Министерство образования и науки Российской Федерации

Алтайский государственный технический
университет им. И.И. Ползунова

Андреева А.Ю. Тушев А.Н.

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Барнаул 2015

УДК 681.3

Андреева А.Ю., Тушев А.Н. Функциональное и логическое программирование: Учебно-методическое пособие/ Алт. госуд. технич. ун-т им. И.И. Ползунова.- Барнаул: 2015.-155 с.

Данное учебной пособие предназначено для дистанционного изучения дисциплины “Функциональное и логическое программирование” для обучающихся на направлении 09.03.04 «Програмная инженерия». Рассматриваются вопросы программирования на языке Лисп, применение этого языка для автоматического построения чертежей в системе Автокад. Во второй части рассматриваются вопросы программирования на языке Пролог, применение языка для решения различных задач.

Рецензент: С.А. Кантор - зав. кафедрой Прикладной математики АлтГТУ

1 ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА ЛИСП

В главах 1-3 язык Лисп рассматривается на примере интерпретатора muLisp, который сохраняет большинство свойств стандарта языка и удобен для изучения. Несоответствия muLisp стандарту Common Lisp отмечены в каждом конкретном случае.

В главе описаны основные структуры данных, встроенные функции muLisp, рекурсивные и итерационные методы обработки списков и приведены упражнения.

1.1 СТРУКТУРЫ ДАННЫХ

1.1.1 Числа

Числа в языке Лисп подразделяются на **целые** и **дробные**. Целые числа в muLisp делятся на *малые целые* (-65536,+65536) и *большие целые* (произвольного размера). Различие между малыми и большими целыми числами влияет только на результат операции сравнения EQ.

Дробные числа представляют собой дробь, числитель и знаменатель которой являются целыми числами. При отображении дробные числа представляются в форме с десятичной точкой, при вводе возможны 2 варианта: форма с десятичной точкой и дробная форма.

Примеры:

-478	малое целое число
1234567890	большое целое число
4.789	дробное число
-1/5	дробное число
-1 / 5	не число (пробелы в записи числа недопустимы)

Все операции, кроме операции сравнения EQ, дают одинаковый результат для всех типов чисел.

В других диалектах Лиспа встречаются различные внутренние представления чисел. Иногда дробные числа в Лиспе хранятся в форме с плавающей точкой, как в других языках программирования (например, числа типа REAL в языке Паскаль). В этом случае в реализации языка предусматриваются функции приведения типов.

1.1.2 Символьные атомы

Символьные атомы представляют собой идентификаторы, т.е. последовательности латинских букв и цифр, начинающиеся с буквы. Заглавные и малые буквы в идентификаторах

неразличимы (как в языке Паскаль). Кроме букв и цифр, имена символьных атомов могут включать и другие символы, такие как * (звездочка), + (плюс) и т.п. Примеры правильных идентификаторов: x1, X1, a, b2, *atom*.

Символьные атомы являются аналогами переменных величин в языках Паскаль и Си: они имеют **значение**, которое присваивается им специальными функциями SETQ и SET.

В отличие от переменных Паскаля и Си, символьные атомы Лиспа не являются *типизированными*, т.е. в Лиспе отсутствует понятие типа переменной (например, REAL, INTEGER в языке Паскаль). В разные моменты времени значением одного и того же символьного атома может быть число, строка, список, символьный атом и т.д. Для определения типа текущего значения символьного атома существуют встроенные функции. Подобным образом организованы переменные в СУБД FoxPro.

Значением символьного атома по умолчанию (до первого присваивания) в muLisp является сам этот атом. В классическом Лиспе только два символьных атома по умолчанию имеют в качестве значения самих себя - это атомы T (истина) и NIL (ложь). Попытка обращения к переменной, не имеющей значения, в классическом Лиспе вызывает сообщение об ошибке: UNDEFINED VARIABLE (неопределенная переменная).

Символьные атомы и числа вместе образуют группу **атомов**.

1.1.3 Списки

По определению список представляет собой выражение вида: (A_1 A_2 ...A_n), где все A_i - это атомы (символьные атомы или числа) или списки. Определение списка является примером **рекурсивного** определения, поскольку в нем имеется ссылка на само себя.

Примеры:

(A 3 1) - список из трех элементов;

((3.5)) - список из одного элемента, который является списком из числа 3.5;

((A 1) C ((4) 3)) - список из четырех элементов, причем первый элемент {(A 1)} является списком из двух элементов, второй {C} – символьным атомом, а третий {((4) 3)} – списком из 2 элементов, первый из которых {(4)} в свою очередь также является списком;

(A() - это выражение не является списком, т.к. количество открывающихся скобок не соответствует количеству

закрывающихся.

Пустой список (список, не содержащий элементов) обозначается пустыми скобками () или идентификатором NIL. Таким образом, NIL в Лиспе обозначает одновременно и символьный атом, и пустой список.

Списки и атомы вместе называются **s-выражениями** (от слова symbolic).

В классическом Лиспе отсутствуют другие типы данных. Наиболее общей структурой является **s-выражение**, которое может быть **атомом** или **списком**. **Атомы** делятся на **символьные атомы** и **числа**.

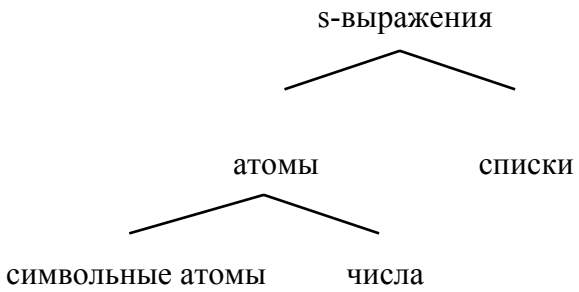


Рис. 1. Иерархия типов данных Лиспа.

В muLisp, как и в других современных вариантах языка Лисп (стандарт Common Lisp), добавлены типы данных, удобные для представления различных структур: строки, векторы (аналоги массивов), матрицы, файловые типы данных, классы и др.

1.2 ПРОГРАММЫ НА ЛИСПЕ

Программа на Лиспе представляет собой последовательность s-выражений, поступающих на вход интерпретатора Лиспа. Каждое s-выражение **оценивается** интерпретатором и результат выводится. В интерпретаторе muLisp приглашением к вводу очередного s-выражения служит символ $.

Итак, в Лиспе программы и данные состоят из одних и тех же объектов, а именно из s-выражений. Любое s-выражение можно оценить, т.е. выполнить как программу. Результатом оценки s-выражения всегда является s-выражение. Способ оценки конкретного s-выражения зависит от его типа:

1. результатом оценки числа всегда является само это число;

2. результатом оценки символьного атома - его значение;

3. результатом оценки списка - значение функции, при этом список должен иметь вид: (A B_1 B_2 ... B_n), где A - символьный атом, являющийся именем выполняемой функции, а B_1...B_n - произвольные s-выражения, представляющие собой аргументы функции. Сначала оцениваются аргументы по изложенным выше трем правилам, затем к результатам их оценки применяется

соответствующая функция. Поскольку аргументами $B_1...B_n$ могут быть списки, правила 1-3 являются рекурсивными.

Таким образом, оценивая s-выражения, Лисп вычисляет значения *функций*, поэтому он относится к языкам **функционального типа**, а способ программирования на подобных языках носит название **функционального программирования**.

При вычислении значений функций в чисто функциональных языках значения аргументов этих функций не изменяются. Лисп имеет ряд отличий от таких языков, в частности, некоторые функции Лиспа не оценивают своих аргументов (например, функции QUOTE, SETQ), другие меняют значения своих аргументов (функции присваивания SET, SETQ).

Функции могут быть встроенными или определенными пользователем. Если при оценке списка оказывается, что функция с заданным именем не существует, то по правилам Лиспа интерпретатор должен выдать сообщение об ошибке и прекратить вычисления. Интерпретатор muLisp ошибки не выдает и считает результатом оценки списка сам этот список.

1.3 ОСНОВНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ ЛИСПА

1.3.1 функция (QUOTE A)

Функция QUOTE не оценивает свой аргумент и выдает его в качестве результата.

Примеры:

```
$ ( QUOTE  A)
A
$ ( QUOTE (A B C))
(A B C)
```

Функция QUOTE встречается настолько часто, что для нее в Лиспе введено сокращенное обозначение:

```
(QUOTE A )       <==>  'A
(QUOTE ( A B))   <==>  '(A B)
```

1.3.2 Арифметические функции

Арифметические функции обозначаются символами **“+”, “-”, “*”, “/”** и выполняют соответствующие действия после оценивания своих аргументов, значения которых должны быть числами.

Примеры:

```
$ (+ 3 7)
10
$ (+ 2 4 9)
15  ;Функция “+” может иметь произвольное число аргументов.
```

```
$ (- 5 2)
3
$ (- 5)
-5  ;Функция “-” может иметь один аргумент, в этом случае
результатом функции является число с противоположным знаком.
$ (* 4 2 5)
40 ;Функция “*”, как и “+”, может иметь больше двух аргументов.
$ (/ (+ 2 4) 3)
2
```

Рассмотрим, как интерпретируется последнее выражение. Сначала оценивается первый аргумент функции деления. Поскольку он является списком, начинается его интерпретация по третьему правилу. В выражении (+ 2 4) аргументы оцениваются сразу, так как числа равны самим себе. В результате функция **“+”** дает 6. Теперь оценивается второй аргумент функции деления: его значение равно 3 - и результат оценки выражения (/ 6 3) с оцененными аргументами равен 2.

1.3.3 Функции присваивания (SETQ A B) и (SET A B)

Функция SETQ не оценивает первый аргумент, который должен быть символьным атомом, оценивает второй аргумент, присваивает его значение первому аргументу и выдает это же значение в качестве результата функции. Функция SET, в отличие от SETQ, оценивает и первый аргумент, результат оценки которого должен быть символьным атомом.

Примеры:

```
$ ( SETQ A 'B)
B
$ A
B
$ ( SETQ X 2)
2
```

В последнем присваивании не требуется апостроф (функция QUOTE) для второго аргумента, так как число равно самому себе.

```
$ ( SETQ A 'D)
D
$ ( SET A 3)
3
$ A
D
$ D
3
```

При выполнении функции SETQ атому A присваивается атом D. Затем при выполнении SET оценивается первый аргумент (атом A), значением которого теперь является D, поэтому атому D и присваивается 3 (оценка второго аргумента SET).

Заметим, что (SETQ A B) <==> (SET 'A B) для произвольных A и B, поэтому имя функции SETQ образовано от имен SET и QUOTE.

1.3.4 Функции обработки списков

Функция (CAR A) оценивает свой аргумент, который должен быть списком, и выдает в качестве значения первый элемент этого списка. Обратите внимание, что элемент списка может быть произвольным s-выражением, т.е. как атомом, так и списком. Примеры:

```
$ (CAR '(A B C))
A
$ (CAR '((A) B (X)))
(A)
$ (CAR '(((A))))
((A))
```

Функция (CDR A) оценивает свой аргумент, который должен быть списком, и выдает в качестве значения этот список без первого элемента. Примеры:

```
$ (CDR '(A B C))
(B C)
$ (CDR '((A) B (X)))
(B (X))
$ (CDR '(((A))))
NIL
```

Первый элемент списка называется **головой** списка, а список без первого элемента - **хвостом** списка. Таким образом, функция CAR выделяет голову списка, а функция CDR - хвост списка. Хвост списка всегда является списком, возможно, пустым.

Заметим, что функция CAR, как и большинство функций Лиспа, оценивает свой аргумент, поэтому при оценке выражения (CAR (A B C)) интерпретатор Лиспа будет считать аргументом функции CAR результат применения функции A к аргументам B, C. Если функция с именем A не была определена пользователем, такие действия приведут к ошибке “Неизвестная функция A””. Для предотвращения оценки аргумента необходимо поставить перед ним апостроф (т.е. применить к нему функцию QUOTE). Интерпретатор muLisp не выдает ошибки “Неизвестная функция””, а считает результатом оценки подобного списка сам этот список, т.е. как бы “подставляет”” пропущенный апостроф.

Функция (CONS A B) оценивает аргументы, причем значение первого аргумента может быть произвольным s-выражением, а значением второго должен быть список. Результатом функции является список с головой A и хвостом B. Примеры:

```
$ (CONS 'A '(B C))
(A B C)
$ (CONS '(A) '(B C))
((A) B C)
$ (CONS 'A NIL)
(A)
```

Для функций CAR и CDR существует сокращенный вариант записи: CxxR, CxxxR, CxxxxR (число символов x не больше 4), где вместо символов x можно подставить символы D или A. Примеры:

```
(CDAAR X)   <=>   (CDR(CAR(CAR X)))
(CADR X)    <=>   (CAR(CDR X))
```

Функция (LIST $A_1...A_n$) имеет произвольное количество аргументов и после их оценки строит список вида ($A_1...A_n$). Значениями $A_1...A_n$ могут быть произвольные s-выражения. Примеры:

```
$ (LIST 'A 2 '(B C))
(A 2 (B C))
$ (LIST (CAR '(A B))(CDR '(A B)))
(A(B))
$ (LIST 'A NIL)
(A NIL)
```

Функция (APPEND A B) выполняется следующим образом: если значениями A и B являются списки вида (A_1 ...A_n) и ($B_1...B_m$), то значение функции APPEND равно списку ($A_1...A_n$ $B_1...B_m$). Примеры:

```
$ (APPEND '(A B C) '(D E))
(A B C D E)
$ (APPEND '(A) '((B)))
(A(B))
$ (APPEND '(A) NIL)
(A)
```

1.3.5 Внутреннее представление данных в Лиспе

В Лиспе все списки хранятся в виде списочных ячеек, или **CONS-ячеек**, имеющих два указателя: на элемент и на следующую ячейку. Эти два указателя называются соответственно CAR и CDR:

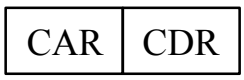


Рис.2. Структура CONS-ячейки

При образовании списка ячейки, соответствующие элементам списка, объединяются в цепочку с помощью указателей CDR. Указатель CDR последнего элемента указывает на NIL. Ниже приведены примеры внутреннего представления списков.

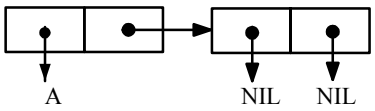


Рис.3. Структура внутреннего представления списка (A NIL)

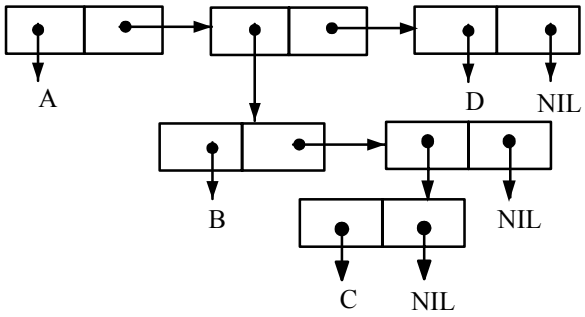


Рис.4. Структура внутреннего представления списка (A (B (C)) D)

Теперь становится ясным принцип работы описанных выше функций.

