следующие группы:

Работа с контекстом:

- QUOTE или блокировка вычисления;
- вызов функции и лямбда-вызов;
- предложения LET и LET*.

Последовательное исполнение:

- предложения PROG1, PROG2 и PROGN.

Разветвление вычислений:

- условные предложения COND, IF, WHEN, UNLESS;
- выбирающее предложение CASE.

Итерации:

- циклические предложения DO, DO*, LOOP, DOTIMES, DOUNTIL

Передачи управления:

- предложения PROG, GO и RETURN.

Динамическое управление вычислением:

- THROW и CATCH, а также BLOCK.

LET создает локальную связь

Вычисление вызова функции создает на время вычисления новые связи для формальных параметров функции. Новые связи внутри формы можно создать и с помощью предложения LET. Эта структура (немного упрощенно) выглядит так:

```
(LET ((ml знач1) (m2 знач2) ...)
        форма1 форма2...)
```

Предложение LET вычисляется так, что сначала статические переменные ml, m2, ... из первого "аргумента" формы связываются (одновременно) с соответствующими значениями знач1, знач2, — Затем слева направо вычисляются значения форм форма1, форма2, … В качестве значения всей формы возвращается значение последней формы. Как и у функций, после окончания вычисления связи статических переменных ml, m2, ... ликвидируются и любые изменения их значений (SETQ) не будут видны извне. Например:

```
_(setq x 2)
2
_(let ((x 0))
   (setq x  1))
1
_x
2
```

Форма LET является на самом деле синтаксическим видоизменением лямбда-вызова, в которой формальные и фактические параметры помещены совместно в начале формы:

```
(LET ((m1 a1) (m2 a2) ... (mn an))
        Форма1 форма2 ...)
>
((LAMBDA
  (m1 m2 ... mn)              ; формальные параметры
  (форма1 форма2 … )          ; тело функции
  (a1 a2 ... an)              ; фактические параметры
```

Тело лямбда-выражения в Коммон Лиспе может состоять из нескольких форм, которые вычисляются последовательно, и значение последней формы возвращается в качестве значения лямбда-вызова.

Значения переменным формы LET присваиваются одновременно. Это означает, что значения всех переменных mi вычисляются до того, как осуществляется связывание

с формальными параметрами. Новые связи этих переменных еще не действуют в момент вычисления начальных значений переменных, которые перечислены в форме позднее. Например:

```
_(let ((x 2)  (y (* 3 x)))
(list x y))                    ;   при вычислении Y
Error:  Unbound atom X         ;    у X нет связи
```

Побочный эффект можно наблюдать при работе с формой LET* подобной LET, но вычисляющей значения переменных последовательно:

```
_(let * ((x 2)  (y (* 3 x)))
      (list x y))
(2 6)
```

Последовательные вычисления: PROG1, PROG2 и PROGN

Предложения PROG1, PROG2 и PROGN позволяют работать с несколькими вычисляемыми формами:

```
(PROG1 форма1 форма2 ... формаN)
(PROG2 форма1 форма2 ... формаN)
(PROGN форма1 форма2 ... формаN)
```

У этих специальных форм переменное число аргументов, которые они последовательно вычисляют и возвращают в качестве значения значение первого (PROG1), второго (PROG2) или последнего (PROGN) аргумента. Эти формы не содержат механизма определения внутренних переменных:

```
_(progn (setq x 2)  (setq y (*3 x)))
6
_x
2
```

Многие формы, как, например описанная выше форма LET(*), позволяют использовать последовательность форм, вычисляемых последовательно, и в качестве результата последовательности возвращают значение последней формы. Это свойство называют неявным PROGN (implicit progn feature).

Разветвление вычислений: условное предложение COND

Предложение COND является основным средством разветвления вычислений. Это синтаксическая форма, позволяющая управлять вычислениями на основе определяемых предикатами условий. Структура условного предложения такова:

```
(COND  (p1 al)
       (p2 a2)
        …
       (pN aN))
```

Предикатами pi и результирующими выражениями ai могут быть произвольные формы. Значение предложения COND определяется следующим образом:

1. Выражения pi, выполняющие роль предикатов, вычисляются последовательно слева направо (сверху вниз) до тех пор, пока не встретится выражение, значением которого не является NIL, т.е. логическим значением которого является истина.

2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения COND.

3. Если истинного предиката нет, то значением COND будет NIL.

Рекомендуется в качестве последнего предиката использовать символ T, и соответствующее ему результирующее выражение будет вычисляться всегда в тех случаях, когда ни одно другое условие не выполняется. В следующем примере с помощью предложения COND определена функция, устанавливающая тип выражения:

```
_(defun тип (l)
        (cond ((null l) 'пусто)
              ((atom l) 'атом)
              (t 'список)))
ТИП
_(тип '(a b c))
СПИСОК
_(тип (atom '(a  т  о  м)))
ПУСТО
```

В условном предложении может отсутствовать результирующее выражение ш или на его месте часто может быть последовательность форм:

```
(COND (pi a11)
    …
   (pi)                        ; результирующее
    …                        ; выражение отсутствует
  (pk akl ak2... akN)     ; последовательность форм
       …)                      ; в качестве результата
```

Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения COND при истинности предиката выдается само значение предиката. Если же условию соответствует несколько форм, то при его истинности формы вычисляются последовательно слева направо и результатом предложения COND будет значение последней формы последовательности (неявный PROGN).

В качестве примера использования условного предложения определим логические действия логики высказываний "и", "или", "не", => (импликация) и <=> (тождество):

```
_(defun и (x y)
  (cond (x y)
        (t nil)))
И
_(и t nil)
NIL
_(defun или (x y)
  (cond (x t)
        (t y)))
ИЛИ
_(или t nil)
T
```

```
_(defun не (x)
   (not x))
НЕ
_(не t)
NIL
_(defun => (x y)
   (cond (x y)
          (t t)))
=>
_(=> nil t)
T
```

Импликацию можно определить и через другие операции:

```
_(defun => (x y)
    (или x (не y)))
=>
_(defun <=> (x y)
    (и (=> x y)  (=> y x)))
<=>
```

Предикаты "и" и "или" входят в состав встроенных функций Лиспа и называются AND и OR. Число их аргументов может быть произвольным.

```
_(and (atom nil)  (null nil)  (eq nil nil))
T
```

Предикат AND в случае истинности возвращает в качестве значения значение своего последнего аргумента. Его иногда используют как упрощение условного предложения по следующему образцу:

```
(AND условие1 условие2 ... условиеN)
<=>
(COND ((AND условие! условие2 ...условиеN-1)
            условиеN
          (T NIL))

_(and (atom nil)  (+23))
 5
```

Такое использование предиката AND не рекомендуется. Предложения COND можно комбинировать таким же образом, как и вызовы функций. Например, предикат "исключающее или" (exclusive or или хог), который ложен, когда оба аргумента одновременно либо истинны, либо нет, можно определить следующим образом:

```
_(defun хог (x y)
   (cond (x (cond (y nil)
                  (t t)))
```

```
        (t y)))
XOR
_(xor t nil)
T
_(xor nil nil)
NIL
```

В этой функции на месте результирующего выражения первого условия вновь стоит предложение COND. На месте, отведенном условию, также можно использовать еще одно условное предложение, и в этом случае мы получим условное условие. Такие построения очень быстро приводят к труднопонимаемым определениям.

Другие условные предложения: IF, WHEN, UNLESS и CASE

Предложение COND представляет собой наиболее общую условную структуру. Ее критикуют за обилие скобок и за то, что структура этого предложения совершенно оторвана от естественного языка. Поэтому в Лисп-системах используются и другие, в различных отношениях более естественные, условные предложения. Но можно обойтись и с помощью лишь COND предложения.

В простом случае можно воспользоваться естественной и содержащей мало скобок формой IF.

```
(IF условие  то-форма  иначе-форма)
<=>
(COND (условие то-форма)
        (T иначе-форма))

_(if (atom t)  'атом 'список)
ATOM
```

Можно еще использовать формы WHEN и UNLESS:

```
(WHEN условие форма1 форма2 ...)
<=>
(UNLESS (NOT условие) форма1 форма2 ...)
<=>
(COND (условие форма1 форма2 ...))
<=>
(IF условие (PROGN форма1 форма2 ...) NIL)
```

Также можно применять подобное используемому в языке Паскаль выбирающее предложение CASE:

```
(CASE ключ
    (список-ключей1 m11 m12 … )
```

(список-ключей2 m21 m22 …)
…)

Сначала в форме CASE вычисляется значение ключевой формы ключ. Затем его сравнивают с элементами списков ключей список-ключеШ, с которого начинаются альтернативы. Когда в списке найдено значение ключевой формы, начинают вычисляться соответствующие формы mil, mi2, ..., значение последней из которых и возвращается в качестве значения всего предложения CASE (неявный PROGN).

Циклические вычисления: предложение DO

В случае повторяющихся вычислений в Лиспе используются вызывающие сами себя (рекурсивные) функции (и условные предложения) либо известные в основном по процедурным языкам циклы, передачи управления и другие подобные механизмы. Прежде всего познакомимся сначала с предлагаемыми Лиспом возможностями по использованию циклов. Самым общим циклическим или итерационным предложением в Коммой Лиспе является DO.

С его помощью можно задать:

1. Набор внутренних статических переменных с их начальными значениями, как в предложении LET(*).
2. Ряд форм, вычисляемых последовательно в цикле.
3. Изменения внутренних переменных после каждой итерации (например, наращивание счетчиков и т.п.).
4. Условие окончания цикла и выражение, значение которого будет значением всего предложения.

Предложение DO имеет следующую форму:

(DO ((varl знач1 шаг1) (var2 знач2 шаг2) …)
(условие-окочания форма11 форма12 …)
форма21
форма22
…)

Первый аргумент предложения описывает внутренние переменные varl, var2, ..., их начальные значения знач1, энач2, ... а также формы обновления шаг1, шаг2, … Вычисление предложения DO начинается с присваивания значений переменным формы таким же образом, как в случае предложения LET. Переменным, начальное значение которых не задано, присваивается по умолчанию NIL. В каждом цикле после присваивания значения переменным вычисляется условие окончания. Как только значение условия не равно NIL, т.е. условие окончания истинно, последовательно вычисляются формы форма1i, и значение последней формы возвращается как значение всего предложения DO. В противном случае последовательно вычисляются формы форма2i из тела предложения DO. На следующем цикле переменным vari присваиваются (одновременно) значения форм шагi, вычисляемых в текущем контексте, проверяется условие окончания и т.д. Если для переменной не задана форма, по которой она

обновляется, то значение переменной не меняется. Для примера с помощью предложения DO определим функцию, вычисляющую n-ю степень числа (n -целое, положительное):

```
_(defun exptl (x n)
   (do ((результат 1))            ; начальное
                                  ; значение
   ((= n 0) результат)            ; условие
                                  ;  окончания
   (setq результат (* результат x))
   (setq n (- n 1))))
EXPT1
_(exptl 2 3)
8
```

В качестве имени функции используется символ EXPT1, чтобы не переопределять лисповскую функцию возведения в степень EXPT.

Идея определения состоит в том, чтобы умножить X на себя N раз, что и является N-й степенью числа X. В каждом цикле значение переменной РЕЗУЛЬТАТ умножается на X до тех пор, пока значение счетчика N не уменьшится до 0 и конечное значение переменной РЕЗУЛЬТАТ можно будет выдать в качестве значения предложения DO.

Так как в предложении DO можно совместно с переменными описать и закон их изменения, то функцию EXPT можно было бы задать и такой формой:

```
_(defun expt2 (x n)
    (do ((результат x (* результат x))
         (разы n (- разы 1)))
        ((= разы 1) результат)))          ; условие
                                          ; окончания
```

В этом определении нет вычисляемого в цикле тела предложения ЕЮ, присутствуют только описания переменных, законов их изменения и условие завершения. Обратите внимание, что условие окончания функции EXPT1 отличается от условия окончания EXPT2. (Как бы думаете, почему?)

Аналогично тому, как предложению LET соответствовало последовательно вычисляющее свои переменные предложение LET*, так и у предложения ЕЮ есть свой вариант DO.

Предложения PROG, GO и RETURN

На Лиспе можно писать программы и в обычном операторном стиле с использованием передачи управления, как, например, в Фортране. Для этой цели уже в первых Лисп-системах существовало предложение PROG или PROG-механизм (prog feature). Значимость PROG-механизма в программировании уменьшилась в связи с введением в современных Лисп-системах более развитых условных и циклических

форм, таких как форма ЕЮ, так что использование PROG-механизма, например в Коммон Лиспе, в общем-то не рекомендуется. Можно показать, что все выразимое предложением PROG можно записать и с помощью предложения DO и, как правило, в более понятной форме. С помощью предложения PROG можно:

1. Осуществлять последовательное вычисление форм.
2. Организовывать циклы с помощью команды перехода.
3. Использовать локальные переменные формы.

Структура предложения PROG такая же, как и в более старых системах:

```
(PROG (ml m2 ... mN)
        оператор1
        оператор2
        …
        операторМ)
```

Перечисленные в начале формы переменные ml являются локальными статическими переменными формы, которые можно использовать для хранения промежуточных результатов так же, как это делается при программировании на операторных языках. Если какая-нибудь форма операторі является символом или целым числом, то это метка перехода (tag). На такую метку можно передать управление оператором GO:

```
(GO метка)
```

GO не вычисляет значение своего "аргумента".

Кроме этого, в PROG-механизм входит оператор окончания вычисления и возврата значения:

```
(RETURN результат)
```

Операторы предложения PROG вычисляются слева направо (сверху вниз), пропуская метки перехода. Оператор RETURN прекращает выполнение предложения PROG; в качестве значения всего предложения возвращается значение аргумента оператора PROG. Если во время вычисления оператор RETURN не встретился, то значением PROG после вычисления его последнего оператора станет NIL (Когда PROG-меха-низм используется для получения побочного эффекта, то возвращаемое значение не играет никакой роли.)

Через список переменных можно определить локальные для предложения PROG программные переменные (program variable). Перед вычислениями им присваиваются значения NIL. Если переменных нет, то на месте списка переменных нужно оставить NIL. После вычисления значения формы связи программных переменных исчезают так же, как и значения переменных форм LET(*) и DO(*) или как связи формальных параметров лямбда-выражения в вызове функции.

В следующем примере предложение PROG используется для определенной нами ранее через DO функции возведения в степень EXPT:

```
_(defun expt3 (x n)
      (PROG (результат)
             (setq результат x)
            loop                                  ; метка
           (if (= n 1)
               (RETURN результат))        ; выход
           (setq результат (* результат x))
           (setq n (- n 1))
           (GO loop)))                          ; передача управления EXPT3
_(expt3 2 3)
8
_результат
Error: Unbound atom РЕЗУЛЬТАТ
```

Это определение явно более громоздко, чем описанные выше версии, основанные на DO.

Механизм передачи управления и предложение RETURN можно использовать наряду с PROG и в некоторых других формах, как, например, DO(*).

Формы GO и RETURN являются примерами статических форм, т.е. они управляют вычислением только в пределах текста определения.

Другие циклические структуры

В Коммон Лиспе есть еще циклические предложения. Форма

(LOOP ml m2 ...)

реализует цикл, в котором формы ml, m2, ... вычисляются снова и снова. Цикл завершается лишь в случае, если в какой-нибудь из вычисляемых форм не встретится явный оператор завершения RETURN (или другая форма, прекращающая вычисления). Часто некоторый цикл надо выполнить определенное количество раз или выполнить его с каждым элементом списка. В Коммон Лиспе для этого имеются формы DOTIMES и DOLIST.

Повторяющиеся вычисления используются и в MAP-функциях, и в других функциях с функцией в качестве аргумента, или в функционалах. В дальнейшем мы рассмотрим функционалы более подробно. В Лиспе, как мы увидим далее, довольно легко и самим описывать новые структуры управления.

Повторение через итерацию или рекурсию

В "чистом" функциональном Лиспе нет ни циклических предложений (DO, PROG и другие), ни тем более операторов передачи управления. Для программирования повторяющихся вычислений в нем используются лишь условные предложения и

определения рекурсивных, или вызывающих самих себя, функций. Например, рекурсивный вариант функции EXPT можно было бы определить так:

```
(defun expt4 (x n)
    (if (= n 1) x
        (* x (expt4 x (- n 1)))))
EXPT4
```

Функция EXPT4 вызывает себя до тех пор, пока используемый как счетчик аргумент N не уменьшится до единицы, т.е. всего N-1 раз. После этого в качестве результата вызова функции EXPT4 возвращается значение X. При каждой передаче значения на предыдущий уровень результат умножается на X. Так X окажется перемноженным на себя N раз.

Рекурсивное определение функции EXPT короче и в большей степени отражает суть функции, чем версии, основанные на DO и PROG.

Рассмотрим еще одну функцию, просто определяемую через рекурсию, - факториал (факториал - это произведение данного числа на все целые положительные числа, меньшие данного. Например, факториал от 5, обозначаемый как 5!, есть 1*2*3*4*5=120. Факториалом нуля считается 1:

```
(defun !  (n)
   (if (= n 0)  1
       (* n (!   (- n 1»)))
!
_(!  5)
120
```

Итеративные и рекурсивные программы теоретически одинаковы по своим вычислительным возможностям, иными словами, множества вычислимых с их помощью функций совпадают (так называемые частично рекурсивные функции). Так что любое вычисление в принципе, можно запрограммировать любым из этих способов. Однако свойства итеративных и рекурсивных вариантов программ могут существенно отличаться. В связи с этим часто приходится решать, какой из способов программирования больше подходит для данной задачи. От сделанного выбора зависит простота и естественность программирования, а также его эффективность с точки зрения времени исполнения и использования памяти.