Функции CAR и CDR возвращают значение соответствующего указателя списочной ячейки, являющейся аргументом функции.

Функция CONS создает новую списочную ячейку (CONS-ячейку), устанавливает указатели CAR и CDR соответственно на первый и второй аргументы и возвращает указатель на созданную ячейку.

Действие функции (LIST A B) аналогично действию выражения (CONS A (CONS B NIL)), т.е. при выполнении функции LIST создается две новых списочных ячейки.

Функция APPEND ищет последнюю ячейку в списке (цепочке ячеек), соответствующей первому аргументу, и присваивает ее указателю CDR значение второго аргумента.

В стандартном Лиспе значение первого аргумента функции APPEND должно быть списком, в противном случае выдается соответствующее сообщение об ошибке.

Указание атома, т.е. не списочной ячейки, в качестве второго аргумента функций CONS и APPEND приведет к образованию **точечной пары**, т.е. списочной ячейки, указатель CDR которой указывает на атом.

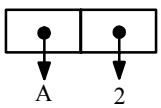


Рис.5. Списочная ячейка, представляющая точечную пару (A . 2)

Точечные пары записываются, как и списки, в круглых скобках, но элементы точечной пары разделяются символом **“.”** (точка).

Примеры:

```
$ (APPEND '(A) 'B)
(A . B)
$ (CONS '(A B) 'C)
```

```
((A B) . C)
```

1.3.6 Функции - предикаты

Функции-предикаты предназначены для проверки логических условий и в качестве значений выдают T(истина) в случае выполнения условия и NIL (ложь) в противном случае.

Функция (NULL X) оценивает аргумент и выдает T, если значением X является NIL, и NIL в противном случае. Эта функция является аналогом NOT в языке Паскаль. Примеры:

```
$ (NULL 2)
NIL
$ (NULL (CDR '(R)))
T
$ (NULL (CAR '(T))
NIL
```

Функция (ATOM X) оценивает аргумент и выдает T, если его значением является атом, и NIL в противном случае. Примеры:

```
$ (ATOM 3)
T
$ (ATOM '(A))
NIL
$ (ATOM 'R)
T
$ (ATOM (CAR '(U 1)))
T
$ (ATOM NIL)
T
```

Функция (NUMBERP X) оценивает свой аргумент и выдает значение T, если значением аргумента является число и NIL в противном случае. Примеры:

```
$ (NUMBERP 2)
T
$ (NUMBERP (+ 2 3))
T
$ (NUMBERP (CAR '(R 4)))
NIL
$ (NUMBERP '(3))
NIL
```

Функция (LISTP X) оценивает свой аргумент и выдает T, если значением аргумента является список, и NIL в противном случае. Примеры:

```
$ (LISTP '(A D))
T
$ (LISTP (CAR '(A B)))
NIL
$ (LISTP (CDR '(A B)))
T
$ (LISTP NIL)
T
```

Так как NIL в Лиспе обозначает и пустой список и символьный атом, обе функции ATOM и LISTP возвращают истину (T) для этого аргумента.

Функция (CONSP X) оценивает свой аргумент и выдает T, если значением аргумента является CONS-ячейка (список или точечная пара), и NIL в противном случае.

```
$ (CONSP '((A B) . C))
T
$ (CONSP (CAR '(A 2)))
NIL ; атом не является CONS-ячейкой
$ (CONSP (CDR '(A 2)))
T   ; список (2) является CONS-ячейкой
$ (CONSP (CDDR '(A 2)))
NIL ; NIL не является CONS-ячейкой, хотя является списком.
```

Функция сравнения двух атомов (EQ A B) оценивает аргументы и дает T, если они оба являются одним и тем же атомом.

Примеры:

```
$ (EQ 'A (CAR '(A B)))
T
$ (EQ 2 (+ 1 1))
T
$ (EQ '(A) '(A))
NIL
```

Функция EQ не позволяет сравнивать списочные ячейки, большие целые и дробные числа. Более общая функция сравнения (EQUAL A B) позволяет сравнить произвольные s-выражения и дает T, если они одинаковы и NIL в противном случае. Примеры:

```
$ (EQUAL '(A) '(A))
T
$ (EQUAL 2 (CAR '(2 U)))
T
$ (EQUAL T (ATOM 'Q))
T
```

Существует также несколько функций-предикатов для сравнения чисел, причем их отличие от остальных предикатов состоит в том, что значения аргументов должны быть числами, в противном случае выводится сообщение об ошибке.

Функция (= A B) дает T, если значения аргументов - равные числа, и NIL, если числа не равны. Примеры:

```
$ (= 4 (* 2 2))
T
$ (= 3 (CAR '(4 5)))
NIL
```

Функции (< A B) (<= A B) (> A B) (>= A B) позволяют сравнивать числа между собой. Примеры:

```
$ (<= 3 (+ 1 4))
T
$ (< 2 6)
```

```
T
$ (> 3 5)
NIL
$ (>= 5 5)
T
```

1.3.7 Логические функции AND и OR

Функция (AND $A_1...A_n$) последовательно оценивает свои аргументы, и если какой-нибудь аргумент равен NIL, то значение всей функции AND становится равным NIL, последующие аргументы при этом не оцениваются, иначе значением функции AND является значение последнего аргумента.

Функция (OR $A_1...A_n$) последовательно оценивает свои аргументы, и если какой-нибудь аргумент не равен NIL, то значением функции OR становится значение этого аргумента, а остальные аргументы не оцениваются, в противном случае значением функции OR является NIL.

Таким образом, при невыполнении условий значение функций OR и AND равно NIL (ложь), но оно не обязательно равно T (истина) при выполнении условий. Такие функции Лиспа называются *расширенными* предикатами, и этим они отличаются от аналогичных операций Паскаля.

Примеры:

```
$ (AND (ATOM 'A) 'B 2)
2
$ (OR (NULL (ATOM '(A B))) '(1 2))
T
```

1.3.8 Функции проверки условия

Функция (COND $A_1...A_n$) оценивает последовательно свои аргументы, которые должны иметь вид $A_i = (B_i\ C_i)$.

Если значение очередного B_i не равно NIL, то производится оценка C_i и его значение становится значением всей функции COND, при этом остальные пары A_i не рассматриваются. Для того, чтобы у функции COND значение всегда было определено, пара A_n обычно имеет вид $(T\ C_n)$.

Примерным аналогом COND в Паскале является оператор CASE. Выражение $(T\ C_n)$ при этом аналогично выражению ELSE C_n.

Функция (IF A B C) имеет 3 аргумента, причем A является условием, при выполнении которого (значение A не равно NIL) оценивается B, и значение B становится значением всей функции IF. Если же значение A равно NIL, значением функции IF будет оценка C. Функция IF является аналогом выражения на Паскале:

```
IF A THEN B
     ELSE C
```
Примеры:
```
$ (COND ((NULL 'A) 1) ((ATOM '(Q)) 2) (T 3))
3
$ (IF (= 5 (+ 2 3)) T NIL)
T
```
Легко заметить, что функция IF следующим образом выражается через COND:
```
(IF A B C)  <=> (COND (A B) (T C))
```

1.3.9 Функция определения новых функций DEFUN

Функция (DEFUN A B C) не оценивает своих аргументов, ее назначение - определить новую функцию, которую в дальнейшем можно будет использовать так же, как и встроенные.

A - символьный атом, являющийся именем определяемой функции;

B - список из символьных атомов, являющихся формальными параметрами функции, если их нет, необходимо поставить пустой список NIL;

C - s-выражение, являющееся телом определяемой функции.

Функция DEFUN в качестве результата возвращает имя определяемой функции.

Рассмотрим ряд простых примеров определения новых функций.

а) Вычисление квадрата функции:
```
$ (DEFUN SQR (X) (* X X))
SQR
$ (SQR 5)
25
```
б) Функция, вычисляющая сумму квадратов двух чисел:
```
$ (DEFUN SUMS (X Y) (+ (SQR X) (SQR Y)))
SUMS
$ (SUMS 3 4)
25
```
В данном определении используется ранее определенная функция SQR.

в) Вычисление второго элемента произвольного списка:
```
$ (DEFUN SECOND (X) (CAR (CDR X)))
SECOND
$ (SECOND '(1 2 3))
2
г) Функция, возвращающая минимум из двух чисел:
$ (DEFUN MIN2 (X Y) (IF (< A B) A B))
MIN2
$ (MIN2 3 8)
3
```
д) Функция, дающая в качестве значения число 2:
```
$ (DEFUN TWO NIL 2)
TWO
$ (TWO)
```

2

1.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему Лисп называют языком функционального программирования?
2. Что такое атом в Лиспе? Какие бывают атомы?
3. Как записываются списки на Лиспе? Что может быть элементом списка?
4. Что такое s-выражение?
5. По каким правилам оцениваются s-выражения?
6. Как хранятся данные в Лиспе?
7. Как записываются на Лиспе арифметические функции?
8. В чем различение между функциями SET и SETQ?
9. Каково назначение функции DEFUN?
10. Каковы особенности функций AND и OR?

1.5 УПРАЖНЕНИЯ

1 Как будет выглядеть выражение 96/(16-(18*0))+(7-8), записанное на Лиспе?

Вспомним, что бинарные арифметические функции в Лиспе представляются списками вида (<операция> <операнд1> <операнд2>). Каждый из операндов может быть либо числовой константой, либо другим s-выражением, например, еще одной арифметической функцией. Поэтому, чтобы правильно записать выражение на Лиспе, необходимо учесть приоритеты арифметических операций.

В первую очередь должна быть выполнена операция умножения, находящаяся в наиболее глубоко расположенных скобках. Соответствующая запись на Лиспе – (* 18 0).

Затем выполним вычитание, причем второй аргумент, являющийся скобкой, мы уже вычислили. Воспользовавшись этим, запишем (- 16 (* 18 0)). Для последней скобки можно сразу записать (- 7 8).

Запись для деления получаем аналогично: (/ 96 (- 16 (* 18 0))). Теперь учтем сложение и запишем окончательный вариант:

(+ (/ 96 (- 16 (* 18 0))) (- 7 8))

2 Как будут оценены интерпретатором Лиспа следующие s-выражения:

(SET (SET B 'C) 'Z)

(SETQ (SET 'A B) 'C)

Оба s-выражения являются списками. По правилам Лиспа, первый элемент оцениваемого списка задает имя функции, а остальные элементы – аргументы этой функции.

Функция SET оценивает оба своих аргумента, затем присваивает оценке первого оценку второго. Первым аргументом этой функции в нашем случае также является список. Вычислим соответствующую ему функцию.

Функция (SET B 'C) также оценивает оба своих аргумента. Первым ее аргументом является символьный атом B. Оценка символьного атома возвращает его значение. Допустим, что значение атому B еще не присвоено. В стандартном Лиспе попытка оценки такого атома привела бы к ошибке и прекращению оценки всего списка (SET (SET B 'C) 'Z). Мы примем правила системы muLisp и будем считать значением атома по умолчанию сам этот атом. Итак, в нашем случае оценкой символьного атома B является сам символьный атом B.

Запись 'C равносильна записи (QUOTE C). Функция QUOTE не оценивает свой аргумент и возвращает его в качестве значения. Таким образом, результатом оценки выражения 'C является символьный атом C.

Оценив оба аргумента функции (SET B 'C), мы можем выполнить присваивание и вычислить значение этой функции. В результате присваивания символьный атом B приобретет значение C. Значением всей функции SET является оценка ее второго аргумента, т.е. символьный атом C.

По аналогии с 'C, оценкой второго аргумента внешней функции SET - 'Z – является символьный атом Z. В результате работы этой функции символьный атом C (оценка первого аргумента) примет значение Z, и значением самой функции SET также будет символьный атом Z.

Рассмотрим оценку второго s-выражения (SETQ (SET 'A B) 'C). Это список. Символьный атом SETQ задает имя функции. Отличительной особенностью этой функции является то, что она не оценивает свой первый аргумент, поэтому оставим пока s-выражение (SET 'A B) без внимания и перейдем к оценке второго аргумента. Здесь мы видим функцию QUOTE, значением которой в данном случае является символьный атом C.

Теперь можно перейти к выполнению присваивания. Однако на этом этапе обнаруживается, что первый аргумент функции SETQ не является символьным атомом, поэтому выдается сообщение о невозможности выполнения присваивания списку.

Итак, правильный ответ:

После оценки первого выражения B=C, C=Z. При оценке второго – сообщение об ошибке “невозможно присвоить значение списку”.

Заметим, что

1) никакие действия “внутри” неоцениваемого аргумента не выполняются, соответственно, значения никаких символьных атомов не могут быть изменены, и никакие сообщения об ошибках не будут выданы, даже если выражение, соответствующее первому аргументу, записано некорректно;
2) оценивание второго аргумента функции SETQ происходит до попытки присваивания, так что, даже если присваивание окажется невозможным (как в нашем случае), никакие действия, выполненные при оценке второго аргумента, например, присваивания, не будут отменены.

3 Записать суперпозицию функций CAR и CDR, выделяющих из списка ((1 (A)) 2) символьный атом A

Атом A, который требуется извлечь, находится внутри головы исходного списка. Применим к списку функцию CAR: (CAR '((1 (A)) 2)), получим (1 (A)).

Теперь атом A находится в хвосте, а не в голове, поэтому необходимо избавиться от головы списка: (CDR '(1 (A))) даст нам список ((A)).

Далее, очевидно, необходимо дважды применить функцию CAR для извлечения атома A: (CAR '((A))) = (A), (CAR '(A)) = A.

Теперь необходимо записать эту последовательность действий в виде суперпозиции функций. Каждая последующая функция применяется к результатам работы предыдущей, поэтому упомянутые последними функции должны находиться на внешнем уровне скобок. Заметим также, что необходимо запретить оценку обрабатываемого списка, и только его, поэтому единственный апостроф (функция QUOTE) в нашей записи должен стоять перед обрабатываемым списком:

(CAR (CAR (CDR (CAR '((1 (A)) 2)))))

или в сокращенной записи

(CAADAR '((1 (A)) 2))

4 Каков результат оценки s-выражения (LIST A (SETQ A 'B) 3)

Это выражение является списком, поэтому его оценкой будет значение функции. В данном случае должна быть вычислена функция LIST от 3 аргументов.

Оценим первый аргумент функции LIST. Он является символьным атомом. Допустим, символьный атом A еще не получал значения, и примем правила системы muLisp (см. обсуждение задачи 2), тогда результатом оценки этого символьного атома будет он сам.

Второй аргумент представляет собой список, в результате оценки которого будет выполнено присваивание символьному атому A символьного атома B. Значением функции SETQ будет символьный атом B. Заметим, что поскольку оценка второго аргумента выполняется после оценки первого, то изменение значения символьного атома A никак не повлияло на оценку первого аргумента.

Третий аргумент является числовой константой, и его оценка равна самой этой константе, т.е. числу 3.