С помощью итеративного программирования, как правило, более длинного и трудного в осуществлении, результат может вычисляться значительно проще и быстрее. Это происходит по двум причинам, во-первых, потому, что вычислительные машины в общем ориентированы на последовательные вычисления, и, во-вторых, потому, что трансляторы не всегда в состоянии преобразовать рекурсивное определение в итеративное и используют при вычислениях стек, несмотря на то что он не всегда нужен.

Рекурсивное программирование в общем более короткое и содержательное. Особенно полезно использовать рекурсию в тех случаях, когда решаемая задача или обрабатываемые данные по сути своей рекурсивны. Например, математическое

определение факториала рекурсивно и его реализация через рекурсивную функцию совершенно естественна:

n! = 1 , если n=0
n! = n*(n-1)! , если n>0

Рекурсивное решение хорошо подходит для работы со списками, так как списки могут рекурсивно содержать подсписки. Рекурсивными функциями можно с успехом пользоваться при работе с другими динамическими структурами, которые заранее не полностью известны. Рекурсивные процедуры занимают важное место почти во всех программах, связанных с искусственным интеллектом.

В главе, посвященной технике функционального программирования, мы попробуем показать, что программирование с помощью рекурсии и условного предложения вполне осуществимо и разумно. Мы увидим, что рекурсия позволяет взглянуть на программирование с другой точки зрения, которая значительно отличается от обычной основанной на операторах и отраженной, например, в предложении PROG.

Формы динамического прекращения вычислений: CATCH и THROW

До сих пор мы рассматривали лишь структуры, вычисление которых производится в одном статическом контексте. В некоторых случаях возникает необходимость прекратить вычисление функции динамически из другого вычислительного контекста, где вычисляются некоторые подвыражения, и выдать результат в более раннее состояние, не осуществляя нормальную последовательность возвратов из всех вложенных вызовов (dynamic non-local exit). Это нужно, например, тогда, когда какая-нибудь вложенная программа обнаружит ошибку, по которой можно решить, что дальнейшая работа бесполезна либо может даже навредить. Возврат же управления по обычным правилам привел бы к продолжению вычислений на внешних уровнях.

Такое динамическое прерывание вычислений можно запрограммировать в Коммон Лиспе с помощью форм CATCH и THROW, которые, как это видно из их имен (ПОЙМАТЬ, БРОСИТЬ), передают управление. Подготовка к прерыванию осуществляется специальной формой CATCH:

(CATCH метка форма1 форма2 ...)
Например:

```
(CATCH  'возврат1
     (делай-раз)  (делай-два)  (делай-три))
```

При вычислении формы сначала вычисляется метка, а затем формы форма1, форма2, ... слева направо. Значением формы будет последнее значение (неявный &PROG) при условии, что во время вычисления непосредственно этих форм или форм вызванных из них не встретится предложение THROW:

(THROW метка значение)

Если аргумент метка вызова THROW представляет собой тот же лисповский объект, что и метка в форме CATCH, та управление передается обратно в форму CATCH и его значением станет значение второго аргумента формы THROW. В приведенном ранее примере в результате вычисления формы

(THROW возврат1 'сделано)

вызов CATCH получает значение СДЕЛАНО, и если это произошло во время вычисления функции ДЕЛАЙ-ДВА, то вычисление ДЕЛАЙ-ТРИ отменяется. Механизм CATCH-THROW позволяет осуществлять возврат управления из динамического окружения, вложенного на любую глубину.

1.8 ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СПИСКОВ

В этой главе рассматриваются представление списков и атомов в памяти машины, а также специальные функции, с помощью которых можно изменять внутреннюю структуру списков. Без знания внутренней структуры списков и принципов ее использования изучение работы со списками и функциями останется неполным.

В чистом функциональном программировании специальные функции, изменяющие структуры, не используются. В функциональном программировании лишь создаются новые структуры путем анализа, расчленения и копирования ранее созданных структур. При этом созданные структуры никогда не изменяются и не уничтожаются структуры, значения которых уже не нужны. В практическом программировании псевдофункции, изменяющие структуры, иногда все-таки нужны, хотя их использования в основном пытаются избежать.

Лисповская память состоит из списочных ячеек

Оперативная память машины, на которой работает Лисп-система, логически разбивается на маленькие области, которые называются списочными ячейками (memory cell, list cell, cons cell или просто cons). Списочная ячейка состоит из двух частей, полей CAR и CDR. Каждое из полей содержит указатель (pointer). Указатель может ссылаться на другую списочную ячейку или на некоторый другой лисповский объект, как, например, атом. Указатели между ячейками образуют как бы цепочку, по которой можно из предыдущей ячейки попасть в следующую и так, наконец, до атомарных объектов. Каждый известный системе атом записан в определенном месте памяти лишь один раз. (В действительности в Коммон Лиспе можно использовать много пространств имен, в которых атомы с одинаковыми именами хранятся в разных местах и имеют различную интерпретацию.)

Графически списочная ячейка представляется прямоугольником, разделенным на части (поля) CAR и CDR. Указатель изображается в виде стрелки, начинающейся водной из частей прямоугольника и заканчивающейся на изображении другой ячейки или атоме, на которые ссылается указатель.

Значение представляется указателем

Указателем списка является указатель на первую ячейку списка. На ячейку могут указывать не только поля CAR и CDR других ячеек, но и используемый в качестве переменной символ, указатель из которого ссылается на объект, являющийся значением символа. Указатель на значение хранится вместе с символом в качестве его системного свойства. Кроме значения системными свойствами символа могут быть определение функции, представленное в виде лямбда-выражения, последовательность знаков, задающая внешний вид переменной (print name), выражение, определяющее пространство, в которое входит имя, и список свойств (property list). Эти системные свойства, которые мы рассмотрим подробнее в следующей главе, не будут отражены на наших рисунках.

Побочным эффектом функции присваивания SETQ является замещение указателя в поле значения символа. Например, следующий вызов:

_(setq список '(a b c))
(A B C)

создает в качестве побочного эффекта изображенную на рис. штриховую стрелку.

список

Изображение одноуровнего списка состоит из последовательности ячеек, связанных друг с другом через указатели в правой части ячеек. Правое поле последней ячейки списка в качестве признака конца списка ссылается на пустой список, т.е. на атом NIL Графически ссылку на пустой список часто изображают в виде перечеркнутого поля. Указатели из полей CAR ячеек списка ссылаются на структуры, являющиеся элементами списка, в данном случае на атомы A, B и C.

CAR и CDR выбирают поле указателя

Работа селекторов CAR и CDR в графическом представлении становится совершенно очевидной. Если мы применим функцию CAR к списку СПИСОК, то результатом будет содержимое левого поля первой списочной ячейки, т.е. символ A:

_(car список)
A
Соответственно вызов
_(cdr список)
(B C)
возвращает значение из правого поля первой списочной ячейки, т.е. список (B C).

CONS создает ячейку и возвращает на нее указатель

Допустим, что у нас есть два списка:
_(setq голова '(b c))
(B C)
_(setq хвост '(a b c))
(A B C)

Вызов функции

_(cons голова хвост)

((B C) A B C)

строит новый список из ранее построенных списков ГОЛОВА и ХВОСТ так, как это показано на рис.

(CONS ГОЛОВА ХВОСТ)

ХВОСТ

A ГОЛОВА B C

CONS создает новую списочную ячейку (и соответствующий ей список). Содержимым левого поля новой ячейки станет значение первого аргумента вызова, а правого - значение второго аргумента. Обратите внимание, что одна новая списочная ячейка может связать две большие структуры в одну новую структуру. Это довольно эффективное решение с точки зрения создания структур и их представления в памяти. Заметим, что применение функции CONS не изменило структуры списков, являющихся аргументами, и не изменило значений переменных ГОЛОВА и ХВОСТ.

У списков могут быть общие части

На одну ячейку может указывать одна или более стрелок из списочных ячеек, однако из каждого поля ячейки может исходить лишь одна стрелка. Если на некоторую ячейку есть несколько указателей, то эта ячейка будет описывать общее подвыражение. Например, в списке

(кто-то приходит кто-то уходит)

символ КТО-ТО является общим подвыражением, на которое ссылаются указатели из поля CAR из первой и из третьей ячейки списка.

Если элементами списка являются не атомы, а подсписки, то на месте атомов будут находится первые ячейки подсписков. Например, построенная вызовом

_(setq список1 '((b c) a b c)

((B C) A B C)

структура изображена на рис.

СПИСОК1

A B C

Из этого рисунка видно, что логически идентичные атомы содержатся в системе один раз, однако логически идентичные списки могут быть представлены различными списочными ячейками. Например, значения вызовов

_(car список1)

(B C)

```
_(cddr список1)
(B C)
```

являются логически одинаковым списком (B C), хотя они и представлены различными списочными ячейками:

```
_(equal (car список1)  (cddr список1))
T
```

Однако список (B C), как видно из рис., может состоять и из тех же ячеек.

Эту структуру можно создать с помощью следующей последовательности вызовов:

```
_(setq bc  '(b c))
(B C)
_(setq abc (cons 'a  bc))
(A B C)
_(setq список2 (cons bc  abc))
((B C)  A B C)
_список2
((B C)  A B C)
```

Таким образом, в зависимости от способа построения логическая и физическая структуры двух списков могут оказаться различными. Логическая структура всегда топологически имеет форму двоичного дерева, в то время как физическая структура может быть ациклическим графом, или, другими словами, ветви могут снова сходиться, но никогда не могут образовывать замкнутые циклы, т.е. указывать назад. В дальнейшем мы увидим, что, используя псевдофункции, изменяющие структуры (поля) (RPLACA, RPLACD и другие), можно создать и циклические структуры.

СПИСОК 2

Логическое и физическое равенство не одно и то же

Логически сравнивая списки, мы использовали предикат EQUAL, сравнивающий не физические указатели, а совпадение структурного построения списков и совпадение атомов, формирующих список.

Предикат EQ можно использовать лишь для сравнения двух символов. Во многих реализациях языка Лисп предикат EQ обобщен таким образом, что с его помощью можно определить физическое равенство двух выражений (т.е. ссылаются ли значения аргументов на один физический лисповский объект) не зависимо от того, является ли он атомом или списком. При сравнении символов все равно, каким предикатом пользоваться, поскольку атомарные объекты хранятся всегда в одном и том же месте. При сравнении списков нужно поступать осторожнее.

Вызовы функции EQ из следующего примера возвращают в качестве значения NIL, так как логически одинаковые аргументы в данном случае представлены в памяти физически различными ячейками:

```
_(eq '((b c)  a b c)   '((b c)  a b c))
NIL
_(eq список1 список2)
NIL
_(eq (car список1)  (cddr список1))
NIL
```

Поскольку части CAR и CDR списка СПИСОК2 представлены при помощи одних и тех же списочных ячеек, то получим следующий результат:

```
_(eq (car список2)   (cddr список2)) T
```

Точечная пара соответствует списочной ячейке

Определяя базовую функцию CONS, мы предполагали, что ее вторым аргументом является список. Это ограничение не является необходимым, так как при помощи списочной ячейки можно было бы, например, результат вызова

```
(cons 'a 'b)
```

представить в виде структуры, изображенной на рис.

```
'(A . B)

                    B

          A
```

На рис. показан не список, а более общее символьное выражение, так называемая точечная пара (dotted pair). Для сравнения на след. рис. мы изобразили список (A B).

Название точечной пары происходит из использованной в ее записи точечной нотации (dot notation), в которой для разделения полей CAR и CDR используется выделенная пробелами точка:

```
_(cons 'a  'b)
(A .  B)
```

Выражение слева от точки (атом, список или другая точечная пара) соответствует значению поля CAR списочной ячейки, а выражение справа от точки - значению поля CDR. Базовые функции CAR и CDR действуют совершенно симметрично:

```
_(car '( a . b )            ;  обратите внимание на
```

A ; пробелы, выделяющие точку
_(cdr '(a . (b . c)))
(B.C)

Точечная нотация позволяет расширить класс объектов, изображаемых с помощью списков. Ситуацию можно было бы сравнить с началом использования дробей или комплексных чисел в арифметике.

Варианты точечкой и списочной записей

Любой список можно записать в точечной нотации. Преобразование можно осуществить (на всех уровнях списка) следующим образом:

(a1 a2 ... aN)
<=>
(al . (a2 . ...(aN . NIL) ...))

Приведем пример:
(a b (c d) e)
<=>
(a . (b . ((c . (d . NIL)) . (e . NIL))))

Признаком списка здесь служит NIL в поле CDR последнего элемента списка, символизирующий его окончание.

Транслятор может привести записанное в точечной нотации выражение частично или полностью к списочной нотации. Приведение возможно лишь тогда, когда поле CDR точечной пары является списком или парой. В этом случае можно опустить точку и скобки вокруг выражения, являющегося значением поля CDR:

(al . (a2 a3)) <=> (al a2 a3)
(al . (a2 . a3)) <=> (al a2 . a3)
(al a2 . NIL) <=> (al a2 . ( )) <=> (al a2)

Точка останется в выражении лишь в случае, если в травой части пары находится атом, отличный от NIL. Убедиться в том, что произведенные интерпретатором преобразования верны, можно, нарисовав структуры, соответствующие исходной записи и приведенному виду:

_'(a . (b . (c . (d)))
(A B C D)
_'((a b) . (b c))
((A B) B C)
_'(a . nil)
(A)
_'(a . (b . c))
(A B . C)
_'((((nil . a) . b) . c) . d)
((((NIL . A) . B) . C) . D)

Использование точечных пар в программировании на Лиспе в общем-то излишне. С их помощью в принципе можно несколько сократить объем необходимой памяти. Например структура данных

(a b c)

записанная в виде списка, требует трех ячеек, хотя те же данные можно представить в виде

(a b . c)

требующем двух ячеек. Более компактное представление может несколько сократить и объем вычислений за счет меньшего количества обращений в память.

Точечные пары применяются в теории, книгах и справочниках по Лиспу. Часто с их помощью обозначают список заранее неизвестной длины в виде

(голова . хвост)

Точечные пары используются совместно с некоторыми типами данных и с ассоциативными списками, а также в системном программировании. Большинство программистов не используют точечные пары, поскольку по сравнению с требуемой в таком случае внимательностью получаемый выигрыш в объеме памяти и скорости вычислений обычно не заметен.

Управление памятью и сборка мусора

В результате вычислений в памяти могут возникать структуры, на которые потом нельзя сослаться. Это происходит в тех случаях, когда вычисленная структура не сохраняется с помощью SETQ или когда теряется ссылка на старое значение в результате побочного эффекта нового вызова SETQ или другой функции. Если, например, изображенному на рис. списку СПИСОК3

```
_(setq список3
      '((это станет мусором) cdr часть))
(ЭТО СТАНЕТ МУСОРОМ) CDR ЧАСТЬ)
```

присвоить новое значение

```
_(setq список3 (cdr список3)) (CDR ЧАСТЬ)
```

то CAR-часть как-бы отделяется, поскольку указатель из атома СПИСОК3 начинает ссылаться так, как это изображено на рисунке при помощи штриховой стрелки. Теперь уже нельзя через символы и указатели добраться до четырех списочных ячеек. Говорят, что эти ячейки стали мусором.

Мусор возникает и тогда, когда результат вычисления не связывается с какой-нибудь переменной. Например, значение вызова

```
_(cons 'a (list 'b))
(A B)
```

лишь печатается, после чего соответствующая ему структура останется в памяти мусором.

Для повторного использования ставшей мусором памяти в Лисп-системах предусмотрен специальный мусорщик (garbage collector), который автоматически запускается, когда в памяти остается мало свободного места. Мусорщик перебирает все ячейки и собирает являющиеся мусором ячейки в список свободной памяти (free Irsi) для того, чтобы их можно было использовать заново.

Пользователь может заметить работу мусорщика только по тому, что вычисления время от времени приостанавливаются на момент, когда система выводит успокаивающее программиста сообщение. Для сборки мусора в разных системах предусмотрены различные процедуры. В некоторых системах мусорщик непрерывно работает на фоне вычислений. В таких системах не происходит внезапных остановок вычислений, которые недопустимы в так называемых системах реального времени.

Вычисления, изменяющие и не изменяющие структуру

Все рассмотренные до сих пор функции манипулировали выражениями, не вызывая каких-либо изменений в уже существующих выражениях. Например, функция CONS, которая вроде бы изменяет свои аргументы, на самом деле строит новый список, функции CAR и CDR в свою очередь лишь выбирают один из указателей. Структура существующих выражений не могла измениться как побочный эффект вызова функции. Значения переменных можно было изменить лишь целиком, вычислив и присвоив новые значения целиком. Самое большее, что при этом могло произойти со старым выражением, - это то, что оно могло пропасть.

В Лиспе всетаки есть и специальные функции, при помощи которых на структуры уже существующих выражений можно непосредственно влиять так же, как, например, в Паскале. Это осуществляют функции, которые, как хирург, оперируют на внутренней структуре выражении.