Теперь, когда оценка всех аргументов закончена без ошибок, можно вычислить значение самой функции LIST, составив список из оценок ее аргументов:

(A B 3)

5 Постройте суперпозицию функций LIST, APPEND, CONS, CAR, CDR (любых из них и в любом количестве), позволяющую построить из s-выражений (A B), (C D) список (A (B C) D)

Для построения требуемого списка воспользуемся функций LIST с тремя аргументами. В этом случае необходимо составить аргументы таким образом, чтобы их оценки были равны соответственно A, (B C) и D.

Символьный атом A получим из первого выражения с помощью функции CAR. Третий аргумент получим из второго выражения с помощью функций CAR и CDR.

Во второй аргумент входят атомы из обоих выражений. Извлечем атом B из первого выражения с помощью функций CAR и CDR, а атом C из второго с помощью функции CAR. Теперь объединим их с помощью еще одной функции LIST: (LIST (CAR (CDR '(A B))) (CAR '(C D))).

Искомая суперпозиция:

(LIST (CAR '(A B)) (LIST (CADR '(A B)) (CAR '(C D))) (CADR '(C D)))

Обратите внимание на расстановку апострофов!

Заметим, что это не единственный вариант построения суперпозиции указанных функций для получения требуемого списка из данных выражений.

6 Каков будет результат оценки системой muLisp выражения (EQ '(A B) (CONS 'A '(B)))

Функция EQ сравнивает оценки своих аргументов в предположении, что они атомы. Сначала оценим эти аргументы. Первый аргумент является списком (QUOTE (A B)), результатом его оценки является список (A B).

Для получения оценки второго аргумента необходимо вычислить функцию CONS. Эта функция строит список из головы и хвоста, т.е. вставляет результат оценки первого аргумента в

список, полученный в результате оценки второго аргумента (подробнее смотри пункт 1.3.4). В нашем случае результатом оценки этой функции является список (A B).

Очевидно, что получившиеся списки одинаковы, т.е. имеют одинаковую структуру и состоят из одних и тех же атомов, однако с точки зрения функции EQ они не равны, поскольку не являются атомами.

Таким образом, результат оценки приведенного выражения – NIL.

7 Каков будет результат оценки системой muLisp выражения (SET (OR (SETQ A NIL)(SET 'A 'B)) (AND (CAR '(A B)) A))

Оценим первый аргумент функции SET - (OR (SETQ A NIL)(SET 'A 'B)). Функция OR оценивает свои аргументы и возвращает в качестве результата первую отличную от NIL оценку или NIL, если таких нет. Оценка первого аргумента – (SETQ A NIL) – даст NIL, причем в процессе вычисления этой функции изменится значение символьного атома A. При вычислении функции SET оба аргумента берутся без оценки (точнее, оцениваются значения функций QUOTE), поэтому атом A приобретает значение B, значением функции SET и, соответственно, функции OR также будет символьный атом B.

Оценим второй аргумент функции SET- (AND (CAR '(A B)) A)). Эта функция прекращает оценку своих аргументов, как только встретился первый NIL, в противном случае значение функции равно оценке последнего аргумента. Оценкой функции CAR является отличный от NIL атом A, поэтому оценка аргументов продолжается. Оценкой символьного атома A заведомо является B, поскольку соответствующее присваивание выполнено при вычислении значения функции OR. Этот символьный атом B и станет результатом функции AND.

Таким образом, оценки обоих аргументов функции SET дали B, поэтому значением символьного атома B станет сам этот символьный атом, и результатом оценки всего s-выражения также будет символьный атом B.

8 Написать функцию, имеющую один аргумент-список и возвращающую список из 2-го и 4-го элемента, если они оба являются атомами, иначе квадрат 1-го элемента, если этот элемент - число, и NIL в противном случае.

Условие, накладываемое на аргументы, является “трехэтажным”, поэтому для программирования такого условия удобнее использовать функцию COND, чем IF. Аргументы функции COND записываются парами (<условие> <результат>), причем анализ таких пар, или веток, идет до тех пор, пока не найдется ветка с истинным условием (смотри пункт 1.3.8). Это дает возможность проверять каждое условие только один раз.

Обозначим формальный параметр нашей функции через X. 2-ой и 4-ый элементы этого списка получим с помощью функций (CADR X) и (CADDDR X) соответственно. Тогда первая

ветка функции COND будет иметь вид ((AND (ATOM (CADR X)) (ATOM (CADDDR X))) (LIST (CADR X) (CADDDR X))).

Без дополнительных проверок записываем вторую ветку COND: ((NUMBERP (CAR X)) (* (CAR X) (CAR X))).

Результат третий ветки должен становиться результатом функции во всех случаях, когда не выполнились условия первых двух веток, т.е. когда эта ветка была достигнута, поэтому условие этой ветки должно выполняться всегда. Простейший вариант записи такого условия – использование символьного атома T.

Теперь, используя синтаксис функции DEFUN, описанный в пункте 1.3.9, запишем определение нашей функции, которую назовем, к примеру, MAIN:

```
(DEFUN MAIN (X)
  (COND
    ((AND (ATOM (CADR X)) (ATOM (CADDDR X)))
      (LIST (CADR X) (CADDDR X)))
    ((NUMBERP (CAR X)) (* (CAR X) (CAR X)))
    (T NIL)
  )
)
```

1.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Как будет выглядеть выражение (7.3-6.8-(-3.4))*(92+3.4), записанное на Лиспе?
   a) (7.3-6.8-(-3.4))*(92+3.4)
   b) (* (- (-7.3 6.8) (-3.4)) (+ 92 3.4))
   c) (* (- 7.3 6.8 -3.4) (+ 92 3.4))
   d) (- (-7.3 6.8) (* (-3.4) (+ 92 3.4)))
   e) (- - * + 7.3 6.8 (-3.4) 92 3.4)
2. Как будут оценены интерпретатором Лиспа следующие s-выражения:
   (SETQ (SETQ A 2) 'Z)
   (SET (SET 'A 'B) 'Z)
   a) При оценке первого будет выдана ошибка “невозможно присвоить значение константе”. После оценки второго A=B, B=Z
   b) После оценки этих выражений A=2, B='A, Z='B
   c) При оценке первого будет выдано сообщение “невозможно присвоить значение списку”. В результате оценки второго A=B, B=Z
   d) После оценки первого A=2, затем выдается сообщение о невозможности присвоения значения константе. При оценке второго будет выдано сообщение “невозможно присвоить значение списку”.

e) Первым символьным атомам первых аргументов обоих списков будет присвоен второй аргумент, т.е. SETQ='Z, SET='Z.

3. Записать суперпозицию функций CAR и CDR, выделяющих из списка ((1 2) (A (3 4))) символьный атом A
   a) (CAADR '((1 2) (A (3 4))))
   b) (CDAAR '((1 2) (A (3 4))))
   c) (CAAAR '((1 2) (A (3 4))))
   d) (CDDDR '((1 2) (A (3 4))))
   e) (CADR '((1 2) (A (3 4))))
4. Каков результат оценки выражения (LIST (SETQ A 'B) 2 A)?
   a) (B 2 B)
   b) (B 2 A)
   c) ((SETQ A 'B) 2 A)
   d) (SETQ A 'B 2 A)
   e) (LIST (SETQ A 'B) 2 A)
5. Постройте суперпозицию функций LIST, APPEND, CONS, CAR, CDR (любых из них и в любом количестве), позволяющую построить из s-выражений (A B), C, D список (A B (C D))
   a) (CONS '(A B) (CONS 'C 'D)
   b) (LIST (A B) (APPEND 'C 'D))
   c) (APPEND '(A B) (LIST (LIST 'C 'D)))
   d) (LIST (CAR '(A B)) (CDR (A B)) (APPEND (CDR C) D))
   e) (CAR (CONS '(A B) (LIST 'C 'D)))
6. Каков будет результат оценки системой muLisp выражения (NULL (LISTP (CAR '(A B))))?
   a) T
   b) NIL
   c) Ошибка
   d) A
   e) NULL

7. Каков будет результат оценки системой muLisp выражения (AND (OR (SETQ A 5) (SETQ B A)) (SETQ C B))?
   a) T
   b) NIL
   c) A
   d) 5
   e) B
8. Напишите функцию, вычисляющую квадрат наименьшего из двух числовых аргументов a и b.
   a) (defun main a,b;
      if a<b then return a^2;
      if a>=b then return b^2;)
   b) (defun main (a b) (if (a<b) then a*a else b*b))
   c) (defun main (a b) if (< a b) (setq main (* a a) (setq main (*b b)))
   d) (defun main (a b) (cond ((< a b) (* a a)) (t (* b b))))
   e) (defun sqr (x) (* x x))
      (defun main (a b) (if (< a b) (sqr a) (sqr b))

Ответы: 1.b; 2.c; 3.a; 4.a; 5.c; 6.a; 7.e; 8.d,e.

1.7 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №1

ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ФУНКЦИЙ ЛИСПА

Необходимо выполнить по одному заданию из каждого упражнения (вариант взять у преподавателя).

Задания выполняются в среде cLisp (или другой). Рузультаты привести в отчете. В случае ошибок интерпритации – привести пояснения.

Быть готовым пояснить любой вопрос по тексту кода.

Упражнение 1

Записать следующие выражения в виде s-выражений Лиспа:

1. (2+3*6)/4/(5*(2+3))
2. (6+3*4)*(6+2)/6+6
3. 2+4.3/6.8-4/7.2+3*5
4. (7.2-3.6)/(4*5-4.8)+3.6*5
5. (3.8-2.4)/3.8+4.2*3/6.8+4
6. (5-3)/(4.8-6.2)+3.4/(7-2)+3
7. 7-6/4+3*4/(7-2)+3*4
8. (3-4.8*2)/(6*4-3.8)-(4.2-2)/(4-3)
9. (-3+4.2*6)/(7.8-3)+4.8/(3.4-2)
10. (3.4-3.2)/(-1.6)-3.8/(2-6)+4.8
11. (2.5+7)/8.1*(6.5+4.3)
12. (1.2-6.8)/3*5+(2-8)
13. (7.8-3)*(5+4)/3-1.8
14. 8.3*(1.1-1.0)+7.6/3.8
15. 6+3/7*8+(3+5)/2.3
16. (36+8)/(7+5)*4
17. (-3.6+2.4)*5/(25/5)
18. 96/(16-(18*0))+7
19. 25*25/(5*5+10*10)
20. (7.3-6.8-(-3.4))*(92+3.4)
21. (2*2+3*3)-(6.4-4.4)
22. -35+78*87/78
23. (5.5+6.6-7.7*8.8)/9.9
24. 56+65/56-25*52/4
25. 77+88-7*(7*8/4)
26. 92/4*8+89-72
27. 72/9+48*(6/3)
28. 2.5+4.6*7.4-4.8
29. 3.9+9.3/3.9*2.5
30. (88-8.8)/((77+7.7)*4.4)

Упражнение 2

Определить, как будут выполнены интерпретатором Лиспа следующие последовательности s-выражений, т.е. что появится на выходе и какие значения получат символьные атомы в результате присваиваний

1.
```
(SETQ A 3)
(SETQ B A)
 (SET (SETQ B 'A) 'B)
```

2.
```
(SETQ B 3)
(SETQ A ‘B)
(SETQ ‘(SET A 4) B)
```

3.
```
(SETQ A ‘C)
(SETQ B 2)
(SETQ (SET A B) 'C)
```

4.
```
(SET ‘B 4)
(SET A B)
(SET '(SETQ A 'B) 2)
```

5.
```
(SETQ A ‘C)
(SETQ B ‘A)
(SETQ B (SETQ B ‘C))
```

6.
```
(SETQ B 2)
(SET ‘(SET B 'C) 'Z)
```

7.
```
(SETQ 'A 4)
(SETQ ‘B ‘A)
(SETQ (SET B A) '3)
```

8.
```
(SETQ  B `A)
(SETQ  C `B)
(SET (SET B C) B)
```

9.
```
(SETQ  B `A)
(SET B (SET B ‘2))
(SETQ (SET 'A B) 'C)
```

10.
```
(SETQ A 2)
(SET ‘C `B)
(SETQ (SET ‘C 'B) A)
```

11.
```
(SETQ A ‘C)
(SET (SET A 'B) (SET ‘C B))
```

12.
```
(SETQ (SETQ A 'B) 'C)
(SET (SET A B) (SET B 'A))
```

13.
```
(SETQ B ‘C )
(SET B ‘A)
(SET B (SETQ A 'C))
```

14.
```
(SETQ B ‘A)
(SET (SETQ C B) (SETQ ‘A C))
```

15.
```
(SETQ C ‘A)
(SET C ‘B)
(SET C (SETQ ‘B ‘2))
```

Упражнение 3

Записать последовательность функций CAR и CDR, выделяющую из приведенных списков атом A.

1. (1 (2 (A) 3) 4)
2. ((1) (2 A) 3)
3. ((1 (2 (A))))
4. (1 2 (3 A))
5. (1 (2) 3 (A))
6. (1 2 ((3) A))
7. (1 2 (3 (A 4)))
8. ((1 2) (3 A) 4)
9. ((1 2) ((3 A) 4))
10. ((1) (2) (A))
11. ((1 2) (3 A))
12. ((1 2) (3 (A)))
13. (1 (2 A))
14. (1 ((2 A) 3))
15. (((A) (1 2)))
16. (3 ((A) 1 2) 3)
17. ((1 (A)) 2)
18. ((1 A) 2)
19. (1 (2) (A 3))
20. (1 (A 2) 3)
21. ((A 1) 2)
22. (1 2 ((A) 3))
23. ((1 (2) A) 3)
24. ((1 2) (A (3 4)))
25. ((A (1)) 2)
26. (1 ((2 (A)) 3))
27. ((1 (A)) 2 3)
28. (1 2 ((A) 3))
29. ((1 2) 3 (4 A))
30. (1 (2 ((A)) 3))

Упражнение 4

Каков результат интерпретации следующих выражений?

1. (CONS NIL '(A B C))
2. (CONS '(NIL) '(NIL))
3. (CONS NIL NIL)
4. (APPEND ‘(A) NIL)
5. (LIST (SETQ A ‘B) 2 ‘(A))
6. (LIST ‘A (SETQ A ‘(B)) 3 A)
7. (CAR (SET ‘A ‘(B C D)))
8. (CAR (CDR '(A (B C))))
9. (CDR (CDR ‘(CDR '(A B C))))
10. (CAR ‘(((A)) (B C)))
11. (CDR (LIST ‘A ‘B NIL))
12. (CAR "A)
13. (LIST (LIST 'A 'B) '(CAR '(C D)))
14. (LIST NIL '(A))
15. (LIST NIL NIL)
16. (APPEND (CAR '((A))) '(B (C)))
17. (APPEND (CDDR '(A B)) '(C D))
18. (APPEND NIL NIL)
19. (LIST (CDR '(A B C)) (CAR '(A B C)))
20. (CONS '(A) '(B))
21. (CONS ‘A (LIST (+ (/ 6 2) 5)))
22. (CONS ‘A ‘(+ (/ 6 2) 5))
23. (APPEND (CDR '((A) B C)) (CAR '((A) B C)))

Упражнение 5

В данном упражнении требуется получить списки справа от стрелки с помощью суперпозиции функций LIST, APPEND, CONS, CAR, CDR. Аргументами функций могут быть лишь s-выражения, указанные слева от стрелки.