Такие функции называют структура-разрушающими (destructive), поскольку с их помощью можно разорвать структуру и склеить ее по-новому.

1.9 СВОЙСТВА СИМВОЛА

У символа могут быть свойства

В Лиспе с символом можно связать именованные свойства (property). Свойства символа записываются в хранимый вместе с символом список свойств (property list, p-list).

У свойств есть имя и значение

Список свойств может быть пуст или содержать произвольное количество свойств. Его форма такова:

(имя1 значение1 имя2 значение2
 ...
 имяN значениеN)

Например, у символа ЯГОДА-РЯБИНЫ может быть такой список свойств:

(вкус кислый цвет красный)

Список свойств символа можно использовать без особых ограничений, его можно по необходимости обновлять или удалять. Программист должен сам предусматривать и обрабатывать интересующие его свойства.

Системные и определяемые свойства

С символом связаны лишь его имя, произвольное, назначенное функцией присваивания (SETQ), значение и назначенное определением функции (DEFUN) описание вычислений (лямбда-выражекке}. Значение и определение функции являются встроенными системными свойствами, которые управляют работой интерпретатора в различных ситуациях. Функции, используемые для чтения и изменения этих свойств (SETQ, SYMBOL-VALUE, DEFUN, FUNCTION-VALUE и другие), мы уже ранее рассматривали. Весь список свойств также является системным свойством. Работающие со свойствами символов прикладные системы могут свободно определять новые свойства.

Чтение свойства

Выяснить значение свойства, связанного с символом, можно с помощью функции GET:

(GET символ свойство)

Если, например, с символом ЯГОДА-РЯБИНЫ связан определенный нами ранее список свойств, то мы получим следующие результаты:

_(get 'ягода-рябины 'вкус)
КИСЛЫЙ
_(get 'ягода-рябины 'вес)
NIL

Так как у символа ЯГОДА-РЯБИНЫ нет свойства ВЕС, то GET вернет значение NIL.

Присваивание свойства

Присваивание нового свойства или изменение значения существующего свойства в основных диалектах языка Лисп осуществляется псевдофункцией PUTPROP (put property) или PUT:

(PUTPROP символ свойство значение)

В Коммон Лиспе функции PUTPROP не существует. Свойства символов находятся в связанных с символами ячейках памяти, для присваивания значений которым используется обобщенная функция присваивания SETF. Присваивание свойства в Коммон Лиспе осуществляется через функции SETF и GET следующим образом:

(SETF (GET символ свойство) значение)

Здесь вызов GET возвращает в качестве значения ячейку памяти для данного свойства, содержимое которой обновляет вызов SETF. Присваивание будет работать и в том случае, если ранее у символа не было такого свойства. Приведем пример:

_(setf (get 'ягода-рябины 'вес) '(2 g))
(2 G)
_(get 'ягода-рябины 'вес)
(2 G)

Побочным эффектом вызова будет изменение списка свойств символа ЯГОДА-РЯБИНЫ следующим образом:

(ВЕС (2 G) ВКУС КИСЛЫЙ ЦВЕТ КРАСНЫЙ)

Псевдофункция SETF меняет физическую структуру списка свойств. Поэтому использование других списков как части списка свойств без их предварительного копирования может привести к неожиданным ошибкам.

Удаление свойства

Удаление свойства и его значения осуществляется псевдофункцией REMPROP:

(REMPROP символ свойство)

Приведем пример:

_(remprop 'ягода-рябины 'вкус)
ВКУС
_(get 'ягода-рябины 'вкус)
NIL

Псевдофункция REMPROP возвращает в качестве значения имя удаляемого свойства. Если удаляемого свойства нет, то возвращается NIL Свой-ство можно удалить, присвоив ему значе-ние NIL. В этом случае имя свойства и значение NIL физически остаются в списке свойств. Читать из списка свойств, создавать и обновлять в нем свойства можно не только по отдельности, но и целиком. Например, в Коммон Лиспе значением вызова

(SYMBOL-PLIST символ)

является весь список свойств:

_(symbol-plist 'ягода-рябины)
(ВЕС (2 G) ЦВЕТ КРАСНЫЙ)

Свойства глобальны

Свойства символов независимо от их значений доступны из всех контекстов до тех пор, пока они не будут явно изменены или удалены. Использование символа в качестве функции или переменной, т.е. изменение значения символа или определения функции, не влияет на другие свойства символа, и они сохраняются.

Список свойств используется во многих системных программах Лисп-систем. Наличие свойств полезно как для поддержки работы самой Лисп-системы, так и во многих типичных случаях представления данных. Использование свойств дает средства для рассматриваемого позже программирования, управляемого данными, с помощью которого можно реализовать различные языки представления знаний и формализмы,

такие как семантические сети (semantic net), фреймы (frame) и объекты объектно-ориентированного программирования (object, flavor).

В некоторых системах можно использовать в качестве обобщения так называемые свободные списки свойств (disembodied property list), несвязанные с каким-либо символом.

1.10 ВВОД И ВЫВОД

Ввод и вывод входят в диалог

До сих пор в определенных нами функциях ввод данных (READ) и вывод (PRINT) осуществлялись в процессе диалога с интерпретатором. Интерпретатор читал вводимое пользователем выражение, вычислял его значение и возвращал его пользователю. Сами формы и функции не содержали ничего, связанного с вводом или выводом.

Если не использовать специальную команду ввода, то данные можно передавать лисповской функции только через параметры и свободные переменные. Соответственно, без использования вывода, результат можно получить лишь через конечное значение выражения. Часто все же возникает необходимость вводить исходные данные и выдавать сообщения и тем самым управлять и получать промежуточные результаты во время вычислений, как это делается и в других языках программирования.

READ читает и возвращает выражение

Лисповская функция чтения READ отличается от ввода в других языках программирования тем, что она обрабатывает выражение целиком, а не одиночные элементы данных. Вызов этой функции осуществляется пользователем (немного упрощенно) в виде

(READ)

Как только интерпретатор встречает предложение READ, вычисления приостанавливаются до тех пор, пока пользователь не введет какой-нибудь символ или целиком выражение:

```
_(read)
(вводимое выражение)          ; выражение
                              ; пользователя
(ВВОДИМОЕ ВЫРАЖЕНИЕ)          ; значение функции
 …                            ; READ
```

Обратите внимание, READ никак не показывает, что он ждет ввода выражения. Программист должен сам сообщить об этом при помощи рассматриваемых позже функций вывода. READ лишь читает выражение и возвращает в качестве значения само это выражение, после чего вычисления продолжаются.

У приведенного выше вызова функции READ не было аргументов, но у этой функции есть значение, которое совпадаете введенным выражением. По своему действию READ представляет собой функцию, но у нее есть побочный эффект, состоящий именно во вводе выражения. Учитывая это, READ является не чистой функцией, а псевдофункцией.

Если прочитанное значение необходимо сохранить для дальнейшего использования, то вызов READ должен быть аргументом какой-нибудь формы, например присваивания (SETQ), которая свяжет полученное выражение:

```
_(setq  input  (read))
(+ 2 3)                    ;  введенное выражение
(+ 2 3)                    ;  значение
_input
(+ 2 3)
```

Форма, вызывающая интерпретатор, и функция READ совместно с другими функциями позволяют читать выражения внешние по отношению к программе. Из них можно строить новые лисповские выражения или целые программы. Построенные структуры можно вычислить, передав их непосредственно интерпретатору:

```
_(eval input)
5
_(eval (list (read)  (read)  (read)))
+  2  3
5
```

Программа ввода выделяет формы

Функция READ основана на работающей на системном уровне процедуре чтения (Lisp reader). Она читает s-выражение, образуемое последовательностью знаков, поступающих из файла или иного источника. Внешние устройства становятся доступными из Лисп-системы через объекты, называемые потоками (stream). На логическом уровне потоки независимо от характера внешнего устройства являются последовательностью читаемых или записываемых знаков или битов. Для ввода и вывода, как и для двустороннего обмена, существуют свои типы потоков и специальные операции.

Макросы чтения изменяют синтаксис Лиспа

Процедура чтения содержит анализатор (parser), проверяющий знаки в читаемой им последовательности. Чтение обычного алфавитно-цифрового знака никаких особых действий не требует, в то время как чтение специального знака, такого как открывающая или закрывающая скобка, пробел, разделяющий элементы, или точка, приводит к специальным действиям. Соответствие между различными знаками и действиями определяется так называемой таблицей чтения (read table), которая задает лисповские функции для знаков.

Знаки, вызывающие специальные действия, называют макрознаками (macro character) или макросами чтения (read macro), поскольку их чтение требует более сложных действии. Таблица чтения доступна программисту, и он может сам определять новые интерпретации знаков и, таким образом, расширять или изменять синтаксис Лиспа.

Действие макроса-чтения определяется в Коммой Лиспе при помощи обыкновенной функции. Она читает и возвращает в качестве значения форму, для построения которой она в свою очередь может предварительно использовать макросы. Определим для примера макрос чтения %, действующий так же, как апостроф. Действие блокировки вычисления пользователь может определить в виде функции, которая рекурсивно читает очередное выражение и возвращает его в составе формы QUOTE:

```
(defun quote-блокировка (поток знак)
    (list 'quote (read)))
```

Функция, определяющая макрос чтения, имеет в Коммон Лиспе два аргумента, первый из которых описывает поток чтения, а значением второго будет сам макрознак. В данном примере мы не использовали параметры.

Запись символов и определенных для них макроинтерпретаций в таблицу чтения осуществляется командой

```
(SET-MACRO-CHARACTER  знак  функция)

_(set-macro-character ~
    #\%  'quote-блокировка)
T
```

Здесь запись #\% обозначает знак процента (%) как объект с типом данных знак (в дальнейшем к типам данных мы вернемся подробнее).

После этих определений можно (с точки зрения пользователя) знак процента использовать так же, как апостроф:

```
_(list %знак %процента)
(ЗНАК ПРОЦЕНТА)
_%(a %b  c)
(A (QUOTE  B) C)
```

Таблиц чтения может быть несколько, но процедура чтения использует в каждый момент времени лишь одну таблицу. Текущая таблица сохраняется как значение системной переменной *READTABLE*.

Встроенными макросами чтения в Коммон Лиспе являются:

```
(    ; начинает ввод списка или точечной пары
)    ; заканчивает ввод списка или точечной пары
'    ; возвращает очередное выражение в виде
     ; вызова QUOTE
;    ; символы до конца строки считаются
     ; комментарием
\    ; выделение одиночного специального знака
|    ; выделение нескольких специальных знаков
 ; | … |
```

" ; строка: "..."

Макрознаки нельзя использовать в составе символов наподобие обычных знаков, поскольку процедура чтения проинтерпретирует их в соответствии с таблицей как макросы чтения. Для включения таких знаков в состав имен нужно использовать специальные выделяющие знаки "\" (backslash) и "|" (bar), которые блокируют макрообработку знаков.

Символы хранятся в списке объектов

Читая и интерпретируя знаки, процедура чтения пытается строить атомы и из них списки. Прочитав имя символа, интерпретатор ищет, встречался ли ранее такой символ или он неизвестен. Для нового символа нужно зарезервировать память для возможного значения, определения функции и других свойств. Символы сохраняются в памяти в списке объектов (object list, oblist) или массиве объектов (obarray), в котором они проиндексированы на основании своего имени. Список объектов содержит как созданные пользователем, так и внутрисистемные символы (например, CAR, CONS, NIL и т.д.). Внесение символа в список объектов называют включением или интернированием (intern).

Пакеты или пространства имен

В более новых Лисп-системах, как и в Коммон Лиспе, можно пользоваться несколькими различными списками объектов, которые называют пакетами (package) или пространствами имен (name space). Символы из различных пространств, имеющие одинаковые имена, могут использоваться различным образом. Это необходимо при построении больших систем, при программировании различных ее подсистем программисты частенько используют одинаковые имена для различных целей. Текущее пространство имен определяется по значению глобальной системной переменной. На атомы из других пассивных пакетов можно сослаться, написав перед символом через двоеточие имя пакета:

пакет: символ

Перед использованием такой записи необходимо, чтобы символ, на который ссылаются, был объявлен внешней (external) переменной пространства имен. (Однако и на остальные, т.е. внутренние (internal) переменные, в принципе, можно сослаться, но более специфическим способом.)

PRINT переводит строку, выводит значение и пробел

Для вывода выражений можно использовать функцию PRINT. Это функция с одним аргументом, которая сначала вычисляет значение аргумента, а затем выводит это значение. Функция PRINT перед выводом аргумента переходит на новую строку, а после него выводит пробел. Таким образом, значение выводится всегда на новую строку

```
_(print ( + 2 3 ))
5                    ; вывод (эффект)
5                    ; значение
```

```
_(print (read))
(+ 2 3)          ; ввод
(+ 2 3)          ; вывод
(+ 2 3)          ; значение
```

Как и READ, PRINT1 является псевдофункцией, у которой есть как побочный эффект, так и значение. Значением функции является значение его аргумента, а побочным эффектом -печать этого значения.

Лисповские операторы ввода-вывода, как и присваивания, очень гибки, поскольку их можно использовать в качестве аргументов других функций, что в других языках программирования обычно невозможно:

```
_(+ (print 2) 3)
2
5
_(setq x '( + 2 3))
(+ 2 3)
_(eval (print x))
(+ 2 3)
5
_(eval (setq y (print x)))
(+ 2 3)
 5
_y
(+ 2 3)
```

FORMAT выводит в соответствии с образцом

Печать выражений в сложной форме с помощью функций PRINx и TERPRI требует либо предварительного построения специальных структур, либо требует последовательного использования большого числа функций вывода. Программирование, таким образом, усложняется, и поэтому не сразу можно понять, что же на самом деле печатается.

Для решения этих проблем в Коммон Лиспе есть функция FORMAT, при помощи которой можно, используя параметры, гибко задать формат печати. О множестве возможностей и, с другой стороны, о сложности функции FORMAT свидетельстует то, что в спецификации Ком мои Лиспа ей посвящено несколько десятков страниц. Мы рассмотрим лишь самые существенные с точки зрения практического программирования форматы вывода. Форма вызова функции FORMAT следующая:

(FORMAT поток образец &REST аргументы)

Первый аргумент формы поток задает файл или устройство, куда осуществляется вывод. Потоку присваивается значение T, если вывод осуществляется на экран. При выводе в файл потоком является поток вывода,

представляющий этот файл (подробнее мы вернемся к работе с файлами чуть позже). Вторым аргументом образец является ограниченная с обеих сторон кавычками

управляющая строка (control string), которая может содержать управляющие коды (directive). Они опознаются по знаку ~ (тильда) перед ними. Остальные аргументы формы ставятся в соответствие управляющим кодам строки.

Если управляющие коды и соответствующие им аргументы не используются, то FORMAT выводит строку так же, как функция PRINC, и возвращает в качестве значения NIL.

```
_(format  t  “Это печатается !”)
Это печатается !
NIL                                    ; значение
```

Задав NIL значением параметра ПОТОК, получим в качестве результата функции строку, построенную с помощью управляющих кодов и аргументов. У такой формы есть лишь значение и нет побочного эффекта, связанного с выводом.

Гибкость функции FORMAT основана на использовании управляющих кодов. Они выводят в порядке их появления слева направо каждый очередной отформатированный аргумент или осуществляют какие-нибудь действия, связанные с выводом. Наиболее важными управляющими кодами и их назначением являются:

КОД	НАЗНАЧЕНИЕ
~%	Переводит строку
~S	Выводит функцией PRIN1 значение очередного аргумента
~A	Выводит функцией PRINC значение очередного аргумента
~nT	Начинает вывод с колонки п. Если уже ее достигли, то выводится пробел.
~~	Выводит сам знак тильды.

Приведем пример:

```
_(format  t  “На~%отдельные~%строки~%”)
    На                                ; печать
отдельные                  ; печать
строки                            ; печать
NIL                          ; результат
_(format  nil
      "Ответ - ~S" (+ 2 3))    ; печати нет
"Ответ - 5"                          ; значением является строка
_(setq x 2)
2
_(setq y 3)
3
_(format  t  "~S плюс ~S будет ~S~%"
        x y (+ x y))                                ; аргумент
2 плюс 3 будет 5
NIL
_(format t "Знак тильды: ~~")
Знак тильды: ~
```

```
NIL
_(defun таблица (a-список)
      (format t “~%Свойство~15TЗначение~%”)
      (do ((пары a-список (rest пары)))
           ((null пары) (terpri))
            (format t "~%~A~15T~A"
               (caar пары) (cadar пары))))
ТАБЛИЦА
_(таблица  '((имя "Zippy")
              (кличка "Pinhead")
              (язык английский)))
Свойство      Значение
ИМЯ           Zippy
КЛИЧКА        Pinhead
ЯЗЫК          АНГЛИЙСКИЙ
```

Различные Лисп-системы дополнительно к основным функциям ввода и вывода содержат целый набор различных встроенных функций, макросов иг других средств, при помощи которых можно задавать более сложные виды печати, например шрифт (font).

1.11 ОСНОВЫ РЕКУРСИИ

Лисп - это язык функционального программирования

По одной из классификаций языки программирования делятся на процедурные (procedural), называемые также операторными или императивными (imperative), и декларативные (declarative) языки. Подавляющее большинство используемых в настоящее время языков программирования, например Бейсик, Кобол, Фортран, Паскаль, Си и Ада, относятся к процедурным языкам. Наиболее существенными классами декларативных языков являются функциональные (functional), или аппликативные, и логические (logic programming) языки. К категории функциональных языков относятся, например Лисп, FP, Apl, Nial, Krc и Logo. Самым известным языком логического программирования является Пролог.

На практике языки программирования не являются чисто процедурными, функциональными или логическими, а содержат в себе черты языков различных типов. На процедуоном языке часто можно написать функциональную программу или ее часть и наоборот. Может точнее было бы вместо типа языка говорить о стиле или методе программирования. Естественно различные языки поддерживают разные стили в разной степени.

Процедурное и функциональное программирование

Процедурная программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. Типичными операторами являются операторы присваивания и передачи управления, операторы ввода-вывода и специальные предложения для организации циклов. Из них можно составлять фрагменты программ и подпрограммы. В основе такого программирования

лежат взятие значения какой-то переменной, совершение над ним действия и сохранение нового значения с помощью оператора присваивания, и так до тех пор пока не будет получено (и, возможно, напечатано) желаемое окончательное значение.

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции, которая в свою очередь вызывает функции, входящие в ее определение и т.д. в соответствии с иерархией определений и структурой условных предложений. Функции часто либо прямо, либо опосредованно вызывают сами себя.

Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор пока запустившая вычисления функция не вернет конечный результат пользователю.

"Чистое" функциональное программирование не признает присваиваний и передач управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения. Повторные вычисления осуществляются через рекурсию, являющуюся основным средством функционального программирования.

Рекурсивный - значит использующий самого себя

Многие практические ситуации предполагают рекурсивное или самоповторяющееся поведение, возвращающееся к самому себе. Предположим, что мы, например, пытаемся прочитать текст на другом языке, пользуясь толковым словарем этого языка. Когда в тексте встречается незнакомое слово, то его объяснение ищется в словаре. В объяснении слова могут, в свою очередь, встретится незнакомые слова, которые нужно найти в словаре и т.д. Эту процедуру можно определить с помощью следующих правил:

ВЫЯСНЕНИЕ ЗНАЧЕНИЯ СЛОВА:

1. Найди слово в словаре.2. Прочитай статью, объясняющую значение этогослова.

3. Если объяснение понятно, т.е. статья не содержит непонятных слов, продолжи чтение с последнего прерванного места.

4. Если в объяснении встречается незнакомое слово, то прекрати чтение, запомни место прекращения и выясни значение слова, придерживаясь совокупности правил ВЫЯСНЕНИЕ ЗНАЧЕНИЯ СЛОВА.

Изображенный выше свод правил называют рекурсивным или использующим себя, поскольку в нем содержится ссылка на собственное определение. Речь идет не о порочном круге определений, а об образе действий, который пригоден для использования и вполне выполним.

Рекурсия всегда содержит терминальную ветвь

В рекурсивном описании действий имеет смысл обратить внимание на следующие обстоятельства. Во-первых, процедура содержит всегда по крайней мере одну терминальную ветвь и условие окончания (пункт 3). Во-вторых, когда процедура доходит до рекурсивной ветви (пункт 4), то функционирующий процесс приостанавливается н новый такой же процесс запускается с начала, но уже на новом

уровне. Прерванный процесс каким-нибудь образом запоминается. Он будет ждать и начнет исполняться лишь после окончания нового процесса. В свою очередь, новый процесс может приостановиться, ожидать и т.д.

Так образуется как бы стек прерванных процессов, из которых выполняется лишь последний в настоящий момент времени процесс; после окончания его работы продолжает выполняться предшествовавший ему процесс. Целиком весь процесс выполнен, когда стек снова опустеет, или, другими словами, все прерванные процессы выполнятся.

Рекурсия может проявляться во многих формах

В реальном мире рекурсия проявляется в виде различных форм и связей. Она может быть как в структуре, так и в действиях.

Каждому известно изображение, повторяющееся в двух зеркалах, установленных друг против друга, картина, изображающая картину, телевизор, в котором виден телевизор, и т.д. В математике рекурсия встречается в связи с многими различными аспектами, такими как ряды, повторяющиеся линии, алгоритмы, процедуры определения и доказательства, одним словом, в самых различных структурах.

Рекурсия встречается обычно и в природе: деревья имеют рекурсивное строение (ветки образуются из других веток), реки образуются из впадающих в них рек. Клетки делятся рекурсивно. В растениях это часто видно уже на макроуровне. Например, семенная чешуя шишек и семена некоторых цветов (например, подсолнечника) часто расположены пересекающимися спиралевидными веерами, определяемыми соотношением чисел Фибоначчи.

Продолжение жизни связано с рекурсивным процессом. Молекулы ДНК и вирусы размножаются, копируя себя, живые существа имеют потомство, которое, в свою очередь, тоже имеет потомство и т.д.

Рекурсия распространена и в языке, и в поведении так же, как в способах рассуждения и познания.

Списки строятся рекурсивно

Рекурсия в Лиспе основана на математической теории рекурсивных функций. Рекурсия хорошо подходит для работы со списками, так как сами списки могут состоять из подсписков, т.е. иметь рекурсивное строение. Для обработки рекурсивных структур совершенно естественно использование рекурсивных процедур.

Списки можно определить с помощью следующих правил Бэкуса-Наура:

```
список > NIL                    ; список либо пуст, либо это
список > (голова . хвост)       ; точечная пара, хвост
                                  ; которой является списком
голова > атом                   ; рекурсия "в глубину”
голова > список
хвост > список                  ; рекурсия "в ширину"
```

Рекурсия есть как в определении головы, так и в определении хвоста списка. Заметим, что приведенное выше определение напрямую отражает определения функций,

работающих со списками, которые могут обрабатывать рекурсивным вызовом голову списка, т.е. "в глубину" (в направлении CAR), и хвост списка, т.е. "в ширину" (в направлении CDR).

Списки можно определить и в следующей форме, которая подчеркивает логическую рекурсивность построения списков из атомов и подсписков:

список > NIL
список > (элемент элемент ...)
элемент > атом
элемент > список ; рекурсия

Лисп основан на рекурсивном подходе

В программировании на Лиспе рекурсия используется для организации повторяющихся вычислений. На ней же основано разбиение проблемы и разделение ее на подзадачи, решение которых, насколько это возможно, пытаются свести к уже решенным или в соответствии с основной идеей рекурсии к решаемой в настоящий момент задаче.

На рекурсии основан и часто используемый при решении задач и при поиске (search) механизм возвратов (backtracking), при помощи которого можно вернуться из тупиковой ветви к месту разветвления, аннулировав проделанные вычисления. Рекурсия в Лиспе представляет собой не только организацию вычислений - это образ мыслей и методология решения задач.

Теория рекурсивных функций

Теория рекурсивных функций наряду с алгеброй списков и лямбда-исчислением является еще одной опорой, на которой покоится Лисп. В этой области математики изучаются теоретические вопросы, связанные с вычислимостью (computabi-lity). Под вычислимыми понимаются такие задачи, которые в принципе можно запрограммировать и решить с помощью вычислительной машины. Теория рекурсивных функций предлагает наряду с машиной Тьюринга, лямбда-исчислением и другими теоретичес кими формализмами эквивалентный им формализм алгоритмической вычислимости (effective computabi-lity).В теории рекурсивных функций сами функции (алгоритмы) и их свойства рассматриваются и классифицируются в соответствии с тем, какие функции можно получить и вычислить, используя различные формы рекурсии. Основная идея рекурсивного определения заключается в том, что функцию можно с помощью рекуррентных формул (recurrence formula) свести к некоторым начальным значениям, к ранее определенным функциям или к самой определяемой функции, но с более "простыми" аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям. Например, числа Фибоначчи

0, 1, 1, 2, 3, 5, …

используя префиксную нотацию Лиспа, можно определить с помощью следующих рекуррентных формул:

(fib 0) = 0,
(fib 1) = 1,
(fib n) = (+ (fib (- n 1)) (fib (- n 2)))

Класс функций, получаемых таким образом, называют классом примитивно рекурсивных (primitive recursive) функций.

Существуют также функции, не являющиеся примитивно рекурсивными. В качестве примера можно привести функцию Аккермана, которую можно задать следующими рекуррентными формулами:

(Ack 0 x y) = (+ y x),
(Ack 1 x y) = (* y x),
(Ack 2 x y) = (^ y x),
…
(Ack (+ n 1) x y) = к значениям y x-1 раз
применяется операция
(lambda (u v) (Ack z u v))

Заданная с помощью приведенной выше схемы функция Ack не является примитивно рекурсивной, хотя она и вычислимая, т.е. ее можно определить и на основе этого определения вычислить ее значение за конечное время.

Можно показать, что существуют функции, значения которых можно вычислить с помощью алгоритма, но которые нельзя алгоритмически описать. Вычисление такой функции может быть бесконечным. В качестве примера приведем функцию (f n m), результатом которой является 1 в случае, если в десятичной записи числа n встречается фрагмент из последовательности повторяющихся цифр m длиной n.

Можно показать, что алгоритм вычисления этой функции существует, но нам неизвестно, каков он. Мы можем лишь пытаться вычислять знаки пи в надежде, что искомая последовательность обнаружится, но определить, закончится ли когда-нибудь вычисление, мы не можем. Такие функции называются общерекурсивными (general recursive).

Класс примитивно рекурсивных функций достаточно интересен с точки зрения многих практических проблем. Применение рекурсивных функций в Лиспе не ограничивается лишь численными аргументами, они используются (и даже в первую очередь) для символьных структур, что открывает новые большие возможности по сравнению с численными вычислениями.

1.12 ПРОСТАЯ РЕКУРСИЯ

Функция является рекурсивной, если в ее определении содержится вызов самой этой функции. Мы будем говорить о рекурсии по значению, когда вызов является выражением, определяющим результат функции. Если же в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции, то будем говорить о рекурсии по

аргументам. Аргументом рекурсивного вызова может быть вновь рекурсивный вызов, и таких вызовов может быть много.

Простая рекурсия соответствует циклу

Рассмотрим сначала случай простой рекурсии. Мы будем говорить, что рекурсия простая (simple), если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.

Определим функцию КОПИЯ, которая строит копию списка. Копия списка логически идентична первоначальному списку:

```
_(defun КОПИЯ ( l )
     (cond ((null l) nil)          ;  условие окончания
     (t (cons (car l)              ;  рекурсия
               (КОПИЯ (cdr l))))))
КОПИЯ
_(копия  '(a b c))
   (ABC)                           ;  логическая копия
_(eq '(a b c)  (копия '(a b c)))
   NIL                             ; физически различные списки
```

Эта функция является рекурсивной по аргументу, поскольку рекурсивный вызов стоит на месте аргумента функции CONS. Условное предложение из тела функции содержит две ветви: ветвь с условием окончания и ветвь с рекурсией, с помощью которой функция проходит список, копируя и укорачивая в процессе этого список по направлению CDR.

Мы можем понаблюдать за работой этой функции при помощи содержащихся в интерпретаторе средств трассировки (TRACE). Трассировка функции включается при помощи директивы, которая в отличие от обычных функций не вычисляет свой аргумент.

```
(TRACE   функция)

_(trace копия)            ;  апостроф ставить не нужно
(КОПИЯ)
```

Трассировку можно отменить аналогичной по форме директивой UNTRACE. После ввода директивы TRACE интерпретатор будет распечатывать значения аргументов каждого вызова функции КОПИЯ перед ее вычислением и полученный результат после окончания вычисления каждого вызова. При печати вызовы на различных уровнях (levels) рекурсии отличаются при помощи отступа. Приведем пример:

```
_(копия  '(a b))
КОПИЯ:                    ;  аргументы вызова
L = (A B)                 ; 1 уровня
```

```
     КОПИЯ:                ; аргументы вызова
   L = (B)                 ; 2 уровня
        КОПИЯ:             ; аргументы вызова
            L = NIL            ; 3 уровня
        КОПИЯ = NIL        ; значение уровня 3
   КОПИЯ = (B)             ;  значение уровня 2
КОПИЯ = (A B)              ;  значение уровня 1
  (A B)                        ;  окончательное  значение
```

Функция вызывается рекурсивно с хвостом списка L в качестве аргумента до тех пор, пока условие (NULL L) не станет истинным и значением функции не станет NIL, который выступает вторым аргументом последнего вызова функции CONS. После этого рекурсивные вызовы будут по очереди заканчиваться и возвращаться на предыдущие уровни, и с помощью функции CONS в рекурсивной ветке определения начнет формироваться от конца к началу новый список. На каждом уровне к возвращаемому с предыдущего уровня хвосту (CDR L) добавляется головная часть с текущего уровня (CAR L).

Обратите внимание, что функция копирует не элементы списка (т.е. в направлении CAR, или в глубину), а лишь составляющие список ячейки верхнего уровня (т.е. список копируется в направлении CDR, или в ширину). Позднее мы вернемся к случаю, когда спнсок копируется и по направлению CAR на любую глубину. Для этого в определение нужно добавить лишь еще один рекурсивный вызов.

Копирование списков представляет собой одно из основных действий над списками, и поэтому соответствующая функция входит в число встроенных функций практически во всех Лисп-системах. И даже если ее нет, то ее очень просто можно определить, как мы это только что сделали. В Коммой Лиспе эта функция называется COPY-LIST.

Мы не использовали это имя, а дали функции свое имя, чтобы напрасно не менять определение встроенной функции. Хотя программист в Лиспе может свободно переопределить по своему усмотрению любую функцию, экспериментировать лучше со своими именами, чтобы потом, когда вашим неверным или неполным определением будет вызвана ошибка, не удивляться, почему система не работает так, как надо.

В дальнейшем мы в целях обучения часто будем программировать функции из числа встроенных функций Коммон Лиспа. Будем систематически использовать их стандартные имена, добавив к ним номер варианта, как мы и поступали ранее, определяя различными способами экспоненциальную функцию EXPT (EXPT1, EXPT2 и другие варианты). Так мы одновременно познакомимся с основными функциями и их именами, принятыми в Коммон Лиспе.

Чтобы облегчить восприятие, мы воспользуемся таким типографским приемом, как запись ПРОПИСНЫМИ или строчными буквами. Рекурсивный вызов функции будет выделяться путем написания ее имени прописными буквами.

MEMBER проверяет, принадлежит ли элемент списку

Рекурсию можно использовать для определения как предикатов, так и функций. В качестве второго примера простой рекурсии возьмем встроенный предикат Лиспа

MEMBER. С его помощью можно проверить, принадлежит ли некоторый элемент данному списку или нет. См. пример на следующей странице.

Тело функции состоит из условного предложения, содержащего три ветви. Они участвуют в процессе вычислений в зависимости от возникающей ситуации:

1. ((null l) nil):

```
_(defun MEMBER1 (a l)
    (cond ((null l) nil)                ; l пуст ?
          ((eql (car l)  a)  l)         ;  a найден?
          (t (MEMBER1 a                 ;  a – в хвосте?
                      (cdr  l )))))
MEMBER
_(member1  'b  '(a b c d))
(B C D)
_(member1 'e  '(a b c d))
NIL
_(member1 '(b c)  '(a (b c) d)          ; сравнение предикатом
NIL                                     ;  EQL
```

Аргумент - пустой список либо с самого начала, либо потому, что просмотр списка окончен.

2. ((eql (car l) a) l):

Первым элементом является искомый элемент. В качестве результата возвращается список, в котором A - первый элемент.

3. (t (MEMBER1 a (cdr l))):

Ни одно из предыдущих утверждений не верно: в таком случае либо элемент содержится в хвосте списка, либо вовсе не входит в список. Эта задача аналогична первоначальной, только она меньше по объему. Поэтому мы можем для ее решения использовать тот же механизм, или, другими словами, применить сам предикат к хвосту списка.

Если список L пуст либо A в него не входит, то функция возвращает значение NIL В противном случае она возвращает в качестве своего значения ту часть списка, в которой искомое A является первым элементом. Это отличное от NIL выражение соответствует логическому значению "истина".

Каждый шаг рекурсии упрощает задачу

В определении предиката MEMBER1 первоначальная задача разбита на три подзадачи. Первые две из них сводятся к простым условиям окончания. Третья решает такую же задачу, но на шаг более короткую. Ее решение можно рекурсивно свести к функции, решающей первоначальную задачу.

Понаблюдаем с помощью трассировки за вычислением предиката MEMBER1:

```
_(trace member1)
(MEMBER1)
```

```
_(memberl  'c  '(a b c d))
MEMBER1:                          ;   вызов уровня 1
A = C
L =  (A B C D)
      MEMBER1:              ;   вызов уровня 2
      A = C
      L = (B C D)
            MEMBER1:        ;  вызов уровня 3
            A = C
            L = (C D)
            MEMBER1 = (C D)   ;  значение уровня 3
           MEMBER1 = (C D)        ;   значение уровня 2
    MEMBER1 = (C D)               ;  значение уровня 1
    (C D)                         ;  значение формы
```

На первых двух уровнях рекурсии вычисления осуществляются по третьей, рекурсивной ветви. В рекурсивном вызове первым аргументом является C, так как искомый элемент на каждом шаге один и тот же. Вторым аргументом берется хвост списка текущего уровня (CDR L). На третьем уровне значением предиката (EQL (CAR L) A) становится T, поэтому на этом уровне значением всего вызова станет значение соответствующего результирующего выражения L=(C D). Это значение возвращается на предыдущий уровень, где оно будет значением вызова MEMBER1 в рекурсивной ветви и, таким образом, станет значением всего вызова на втором уровне. Далее это значение возвращается далее на уровень и, в конце концов, выводится пользователю.