Пример: (A B); C; D ⇒ (A B (C D))

Решение: (APPEND '(A B) (LIST (LIST ‘C ‘D)))

1. (A B) C (D) ⇒ ((A B) (C) D)
2. (A B) (C); (D) ⇒ ((A B) C D)
3. (A B) (C D) ⇒ (A (B C) D)
4. (A); (B); (C); (D) ⇒ ((A B) (C D))
5. (A); B; C; D ⇒ (A B (C) (D))
6. (A); B; C; D ⇒ (((A) (B)) C D)
7. A; B; (C D) ⇒ (A (B C) D)
8. A; (B C); D ⇒ ((A) (B) C (D))
9. (A); (B); (C D) ⇒ (A (B C) D)
10. (A B); (C (D)) ⇒ (A (B (C)) D)
11. (A B); (C D) ⇒ ((A) B C (D))
12. A; B; (C D) ⇒ ((A B) C D)
13. A; (B C D) ⇒ ((A) B (C D))
14. A; (B) ; C; (D) ⇒ ((A) B (C D))
15. A; (B C); (D) ⇒ ((A B) (C) D)
16. (A); B; (C D) ⇒ (A (B) (C) D)
17. A; (B C); (D) ⇒ (A (B) (C D))
18. (A); (B C); D ⇒ (A (B (C) D))

Упражнение 6

Какое из выражений дает значение T, а какое NIL?

1. (ATOM '(CAR '(A B)))
2. (EQUAL '(A B) (CONS '(A) '(B)))
3. (EQ '(A B) (CONS 'A '(B)))
4. (EQ 1234567890 1234567890)
5. (EQUAL ‘1234567890 1234567890)
6. (EQ 5 20/4)
7. (EQ 1/5 0.2)
8. (ATOM (* 2 (+ 2 3)))
9. (NULL (NULL T))
10. (EQUAL (ATOM NIL) (CAAR '((T))))
11. (ATOM (SETQ A 'B))
12. (NULL (LISTP (CAR '(A B))))
13. (NUMBERP '(+ 2 3))
14. (LISTP (CAR '(A)))
15. (LISTP (EQ 'A 'B))
16. (ATOM (CAR '(SETQ A 'B)))
17. (NULL (LISTP NIL))
18. (NUMBERP ‘(SETQ A 4))
19. (ATOM '(CAR '(A)))
20. (ATOM ‘(LISTP ‘(A)))
21. (ATOM (LISTP ‘(A)))
22. (CONSP (CONS ‘A ‘B))
23. (CONSP (CAR ‘(A B)))
24. (CONSP (CDR ‘(A)))

25. (EQ (SETQ A 'B)A)
26. (EQ (SETQ A 'B)(CAR '(A X)))
27. (EQ (SETQ A 'B)(SETQ A 'C))
28. (EQ (SETQ A 'B)(SET 'A A))
29. (ATOM (CAR '(CAR '((A) B))))
30. (CAR (LIST NIL)))

Упражнение 7

Чему равны значения следующих выражений?

1. (AND (CDDR '(A B)) 'A 2)
2. (OR (CAR '(NIL 2 'A)) (ATOM '(A B)))
3. (OR (SET 'A NIL) (AND (NULL A) (CDR '(A))))
4. (AND (> 5 3) (< (SETQ A 1) 0))
5. (OR (SETQ A NIL) (CAR '(A B)))
6. (AND (SETQ A '5) (SETQ 'C (= A 4)))
7. (OR '(A 4) (SETQ C (A 4)))
8. (OR (SETQ C (= 4 8)) (CAR '(NIL C)))
9. (AND (OR (SETQ A 5) (SETQ B A)) (SETQ C B))
10. (OR (ATOM (< 5 3)) (SETQ C '8))
11. (OR (SETQ A ‘B) (SETQ ‘B C))
12. (AND (SETQ B ‘A) (EQ A B))
13. (SET (OR (SETQ A NIL)(SET 'A 'B)) (AND(CAR '(A B)) A))
14. (AND (OR '(SETQ A NIL) NIL)(OR NIL A))
15. (AND (EQUAL (SETQ A NIL)NIL) (EQ (OR NIL A 2) 2))
16. (AND (NULL (SETQ A NIL)) (EQ A (ATOM '(A B)))
17. (OR(APPEND NIL NIL) (CONS NIL NIL)2)
18. (AND (LIST NIL NIL) (CAR (LIST NIL)) (CONS NIL NIL))
19. (SETQ (OR (SETQ A NIL) (SETQ A 'B) (SETQ B 'C))(SET B 2))
20. (OR (CDR '(NIL)) (CAR '(NIL)) (LIST NIL))

Упражнение 8

Записать определения следующих функций, используя функции проверки условия IF и COND (т.е. 2 варианта!):

1. Функция, вычисляющая выражение 1/a + 1/b для заданных чисел a и b. В случае, если какое-либо из чисел равно нулю, функция должна давать нуль.
2. Функция, вычисляющая максимум из трех чисел a, b, c.
3. Функция, вычисляющая квадрат наименьшего из двух чисел a и b.
4. Функция, имеющая один аргумент-список и возвращающая квадрат третьего элемента этого списка, если он является числом, голову третьего элемента, если он является списком, иначе сам третий элемент.
5. Функция, имеющая один аргумент-список и возвращающая сумму 4-го и 5-го элемента этого списка, если оба этих элемента являются числами, иначе 1-й элемент списка.
6. Функция, имеющая один аргумент-список и возвращающая квадрат первого элемента этого списка, если этот элемент является числом, и 3-й элемент списка в противном случае.
7. Функция, имеющая один аргумент-список и возвращающая список из 2-го и 4-го элемента, если они оба являются атомами, иначе квадрат 1-го элемента, если этот элемент - число, и NIL в противном случае.

8. Функция, имеющая два аргумента - числовых списка и возвращающая тот список, голова которого меньше.

9. Функция, аргументами которой являются координаты двух точек (x_1 y_1) (x_2 y_2), а результатом - расстояние межу точками

10. Функция, имеющая три аргумента и возвращающая их сумму, если все они числа, и список, состоящих из них, в противном случае.

11. Функция, аргументом которой является числовой список, а результатом максимум из квадрата головы и суммы второго и третьего элементов.

12. Функция, аргументом которой является некоторый список, а результатом голова этого списка, если в списке больше трех элементов, иначе второй элемент списка.

13. Аргументом функции является список из трех неравных чисел. Функция должна вернуть число, которое не является максимумом и минимумом.

14. Функция имеет 2числовых аргумента a и b описывающих уравнение ax+b=0. Значением функции должен быть корень уравнения.

15. Аргументом функции является список из трех чисел. Функция должна вернуть список из тех же чисел, расположенных в порядке возрастания.

2 РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

2.1 ПОНЯТИЕ РЕКУРСИИ

Очень часто в программировании необходимо повторить некоторую последовательность действий несколько раз. Один из приемов, позволяющих это сделать, - использование циклов. Второй способ - **рекурсивный вызов** функций. Рекурсивной называется функция, прямо или через обращение к другой функции вызывающая сама себя.

В любом цикле можно выделить условие окончания цикла и тело цикла. В рекурсивной функции также присутствуют две составляющие: терминальная ветвь (условие) и рекурсивная. Результат рекурсивной ветви формируется с использованием рекурсивных вызовов, в терминальной ветви функция не вызывает сама себя.

Как в цикле может быть несколько условий окончания, так и в рекурсивной функции может быть несколько терминальных ветвей. В теле цикла в зависимости от ситуации могут выполняться различные действия (например, для четного и нечетного значения счетчика). Рекурсивных ветвей также может быть несколько.

Рассмотрим пример. Пусть мы хотим определить функцию, вычисляющую последний элемент заданного списка. Эта задача сложнее, чем вычислить первый или второй элемент, поскольку список может содержать произвольное число элементов. Однако рекурсивно такую функцию определить очень просто:

1. Если список состоит ровно из одного элемента, то его голова равна последнему элементу списка. Это условие соответствует терминальной ветви рекурсивной функции.

2. В случае нескольких элементов последний элемент списка совпадает с последним элементом списка, у которого удалена голова. Это рекурсивная ветвь функции.

Проверить, что список состоит из одного элемента, можно, взяв от него хвост, который должен быть пустым. Теперь мы можем записать определение рекурсивной функции:

```
(DEFUN LAST (X)
  (COND
    ((NULL (CDR X)) (CAR X))    ; терминальная ветвь
    (T (LAST (CDR X)))))        ; рекурсивная ветвь
```

Обратите внимание, что терминальная ветвь в записи функции обязательно должна предшествовать рекурсивной.

Еще один пример. Определим рекурсивно вычисление факториала натурального числа.

1. Факториал нуля равен единице.

2. Факториал для n>0 равен факториалу числа на единицу меньше, умноженному на n.

Запишем функцию:

```
(DEFUN F (N)
(IF (= N 0) 1
    (* (F (- N 1)) N)))
```

Заметьте, что в рекурсивной функции значения параметров не меняются, поскольку нет ни одной операции присваивания. Эффект изменения значений параметров достигается при вычислении параметров рекурсивного вызова. Например, в функции факториала мы не изменяем значение параметра N, но вызываем функцию F с аргументом (- N 1), т.е. на единицу меньшим текущего значения N.

2.2 ТИПЫ РЕКУРСИИ

В случае, когда требуется обработка *элементов* списка (на верхнем уровне скобок) принцип рекурсии следующий: список разбивается на голову и хвост, голова обрабатывается непосредственно, а к хвосту рекурсивно применяется определяемая функция. Такой принцип называется **частной** или **хвостовой рекурсией**. Поскольку последовательное взятие хвостов приводит в итоге к пустому списку, выбор условия терминальной ветви очевиден.

Рассмотрим примеры:

а) В произвольном списке вычислить количество его элементов.

```
(A (B B) (D(1)))  ⇒ 3
```

В пустом списке содержится нуль элементов, а в непустом на единицу больше, чем в списке без головы:

```
(DEFUN KOLELEM (X)
(IF (NULL X) 0                    ; терминальная ветвь
    (+ 1 (KOLELEM (CDR X)))))     ; рекурсивная ветвь
```

б) Дан числовой список. Каждый элемент этого списка увеличить на единицу.

Возьмем голову списка, увеличим ее на единицу, к хвосту применим определяемую функцию и поставим измененную голову на место. Предельный случай- пустой список, результат применения функции к которому должен быть равен NIL.

```
(DEFUN PLUS1 (X)
  (COND
    ((NULL X) NIL)
    (T (CONS (+ 1 (CAR X)) (PLUS1 (CDR X))))))
```

В случае, когда требуется обработать каждый отдельный *атом* списка, этот список также разбивается на голову и хвост, но функция применяется как к голове списка, так и к хвосту. Такой принцип называется **общей рекурсией**. При этом, поскольку последовательное взятие хвостов приводит к пустому списку, а взятие голов приводит к атому, следует рассматривать оба эти предельных случая в качестве терминальных ветвей. Рассмотрим примеры.

а) В произвольном списке вычислить количество отличных от NIL атомов.

```
(A (B 1) ((C) 2 3)) ⇒ 6
```

Количество атомов любого списка равно сумме количества атомов в его голове и количества атомов в его хвосте.

```
(DEFUN KOLATOM (X)
  (COND
    ((NULL X) 0       ; терминальная ветвь - пустой список
    ((ATOM X) 1)      ; терминальная ветвь – атом
    (T (+ (KOLATOM (CAR X))
          (KOLATOM (CDR X))))))   ; рекурсивная ветвь
```

Обратите внимание, что условие (NULL X) предшествует условию (ATOM X), т.к. пустой список NIL может рассматриваться и как символьный атом NIL, поэтому при записи (COND ((ATOM X) 1) ((NULL X) 0) ...) функция KOLATOM никогда не вернула бы 0.

б) В произвольном списке, состоящем из чисел, требуется каждый атом увеличить на единицу.

```
(1 (2 3) ((4))) ⇒ (2 (3 4) ((5)))
```

Определяемую функцию PLUSATOM применяем к голове и хвосту, функцией CONS объединяем результаты действия функции PLUSATOM, а в предельном случае, когда получаем атом, его и увеличиваем на единицу.

```
(DEFUN PLUSATOM (X)
  (COND
    ((NULL X) NIL)
    ((ATOM X)(+ X 1))
    (T (CONS (PLUSATOM (CAR X)) (PLUSATOM (CDR X))))))
```

Рассмотрим еще один принцип рекурсивного программирования, который называется **методом накапливающихся параметров**. В функцию добавляется параметр, который рекурсивно меняется и позволяет в предельном случае получить результат. Этот параметр не является исходным данным, а вводится искусственно и по назначению соответствует промежуточным переменным в языке Паскаль.

Рассмотрим пример. Дан числовой список, требуется найти сумму его элементов. Сначала приведем решение методом хвостовой рекурсии.

```
(DEFUN SUMELEM (X)
  (IF (NULL X) 0
      (+ (CAR X) (SUMELEM (CDR X)))))
```

А теперь решим эту задачу методом накапливающихся параметров, для этого используем вспомогательную функцию, второй параметр которой дает окончательный ответ в предельном случае.

```
(DEFUN SUMELEM (X) (SUMELEM1 X 0))
(DEFUN SUMELEM1 (X N)
  (IF (NULL X) N
    (SUMELEM1 (CDR X) (+ N (CAR X)))))
```

Для некоторых задач метод накапливающихся параметров позволяет значительно упростить решение. Пример: дан список из чисел, первый элемент увеличить на 1, второй на 2 и т.д. до последнего элемента.