В процессе спуска по ходу рекурсии на более низкие уровни значение параметра A не меняется, в то время как значение параметра L меняется при переходе на следующий уровень. Значения предыдущего уровня сохраняются, поскольку связи переменных ассоциируются с уровнем. Значения предыдущих уровней скрыты до тех пор, пока на них не вернется управление, после этого старые связи вновь становятся активными. В приведенном примере после возврата на предыдущие уровни эти связи не используются. Обратите внимание, что связи параметра A на различных уровнях физически различны, хотя значение остается тем же самым.

Порядок следования ветвей в условном предложении существенней

В рекурсивном определении существенней порядок следования условий. Их нужно располагать так, чтобы нужная ветвь была выбрана после отбрасывания условия или ряда условий, а чаще в результате выполнения некоторого условия. Последовательность условий может также повлиять на эффективность вычислений. Правильную последовательность можно получить путем логического рассуждения на основе учета возможных ситуаций. Порядок следования может влиять и на выбор предикатов, например использовать ли сам предикат или его отрицание и т.п.

При определении порядка следования условий основным моментом является то, что сначала проверяются всевозможные условия окончания, а затем ситуации, требующие продолжения вычислений. При помощи условий окончания последовательно проверяются все особые случаи и программируются соответствующие результирующие

выражения. На исключенные таким образом случаи можно в последующем не обращать внимания.

Например, в рассмотренном ранее предикате MEMBER1 первое условие (NULL L) будет истиной, когда аргумент L с самого начала пуст или когда первоначально непустой список в процессе рекурсивных вызовов пройден до конца:

```
_(member1 'x  '( ))        ;  L первоначально пуст
NIL
_(memberl  'x '(a b c d))
   NIL                          ;  L пройден до конца
```

Вычисления в обоих случаях оканчиваются на условии (NULL). Обратите внимание, что в приведенных ниже вызовах вычисления останавливаются не по этому условию:

```
_(member1 nil '(a b nil d))
(NIL D)
_(member1 nil '(a b c nil))
(NIL)
```

Рассмотрим в качестве полезного урока вариант предиката MEMBER, в котором условия перечислены в неверном порядке. Для того чтобы была виднее неправильная работа предиката, определим его в отличие от Коммон Лиспа так, чтобы в случае, если элемент найден, он возвращал не хвост списка, а значение T.

```
_(defun MEMBER2 (a l)
        (cond ((eql a (car l ))  t)              ;  сначала EQL
                  ((null l) nil)                    ;  затем NULL
               (t (MEMBER2 a (cdr l)))))
MEMBER2
_(member2  'd  '(a b c d))
                                          ;  D является элементом списка
   T                                          ; результат верен
_(member2 nil  '(nil))
                              ;  NIL является элементом (NIL)
    T                            ;  результат верен
_(member2 nil  nil)
                              ;  NIL не является элементом
    T                            ;  пустого списка.  Ошибка!
```

Ошибочный результат является следствием того, что в момент применения предиката EQL неизвестно, завершился ли список или проверяется список (NIL . хвост). Причиной возникновения ошибки служит принятое в Лисп-системе соглашение, что головой пустого списка является NIL, а это приводит к тому, что как (CAR NIL), так и (CAR '(NIL. хвост)) возвращают в качестве результата NIL. В "чистом" Лиспе значение

формы (CAR NIL) не определено, поскольку NIL - атом. Если бы функция MEMBER2 была определена так, как это сделано у нас, то вызов функции выдавал бы ошибочный результат во всех случаях, когда искомый элемент не входит в список. Если бы ветви EQL и NULL были расположены в том же порядке, как в MEMBER1, то функция работала бы правильно, поскольку ранее проверяемое условие (NULL L) исключало бы возможность применения CAR к пустому списку.

Ошибка в условиях может привести к бесконечным вычислениям

Отсутствие проверки, ошибочное условие или неверный их порядок могут привести к бесконечной рекурсии. Это произошло бы, например, если в предикате MEMBER значение аргумента L не укорачивалось на каждом шагу рекурсии формой (CDR L). То же самое случилось бы, если рекурсивная ветвь находилась в условном предложении перед условием окончания. Поэтому существенно, чтобы каждый шаг рекурсии приближал вычисления к условию окончания. На практике рекурсивные вычисления не могут оказаться бесконечными, поскольку каждый рекурсивный вызов требует некоторого количества памяти (если только интерпретатор или транслятор не преобразовал рекурсию в цикл), а общий объем памяти ограничен. При простой рекурсии память заполняется быстро, но в более сложных случаях вычисления могут оказаться практически бесконечными, другими словами, до исчерпания памяти будет бесполезно истрачено много машинного времени.

Замечание: Предикат MEMBER в Коммон Лиспе определен в более общем виде, чем мы его представили. Ему можно вместо предиката сравнения EQL, являющегося умолчанием, задать при ызове с помощью ключевого параметра :TEST другой предикат. Например, в следующем примере для сравнения элементов используется предикат EQUAL, который применим и к спискам:

```
_(member  '(c d)   '((a b)  (c d))
        :test 'equal)
((C D))
```

Во многих системах (например, в Маклиспе) в MEMBER по умолчанию используется сравнение при помощи EQUAL.

APPEND объединяет два списка

Рассмотрим встроенную функцию Лиспа APPEND, объединяющую два списка в один новый список. Функция APPEND, подобно функции COPY-LIST (КОПИЯ), строит новый список из значений, сохраняемых на различных уровнях рекурсии:

```
_(defun APPEND1 (x y)
      (cond ((null x) y)
            (t (cons (car x)
                     (APPEND1 (cdr x) y)))))
APPEND1
```

Приведем пример:

```
_(append1 '(с  л  и)  '(я  н  и  е))
(С Л И Я Н И Е)
_(append1 '(a b)  nil)
(A B)
_(append1  '(a  b nil) '(nil))
(A B NIL NIL)
```

Идея работы функции состоит в том, что рекурсивно откладываются вызовы функции CONS с элементами списка X до тех пор, пока он не исчерпается, после чего в качестве результата возвращается указатель на список Y и отложенные вызовы, завершая свою работу, формируют результат:

```
_(trace APPEND1)
(APPEND1)
_(append1 '(a  b)  '(c  d))
APPEND1:
X = (A  B)
Y = (C  D)
     APPEND1:
     X = (B)
     Y = (C D)
          APPEND1:
          X = NIL
          Y = (C D)
          APPEND1 = (C D)
     APPEND1 = (BCD)
APPEND1 = (A B C D)
(A B C D)
```

Обратите внимание, что результирующий список строится от конца первого списка к началу, поскольку вычислявшиеся в процессе рекурсии вызовы функции CONS начинают вычисляться из глубины наружу по мере того, как осуществляется процесс возврата из рекурсии.

В функции APPEND1 мы использовали рекурсию по аргументам. APPEND можно было бы определить и в форме рекурсии по значению:

```
_(defun APPEND2 (x y)
      (cond ((null x) y)
            (t (APPEND2 (cdr x)
                        (cons (car x) y)))))
APPEND2
```

Это определение отличается тем, что результат теперь строится непосредственно во втором аргументе, а не где-то на стороне, как это осуществлялось в предыдущем определении APPEND. Трассировку вычислений пусть читатель попробует сам.

Как видно из предыдущих определений, APPEND копирует список, являющийся первым аргументом. Эту функцию часто так и используют в виде (APPEND список NIL), когда необходимо сделать копию верхнего уровня списка. Но все-таки лучше использовать форму (COPY-LIST список).

Во многих Лисп-системах функция APPEND может иметь переменное число параметров. Такую функцию APPEND можно определить через двуместную функцию APPEND1 следующим образом:

```
_(defun APPEND3 (&rest  args)
     (cond ((null args) nil)
            (t (append1 (car  args)
                    (append3 (cdr args))))))
APPEND3
_(append3  '(a b)  '(c d)  '(e f))
(A B C D E F)
```

Последним аргументом функции APPEND в Коммон Лиспе не обязательно должен быть список. В этом случае результатом будет точечное выражение.

REMOVE удаляет элемент из списка

Все предыдущие определения функций содержали лишь один рекурсивный вызов. Рассмотрим в качестве следующего примера содержащую две рекурсивные ветви встроенную функцию REMOVE, которая удаляет из списка все совпадающие с данным атомом (EQL) элементы и возвращает в качестве значения список из всех оставшихся элементов. REMOVE можно определить через базисные функции и ее саму следующим образом:

```
_(defun REMOVE1 (a l)
      (cond ((null l )  nil)
            ((eql a (car l ))
                  (REMOVE1 a                  ; убрали элемент a
                              (cdr l)))
            (t (cons (car l)                  ;a  в CAR не было
                    (REMOVE1 a (cdr l))))))
REMOVE1
_(remove1 'л   '(с л о н))
(С О Н)
_(remove1  'b  '(a (b c)))      ;  элементы
   (A (B C ))                       ;  проверяются лишь в
                                ;    направлении CDR
_(remove1  '(a b)   '((a b)  (c d)))
((A B)  (C D))                  ;  сравнение EQL
```

Список L сокращается путем удаления всех идентичных A в смысле EQL элементов (вторая ветвь) и копирования в результирующий список (CONS) остальных элементов (третья ветвь) до тех пор, пока условие окончания (NULL) не станет истинным. Результат формируется в процессе возврата аналогично функции APPEND1.

Замечание: Во многих Лисп-системах функция REMOVE определена с использованием предиката EQUAL вместо EQL, с помощью такой функции можно удалять элементы, являющиеся списками. В Коммой Лиспе предикат сравнения для функции REMOVE можно задать непосредственно при ее вызове ключевым параметром :TEST таким же образом, как и для функции MEMBER. Приведем пример:

```
_(remove  '(a b)  '((a b)  (c d))
        :test 'equal)
((C D))
```

SUBSTITUTE заменяет все вхождения элемента

Функция SUBSTITUTE, заменяющая все вхождения данного элемента СТАРЫЙ в списке L на элемент НОВЫЙ, работает подобно функции REMOVE.

Обратите внимание, что и здесь замена производится лишь на самом верхнем уровне списка L, т.е.

```
_(defun SUBSTITUTE1 (новый  старый  l)
     (cond
        ((null l) nil)
        ((eql  старый (car l ))
           (cons   новый                        ;  замена головы
               (SUBSTITUTE1  новый
                                 старый         ;  обработка хвоста
                                  (cdr l ))))
       (t (cons (car l)                         ;  голова не меняется
              (SUBSTITUTE1  новый
                                 старый         ;  обработка хвоста
                                 (cdr 1))))))
SUBSTITUTE1
_(substitute1  'b  'x  '(a x x a))
(A B B A)
```

рекурсия осуществляется только по хвостовой части списка (в направлении CDR). Как и при копировании списка процедуру замены элемента можно обобщить так, чтобы список обрабатывался и в глубину, для этого в случае, когда голова списка является списком, нужно осуществить рекурсию и по головной части. К такому виду рекурсии мы вернемся позднее.

Замечание: Функции SUBSTITUTE в Коммой Лиспе можно через ключевой параметр :TEST передать предикат сравнения таким же образом, как и функции REMOVE. Мы сейчас не будем останавливаться на том, что эти функции, как это станет

очевидным при рассмотрении типов данных, в Коммон Лиспе заданы в значительно более общем виде.

REVERSE обращает список

В приведенных примерах мы просматривали список в соответствии с направлением указателей в списочных ячейках слева направо. Но что делать, если нужно обрабатывать список справа налево, т.е. от конца к началу?

Рассмотрим для примера функцию REVERSE, которая также является встроенной функцией Лиспа. REVERSE изменяет порядок элементов в списке (на верхнем уровне) на обратный.

Для обращения списка мы должны добраться до его последнего элемента и поставить его первым элементом

```
_(defun REVERSE1 (l)
     (cond ((null l ) nil)
              (t (append (REVERSE1 (cdr l ))
                     (cons (car l ) nil )))))
REVERSE1
_(reverse1  '(a b c))
(C B A)
_(reverse1 '((A B)  (C D)))          ;  обращается
   ((C D)  (A B))                          ;  лишь верхний уровень
```

обращенного списка. Хотя нам непосредственно конец списка не доступен, можно, используя APPEND, описать необходимые действия. Идея определения состоит в следующем: берем первый элемент списка (CAR L), делаем из него с помощью вызова (CONS (CAR L) NIL) одноэлементный список и объединяем его функцией APPEND с перевернутым хвостом. Хвост списка сначала обращается рекурсивным вызовом (REVERSEl (CDR L)). Попробуем проследить, как происходит такое обращение:

```
_(trace  reverse1)
(REVERSE1)
_(reverse1  '(a b c))
REVERSE1:
L = (A B C)
     REVERSE1:
     L = (B C)
          REVERSE1:
          L = (C)
               REVERSE1:
               L = NIL
               REVERSE1 = NIL
          REVERSE1 = (C)
     REVERSE1 = (C B)
REVERSE1 = (C B A)
```

(C B A)

Добраться до последнего элемента списка можно, лишъ пройдя всю образующую список цепочку слева направо. В функции REVERSE1 список проходится до конца и по пути подходящие элементы списка откладываются в аргументы незавершенных вызовов. Построение обращенного списка в порядке, противоположном следованию элементов исходного списка может начатся лишь после завершения рекурсии. Результат будет сформирован, когда исчерпается стек рекурсивных вызовов.

Использование вспомогательных параметров

Список является несимметричной структурой данных, которая просто проходится слева направо. Во многих случаях для решения задачи более естественны вычисления, производимые справа налево. Например, то же переворачивание списка было бы гораздо проще осуществить, если бы был возможен непосредственный доступ к последнему элементу списка. Такое противоречие между структурой данных и процессом решения задачи приводит к трудностям программирования и может служить причиной неэффективности.

В процедурных языках программирования существует возможность использования вспомогательных переменных, в которых можно сохранять промежуточные результаты. Например, обращение списка можно осуществить простым переносом элементов списка друг за другом из списка L в результирующий список, используя функцию CONS:

```
_(defun reverse2 (l)
     (do ((остаток l (cdr  остаток))
          (результат  nil
               (cons (car  остаток)
                     результат)))
         ((null остаток)  результат)))
REVERSE2
```

В функциональном программировании переменные таким образом не используются. Но соответствующий механизм можно легко осуществить, используя вспомогательную функцию, у которой нужные вспомогательные переменные являются параметрами. Тогда для функции REVERSE мы получим такое определение:

```
_(defun reverse3 (l)
     (ПЕРЕНОС  l  nil))
REVERSE3
_(defun ПЕРЕНОС  (l  результат)
         (cond ((null l)  результат)
               (t (ПЕРЕНОС  (cdr  l)
                            (cons (car  l)
                             результат)))))
ПЕРЕНОС
```

Вспомогательная функция ПЕРЕНОС рекурсивна по значению, так как результирующим выражением ее тела является либо непосредственно рекурсивный вызов, либо готовое значение. С помощью этой функции элементы переносятся таким образом, что на каждом шаге рекурсии очередной элемент переходит из аргумента L в аргумент РЕЗУЛЬТАТ. Обращенный список строится элемент за элементом функцией CONS в аргументе РЕЗУЛЬТАТ так же, как и в итеративном варианте. Вычисления производятся по списку L слева направо и соответствуют итеративным вычислениям.

1.13 ДРУГИЕ ФОРМЫ РЕКУРСИИ

В определении функции (или чаще в определениях вызывающих друг друга функций) рекурсия может принимать различные формы. Ранее мы рассмотрели простую рекурсию, когда одиночный рекурсивный вызов функции встречается в одной или нескольких ветвях.

В этой главе мы продолжим изучение различных форм рекурсии, в том числе:

1) параллельную рекурсию, когда тело определения функции f содержит вызов некоторой функции g, несколько аргументов которой являются рекурсивными вызовами функции f :

```
(defun f ...
     ... ( g ... (f ... ) ... ( f ... ) ... )
     …)
```

2) взаимную рекурсию, когда в определении функции f вызывается некоторая функция g , которая в свою очередь содержит вызов функции f:

```
(defun f ...
     ... ( g ... ) ... )
    (defun g ...
          ... ( f ... ) ... )
```

3) рекурсию более высокого порядка, когда аргументом рекурсивного вызова является рекурсивный вызов:

```
(defun f ...
     ... ( f ... ( f ...) ... )
     … )
```

Параллельное ветвление рекурсии

Рекурсию называют параллельной, если она встречается одновременно в нескольких аргументах функции. Так выглядят повторяющиеся вычисления, соответствующие следующим друг за другом (текстуально) циклам в операторном программировании.