Оказывается, проще решить более общую задачу: первый элемент увеличить на k, второй элемент на k+1 и т.д., передавая число, на которое увеличивается голова списка, в качестве накапливающего параметра.

```
(DEFUN F1 (X K)
(IF (NULL X) NIL
    (CONS (+ K (CAR X)) (F1 (CDR X) (+ K 1)))))
```

Теперь исходная задача решается тривиально:

```
(DEFUN F (X) (F1 X 1))
```

Рассмотрим еще один пример применения метода накапливающихся параметров. Дан список, переставить его элементы в обратном порядке:

```
(A (B C) (D (E)))    ⇒    ((D (E)) (B C) A)
```

Основная функция будет иметь вид:

```
(DEFUN REVERSE (X) (REVERSE1 X NIL))
```

Функция REVERSE1(A B) рекурсивно убирает голову из списка A и добавляет его в качестве головы списка B, поэтому, когда A оказывается пустым, второй параметр содержит результат функции:

```
(DEFUN REVERSE1 (A B)
  (IF (NULL A) B
    (REVERSE1 (CDR A) (CONS (CAR A) B))))
```

Рассмотрим пример работы функции REVERSE1. Пусть исходный список A имеет вид (1 2 3). Тогда при последовательных вызовах функции REVERSE1 аргументы A и B получат следующие значения:

Номер итерации	A	B	
1	(1 2 3)	()	- начальное значение
2	(2 3)	(1)	
3	(3)	(2 1)	
4	()	(3 2 1)	- результат

Обобщим задачу. Пусть требуется выполнить перестановку атомов списка на всех уровнях.

```
(A (B C) (D (E))) ⇒ (((E) D) (C B) A)
```

Для этого используем метод, который называется **взаимной рекурсией**. Главная функция определяется следующим образом:

```
(DEFUN REVERSE (X)
  (COND ((ATOM X) X) (T REVERSE1 X NIL))))
```

В функции REVERSE добавлена проверка на атом, поскольку к ней будет производиться обращение из функции REVERSE1 с аргументом - символьным атомом.

```
(DEFUN REVERSE1 (A B)
  (COND
    ((NULL A) B)
    (T (REVERSE1 (CDR A)
    (CONS (REVERSE (CAR A)) B)))))
```

Функция REVERSE1 действует так же, как и в предыдущем случае, но вставляет "перевернутую голову", обращаясь к функции REVERSE, которая *по своему определению* и предназначена для переворачивания списка. Таким образом, функции REVERSE и REVERSE1 взаимно вызывают друг друга.

2.3 ОТЛИЧИЯ ТИПОВ РЕКУРСИИ

Мы познакомились с тремя типами рекурсии. Необходимо отличать их друг от друга, поскольку правильное определение типа рекурсии позволяет выбрать наиболее подходящий конкретной задаче метод решения и, в конечном итоге, улучшить качество разрабатываемого программного обеспечения.

Проще всего распознать взаимную рекурсию. Для этого достаточно обнаружить кольцо из двух или более функций, вызывающих друг друга. Например, даже не вникая в смысл приведенной ниже программы, можно установить, что в ней используется взаимная рекурсия. Во взаимных вызовах участвуют все 4 функции:

```
(DEFUN F1(X)
  (IF (ATOM X) X ELSE (F2 X))
(DEFUN F2(X)
  (CONS (F4 (CAR X)) (F3 (CDR X))))
(DEFUN F3(X)
  (IF (NULL X) X (CONS (F4 (CAR X)) (F3 (CDR X)))))
(DEFUN F4(X)
  (F1 X))
```

Труднее различить частную и общую рекурсии. Для этого необходимо выяснить, как именно организованы вычисления в рекурсивной ветви. Если от параметра отделяется часть и обрабатывается отдельно, а в единственный рекурсивный вызов передается оставшаяся часть аргумента, то это несомненно частная рекурсия. Если же параметр разделяется каким-либо способом на несколько (две и более) частей, и функция рекурсивно применяется ко *всем* частям, то мы имеем дело с общей рекурсией.

Как различить частную (хвостовую) и общую рекурсии при обработке списков, видно из рассмотренных в предыдущем пункте примеров. Приведем пример функций с общей и частной рекурсией, обрабатывающих числа.

Пусть необходимо вычислить N-ое число Фибоначчи. 1-е и 2-е числа Фибоначчи равны 1, все остальные – сумме двух предыдущих чисел Фибоначчи. Записав это определение на Лиспе, получим:

```
(DEFUN FIB(N)
  (IF (OR (= N 1) (= N 2)) 1
      (+ (FIB (- N 1)) (FIB (- N 2)))))
```

Легко видеть, что в рекурсивной ветви функция FIB вызывается дважды с различными аргументами. Исходя из этого, можно сделать вывод, что в функции FIB применяется общая рекурсия.

Рассмотрим функцию, $f(N)=\sum_{i=1}^{N} i^2$. Для ее реализации на Лиспе запрограммируем дополнительную функцию возведения в квадрат. Наша программа будет иметь вид:

```
(DEFUN SQR(X) (* X X))
(DEFUN F(N) (F1 N 1))
(DEFUN F1(N I)
  (IF (> I N) 0
      (+ (SQR I) (F1 N (+ I 1)))))
```

В рекурсивной ветви функции F параметр I использован дважды, но лишь один раз в рекурсивном вызове, поэтому можно сделать вывод, что в функции F использована частная рекурсия.

2.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие функции называются рекурсивными?
2. Каково назначение терминальных ветвей в рекурсивных функциях?
3. Сколько рекурсивных ветвей может иметь функция?
4. Какие типы рекурсии вы знаете?
5. Может ли в одной отдельно взятой функции использоваться взаимная рекурсия?
6. Чем отличается частная рекурсия от общей?
7. В чем особенности метода накапливающегося параметра?
8. С какими типами рекурсии может использоваться метод накапливающегося параметра?
9. Почему в рекурсивных функциях терминальные ветви должны предшествовать рекурсивным?
10. Каково назначение дополнительного параметра при использовании метода накапливающегося параметра?

2.5 УПРАЖНЕНИЯ

1 Написать функцию, вычисляющую сумму всех чисел в числовом списке произвольной структуры

Для обработки всех атомов в списке произвольной структуры нам необходимо анализировать *внутреннюю структуру* каждого элемента списка. Для этого нужно применять нашу функцию рекурсивно к голове и хвосту списка, то есть использовать *общую рекурсию*.

Выбрав тип рекурсии, определим терминальные ветви. При обработке списка их как минимум две: пустой список и атом. Поскольку в условии сказано, что список состоит только из чисел и не содержит символьных атомов, то необходимости в проверке типа атома нет. Теперь мы можем записать "скелет" нашей функции:

```
(DEFUN MYFUN(X)
  (COND
    ((NULL X) …)
    ((ATOM X) …)
    (T …)
))
```

Заполним пустые места в терминальных ветвях. Пустой список не содержит элементов, значит, сумма элементов пустого списка равна 0. Сумма одного числа, очевидно, равна самому этому числу. Получаем:

```
(DEFUN MYFUN(X)
  (COND
    ((NULL X) 0)
    ((ATOM X) X)
    (T …)
))
```

Теперь напишем рекурсивную ветвь. Сумма элементов списка равна сумме элементов головы и хвоста. Заметим, что поскольку наша функция корректно обрабатывает аргумент-число, то проверки, является ли голова списка списком или атомом, делать не нужно. Теперь мы можем записать окончательный вариант:

```
(DEFUN MYFUN(X)
  (COND
    ((NULL X) 0)
    ((ATOM X) X)
    (T (+ (MYFUN (CAR X)) (MYFUN (CDR X))))
))
```

Существует еще один способ обработки всех элементов списка произвольной структуры. Каждый раз, когда голова списка оказывается списком, мы можем понизить ее уровень вложенности, просто соединив ее с исходным списком с помощью функции APPEND. Если же голова оказывается числом, то мы ее обрабатываем так же, как в предыдущей программе.

Поясним эту технологию на примере. Пусть необходимо вычислить сумму элементов списка ((1 2) 3 ((4))). Головой этого списка является список (1 2). Объединив его с хвостом списка, получим новый список из тех же чисел: (1 2 3 ((4))). Последующие три попытки взятия головы приведут нас к числам, а к списку ((4)) процедуру понижения вложенности придется применить дважды. Таким образом, последовательность вычислений будет такой:

```
((1 2) 3 ((4))) ⇒ (1 2 3 ((4))) ⇒ 1+(2 3 ((4))) ⇒
1+2+(3 ((4))) ⇒ 1+2+3+(((4))) ⇒1+2+3+((4)) ⇒1+2+3+(4) ⇒
1+2+3+4+() ⇒1+2+3+4+0 ⇒ 10
```

Напишем функцию MYFUN2, реализующую такой алгоритм обработки:

```
(DEFUN MYFUN2(X)
(COND
  ((NULL X) 0)
  ((ATOM (CAR X))(+ (CAR X) (MYFUN2 (CDR X))))
  (T (MYFUN2 (APPEND (CAR X) (CDR X))))
))
```

Заметим, что первая функция корректно обработает аргумент-число, а вторая будет работать *только* со списками.

2 Написать функцию, удаляющую все числовые константы из одноуровневого списка

В этой задаче нас не интересует внутренняя структура элементов списка, нам достаточно знать, является ли элемент числом или нет. Для решения такой задачи достаточно использования *хвостовой рекурсии*.

Терминальным случаем для нашей функции будет пустой список. В этом случае значение функции также будет равно пустому списку. Рекурсивных ветви должно быть две: для удаления головы списка и для ее сохранения. В обеих ветвях для обработки остальных элементов рекурсивно вызывается наша функция с хвостом списка.

```
(defun myfun(x)
(cond
  ((null x) nil)
  ((numberp (car x)) (myfun (cdr x)))     ; удаление головы
  (t (cons (car x) (myfun (cdr x))))
))
```

3 Написать функцию, проверяющую, является ли число простым

Простым называется число, не имеющее делителей, кроме 1 и самого этого числа. Для проверки, является ли число X простым, необходимо проверить все числа в диапазоне от 2 до X-1. Если среди этих чисел найдется хотя бы одно число, на которое X делится нацело, то число X не является простым, и остальные делители можно не проверять.

Пусть текущий проверяемый делитель равен I. Если остаток от деления X на I равен 0, то можно сразу сказать, что значением функции является NIL. Если число не принадлежит указанному диапазону, то перебор закончен, и значение функции равно T. В противном случае

необходимо проверить следующее число из диапазона. Последнее предложение описывает рекурсивную ветвь нашей функции, два предыдущих – терминальные. Мы видим, что в рекурсивной ветви наша функция будет вызываться только один раз, поэтому вся функция должна быть написана по схеме хвостовой, или *частной*, рекурсии.

Функция, которую нам нужно написать, имеет только один параметр – исследуемое число N, но для проведения проверок нам необходимо передавать и текущее значение числа из проверяемого диапазона, поэтому мы должны написать две функции: рекурсивную функцию проверки и вызывающую ее нерекурсивную функцию.

Запишем функции:

```
(DEFUN MYMAIN1 (X) (MYFUN1 X 2))
(DEFUN MYFUN1 (X I)
(COND
  ((>= I X) T)
  ((= (MOD X I) 0) NIL)
  (T (MYFUN1 X (+ I 1)))
))
```

В функции MYFUN1 делители перебираются в порядке возрастания. Можно написать еще один вариант, в котором числа будут перебираться в убывающем порядке.

```
(DEFUN MYMAIN2 (X) (MYFUN2 X (- X 1)))
(DEFUN MYFUN2 (X I)
(COND
  ((<= I 2) T)
  ((= (MOD X I) 0) NIL)
  (T (MYFUN2 X (- I 1)))
))
```

Результаты работы функций MYMAIN1 и MYMAIN2 будут одинаковы, но поскольку вероятнее обнаружить делители среди малых чисел (например, 2 является делителем всех четных чисел), то первая функция вернет результат быстрее второй.

2.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Функция на Лиспе, реализующая вычисление суммы элементов одноуровневого числового списка выглядит следующим образом:

   a)
   ```
   (defun sumel(x)
   (if (null x) 0
      (+ (car x) (sumel (cdr x)))
   ))
   ```

   b)
   ```
   (defun sumel(x)
   (if (null (cdr x)) (car x)
      (+ (sumel (car x)) (sumel (cdr x)))
   ))
   ```

   c)
   ```
   (defun sumel(x)
   (cond
   ```

```
    ((null x) 0)
    ((cdr x)
      (sumel (cons (+ (car x) (cadr x)) (cddr x))))
    (t (car x))
   ))
```

d)
```
(defun sumel(x)
   (if x (+ (sumel (cdr x)) (car x)) 0
   ))
```

e)
```
 (defun sumel(x)
   (if (> (sumel (cdr x)) 0) (+ (car x) (sumel (cdr x)))
      (car x)
   ))
```

2. Функция поиска максимального элемента в непустом числовом списке произвольной структуры выглядит следующим образом:

a)
```
(defun mx (x)
   (cond
    ((null (cdr x)) (car x))
    ((> (car x) (cadr x)) (mx (cons (car x) (cddr x))))
    (t (mx (cdr x)))
   ))
```

b)
```
(defun mx (x)
   (cond
    ((null (cdr x)) (mx (car x)))
    ((atom x) x)
    ((> (mx (car x)) (mx (cdr x))) (mx (car x)))
    (t (mx (cdr x)))
   ))
```

c)
```
(defun mx (x)
   (cond
    ((null (cdr x)) (mx (car x)))
    ((> (mx (car x)) (mx (cdr x))) (mx (car x)))
    (t (mx (cdr x)))
   ))
```

d)
```
(defun mx (x)
   (cond
    ((null (cdr x)) (mx (car x)))
    ((> (car x) (mx (cdr x))) (car x))
    (t (mx (cdr x)))
   ))
```

e)
```
(defun mx (x)
   (cond
    ((null (cdr x)) (mx (car x)))
    ((atom x) x)
    ((atom (car x))
      (if (> (car x) (mx (cdr x))) (car x) (mx (cdr x))))
    (t (mx (append (car x) (cdr x))))
   ))
```

3. Для того чтобы исправить ошибку в приведенной ниже функции необходимо:

```
(DEFUN FUN (N M)
  (*                          ; строка 1
    (FUN (- N 1) M)           ; строка 2
```

```
          (IF (= N 0) 1        ; строка 3
              M))              ; строка 4
)
```

a) поставить строку 2 на последнее место

b) поставить строку 3 на первое место

c) изменить условие в строке 3 на (= N 1)

d) заменить в строке 2 выражение (- N 1) на (+ N 1)

e) заменить строку 2 на (FUN (- N 1) (- M 1)).

4. Какие методы использованы в приведенной функции FUN

```
(DEFUN FUN (X N)
  (COND
    ((NULL X) NIL)
    ((<= N 0) NIL)
    (T (CONS (CAR X) (FUN (CDR X) (- N 1))))
))
```

a) частная рекурсия

b) частная рекурсия и метод накапливающегося параметра

c) общая рекурсия

d) общая рекурсия и метод накапливающегося параметра

e) взаимная рекурсия

5. Сколько раз будет вызвана функция F1 при оценке s-выражения (F1 '(1 2 (3 4) ((5)))?

```
(DEFUN F2(X)
  (IF (ATOM X) (LIST X) (F1 X)))
(DEFUN F1(X)
  (IF (NULL X) NIL
    (CONS (F2 (CAR X)) (F1 (CDR X)))))
```

a) 12

b) 8

c) 5

d) 4

e) 20

Ответы: 1.a,c,d; 2.b,e; 3b; 4.a; 5.a.

2.7 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №2

МЕТОДЫ ХВОСТОВОЙ РЕКУРСИИ ДЛЯ ОБРАБОТКИ СПИСКОВ

Задание: Реализовать обработку списков с использованием механизма общей рекурсии.

Задания выполняются в среде cLisp (или другой). Код программ и тестовые примеры привести в отчете.

Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

1. Написать функцию вычисления m^n.
2. Для произвольного списка определить, является ли последний элемент списка ком или атомом.
3. Получить список, состоящий из поэлементной суммы двух списков.
   (2 4 1 5 6) (3 4 1 2 2) —> (5 8 2 7 8)
4. Написать функцию-предикат, проверяющую, принадлежит ли данный элемент нному списку:
   (MEMBER 'A '(S D A H)) —> T
   (MEMBER 'D '(H K (D) G)) —> NIL
5. Для произвольного списка построить список той же длины, состоящий из атомов NIL в зависимости от того, является ли соответствующий элемент исходного ка списком или атомом соответственно.
6. Написать функцию, которая по заданному s-выражению A и числу N строит ок вида ((((((A)))))), где количество скобок с каждой стороны от выражения равно

7. Написать функцию, преобразующую список к виду
   (A B C) —> (A (B (C)))
8. Определить функцию, разбивающую список на пары:
   (A B C D) —> ((A B) (C D))
9. Написать функцию, которая, чередуя элементы двух списков, строит новый ок:
   (A B C) (1 2 3) —> (A 1 B 2 C 3)
10. Определить функцию, которая строит список (1 3 5 7 ... 2*N-1) для заданного N.
11. Для заданного числового списка построить новый список, в котором вначале дятся все отрицательные элементы, а затем неотрицательные.
12. Два множества представлены списками. Найти пересечение множеств.
13. Два множества представлены списками. Найти вычитание множеств.
14. Два множества представлены списками. Найти объединение множеств.
15. Дан список из положительных чисел, построить список, элементами которого ются количество единиц в исходном списке, количество двоек и т.д. до имального значения числа в исходном списке:
    (2 1 1 3 1 3 5) —> (3 1 2 0 1)

16. Из данного списка удалить все элементы, которые являются списками длиной
е K.
17. Выполнить попарные перестановки элементов списка
(A B D F B A C) —> (B A F D A B C)
18. Дан список, число и произвольное s-выражение. Поставить s-выражение в список
д элементом, номер которого равен числу:
(A B C D), T, 3 —> (A B T C D)
19. Написать функцию, превращающую список в множество, т.е. удаляющую все
повторяющиеся элементы:
(A B A B B C A) —> (A B C)
20. Написать функцию, проверяющую, является ли список множеством, т.е. каждый
ент в нем должен встречаться по одному разу.
21. Проверить, состоят ли два списка из одних и тех же элементов.
22. Дан список и число N. Разбить список на подсписки длины N:
(A B (C D) 1 2 NIL) 4 —> ((A B (C D) 1) (2 NIL))
23. Разложить число на простые множители.—>
30 —> (2 3 5)
24. Дан одноуровневый список, состоящий из чисел. Построить список из
едовательных сумм элементов.
(A1,A2,A3,...,AN) —>—>(A1,A1+A2,A1+A2+A3,...,A1+A2+...+AN)
25. Для двухуровневого списка вернуть подсписок максимальной длинны.
26. Для одноуровневого списка удалить элемент, стоящий следом за заданным
(A B A B B C A), A —>(A A B C A)
27. Написать функцию, выполняющую над числовым одноуровневым списком
ующую операцию:
(A B C ...) —> A^B^C...
Причем последовательное возведение в степень выполняется справа налево.
28. Написать функцию, которая для заданных произвольного списка и числового
ит список вида:
(a c g) (2 3 5) —> (((a)) (((c))) (((((g)))))
29. Написать функцию, вычисляющую произведение всех элементов одноуровневого
ового списка в предположении, что отсутствует операция умножения и можно
льзовать лишь сложение.
30. Из произвольного списка и числового списка построить новый список:
(A B C) (1 2 3) —> ((A) (B B) (C C C))
31. Даны два множества, представленные списками. Проверить, является ли первое
кество подмножеством второго.
32. Построить список, определяющий сколько раз встречается каждый элемент в
ке: (A B A C B C A B D) —> ((A 3) (B 3) (C 2) (D 1))
33. Даны два числовых списка. Построить третий список, количество элементов
рого равно длине второго списка и каждый элемент равен количеству элементов
ого списка, меньших соответствующего элемента второго списка:
(1 7 3 0 2) (3 7 10) —> (3 4 5)
34. Написать функцию, удаляющую из списка каждый K-й элемент.
35. Разбить список на подсписки длинны 1, 2, 3…

36. Дан список и числа N и M. Вырезать из списка подсписок от элемента N до ента M:
    (A B C D E) 2 4 —> (B C D)
37. Даны два множества, представленные списками. Определить функцию, строящую ртово произведение множеств:
    (A B C) (1 2) —> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
38. Дан список из произвольных элементов. Найти наиболее часто встречающийся ент.
39. Написать функцию, вычисляющую множество всех подмножеств данного кества:
    (A B C) —> (NIL (A) (B) (C) (A B) (A C) (B C) (A B C))
40. Дано множество A и список из подмножеств данного множества B. Разбить A на подмножества так, чтобы ни одно из подмножеств B не содержалось целиком в ом из двух подмножеств. Например:
    (A B C D E) ((A B) (A C D) (D E)) —> ((A C E) (B D))
41. Дано натуральное число N. Построить список из подсписков натуральных чисел, в е дающих N. Пример:
    3 —> ((1 2) (2 1) (1 1 1))

2.8 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №3

МЕТОДЫ ОБЩЕЙ РЕКУРСИИ ДЛЯ ОБРАБОТКИ СПИСКОВ

Задание: Реализовать обработку списков с использованием механизма общей рекурсии.

Задания выполняются в среде cLisp (или другой). Код программ и тестовые примеры привести в отчете.

Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

1. Написать функцию-предикат, проверяющую, содержится ли данный атом в списке:
   (MEMQ 'A '(S W F (A S))) —> T
   (MEMQ 'U '(G J (J))) —> NIL
2. Написать функцию, которая из заданного списка строит одноуровневый список:
   (A (B (C 1) D)) —> (A B C 1 D)
3. Для произвольного списка вычислить количество чисел на заданной глубине
   ((1 A) ((3 4) (D C)) (2) (A)), 2 –> 2
4. Найти максимальный уровень вложенности заданного списка.
5. Найти последний отличный от NIL атом в заданном списке:
   (A B (S 1) ((C)) NIL) —> C

6. Дан произвольный список из числовых атомов. Найти минимальный атом в списке.
7. Для произвольного списка найти глубину, на которой находится первое вхождение заданного атома

   ((1 A) ((3 4) (D C)) (2) (A)), D –> 2

8. Найти первый отличный от NIL атом в произвольном списке:

   ((() A) B C) –> A
9. Даны два списка одинаковой структуры. Построить список такой же структуры, состоящий из пар элементов.

   ((A B(B)C)D) , ((1 C (2)3)4) —>—>(((A 1) (B C)((B 2))(C 3))(D 4))
10. Дан список произвольной структуры, в который входят как числа, так и символьные атомы. Написать функцию, которая возвращает список из двух чисел, первое из них равно сумме чисел исходного списка, а второе их количеству.

    ((1 A)(2 (3) B (4))) —> (10 4)
11. Дан список произвольной структуры. Каждый числовой атом списка необходимо увеличить на единицу.

    ((A (1)2)B 3) —> ((A (2)3)B 4)
12. Найти самый глубоко расположенный атом в произвольном списке. Если таких несколько, вывести первый из них.

    (A (B C)(D (E)F)) —> E
13. Для произвольного списка вычислить сколько раз встечается заданный атом

    ((1 A) ((3 4) (D C)) (2) (A)), A –> 2
14. Имеется произвольный список, состоящий из числовых атомов. Преобразовать список по следующему правилу: если элементы некоторого подсписка являются числами, заменить подсписок суммой его элементов.

    (1 (2 (3 4 5) 6)(7 8 9)) —> (1(2 (12) 6)(24))
15. Дан список произвольной структуры. Каждый подсписок, состоящий из одного атома, заменить на сам атом.

    (A ((B)C)((D))) —> (A(B C)D)
16. Написать функцию, проверяющую, что каждый атом произвольного списка является числом.
17. Определить, сколько атомов находится на заданном уровне вложенности. Элемент может быть как атомом, так и списком.

    (A B (B A (C)) A C (A)), 2 —> 3
18. Проверить, состоят ли два списка из одних и тех же атомов.
19. Удалить все числовые атомы в произвольном списке.
20. Имеется список произвольной структуры, но все одноуровневые списки в нем содержат ровно два элемента. Необходимо преобразовать список, удалив второй элемент.

((1 2) ((3 4) (5 6)) (((7 8)))) —> ((1)((3)(5))(((7))))

21. Дан список произвольной структуры, состоящий из чисел. Каждый атом увеличить на его уровень вложенности.

    (1 (2 3((4)5))) —> (2(4 5((8)8)))

22. Дан список произвольной структуры, каждый атом встречается по одному разу. Написать функцию, которая для каждого атома определяет его уровень вложенности и строит соответствующий список:

    (A ((B) C)) —> ((A 1) (B 3) (C 2))

23. Найти сумму первых k числовых атомов списка произвольной структуры.

24. Построить список, определяющий сколько раз встречается каждый атом в списке: (A (B A C B) C (A (B)) D) —> ((A 3) (B 3) (C 2) (D 1))

25. Дан список произвольной структуры. Найти наиболее часто встречающийся атом в списке.

26. Дано множество, представленное списком. Построить все перестановки этого множества:

    (A B C) —> ((A B C) (A C B) (B A C) (B C A) (C A B) (C B A))

27. Дано множество, представленное списком, и число K. Построить все сочетания из данного множества длиной K.

    (A B C D) 3 —> ((A B C) (A B D) (B C D) (A C D))

28. Для произвольного списка вычислить сумму чисел на заданной глубине

    ((1 A) ((3 4) (D C)) (2) (((7 8)))), 2 –> 3

3 ДРУГИЕ ВОЗМОЖНОСТИ ЯЗЫКА ЛИСП

3.1 ОПРЕДЕЛЕНИЕ ИТЕРАЦИОННЫХ ФУНКЦИЙ НА ЛИСПЕ

С алгоритмической точки зрения циклы эквивалентны хвостовой рекурсии, поэтому необходимости в операторах цикла в Лиспе нет, однако запись алгоритма в виде цикла часто оказывается удобнее рекурсивной записи, поэтому во многие диалекты Лиспа введены операторы цикла.

Для организации циклов в системе muLisp используется функция LOOP (цикл). Она имеет вид:

(LOOP A_1 ... A_n)

Эта функция последовательно вычисляет s-выражения A_1...A_n, затем начинает оценивать их заново, пока не будет выполнено условие выхода из цикла.

Если A_i имеет вид (U_i R_i), то оценивается U_i, и если его значение не равно NIL, выполнение функции LOOP прекращается, и ее значением становится оценка R_i. Иначе LOOP сразу переходит к оценке A_{i+1} не оценивая R_i.

Если какое-либо A_i или R_i имеет вид (RETURN D), то происходит оценивание D, и полученное значение становится значением всей функции, независимо от того, какова была вложенность функций LOOP.

Ниже приведена итерационная функция вычисления факториала, полностью аналогичная функциям языков Паскаль и Си:

```
(DEFUN F (N)
(SETQ I 1)
(SETQ K 1)
(LOOP
  ((= I N) K)
  (SETQ I (+ I 1))
  (SETQ K (* K I))))
```

В данном примере видно, что функция DEFUN в muLisp может иметь вид (DEFUN A B C_1...C_n) и при выполнении функции последовательно оцениваются s-выражения C_1...C_n. В классическом варианте Лиспа возможно только одно s-выражение. Для последовательной оценки s-выражений используется специальная функция PROG.

Функция PROG имеет вид: (PROG A B_1...B_n), где: A - список локальных переменных, используемых внутри функции PROG, а B_1...B_n – s-выражения, которые последовательно оцениваются, если являются списками. Значение всей функции PROG становится равным

оценке последнего выражения. Если среди B_i встретится выражение (RETURN C), функция, содержащая PROG, заканчивает работу, возвращая оценку C.

Если среди $B_1...B_n$ встретится символьный атом, он не оценивается и считается меткой, на которую можно выполнить переход функцией (GO Z), где Z - символьный атом (метка).

Обратите внимание, что в отличие от функции LOOP PROG не выполняет «зацикливания» автоматически, его надо организовывать явно с помощью функции GO!

Рассмотрим пример вычисления факториала, записанный на классическом Лиспе:

```
(DE F (N)
(PROG (I K)
  (SETQ I 1)
  (SETQ K 1)
A
  (COND ((EQUAL I N)(RETURN K)))
  (SETQ K (* K I))
  (SETQ I (+ 1 I))
  (GO A)))
```

В приведенном примере функция RETURN использована для прекращения работы цикла и всей функции F. «Естественный» выход из цикла PROG не используется, последним выражением внутри цикла является функция перехода GO.

Интерпретатор muLisp не поддерживает функцию PROG. Для того, чтобы эта и многие другие функции стандартного Лиспа начали работать, необходимо выполнить файл COMMON.LSP, например, вызвав функцию RDS интерпретатора muLisp:

```
$(RDS 'COMMON)
```

3.2 ДОПОЛНИТЕЛЬНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ MULISP

3.2.1 Функция (EVAL A)

Функция EVAL оценивает свой аргумент, а затем выполняет оценку уже оцененного аргумента. Иначе говоря, функция EVAL выступает в качестве интерпретатора Лиспа.

Примеры:

```
$ (SETQ A 'B)
B
$ (SETQ B 'C)
C
$ A
B
$ (EVAL A)
C
$ (EVAL '(CAR '(A B C)))
A
```

```
$ (SETQ A 'B)
B
$ (EVAL (CAR '(A B C)))
B
$ (EVAL (QUOTE (QUOTE QUOTE)))
QUOTE
```

3.2.2 Функции ввода-вывода (READ) и (PRINT A)

При выполнении функции ввода READ происходит останов интерпретатора до ввода s-выражения, которое и является результатом функции.

Пример:

```
$ (DEFUN F NIL (CONS (READ) NIL))
F
$ (F)
A       ; введенный символ
(A)     ; выводимое выражение - результат действия
        ; функции F.
```

Из функций вывода рассмотрим лишь простейшую функцию вывода PRINT. Эта функция оценивает свой единственный аргумент и выводит его в качестве результата.

Примеры:

```
$ (PRINT (+ 2 3))
5          ; действие функции PRINT
5          ; результат PRINT как функции
$ (SETQ X '(+ 2 3))
(+ 2 3)
$ (EVAL (PRINT X))
(+ 2 3)    ; действие функции PRINT
5          ; результат функции EVAL.
```

3.2.3 Функции работы со свойствами атомов (PUT A B C), (GET A B) и (REMPROP A B)

У каждого символьного атома может быть ряд свойств, которые принимают различные значения.

Рассмотрим пример. Пусть символьные атомы представляют собой имена людей: JOHN, VICTOR, IVAN. Свойства - возраст (AGE) и должность (STAFF). Функция (GET A B) извлекает значение свойства B символьного атома A.

```
$ (GET 'IVAN 'AGE)
35
$ (GET 'VICTOR 'STAFF)
PROFESSOR
$ (GET 'IVAN 'STAFF)
NIL
```

Значение NIL выдается, если атом не имеет данного свойства (значение свойства не установлено).