Рассмотрим в качестве примера копирование списочной структуры на всех уровнях. Ранее уже был представлен пример функции, копирующей список в направлении CDR на верхнем уровне: функция COPY-LIST (КОПИЯ) строила копию верхнего уровня списка. На возможные подсписки не обращалось внимания, и они не

копировались, а брались как и атомы в том виде, как они есть (т.е. как указатель). Если нужно скопировать список целиком как в направлении CDR, так и в направлении CAR, то рекурсия должна распространяться и на подсписки. Таким образом, мы получим обобщение функции COPY-LIST Коммон Лиспа COPY-TREE. Слово TREE (дерево) в названии функции возникло в связи с тем, что в определении функции список трактуется как соответствующее точечной паре бинарное дерево (binary tree), у которого левое поддерево соответствует голове списка, а правое поддерево - хвосту:

```
дерево > NIL                          ; пустое дерево
дерево > атом                     ; лист дерева
дерево > (дерево . дерево)        ; точечная пара – дерево

_(defun COPY-TREE1 (l)
     (cond ((null l ) nil)
           ((atom l ) l )
           (t (cons
                 (COPY-TREE1          ; копия головы
                       (car l ))
                 (COPY-TREE1          ; копия хвоста
                       (cdr l ))))))
COPY-TREE1
```

Функция COPY-TREE отличиается от COPY-LIST тем, что рекурсия применяется как к голове, так и к хвосту списка. Отличие состоит также в использовании условия (ATOM L) для определения атомарного подвыражения (листа дерева). Поскольку рекурсивные вызовы представляют собой два аргумента вызова одной функции (CONS), то мы имеем дело с параллельной рекурсией. Заметим, что параллельность является лишь текстуальной или логической, но никак не временной, так как вычисление ветвей естественно производится последовательно.

Таким же образом рекурсией по голове и хвосту списка можно проверить логическую идентичность двух списков на основе сравнения структуры и атомов, составляющих список. Определим далее уже известный нам предикат EQUAL через применяемый к символам предикат EQ: (Ограничим наше рассмотрение лишь символами и списками, состоящими из списков.)

```
(defun EQUAL1 (x y)
   (cond  ((null x )(null y))
          ((atom x)
              (cond ((atom y)  (eq x y))
                    (t nil)))
          ((atom y) nil)
          (t (and (EQUAL1 (car x)       ;  идентичные
                     (car y)            ;  головы
                  (EQUAL1 (cdr x)   ;  идентичные
                       (cdr y))))))     ;  хвосты
```

Мы использовали предикат EQ, исходя из первоначального ограничения, т.е. исходя из предположения, что EQ определен лишь на атомарных аргументах. Практически вторую и третью ветви условного предложения можно было бы объединить более короткой ветвью ((EQ X Y) T).

Приведем еще один пример параллельной рекурсии. Рассмотрим функцию ОБРАЩЕНИЕ. Эта функция предназначена для обращения порядка следования элементов списка и его подсписков независимо от их места и глубины вложенности. Ранее определенная нами простой рекурсией функция REVERSE оборачивала лишь верхний уровень списка:

```
_(defun ОБРАЩЕНИЕ (l)
     (cond
          ((atom l )  l )
          ((null (cdr l ))
              (cons (ОБРАЩЕНИЕ (car l ))  nil))
          (t (append (ОБРАЩЕНИЕ (cdr l ))
                 (ОБРАЩЕНИЕ (cons (car l
                                          nil ))))))
ОБРАЩЕНИЕ
_(ОБРАЩЕНИЕ  '(a  (b (c  (d)))))
((((D)  C)  B)  A)
_(setq палиндром
            '(( A
               (р о з а)
               (у п а л а)
               (н а)
               (л а п у)
               (А з о р а)))
((А)   (Р 0 3 А)   ...)
_(обращение  палиндром)
((А Р 0 3 А)   (У П А Л)   (А Н)   (А Л А П У) (А  З О Р)   (А))
```

Функция ОБРАЩЕНИЕ обращает голову списка, формирует из него список и присоединяет к нему спереди обращенный хвост.

Применяя параллельную рекурсию, можно списочную структуру (двоичного дерева) ужать в одноуровневый список, т.е. удалить все вложенные скобки. Сделать это при помощи структурных преобразований довольно сложно, но с помощью рекурсии это осуществляется просто:

```
_(defun В-ОДИН-УРОВЕНЬ (l)
     (cond
          ((null l ) nil)
          ((atom l) (cons (car l) nil))
          (t (append
            (В-ОДИН-УРОВЕНЬ                    ; сначала голову
```

```
                (car l ))
            (В-ОДИН-УРОВЕНЬ                    ; потом хвост
                (cdr l ))))))
В-ОДИН-УРОВЕНЬ
_(в-один-уровень '(a (((((b)))) (c d) e)))
(A B C D E)
_(equal (в-один-уровень  палиндром)
     (в-один-уровень (обращение палиндром)))
    T
```

Функция В-ОДИН-УРОВЕНЬ объединяет (функцией APPEND) ужатую в один уровень голову списка и ужатый хвост. Если голова списка является атомом, то из него формируется список, поскольку аргументы функции APPEND должны быть списками.

Взаимная рекурсия

Рекурсия является взаимной (mutual) между двумя или более функциями, если они вызывают друг друга. Для примера можно представить ранее определенную нами функцию обращения или зеркального отражения в виде двух взаимно рекурсивных функций следующим образом:

```
(defun ОБРАЩЕНИЕ (l )
     (cond ((atom l ) l )
              (t (ПЕРЕСТАВЬ l  nil))))
(defun ПЕРЕСТАВЬ ( l  результат)
    (cond ((null l )  результат)
            (t (ПЕРЕСТАВЬ (cdr l )
                    (cons (ОБРАЩЕНИЕ (car l ))
                            результат)))))
```

Функция ПЕРЕСТАВЬ используется в качестве вспомогательной функции с дополнительными параметрами таким же образом, как и ранее вспомогательная функция ПЕРЕНОС использовалась совместно с функцией REVERSES. В процессе построения обращенного списка она заботится и о том, чтобы возможные подсписки были обращены. Она делает это не сама, а передает эту работу специализирующейся на этом функции ОБРАЩЕНИЕ.

Результат получен взаимной рекурсией. Глубина и структура рекурсии зависят от строения списка L. Кроме участия во взаимной рекурсии функция ПЕРЕСТАВЬ рекурсивна и сама о себе.

Программирование вложенных циклов

Соответствующие елешенньш (nested) циклам процедурного программирования многократные повторения в функциональном программировании осуществляются обычно с помощью двух и более функций, каждая из которых соответствует простому циклу. Вызов такой рекурсивной функции используется в определении другой функции в качестве аргумента ее рекурсивного вызова. Естественно, аргументом рекурсивного вызова в определении функции может быть другой рекурсивный вызов. Зто уже будет рекурсией более высокого порядка.

Рассмотрим сначала программирование вложенных циклов с помощью двух различных функций, а затем уже с помощью рекурсии более высокого порядка. Вложенный цикл можно выразить и с помощью циклических предложений (DO, LOOP и др.) или спецкализи рованных повторяющих функций (функции MAP, например). Такие явные способы мы сейчас использовать не будем.

Мы начнем рассматривать программирование вложенных циклов с сортировки списков. Определим сначала функцию ВСТАВЬ, которая добавляет элемент A в упорядоченный список L так, чтобы сохранилась упорядоченность, если порядок любых двух элементов задается предикатом РАНЬШЕ-Р:

```
_(defun ВСТАВЬ (a  l  порядок)
     (cond ((null l ) (list a))
            ((раньше-р a  (car l )  порядок)
                (cons a l ))
            (t (cons (car l )
                  (ВСТАВЬ  a  (cdr l )
                                 порядок)))))
ВСТАВЬ
```

Предикат РАНЬШЕ-Р проверяет, находится ли элемент A ранее элемента B, в соответствии с расположением определенным порядком следования элементов в списке ПОРЯДОК:

```
_(defun РАНЬШЕ-Р (a b порядок)
     (cond
          ((null порядок)  nil)
          ((eq  a (car порядок))  t)      ;  A раньше
          ((eq  b (car порядок))  nil)
                                          ;  B раньше
         (t  ( РАНЬШЕ-Р a   b (cdr  порядок)))))
РАНЬШЕ-Р
_(раньше-р  'b  'e  '(a  b  c  d  e))
T
_(вставь  'b  '(a  c  d)  '(a  b  c  d  e))
(A B C D)
```

ВСТАВЬ и РАНЬШЕ-Р образуют двухуровневую вложенную итеративную структуру.

Неупорядоченный список можно отсортировать функцией СОРТИРУЙ; которая рекурсивно ставит первый элемент списка на соответствующее место в предварительно упорядоченном хвосте списка:

```
_(defun СОРТИРУЙ ( l  порядок)
     (cond ((null l )  nil)
            (t (вставь (car l )
                  (СОРТИРУЙ (cdr  l)  порядок)
```

```
                              порядок))))
СОРТИРУЙ
_(СОРТИРУЙ '(b a c) '(a b c d e))
(A B C)
```

Теперь рекурсия функций СОРТИРУЙ, ВСТАВЬ и РАНЬШЕ-Р образует уже трехуровневую вложенную повторяющуюся структуру.

Приведем еще функцию РАССТАВЬ, напоминающую своей структурой функцию СОРТИРУЙ и позволяющую элементы списка НОВЫЙ вставить в упорядоченный список L. Функция РАССТАВЬ повторяет процедуру вставки для каждого элемента списка НОВЫЙ:

```
_(defun РАССТАВЬ (новый l порядок)
     (cond
          ((null новый) l )
          (t (вставь (car новый)
                 (РАССТАВЬ (cdr новый) l
                       порядок)
                   порядок))))
РАССТАВЬ
_(расставь '(b c) '(a b d) '(a b c d e))
(A B B C D)
```

Эту рекурсивную по аргументам функцию можно записать и в виде функции, рекурсивной по значению:

```
(defun РАССТАВЬ (новый l порядок)
   (cond
       ((null новый) l )
       (t (РАССТАВЬ (cdr новый)
                (вставь (car новый) l порядок) порядок))))
```

Рекурсия более высокого порядка

Выразительные возможности рекурсии уже видны из приведенных выше содержательных и занимающих мало места определений. Определения функций В-ОДИН-УРОВЕНЬ и ОБРАЩЕНИЕ в итеративном варианте не поместились бы на один лист бумаги (попробуйте!). С помощью рекурсии легко работать с динамическими, заранее не определенными целиком, но достаточно регулярными структурами, такими как списки произвольной длины и глубины вложения.

Используя все более мощные виды рекурсии, можно записывать относительно лаконичными средствами и более сложные вычисления. Одновременно с этим, поскольку определения довольно абстрактны, растет сложность программирования и понимания программы.

Рассмотрим теперь программирование вложенных циклов в такой форме, при которой в определении функции рекурсивный вызов является аргументом вызова этой же самой функции. В такой рекурсии можно выделить различные порядки (order) в соответствии с тем, на каком уровне рекурсии находится вызов. Такую форму рекурсии

будем называть рекурсией более высокого порядка. Функции, которые мы до сих пор определяли, были функциями с рекурсией нулевого порядка.

В качестве классического примера рекурсии более высокого порядка часто приводится известная из теории рекурсивных функций функция Аккермана, пользующаяся славой "плохой" функции:

```
_(defun АККЕРМАН (m  n)
     (cond   (( = m  0) (+  n  1 ))
             (( =  n  0) (АККЕРМАН (-  m 1 ) 1 ))
             (t (АККЕРМАН  (- m 1)
                    (АККЕРМАН  m  (- n 1))))))
АККЕРМАН
_(аккерман  2 2 )
7
_(аккерман  3 2 )
27
```

Функция Аккермана является функцией с рекурсией первого порядка. Ее вычисление довольно сложно, и время вычисления растет лавинообразно уже при малых значениях аргумента.

В качестве другого примера функции с рекурсией первого порядка приведем функцию В-ОДИН-УРО БЕНЬ, располагающую элементы списка на одном уровне, которую мы ранее определили, используя параллельную рекурсию:

```
_(defun в-один-уровекь ( l )
            (уравнять  l  nil))
В-ОДИН-УРОВЕНЬ
_(defun  ВЫРОВНЯТЬ ( l  результат)
      (cond ((null  l )  результат)
            ((atom l ) (cons  l  результат))
            (t  (ВЫРОВНЯТЬ (car  l )
                    (ВЫРОВНЯТЬ (cdr  l )
                                  результат)))))
ВЫРОВНЯТЬ
```

В этом определении работа функции непосредственно сведена к базовым функциям и рекурсии первого порядка, где аргументом рекурсивного вызова является один рекурсивный вызов. В более раннем определении дополнительно к базовым функциям и рекурсии нулевого порядка мы использовали функцию APPEND. Применяя рекурсию более высокого порядка вычисления можно представить более абстрактно и с помощью более короткого определения, однако представить себе работу такой функции довольно сложно.

Функция ВЫРОВНЯТЬ работает следующим образом. Результат строится в списке РЕЗУЛЬТАТ. Если L - атом, то его можно непосредственно добавить в начало списка РЕЗУЛЬТАТ. Если L - список и его первый элемент является атомом, то все сводится к предыдущему состоянию на следующем уровне рекурсии, но в такой ситуации, когда список РЕЗУЛЬТАТ содержит уже вытянутый в один уровень оставшийся хвост.

В том случае, когда и голова списка L является списком, то его сначала приводят к одному уровню. Это делается с помощью рекурсивных вызов, погружающихся з головную ветвь до тех пор, пока там не встретится атом, который можно добавить в начало вытянутой к этому моменту в один уровень структуры. Встречающиеся таким образом атомы добавляются по одному к вытянутому хвосту. На каждом уровне при исчерпании списка на предыдущий уровень выдается набранный к данному моменту РЕЗУЛЬТАТ.

Следующее определение функции REVERSE, использующей лишь базовые функции и рекурсию, является примером еще более глубокого уровня рекурсии:

```
_(defun REV ( l )
     (cond
        ((null l ) l )
        ((null (cdr l ))  l )
        (t (cons
              (car (REV (cdr l )))
              (REV (cons (car l )
                        (REV  (cdr  (REV (cdr l )
                                    ))))))
```

В определении использована рекурсия второго порядка. Вычисления, представленные этим определением, понять труднее, чем прежние. Сложная рекурсия усложняет и вычисления. В этом случае невозможно вновь воспользоваться полученными ранее результатами, поэтому одни и те же результаты приходится вычислять снова и снова. Обращение списка из пяти элементов функцией REV требует 149 вызовов. Десятиэлементный список требует уже 74 409 вызовов и заметное время для вычисления! Как правило, многократных вычислений можно избежать, разбив определение на несколько частей и используя подходя щие параметры для сохранения и передачи промежуточных результатов.

Обычно в практическом программировании формы рекурсии более высокого порядка не используются, но у них по крайней мере есть свое теоретическое и методологическое значение.

Часть 2

Логическое программирование

2.1. Основные понятия

Индивидуумы. Отношения. Факты

Индивидуумы – произвольные дискретные объекты из заданной предметной области. Они отражают контекст конкретной задачи.

Отношение – отражают взаимосвязи между индивидуумами в данной предметной области. Математически, ***отношения*** – это произвольное подмножество декартового произведения n произвольных множеств.

Говорят, что задано n-местное отношение

S1 x S2 x … x Sn

Пример: бинарное отношение N x N подмножество {(2,4),(3,9),(4,16),…}

Тут второй элемент пары – квадрат первого числа. Оно определяет отношение квадрата числа, т.е. (m, k), где $k=m^2$.

Отметим, что функция – это частный случай отношения. Обратное утверждение не верно, т.е.

$y=\sqrt{x}$ - отношение

$y=\left|\sqrt{x}\right|$ - функция

Одноместные отношения называются свойствами.

N {1, 3, 5, …} – множество чисел, обладающих свойством нечётности.

Будем давать имена индивидуумам и отношениям. Индивидуумы будут представлять самих себя (типа атомов в Lisp) чаще всего.

Символьные константы с маленькой буквы

имя_отношения (об1, об2, …, обn)

квадрат (2,4) – TRUE квадрат (2,5) – FALSE

чётное (8)

студент (Иванов)

имя_отношения – предикатный символ.

Его имя тоже с маленькой буквы. Отношения в таком виде, можно рассматривать как высказывания об объектах в рамках данной задачи.

Отношения могут быть истинными и ложными. Индивидуумы можно рассматривать, как аргументы высказывания и предиката.

Предикат, истинность которого не вызывает сомнения в рамках предметной области, называется фактом.

В Пролог факты записываются с точкой в конце.

Пример: квадрат (2, 4).

Совокупность фактов можно рассматривать как базу данных. В простом случае, как логическая программа.

Пример: Определим некую БД. Область – имущественные отношения в детском саду

имеет (петя, мячик).

имеет (петя, кошка).

имеет (вася, книга).
…
имеет (вася, кошка).

Цель – это предикат, истинность или ложность которого необходимо установить.

Цель инициирует логическую программу и запускает механизм поиска Пролог.

Цель обозначается: *? имеет (вася, кошка)* или *Goal: имеет (петя, книга)*

В зависимости от TRUE или FALSE, Пролог ответит Yes или No. Если No, то в БД нет таких фактов, но не значит, что у Пети нет книги, просто в БД нет такого факта.

Переменные и сложные цели

Для формирования сложных запросов, используются переменные. Переменная – объект особого вида, служащий для обозначения объекта, на который нельзя сослаться непосредственно по его имени. Всегда начинается с большой буквы.

Пример: X, Студент и т.д.

Так в Turbo Prolog. При записи цели переменная записывается вместо искомого объекта.