Функция (PUT A B C) задает свойству B символьного атома A значение C. В качестве значения функция PUT выдает значение C.

```
$(PUT 'IVAN 'STAFF 'DOCTOR)
DOCTOR
$(GET 'IVAN 'STAFF)
DOCTOR
```

Функция (REMPROP A B) удаляет свойство B у символьного атома A. В качестве значения функция REMPROP выдает прежнее значение свойства.

```
$(REMPROP 'IVAN 'STAFF)
DOCTOR
$(GET 'IVAN 'STAFF)
NIL
```

Действие функции (REMPROP 'IVAN 'STAFF) отличается от действия выражения (PUT 'IVAN 'STAFF NIL), поскольку во втором случае свойству задается значение NIL и информация о наличии свойства остается в памяти, а функция REMPROP полностью удаляет информацию о свойстве из памяти.

Свойства атомов и их значения можно записать в виде прямоугольной таблицы:

	AGE	STAFF
JOHN	35	
VICTOR	26	DOCTOR
IVAN	NIL	PROFESSOR

В современной трактовке таблицы столбец называется *полем базы данных*, а строка - *записью базы данных*. Современные языки программирования типа СУБД FOXPRO работают с подобной структурой базы данных, которая называется *реляционной*.

3.3 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЛИСПЕ

Еще одно применение свойств атомов - *объектно-ориентированное программирование* на Лиспе. Поскольку значением свойства может быть любое s-выражение, в свойствах можно хранить не только значения, но и функции-методы.

Рассмотрим реализацию на Лиспе классического примера объектно-ориентированного программирования. Пусть объекты, реализуемые в виде символьных атомов со свойствами, соответствуют животным. Пусть каждый из них имеет имя и метод, возвращающий “крик” этого животного. Значением атома является название животного. Определим объект-атом Mouse :

```
$ (setq m1 "Mouse")
|Mouse|
```

```
$ (put m1 'name "Micky")
|Micky|
```

Значением M1 является атом Mouse. Свойство NAME установлено для атома Mouse, а не M1. Обратите внимание, что имена атомов, указанные в кавычках, становятся регистрочувствительными, т.е. MOUSE и Mouse - разные атомы:

```
$ (get mouse 'name)  ; обращение к атому MOUSE
NIL
$ (get "Mouse" 'name); обращение к атому Mouse
|Micky|
```

Присвоим свойству CRY атома Mouse функцию-метод:

```
$ (put m1 'cry '(princ "Pi-pi-pi!"))
(PRINC |Pi-pi-pi!|)
```

Текстовые строки в Лиспе обычно задаются в кавычках или символах вертикальной черты. С точки зрения Лиспа обе эти записи эквивалентны. При выводе система muLisp обрамляет строковые константы вертикальными чертами. Чтобы избежать вывода этих ограничителей при печати, вместо функции PRINT в приведенном примере используется функция PRINC. Функция PRINC, в отличие от функции PRINT, не переводит строку после вывода аргументов (сравните операторы WRITE и WRITELN в языке Паскаль).

Создадим еще два объекта:

```
$ (setq m2 "Dog")
|Dog|
$ (put m2 'name "Bobik")
|Bobik|
$ (put m2 'cry '(princ "Wow-wow!"))
(PRINC |Wow-wow!|)
$ (setq m3 "Cat")
|Cat|
$ (put m3 'name "Murka")
|Murka|
$ (put m3 'cry '(princ |Mau-mau!|))
(PRINC |Mau-mau!|)
```

Объединим определенные атомы-объекты в список ANIMALS:

```
$(setq animals '(m1 m2 m3))
(M1 M2 M3)
```

С точки зрения функций, работающих с элементами этого списка, все его элементы одинаковы, в терминологии объектно-ориентированного подхода они являются *объектами класса* ЖИВОТНОЕ и обладают следующими общими свойствами: значение атома-объекта соответствует виду животного, а значения его свойств NAME и CRY соответствуют *элементу данных* и *методу* этого класса.

Для удобства определим функции, осуществляющие доступ к определенным свойствам атомов-объектов. Функция GETNAME возвращает значение свойства NAME объекта WHO:

```
$ (defun getname (who) (get who 'name))
GETNAME
```

```
$ (getname m1)
|Micky|
```

Функция DOCRY извлекает значение свойства CRY, соответствующего методу класса, и выполняет его с помощью функции EVAL:

```
$ (defun docry (who) (eval (get who 'cry)))
DOCRY
$ (docry m2)
Wow-wow!|Wow-wow!|
```

Функция PRINC не выполнила перевода строки после вывода “крика”” объекта M2, поэтому результат функции DOCRY (такая же строка, но в ограничивающих символах) был выведен на той же строке.

Функция DESCRIBE выводит полную информацию об объекте WHO. Для перевода строки в ней используется встроенная функция TERPRI:

```
$ (defun describe (who) (progn
  (princ "Animal:    ") (princ who) (terpri)
  (princ "Name:      ") (princ (getname who)) (terpri)
  (princ "Cry:       ") (docry who) (terpri)
))
DESCRIBE
$ (DESCRIBE M3)
Animal: Cat
Name:   Murka
Cry:    Mau-mau!
NIL
```

При создании объектов свойства NAME и CRY были установлены для атомов Mouse, Dog и Cat (в вызовах функций PUT атомы M1, M2, M3 указаны без апострофа), поэтому и функцию DESCRIBE необходимо вызывать с аргументами Mouse, Dog и Cat, т.е. оценками атомов M1, M2, M3. Вызов этой функции с неоцененными атомами Mi не приведет к желаемому результату:

```
$ (describe 'm1)
Animal: M1
Name:   NIL
Cry:
NIL
```

Теперь определим функцию, которая выведет информацию обо всех животных из списка ANIMALS. Обратите внимание, что вызов (DESCRIBE (CAR ANIMALS)) эквивалентен вызову (DESCRIBE ‘M1), поэтому элементы списка перед вызовом функции DESCRIBE принудительно оцениваются с помощью функции EVAL. Для повышения удобочитаемости блоки описаний животных отделяются друг от друга пустой строкой.

```
$ (defun everybody (who)
  (loop
    ((null who) (return nil))        ; конец цикла
    (describe (eval (car who)))      ; печать информации
    (terpri)                         ; перевод строки
    (setq who (cdr who))
```

```
  ))
EVERYBODY
$ (everybody animals)
Animal: Mouse
Name:   Micky
Cry:    Pi-pi-pi!

Animal: Dog
Name:   Bobik
Cry:    Wow-wow!

Animal: Cat
Name:   Murka
Cry:    Mau-mau!

NIL
```

Мы рассмотрели пример использования объектно-ориентированной технологии при программировании в среде muLisp. Мы использовали только стандартные средства, доступные в любой реализации языка Лисп. Многие современные реализации этого языка имеют специальные расширения для поддержки объектно-ориентированного программирования.

3.4 ЗАГРУЗКА ВНЕШНИХ МОДУЛЕЙ

При работе системы muLisp необходимо иметь возможность выполнять загрузку внешних текстовых модулей, содержащих запись функций, необходимых для решения данной задачи.

Простейшим вариантом считывания текстового файла является переключение на него текущего ввода с консоли. Это выполняется командой (RDS 'имя_файла.тип). Пример команды: (RDS 'Animal.lsp). Если расширение отсутствует, то по умолчанию используется расширение LSP. При выполнении команды RDS производится загрузка указанного файла в систему. Последней командой в загружаемом файле должна быть команда (RDS), переключающая текущий ввод на консоль. Если в файле содержатся ошибки, происходит прерывание ввода, в противном случае можно выполнять определенные в файле функции, набрав соответствующие команды.

Выход в ДОС в системе muLisp выполняется функцией (SYSTEM).

В системе muLisp имеется также программа-редактор EDIT.LSP, позволяющая набирать, сохранять и загружать текстовые файлы. Редактор написан на Лиспе и позволяет выполнять многие стандартные операции над текстом. При использовании этого редактора не возникает необходимости использовать внешний редактор для набора программ, а также редактирование и запуск программ можно выполнять не выходя из среды muLisp. Загрузка редактора выполняется командой (RDS 'EDIT), последующий его запуск командой (EDIT).

3.5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какому типу рекурсии эквивалентны циклы?
2. Какая функция для организации циклов предусмотрена в стандартном Лиспе?
3. Какая дополнительная функция для организации циклов имеется в системе muLisp?
4. Зачем в функции PROG описываются локальные параметры?
5. Каковы синтаксические особенности функции LOOP?
6. Как в Лиспе описываются и как используются метки для организации циклов?
7. Как работает функция EVAL?
8. Какие значения возвращают функции READ и PRINT?
9. Какие значения могут принимать свойства символьного атома?
10. Как осуществляется доступ к свойствам символьного атома?
11. Для чего используются свойства символьных атомов?
12. Как загрузить файл с программой на Лиспе в систему muLisp?

3.6 УПРАЖНЕНИЯ

1 Написать итерационную функцию, эквивалентную данной:

```
(DEFUN MYFUN (X A) (COND
  ((NULL X) NIL)
  ((EQUAL (CAR X) A) T)
  (T (MYFUN (CDR X) A))))
```

Функция MYFUN написана с использованием схемы частной рекурсии, поэтому может быть легко записана в итерационной форме. Легко видеть, что это функция-предикат, проверяющая наличие s-выражения A в одноуровневом списке X.

Для начала напишем эту функцию с использованием функции LOOP. Двум терминальным ветвям будут соответствовать выражения вида (A B). В рекурсивной ветви исходной функции происходила обработка списка, укороченного на голову. Для реализации такой операции используем обыкновенное присваивание, обработка нового значения произойдет при повторном выполнении цикла.

Функция MYFUN_LOOP, написанная с использованием функции LOOP и эквивалентная функции MYFUN, будет выглядеть так:

```
(DEFUN MYFUN_LOOP(X A)
(LOOP
  ((NULL X) NIL)
  ((EQUAL (CAR X) A) T)
```

```
  (SETQ X(CDR X))
))
```

Заметим, что благодаря особенностям синтаксиса функции LOOP нам не пришлось использовать функций проверки условия.

Теперь напишем такую же функцию, но на сей раз для организации цикла используем функцию PROG.

При использовании функции LOOP зацикливание происходило автоматически. При использовании PROG нам придется самим организовывать повторное выполнение тела цикла с помощью функции перехода GO и символьного атома MYLABEL, используемого в качестве метки. Дополнительных символьных атомов для хранения промежуточных значений мы использовать не будем, поэтому сразу после PROG необходимо указать пустой список. Символьный атом MYLABEL, используемый в качестве метки, локальной переменной не является.

Функция MYFUN_PROG, написанная с использованием функции PROG и эквивалентная функции MYFUN, будет выглядеть так:

```
(DEFUN MYFUN_PROG(X A)
(PROG ()
MYLABEL
  (IF (NULL X) (RETURN NIL) NIL)
  (IF (EQUAL (CAR X) A) (RETURN T) NIL)
  (SETQ X(CDR X))
  (GO MYLABEL)
))
```

Напомним, что система muLisp не поддерживает функцию PROG, и для правильной работы функции MYFUN_PROG в системе muLisp необходимо загрузить файл ‘COMMON.LSP’.

2 Каков будет результат оценки следующего s-выражения

(EVAL (PRINT (LIST '+ (PRINT (* 8 (EVAL 2))) (CAR '(1 2 3)))))?

Функция EVAL возвращает оценку своего оцененного аргумента, то есть фактически оценивает свой аргумент дважды, в отличие от всех остальных функций, которые делают это максимум один раз. Функция PRINT печатает оценку своего аргумента и переводит строку. Значением этой функции является значение ее аргумента. Рассмотрим процесс оценки приведенного s-выражения подробно.

Внешняя EVAL функция начинает оценку своего аргумента. Функция PRINT вызывает вычисление функции LIST, вторым аргументом которой является еще одна функция PRINT. Углубляясь таким образом внутрь приведенного s-выражения, мы придем к необходимости вычисления функции (EVAL 2). Оценка ее аргумента – числовой константы 2 – даст число 2.

Повторная попытка оценки 2 с помощью функции EVAL приведет к тому же результату – числу 2.

Теперь можно вычислить значение произведения (* 8 (EVAL 2)) ⇒ (* 8 2) ⇒ 16.

После этого функция выдаст PRINT на печать значение 16 и вернет его в качестве собственной оценки. Оценка s-выражения (CAR '(1 2 3)) даст значение 1. Таким образом, результат оценки аргумента (LIST '+ (PRINT (* 8 (EVAL 2))) (CAR '(1 2 3))) внешней функции PRINT равен (+ 16 1). Это выражение будет выдано на печать и в качестве результата самой функции PRINT. Внешняя функция EVAL еще раз оценит это выражение, получит 17, и уже сам интерпретатор Лиспа выдаст это значение в качестве результата оценки всего s-выражения.

Таким образом, на экране мы увидим:

```
$(EVAL (PRINT (LIST '+ (PRINT (* 8(EVAL 2))) (CAR '(1 2 3)))))
16
(+ 16 1)
17
```

3 Что делает функция

```
(DEFUN F1 (X)
  (IF (ATOM X) X
        (IF (AND (LISTP X) (EQ (CAR X) 'QUOTE))
              (F1 (CADR X))
              (CONS (F1 (CAR X)) (F1 (CDR X))))))?
```

Эта функция имеет три ветви. Для того, чтобы их стало лучше видно, перепишем эту функцию с использованием функции COND:

```
(DEFUN F1 (X)
  (COND
    ((ATOM X) X)
    ((AND (LISTP X) (EQ (CAR X) 'QUOTE))
        (F1 (CADR X)))
    (T (CONS (F1 (CAR X)) (F1 (CDR X))))))
```

Первая проверка соответствует терминальной ветви. Аргумент функции F1 станет значением самой функции, если он является атомом. Последняя ветвь соответствует схеме общей рекурсии, поэтому первая ветвь будет обрабатывать как конец списка – NIL, так и произвольный символьный атом, являющийся головой некоторого списка.

Во второй ветви проверяется, что аргумент функции F1 является списком с головой QUOTE. При выполнении этого условия возвращается результат применения этой же функции ко второму аргументу списка. Вспомним, что функция QUOTE имеет всего один аргумент. Таким образом, результат применения F1 к функции QUOTE равен результату применения F1 к ее аргументу.

Таким образом, функция F1 “очищает” переданный список от апострофов. Заметим, что символьный атом QUOTE, не являющийся именем функции, не будет удален, то есть (F1 '(QUOTE QUOTE)) даст QUOTE. Функция F1 предполагает также, что функция QUOTE всегда имеет ровно один параметр, то есть длина списка с головой QUOTE равна 2. Это означает, что (F1 '(QUOTE A B C)) равно просто A, остальные аргументы игнорируются. Последнее замечание: функция F1 удаляет апострофы на любом уровне вложенности, то есть

(F1 '(QUOTE (LIST (QUOTE (QUOTE (QUOTE A))))))

равно (LIST A).