Пример: Goal: имеет (петя, X)

Ответом будет:

1.) признание истинности Yes
2.) все значения переменной X в БД
x=мячик
x=кошка
2 solutions

Пример:

Goal: имеет (X, велик) → No solutions
Goal: имеет (Y, X)
Y=вася X=кошка
Y=петя X=мячик
…
Попарно 4 solutions

Подстановка обеспечивает истинность цели.

Сложные цели имеют следующий вди:

Goal: <подцель 1>, <подцель 2>, …, <подцель N>

Тут “,” имеет смысл конъюнкций.

Сложная цель – попытка доказательства нескольких целей одновременно. С помощью такой записи можно формировать более сложные запросы.

Пример:

Goal: имеет (вася, X), имеет (петя, X)
X=кошка

X – один и тот же объект. Перед попыткой доказать цель X является свободной или не конкретизированной.

Пролог начинает с крайней левой, т.е. первой цели и сверху БД, т.е. с начала БД, сам поиск. Процесс поиска происходит с использованием сопоставления или унификации, сопровождаемого связыванием и конкретизацией переменных.

Сопоставление определяется следующим образом:

1. факт и цель будут сопоставимы, если имеют одинаковый предикатный символ, а их элементы попарно сопоставимы
2. индивидуум – константа сопоставима с идентичной константой или с переменной конкретизированной эквивалентным значением
3. не конкретизированная переменная сопоставима с любым объектом, при этом, если переменная сопоставляется с константой или конкретизированной переменной, происходит её конкретизация. Если переменная сопоставляется с другой конкретизированной переменной, то происходит их связывание.

Рассмотрим действия Prolog системы:

Пример: Goal: имеет (петя, X), имеет (вася, X)

1. берётся подцель 1 (имеет (петя, X)) сопоставимую с первым фактом БД и происходит конкретизация переменной X как “мячик”. Подцель 1 считается доказанной. Каждому X будет соответствовать “мячик”
2. для поиска берётся подцель 2. Вместо X уже подставили “мячик”. Поиск для новой цели начинается с начала БД. Ответом получим “No”. Инициируется процесс возврата (Back Tracking)
3. возврат осуществляется к последней доказанной цели. Тут это подцель 1. Происходит освобождение переменной X. В общем случае, при возврате, освобождаются все переменные, которые были конкретизированы при доказательстве предыдущей цели
4. для поиска вновь выбирается подцель 1, но поиск продолжается по БД с факта 2. Факт 1 помечается в БД как пройденный. С фактом 2 сопоставление завершено успешно
5. X получает значение “кошка”. Вновь выбирается подцель 2. И начинается поиск с начала БД
6. доказаны одновременно обе подцели. И т.о. X=кошка
7. вновь инициируется механизм возврата, чтобы найти другие решения. Вновь выдирается подцель 1, X освобождается и поиск по БД начинается с факта 3 и подцель 1 не может быть доказана другим способом, поэтому Turbo Prolog скажет:

   1 Solution.
   X=кошка.

2.2. Правила в Пролог

Программа на Пролог представляет из себя набор утверждений: фактов, целей и правил.

Правила отражают некую логическую зависимость некого предиката от других предикатов. Правило в Пролог состоит из заголовка и тела. Заголовок правила – некий предикат, возможно содержащий переменные.

Тело правила (хвостовые цели) – список предикатов, разделённых запятыми.

Заголовок if подцель1, подцель2, …, подцельN.

Правило в общем случае гласит, что предикат, составляющий заголовок правила, будет истинным, если истинны все подцели, входящие в его тело, т.е. “,” – имеет смысл конъюнкций.

И заголовок, и подцель могут содержать переменные. Одноимённые переменные имеют смысл только в рамках одного правила, т.е. областью действия переменной в Пролог является ***утверждение*** (как факт правила или цель).

Правила записываются вместе с фактами в базе данных. Работают следующим образом: если в процессе поиска доказательства для некоторой цели происходит сопоставление текущей цели с заголовком правил, то данная цель заменяется списком подцелей из тела правила, при этом конкретизация и связывание переменных распространяется на всё правило.

Goal: заголовок
подцель1, подцель2, …

Например:
| любит (саша, леденцы).
| любит (маша, X) if любит (саша, X).

// Маша любит нечто, если это же самое любит Саша

// Выяснить: любит ли Маша леденцы

Goal: любит (маша, леденцы)

Переменная X конкретизируется значением «леденцы» во всех частях правил. Порождается новая цель, выбирается первая подцель из тела правила: любит (саша, леденцы). Эта цель новая и для неё поиск ведётся с начала данных. И мы убеждаемся, что она согласуется. Переменная X получила новое значение.

Пример: Королевское семейство

1. мужчина (альберт).
2. мужчина (эдуард).
3. женщина (алиса).
4. женщина (виктория).
5. родители (эдуард, виктория, альберт).
6. родители (алиса, виктория, альберт).
7. сестра (X, Y) if женщина (X), родители (X, M, F), родители (Y, M, F).
   Goal: сестра (алиса, X)

- Исходная цель сопоставляется с заголовком утверждения 7. Переменная X из правила конкретизируется значением «алиса». Целевая переменная X связывается с переменной Y из правила.
- Выбираем первую подцель из тела правила и доказываем её с помощью факта 3.
- Вторая цель правила 7: родители (X, M, F). Очевидно, что эта подцель сопоставима с утверждением 6. При этом происходит конкретизация промежуточных переменных M и F.
- Третья подцель имеет теперь вид: родители (Y, виктория, альберт). Она сопоставима с утверждением 5. И Y получает конкретное значение.
- Доказаны три подцели, для утверждения 7 и следовательно целевая переменная X = эдуард является решением.

- Делается возврат к последней доказанной цели: родители (Y, виктория, альберт). И она была доказана с помощью факта 5. И ее Пролог попытается передоказать. И она будет доказана с помощью утверждения 6 и Y конкретизируется новым значением «алиса». И вновь все 3 подцели доказаны и целевая переменная X получает значение «алиса».

2.3. Структура программ на Turbo Prolog

Программа на Пролог состоит из нескольких разделов. Вначале каждого раздела соответствующее ключевое слово.

Раздел	***Ключевое слово***
домены	domains
глобальные домены	global domains
динамические базы данных	database
предикаты	predicates
глобальные предикаты	global predicates
goal (задать цель)	goal
клозы	clauses

Может присутствовать несколько разделов доменов, предикатов и клозов. Остальные разделы представлены по одному. Обязательными являются только разделы предикатов и клозов.

Раздел доменов

Аргументы предикатов в Пролог должны принадлежать строго определённой области или типу данных.

Объявление доменов выглядит следующим образом:

$$имя\_домена1, имя\_домена2, ... = \begin{cases} определение\_типа \\ стандартный\_домен \end{cases}$$

имя_домена – используется при объявлении предикатов.

Существует 5 стандартных типов:

1. char – объекты представляют собой любой восьми битовый код ASCII ‘+’, \ десятичный код ASCII, \n (перевод на новую строку)
2. integer – (целочисленный) диапазон: -32767 до +32768
3. string – цепочка слов в двойных кавычках. Например: “Ура!”. Размер: строковая константа 250 символов, строковая переменная может конкретизироваться до 64 kb
4. symbol – “Ку-Ку!”. Либо в виде идентификатора символьного имени, т.е. последовательность букв, символов, _, причём первая буква всегда маленькая. Например: саша_1

string и symbol – символы взаимозаменяемые (исключение: саша_1), они сопоставлены друг с другом.

Раздел предикатов

Объявление: имя_предиката (домен1, домен2, …)

домен – тип существующего аргумента, либо стандартный, либо определяемый пользователем.

domains

имеет (имя, вещь) // определяемый пользователем

нравится (symbol, symbol) // стандартный

стоит (вещь, цена)

истина // предикат может не иметь аргументов, но должен быть объявлен

Раздел клозов

Сюда входят утверждения (факты, правила). Утверждения для одного предиката должны быть структурированы.

Раздел Goal

Содержит только одну цель. Он необязателен. Если он определён, то Пролог система будет инициировать возникновение окошечка

Goal:

, чтобы самому ввести цель. Желательно писать программу без раздела Goal.

2.4. Синтаксис переменных.

Начинается с большой буквы или знака подчеркивания.

_1 _переменная

Особый случай: анонимная переменная "_" – ссылается на объект, значение которого нас не интересует. Родители(x,_,_) – не означает, то что ищем X у которого родители одинаковые, они просто есть. Анонимная переменная не принимает конкретного значения. При сопоставлении происходит контроль принадлежности переменной конкретному домену.

Исключения:

- символы и строковые домены (символы и строки могут быть сопоставлены).
- Целые числа могут быть преобразованы в действительные.

2.5. Арифметика в TurboProlog.

4 действия: +, -, *, /.

Унарные: -, +.

Используются скобки.
Приоритеты по убыванию: +, -, *, /, mod, dir, '+', '-'. (x / y) / z (-> порядок выполнения).

Операции сравнения.

<, <=, <>, ><, = в прологе это встроенные предикаты.

= работает двояко: и как предикат и как оператор сравнения. Если слева = стоит некокретизирующая переменная, то выражение справа вычисляется, результат связывается с переменной и цель считается доказанной.

Goal: x = 2* 2 -> x = 4

Если кокретизирущая, то происходит попытка сравнения с правой частью. Переменные, входящие в арифметическое выражение должны быть конретизируемы к моменту вычисления. X = X +1 не пройдет. Могут использоваться: abs(x), cos, sin, tan, exp, ln, log, sqrt(x).

Predicates
Население (symbol, integer)
Площадь (symbol, integer)
Плотность (symbol, real)

Clauses
Население (США, 203)
Население (Китай,1000)
. . .
Площадь (Китай, 9)
. . .
Плотность (x, y) if население (x, p), площадь (x, a), y = p/a.

Goal:
Плотность(Китай, x)
X = …
Плотность(Китай, 89)
В пределах (s, x, y) if плотность(s, z), x<z, y>z.

Goal: в пределах (x, 10, 100)

Predicates
Корень (real, real, real, real, real)

Clauses
Корень (A, B, C, x1, x2) if D>= 0, D = B * **B – 4** * A * C,
x1 = (-B + sqrt(D)) / (2 * A) ,
x2 = (-B - sqrt(D)) / (2 * A) .
----------//---------- if D = 0, x1 = …, x1 = x2.

Goal: (8, 4, 5, x, y)

2.6. Простейший ввод/вывод.

- Согласуются с БД
- Согласуются с БД только 1 раз
- Производят побочный эффект

Ввод:

Readln (x) x – не должен быть конкретизирован к моменту выполнения(string, symbol).
Readint (x) x – integer
Readreal (x)
Readchar (x)

Goal: readreal(A), … , корень (A,B, …)

Вывод:

Write (x1, x2, … xn) x – либо константы, либо конкретизируемые переменные.
Goal: … корень (…, x1, x2)
Write ("корни уравнения", A, B, C, "равны", x1, "и", x2)

Nl
Readreal (A1, nl, read…)

2.7. Функции в Прологе.

0! = 1
N! = N * (N – 1)!, N > 0
f (N, F) F – факториал числа N
f (0, 1) – факт
f (N, F) if N > 0, Nпр = N – 1, f (Nпр, Fпр), F = n * Fпр.

Рассмотрим динамику поиска докозательства.

f (3, F) N1 = 3; → f (2, F1) N2 = 2 → f (1, F2) N3 = 1 → f (0, F3) N4 = 0 → F3 = 1 →

F2 = 1 * N3 = 1 → F1 =F2 * N2 = 2 → F = F1 * N1 = 6.

1. Общие терм. случая.
2. Предположить, что уже есть предикат, который правильно работает для неотрых предыдущих случаев.
3. Создать рекурсивные правила с предикатами предыдущего случая.

4. Проверить 1, 2, 3 шаги.

Пример:

$\sum i^2$

Сум (N, S)
Сум (1,1)
Сум (N, S) if N > 1, N1 = N1 – 1, Сум (N1, SS), S = SS + N1 * N1.

Пример:

Cmen (x, D, y) $x^D = y$
Cmen (x, 0, 1)
Cmen (x, D, y) if D > 0, D1 = D –1, Cmen (x, D1, y1), y = x * y1.

Пример:

$f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$

F(0 , 0)
F(1, 1)
F(N, x) if N > 1, N1 = N – 1, N2 = N – 2, F(N1, x1), F(N2, x2), x = x1 + x2.

Пример: (Алгоритм Евклида)

НОД (A, B, x)
НОД (A, B, x) if A = B, x = A.

НОД (A, A, A).
НОД (A, B, R) if A > B, A1 = A – B,
НОД (A1, B, R).
НОД (A, B, R) if A < B, B1 = B – A,
НОД (A, B1, R).

Пример (Ханои):
Ханои (N, U1, U2, U3)

Взять с первой, положить на третью.

C_на (X, Y) if writeln (“Взять с”, X, “положить на ”, Y).
Ханои (1, X, Y, Z) if C_на (X, Z).
Ханои (N, X, Y, Z) if N > 1,
N1 = N – 1,
Ханои (N1, X, Z, Y),
C_на (X, Z),

Ханои (N1, Y, X, Z).

Пример:

Sqrt(x) = y_i $\quad | y_i^2 - x | <= \varepsilon$

$Y_1 = 1$, $y_i = ½ * (y_{i-1} + x/y_{i-1})$, $i > 1$

Корень (10, 1 Z, 0.0001) (X, Y, Z, Eps)

Корень (X, Z, Z, Eps) if abs (Z * Z - X) <=Eps.

Корень (X, Y, Z, Eps) if abs (Y * Y - X) > Eps, Yn = (Y + X/Y) / 2, Корень (X, Yn, Z, Eps).

2.8. Нисходящая и восходящая рекурсии.

Нисходящая рекурсия.

Это такая логическая программа, когда в теле правила, определяется данный предикат, рекурсивное обращение к этому предикату происходит до полного завершения вычислений в данном правиле.

Общий вид: P - предикат

A’, A” - утверждения

P’, P” – обращение к предикату

B’, B” – остатки вычислений

P if A’, A”, … P’, P”, …B’, B”. Остатки вычислений являются цепочкой отложенных вычислений (основные вычисления). F_n if … f_{n-1}, x = … по аналогии с предыдущим x = и является остаточными вычислениями.

Восходящая рекурсия.

Это такая организация вычислительного процесса, когда часть вычислений, подлежащих повторению, полностью завершается до рекурсивного вызова определяемого предиката.

Общий вид:

P if A’, A”, … P’. Только один вызов стоит на последнем месте.

Примеры (Факториал):

1. f (N, F) if f (0, 1, N, F).

   f (N, F , N, F). (f (Счетчик, Параметр, N, F) if Счетчик =N , Параметр =F .)
   f (N1, F1, N, F) if N > N1,
   N2 = N1 + 1,
   F2 = N2 * F1,
   f (N2, F2, N, F).

2. f (N, F) if f (N, 1, F).

   f (0, F, F).
   f (N, F1, F) if N > 0,
   F2 = F1 *N,
   N1 = N –1,
   f (N1, F2, F).

Сравним нисходящую и восходящую рекурсии на числах Фибоначчи.

Нисходящая.

f (0 ,0).

f (1, 1).

f (N, X) if N > 1,

N1 = N –1,

N2 = N – 2,

f (N1, X1), f (N2, X2),

X = X1 + X2.

Чтобы избежать дублирование вычислений, воспользуемся восходящей рекурсией.

f (N , F) if f (N, 2, 0, 1, 0, F). (f (N, i, i-2 ,i – 1, F_n, F).)

f (N, N, _, _, F, F).

f (N, I, F_{i-2}, F_{i-1}, F_i, F_n) if I < N,

I >= 2,

$I_{сл}$ = I + 1,

$F_{сл}$ = F_{i-2} + F_{i-1},

f (N, $I_{сл}$, F_{i-1}, F_i , $F_{сл}$, F_n).

Рекурсия с недетерминированным выбором.

Когда для одного предиката содержится более одного вызова.

Задачи:

Построить следующий ряд чисел: 2/3 2/3 4/3 4/5 6/5 6/7 … и найти их произведение.

Если n нечетное p(N) = (p(n -1) * (n +1)) / n

Если n четное p(N) = (p(n -1) * n) / n

P(1, 2).

P(I, X) if I > 1, I mod 2 = 0,

I1 = I –1,

P (I1, X1),

X = X1 * (I + 1) / I.

P(I, X) if I > 1, I mod 2 = 1,

I1 = I –1,

P (I1, X1),

X = X1 * (I + 1) / I.

Дано натуральное n определить в нем количество цифр.

P(N, 1) if N < 10.

P(N, K) if N >= 10,

N1 = N div 10,

P(N1, K1),

K = K1 + 1.

2.9. Списки

Список – это упорядоченный (порядок элементов имеет значение) набор элементво одного типа, домена.

Элементы: любые объекты, но обязательно одного домена.

Синтаксис: [8, 15, 24]

[вася, петя]

[ ] – пустой список

Объявление: в разделе доменов мы пишем

```
domains
    имя_списка = имя_домена
    int_list=integer *
    symb_list=symbol *
```

Пустой список принадлежит любому типу.

Списки могут быть вложены:

[[ ], [1, 8], [15]]

Могут также быть списки списков.

Список списков целых чисел:

int_list_list=int_list *

Сопоставление списков: списки сопоставимы, если попарно сопоставимы их элементы.

[8, 2]=[8, 2] – сопоставимы

[2, 8]=[8, 2] – несопоставимы

В списках можно использовать переменные:

[2, X]=[8, 2] – несопоставимы

[2, X]=[2, 8] – сопоставимы, x=8

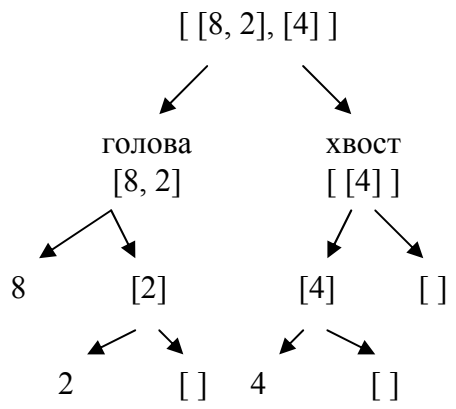
[2, X]=[2, Y] – если X и Y не конкретизированы, то произойдёт сопоставление и X свяжется с Y (X <-> Y)

[[8, 2], [4]] = [X, Y] – X и Y не конкретизированы, X=[8, 2] и Y=[4]

Операция разделения

Любой список, кроме пустого, состоит из головы и остатка (хвоста). Голова – первый элемент списка, а хвост – это ***список*** (возможно пустой), полученный выбрасыванием первого элемента из исходного списка.

Например:



Запись в Пролог: [X | Y], где X – голова, а Y – хвост.
Например: [1, 2, 3] = [X | Y] – попарно-сопоставимы, X=1, Y=[2, 3]
[a, b, c] = [X, Y | Z] – X=a, Y=b, Z=[c]
[a, b, c] = [X | Y, Z] – нельзя
[[a, Y] | Z] = [[X, b], [c, d]] – X=a, Y=b, Z=[[c, d]]
Задача: Попробуем построить предикат, который печатает в столбик свой аргумент

```
domains
    i_l=integer *
predicates
    печать (i_l)
clauses
    печать ([ ])
    печать ([H | T]) if write(…H…), печать (T).
```

Множественность решений не получим, потому что список [H | T] не пустой.
Задача: Принадлежность элемента списку

```
member (X, [X | _]).
member (X, [_ | Y]) if member (X, Y).
```

Goal: member (a, [b, a, c]) – ответ будет YES
Goal: member (X, [a, b, c]) – X=a, X=b, X=c
Так мы можем использовать перебор элементов в списке
Задача: Выделить последний элемент списка

```
last (X, [X]).
last (X, [_ | Y]) if last (X, Y).
```

Задача: Проверить является ли некий список упорядоченным

```
поряд ([ ]). //пустой
поряд ([_]). //состоит из одного элемента
поряд([X, Y | Z]) if X<=Y, поряд ([Y | Z]). //более одного элемента
```

Задача: Склейка списков – [a, b] [c,d] → [a, b, c, d]

```
append ([ ], L, L).
append ([X | L1], L2, [X | L3]) if append (L1, L2, L3).
```

Задача: Реверсирование списка (восходящая и нисходящая рекурсия)
//нисходящая

```
reverse ([ ], [ ]).
reverse ([X | Xs], Zs) if reverse (Xs, Ys), append (Ys, [X], Zs).
//восходящая
        reverse (Xs, Ys) if rev (Xs, [ ], Ys).
        rev([ ], Ys, Ys).
        rev([X | Xs], A, Ys) if rev (Xs, [ X | A], Ys).
```

2.10. Механизмы управления поиском в Пролог

Отсечение

```
        след (X, Y, [X, Y | _ ] ).
        след (X, Y, [ _ | Z] ) : - след (X, Y, Z).
        след (1, X, [1, 2, 1, 3, 8, 15] ).
```

Исключить возможность альтернативного выбора при доказательстве цели можно с помощью оператора “!”.

```
        след (X, Y, [X, Y | _] ) if ! .
```

При активации отсечения, т.е. в процессе доказ. механизм. дошёл до этого предикаты отсек. все неизведанные пути для той цели, попытки доказательства которой привели к исполнению этого самого предиката исполнения.

Пример:

```
1.  нод (A, A, A) if ! .
    нод (A, B, N) if A>B, ! , A1=A-B, нод (A1, B, N).
    нод (A, B, N) if B1=B-A, нод (A, B1, N).

2.  10, 5, 1
    сум (0).
    сум (S) if S>=10, !, write (10), S1=S-10, сум (S1).
    сум (S) if S>=5, !, write (5), S1=S-5, сум (S1).
    сум (S) if write (1), S1=S-1, сум (S1).
```

```
? ..., pred(...),...
1. pred( ).
2. pred( )...
3. pred( ) if a, b, !, c, d.
4. pred( )
5. ...
```

До тех пор пока не будет выбрано утверждение 3, механизм поиска работает обычным образом. Когда выбир. утверждение 3 подцели a и b доказываются обычным образом, но как только они доказаны, исп. отсечение и таким образом путь между порождающей целью и ! фиксир., т.е. не будет делаться попыток повторного

доказательства текущей цели и подцели a и b. Справа от отсечения механизм возврата будет работать обычным образом.

Добавить элемент в список, если его там нет.

```
member
add (X, L, L) if member (X, L), !.
add (X, L, [X | L] ).

pr(...) if условие1, !, действие1.
pr(...) if условие2, !, действие2.
…
pr(...) if else_условие.

a (x) if b (x), !, c (x).
a (x) if d (x).
b (e).
b (f).
c (e).
c (f).
d (g).

goal: a(x)
      x=e
```

Странности отсечения

```
append ([ ], X, X) if !.
```

Использование отсечения может привести к явным ошибкам.

- min (A, B, A) if A<=B.
  min (A, B, B) if B<A.
- min (A, B, A) if A<=B, !.
  min (A, B, B).
- min (A, B, C) if A<=B, !, C=A.
  min (A, B, B).

Существует 2 встроенных предиката: true и fail:

true всегда согласуется с базой данных

fail никогда не согласуется с базой данных

сум (84), fail передоказывается

Использование *комбинаций !, fail*

С ее помощью реализуются аварийный выход из программы. Fail – отрицание, как безуспешное выполнение. В общем виде конструкция выглядит следующим образом:

```
не_равно (X, Y) if !, fail.
не_равно (X, Y).
```

не_утв if утв, !, fail.
не_утв.

Отрицание

Реализация с помощью встроенного предиката not.

not (pr (org))

Цель такого вида истинна, если предикат не согласуется с базой данных и ложно, если предикат – аргумент согласующ. с базой данных.

not (Предикат) if Предикат, !, fail
not (Предикат)

Добавить элемент в список, если его там нет.

add (X, L, L) if member (X, L).
add (X, L, [X | L]) if not (member (X, L)).

Странности отрицания

Никаких конкретизация, никаких присваиваний не производится.

member (X, [a |b]) not (not (member (X, [a |b])))
студент (Иванов)
женат (Петров)
неженатый_СТ (X) if not (женат (X)), студент (X).
Goal: неженатый_СТ (X)
abnormal program terminated: unknoun X

Определяемое пользователем повторение (цикл).

повтор if a, b, c, fail.

Искусственная точка возврата.

repeat.
repeat if repeat.

Эхо if repeat, readln(X), write(X), конец(X), конец(STOP).

2.11. Структуры данных.

Domains
Имя_типа = функтор(объект1, объект2,…)
Функтор – символьное имя.

Domains
Book = книга(автор, название, год)

Автор = symbol
Название = string

Год = integer

Clauses
Ex_libris(Иванов, книга(Толстой, “Война и мир”, 1999))
Goal:
Ex_libris(Иванов, X)
Ex_libris(Иванов, книга(Толстой, X, _))
Правило унификации:
2 составных терма сопоставимы, если они имеют одинаковый функтор, одну арность, а их аргументы попарно сопоставимы.

Domains
Кн = книга(авт, назв, год,издат)
Авт = автор(имя, фамилия, умер)
Изд = издательство(имя_назв, год)

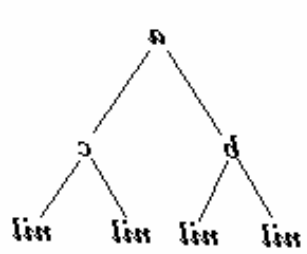
Альтернативные домены.

Что-то = книга(…); аудио(…); видио(…);бумер
Вещи(Иванов, аудио(…))
Вещи(Иванов, бумер)

День = понедельник; вторник; среда; четверг; пятница; суббота; воскресенье;

2.12. Бинарные деревья.

Двоичное дерево состоит из корневой вершины, левого поддерева, правого, либо пустое дерево. Корень дерева – элемент не являющийся деревом. Nil – пустое дерево.



Дерево описывается с помощью вложенных структр:
дерево(a, дерево(b, nil, nil), дерево(c, nil, nil))
дв_дер = дерево(корень, лев_дер, пр_дер)
лев_дер, пр_дер = дв_дер
корень = symbol

Предикат, который проверяет, является ли элемент элементом двоичного дерева:
b_tree(nil).
b_tree(дерево(Вершина, ЛД, ПД)) if b_tree(ЛД), b_tree(ПД).

Принадлежность вершины дереву.

Эл(X, дер(X, _, _)).
Эл(X, дер(Y, Л, П)):- Эл(X, Л), Эл(Y, П).

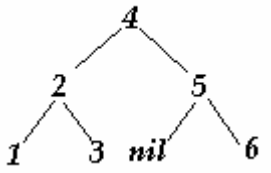
замена(X, Y, TX, TY)
замена(X, Y, nil, nil).
замена(X, Y, дерево(X, Л, П), дерево(Y, Л1, П1)) if замена(X, Y, Л, Л1),
замена(X, Y, П, П1).
замена(X, Y, дерево(Z, Л, П), дерево(Z, Л1, П1)) if X <> Z,
замена(X, Y, Л, Л1),
замена(X, Y, П, П1).

Построить линейный одномерный список, содержащий вершины дерева.
Сверху – вниз:
[a, b…(левое поддерево), c…(правое поддерево)]
обход1(nil, []).
обход1(дерево(X, Л, П), Y) if обход1(Л, $Л_s$),
обход1(П, $П_s$),
append([x| $Л_s$], $П_s$, Y).

Слева – направо:
обход2(nil, []).
обход2(дерево(X, Л, П), Y) if обход2(Л, $Л_s$),
обход2(П, $П_s$),
обход2($Л_s$, [X| $П_s$], Y).

Снизу – вверх:
обход3(nil, []).
обход3(дерево (X, Л, П), Y) if обход3(Л, $Л_s$),
обход3(П, $П_s$),
append($П_s$,[X], $П_s$1),
append($П_s$, $П_s$, Y).

Упорядочивание бинарных деревьев.



эл(X, дер(X, _, _)).
эл(X, дер(Y, Л, П)) if X > Y, эл(X, П).
эл(X, дер(Y, Л, П)) if X <= Y, эл(X, П).

Добавить заданный элемент к упорядоченному дереву:

влк(дер_исх, X, дер_нов).

вкл(nil, X, дер(X, nil, nil)).

вкл(дер(Y, Л, П), X, дер(Y, $Л_x$, П)) if X < Y, вкл(X, Л, $Л_x$).

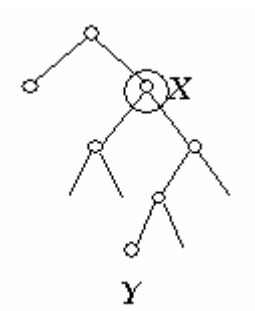
вкл(дер(Y, Л, П), X, дер(Y, Л, $П_x$)) if X > Y, вкл(X, П, $П_x$).

В данном случае ключ считается уникальным.

Удаление элемента.

Уд(дер_сX, X, дер_безX).

{if вкл(дер_безX, X, дер_сX) не пройдет}



Y – крайняя левая вершина правого поддерева элемента X. Если Y перенести в X, то упорядоченность дерева сохраниться.

Уд_лев(исх, У, рез). Удаляет крайнюю левую вершину правого поддерева исходного дерева.

Уд_лев(дер(У, nil, пр), X, пр).

Уд_лев(дер(кор, Л, П), У, дер(кор, Л1, П)) if Уд_лев(Л, У, Л1).

Удаление произвольной вершины.

Уд(дер(X, nil, П), X, П).

Уд(дер(X, Л, nil), X, Л).

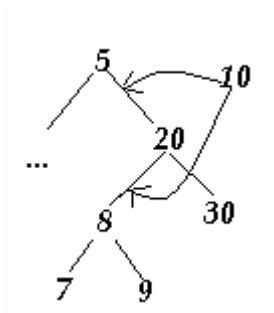
Уд(дер(X, Л, П), X, дер(Y, Л, П1)) if уд_лев(П, Y, П1).

Уд(дер(кор, Л, П), X, дер(кор, Л1, П)) if кор > X, Уд(Л, X, Л1).

Уд(дер(кор, Л, П), X, дер(кор, Л, П1)) if кор > X, Уд(П, X, П1).

Включение вершины на произвольный уровень:

Надо включить вершину 10. Включается в качестве корня, если корень > X, то левое дерево, если < X, то правое.



вклкор(nil, X, дер(X, nil, nil)).
вклкор(дер(Y, Л, П), X, дер(X, Л1, дер(Y, Л1, Л2))) if X < Y,
вклкор(Л, X, дер(Ч, Л1, Л2)).
вклкор(дер(Y, Л, П), X, дер(X, дер(Y, Л, П1), П2)) if X > Y,
вклкор(П, X, дер(Ч, П1, П2)).

Рассмотрим программу включения произвольной вершины в произвольное дерево.

вкл(исх, X, рез) if вклкор(исх, X, рез).
вкл(дер(Y, Л, П), X, дер(Y, Л1, П)) if Y > X, вкл(Л, X, Л1).
вкл(дер(Y, Л, П), X, дер(Y, Л, П1)) if Y < X, вкл(П, X, П1).

С помощью вкл() можно строить упорядоченные деревья из неупорядоченного списка.

уд(исх, X, рез) if вкл(рез, X, исх).

строить([], nil).
строить([н|т], дер) if строить(т, дер2), вкл(н, дер2, дер).

Задан список, например[1, 2, 3, 4, 5] – обход2() – это его преобразование в дерево.
Сортировка по дереву:

Сорт(X, Y) строить(X, дер), обход2(дер, Y).

2.13. Метод ”образовать и построить”

Это метод построения простых и недетерминированных программ. Его суть заключается в следующем: одна программа(предикат) генерирует множество возможных или предполагаемых решений, а другая программа(предикат) проверяет решения, выбирая подходящие.
Решение(X) if породить(X), проверить(X).
Порождение базируется на поиске с возвратом. Самым подходящим генератором является предикат member(X, [8, 15, 44,…]).
Выбрать из списка все элементы меньшие нуля.
Отр(X, L) if member(X, L)? X < 0.

Порождение натуральных чисел:
Нат(1).
Нат(I) if Нат(J), I = J +1.

В пределах от 1 до N:

Нат(1, _).
Нат(I, N) if Нат(J, N), I = J + 1, I <= N.
Тут есть два заблуждения:

1) Нельзя утверждать, что если J натуральное число, то и J + 1 тоже является натуральным в тех же пределах.
2) Надо делать проверки перед рекурсивным вызовом предиката.

Нат(1, _).
Нат(I, N) if N > 1, M = N – 1, Нат(J, M), I = J + 1.

Применение этого метода для разложения N на сумму двух квадратов:

```
кв(N, I, J) if K = sqrt(N), взять(I, K),
                KJ = I, взять(J, KJ),
                I >= J, N =  I * I + J * J.
```

Проверка при выборе отрицательных из списка, без генератора.

```
Отр_mem(X, [X|_]) if X < 0.
Отр_mem(X, [_|Y]) if Отр_mem(X, Y).
```

Построим сортировку вставками. Генератором будет отсортированный ранее список.

```
Сорт([], []).
Сорт([X|T], Y) if Сорт(T, Z), вставить(X, Z, Y).
вставить(X, [], []).
вставить(X, [Y|TY], [X, Y|TY]) if X <=Y.
вставить(X, [Y|TY], [Y|Z]) if X > Y, вставить(X, TY, Z).
```

Быстрая сортировка может быть представлена следующим образом: выбираем произвольный элемент списка и вставляем в 2 списка все оставшиеся элементы, которые больше или меньше выбранного.

```
б_сорт([X|T], Y) if разбить(T, X, M, Б)
                    б_сорт(M, Lм),
                    б_сорт(Б, Lб),
                    append(Lм, [X|Lб], Y).
б_сорт([], []).

разбить([], _, [], []).
разбить([X|T], Y, [X|Lм], Lб) if X <= Y,
                                 разбить(T, Y, Lм, Lб).
разбить([X|T], Y, Lм, [X|Lб]) if X > Y,
                                 разбить(T, Y,Lм, Lб).
```

Способы составления суммы из монет заданного наминала.

```
наборы(X, [10, 5 ,3], sum)
                30
            [10, 10, 10]
            [5, 5, 10, 10] …
наборы([], _, 0).
```

```
уаборы(X, L, M) if M > 0 append(_, [X2|T2], L),
                         M1 = M – X2,
                         M1 >= 0,
                         наборы(Y, [X2|T2], M1), X = [X2|Y].
```