Таким образом, правильный ответ – функция F1 заменяет все функции QUOTE их первыми аргументами на всех уровнях вложенности.

4 Значения столбца некоторой таблицы заданы свойством ARR символьных атомов и содержат списки чисел. Напишите функцию, аргументом которой является список из символьных атомов, соответствующих строкам такой таблицы, а результатом – символьный атом, для которого сумма чисел в столбце ARR максимальна.

Сначала напишем вспомогательную функцию, вычисляющую сумму элементов одноуровневого числового списка.

```
(DEFUN SUM (X)
  (IF (NULL X) 0 (+ (CAR X) (SUM (CDR X)))))
```

Вторая вспомогательная функция будет возвращать для заданного символьного атома сумму элементов числового списка, заданного свойством ARR. Для получения списка воспользуемся функцией GET.

```
(DEFUN FOR_ATOM (X) (SUM (GET X 'ARR)))
```

Осталось написать главную функцию, которая будет выполнять поставленную задачу:

```
(DEFUN MAIN(X) (COND
  ((NULL X) NIL)
  ((NULL (CDR X)) (CAR X))
  ((> (FOR_ATOM (CAR X)) (FOR_ATOM (MAIN (CDR X))))
      (CAR X))
  (T (MAIN (CDR X)))
))
```

Обратите внимание, что в функции MAIN сравниваются не символьные атомы, а результаты применения к ним функции FOR_ATOM. В функциях SUM и MAIN использована хвостовая рекурсия, функция FOR_ATOM не содержит рекурсивных вызовов. С использованием циклов можно решить эту задачу с использованием всего одной функции. Используемые в приведенной ниже функции MAIN_C дополнительные переменные имеют следующий смысл:

M – максимальная сумма списка ARR для уже обработанных элементов;

Y – копия списка ARR текущего элемента, используемая для вычисления суммы;

S – сумма элементов списка Y;

A – первый элемент, для которого сумма списка ARR равна M.

```
(DEFUN MAIN_C(X)
(IF (NULL X) NIL          ; случай совсем пустого списка
  (SETQ M 0)              ; начальные присваивания
  (LOOP                   ; этот цикл соответствует MAIN
    (SETQ Y (GET (CAR X) 'ARR))
    (SETQ S 0)
    (LOOP                 ; этот цикл соответствует SUM
      ((NULL Y) S)
      (SETQ S (+ S (CAR Y)))
      (SETQ Y (CDR Y))
    )                     ; конец цикла вычисления суммы
    (IF (> S M) (SETQ A (CAR X)) NIL)
    (IF (> S M) (SETQ M S) NIL)
    (SETQ X (CDR X))
    ((NULL X) (RETURN A)) ; см. терминальную ветвь MAIN
  )
))
```

Обратите внимание на порядок ветвей с функцией IF. Можно ли поменять их местами?

3.7 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Какая из приведенных ниже функций эквивалентна функции MYFUN?

```
(DEFUN MYFUN(N) (FUN1 1 1 N 2))
(DEFUN FUN1(X1 X2 N I)
      (IF (>=I N) X1 (FUN1 (+ X1 X2) X1 N (+ I 1))))
```

a)
```
(DEFUN MF2 (N)
    (SETQ A 1)
    (SETQ B 1)
    (SETQ C 2)
    (LOOP
      ((>= C N) (RETURN A))
      (SETQ A (+ A B))
      (SETQ B A)
      (SETQ C (+ C 1))
    ))
```

b)
```
(DEFUN MF2 (N)
    (SETQ A 1)
    (SETQ B 0)
    (SETQ C 1)
    (LOOP
      ((>= C N) (RETURN A))
      (SETQ X A)
      (SETQ A (+ A B))
      (SETQ B X)
```

```
      (SETQ C (+ C 1))
   ))
```

c)
```
(DEFUN MF2 (N)
 (PROG (A B C X)
   (SETQ A 1)
   (SETQ B 0)
   (SETQ C 1)
LAB
   ((>= C N) (RETURN A))
   (SETQ X A)
   (SETQ A (+ A B))
   (SETQ B X)
   (SETQ C (+ C 1))
   (GO LAB)
))
```

d)
```
(DEFUN MF2 (N)
 (PROG (A B C X)
   (SETQ A 1)
   (SETQ B 1)
   (SETQ C 2)
LAB
   (IF (>= C N) (RETURN A) NIL)
   (SETQ X A)
   (SETQ A (+ A B))
   (SETQ B X)
   (SETQ C (+ C 1))
   (GO LAB)
))
```

e)
```
(DEFUN MF2 (N)
 (SETQ A 1)
 (SETQ B 0)
 (LOOP
   ((>= 1 N) (RETURN A))
   (SETQ X A)
   (SETQ A (+ A B))
   (SETQ B X)
   (SETQ N (- N 1))
))
```

2. Для каких из приведенных ниже функций можно написать итерационный эквивалент?

```
(DEFUN FUN1 (X Y) (COND
  ((= X 1) Y)
  ((= Y 1) X)
  (T (+ (FUN1 (- X 1) Y) (FUN1 X (- Y 1))))))
(DEFUN FUN2 (X Y) (COND
  ((= X 1) Y)
  (T (+ Y (FUN2 (- X 1) Y)))))
(DEFUN FUN3 (X Y) (COND
  ((AND (= X 1) (= Y 1)) (+ X Y))
  ((= X 1) (+ 1 (FUN3 X (- Y 1))))
  ((= Y 1) (FUN3 Y X))
  (T (+ 1 (FUN3 (- X 1) Y)))))
```

a) ни для какой

b) для всех

c) только для fun1

d) только для fun1 и fun2

e) только для fun2 и fun3

3. Что делает приведенная ниже функция?

```
(DEFUN MYFUN(X) (PROG (Y I A)
  (SETQ Y NIL) (SETQ I 0) (SETQ A NIL)
LABEL
  (IF (NULL X) Y NIL)
  (IF (NULL X) I 0)
  (IF (NULL X) A NIL)
  (SETQ A (CAR X))
  (SETQ I (+ I 1))
  (SETQ X (CDR X))
  (GO LABEL)
  (SETQ Y (APPEND Y (LIST A)))
))
```

a) переворачивает одноуровневый список

b) переворачивает список произвольной структуры

c) возвращает последний элемент списка

d) считает количество элементов в списке

e) возвращает NIL для любого списка X

4. Что выведет система muLisp, если в ответ на приглашение ввести следующее выражение: (PRINT (EVAL '(CAR (LIST (* 8 5) (EVAL (+ 3 (PRINT 2)))))))?

a)
```
2
40
40
```

b)
```
2
(+ 3 2)
45
45
```

c)
```
2
45
45
```

d)
```
(CAR (LIST (* 8 5) (EVAL (+ 3 (PRINT 2)))))
2
40
```

e)
```
2
(CAR (LIST (* 8 5) (EVAL (+ 3 (PRINT 2)))))
45
```

5. Каков будет результат оценки s-выражения (EVAL '(LIST (CONS A (EVAL '(LIST B C))) (CDR '(EVAL (LIST A B))))?
   a) система выдаст сообщение об ошибке
   b) (LIST (CONS A (EVAL '(LIST B C))) (CDR '(EVAL (LIST A B))))
   c) ((A B C) ((LIST A B)))
   d) ((A B C) (B))
   e) (A B C B)
6. Для всех символьных атомов в одноуровневом списке, содержащем символьные атомы и списки, необходимо вычислить значение функции, заданной свойством FUN, если такое свойство определено, и присвоить полученное значение свойству VAL того же атома. С помощью какой из приведенных функций можно выполнить требуемые действия?

   a)
```
(DEFUN F1(X) (COND
 ((NULL X) NIL)
 ((ATOM (CAR X))
   (IF (GET (CAR X) 'FUN)
      (PUT (CAR X) 'VAL (EVAL (GET (CAR X) 'FUN)))
      (CONS (CAR X) (F1 (CDR X)))))
 (T (CONS (CAR X) (F1 (CDR X))))))
```
   b)
```
(DEFUN F1(X) (COND
 ((NULL X) NIL)
 ((AND (ATOM (CAR X)) (GET (CAR X) 'FUN))
      (PUT (CAR X) 'VAL (EVAL (GET (CAR X) 'FUN))))
 (T (CONS (CAR X) (F1 (CDR X))))))
```
   c)
```
(DEFUN F1(X) (COND
 ((NULL X) NIL)
 ((ATOM (CAR X))
   (IF (GET (CAR X) 'FUN)
      (PUT (CAR X) 'VAL (GET (CAR X) 'FUN))
      (CONS (CAR X) (F1 (CDR X)))))
 (T (CONS (CAR X) (F1 (CDR X))))))
```
   d)
```
(DEFUN F1(X) (COND
 ((NULL X) NIL)
 ((ATOM (CAR X))
      (PUT (CAR X) 'VAL (GET (CAR X) 'FUN)))
 (T (CONS (CAR X) (F1 (CDR X))))))
```
   e)
```
(DEFUN F1(X) (COND
 ((NULL X) NIL)
 ((AND (ATOM (CAR X)) (GET (CAR X) 'FUN))
      (SETQ (GET (CAR X) 'VAL) (GET (CAR X) 'FUN)))
 (T (CONS (CAR X) (F1 (CDR X))))))
```

Ответы: 1.b,d; 2.e; 3.e; 4.a; 5.c; 6.a,b.

3.8 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №4

ПОСТРОЕНИЕ ИТЕРАЦИОННЫХ ПРОГРАММ ДЛЯ ОБРАБОТКИ СПИСКОВ

Задание: Реализовать обработку списков с использованием функции LOOP для вариантов из лаюораторной работы № 2.

Задания выполняются в среде cLisp (или другой). Код программ и тестовые примеры привести в отчете.

Быть готовым пояснить любой вопрос по тексту кода.

4 ПОСТРОЕНИЕ ЧЕРТЕЖЕЙ В СИСТЕМЕ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ АВТОКАД

Автокад является наиболее распространенной в настоящее время системой по автоматизации проектирования в машиностроении, строительстве. архитектуре и других областях. Основой Автокада является мощный графический редактор, позволяющий конструктору создавать сложные чертежи из большого количества графических примитивов. Изображения строятся в выбранной системе координат и с высокой точностью отображаются на внешнем носителе (принтере, графопостроителе, плоттере) с заданным масштабом.

Особенностью Автокада является способ хранения чертежей в виде списков, элементами которых являются графические примитивы, и внутренний язык для обработки и анализа этих списков - Автолисп, являющийся диалектом стандартного Лиспа.

Преимущества разработки технической документации средствами компьютерной графики

Применение компьютерных графических систем для разработки технической документации даёт конструктору практически неограниченные возможности:

- разрабатывать чертежи-аналоги по чертежам-прототипам;
- создавать библиотеки изображений стандартных элементов (изображения крепежных элементов − болтов, шпилек, гаек и т. п.);
- моделировать трехмерные геометрические объекты с помощью объемных примитивов и операций, выполняемых с ними;
- адаптировать графическую систему к решаемым задачам пользователя путем расширения графической системы разработкой собственного меню пользователя и внедрения в систему языков программирования высокого уровня.

4.1 КРАТКАЯ ХАРАКТЕРИСТИКА ГРАФИЧЕСКОГО РЕДАКТОРА

Visual LISP – это интегрированная среда разработки программ на языке программирования АВТОЛИСП в системе АВТОКАД. Она значительно облегчает процесс создания программы, её изменения, тестирования и отладки. Visual LISP имеет собственный набор окон и меню, который отличается от соответствующего набора AutoCAD. Однако запуск интегрированной среды Visual LISP производится из системы AutoCAD. Запуск Visual LISP: **Tools – AutoLISP - Visual LISP**

Для создания файла необходимо выбрать пиктограмму – New File (Новый файл). Если файл уже ранее был создан, необходимо выбрать пиктограмму – Open File (Открыть файл). Главное меню представляет собой систему, которая обеспечивает доступ ко всем средствам Visual LISP.

При открытии файла с программой на языке AutoLISP автоматически вызывается окно текстового редактора. Следует помнить, что одновременно можно работать только с одним файлом в одном окне текстового редактора. Каждый раз при открытии файла, с которым мы работаем в интегрированной среде, Visual LISP отображает файл в новом окне текстового редактора.

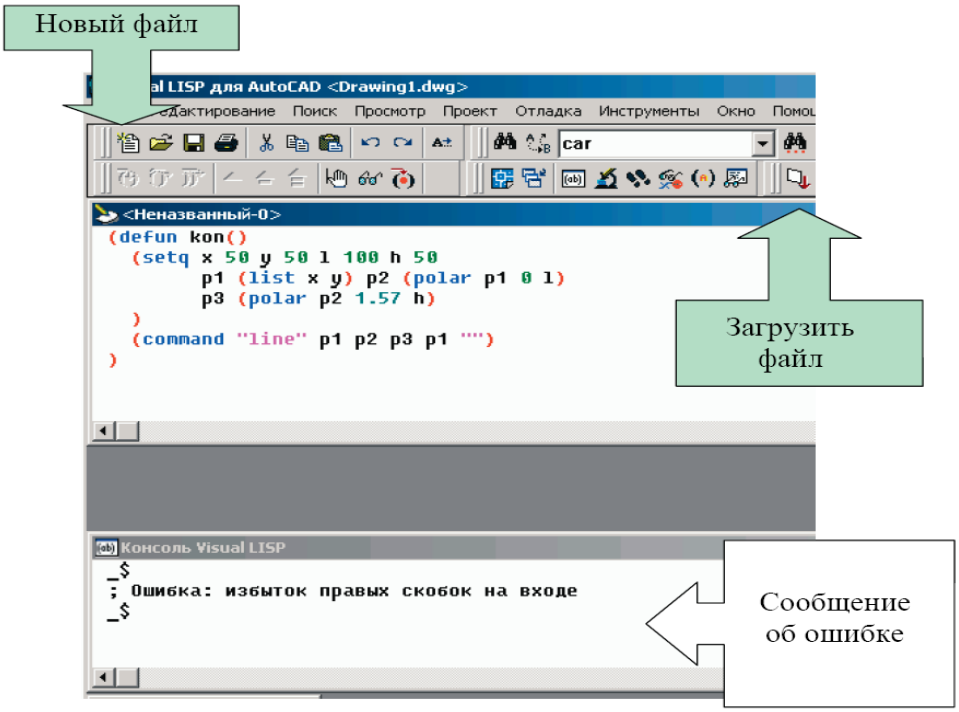


Рис. 1 Окно АвтоЛИСПа

Чтобы загрузить и выполнить программу, надо указать кнопку панели инструментов – Load active edit window. Произойдет загрузка программы в оперативную память. Откроется окно консоли. Для выполнения программы необходимо ввести в консоль имя выполняемой функции.

Отладка в режиме анимации

Надо указать в падающем меню Debug (Отладка) \ Animate (Анимация), а затем загрузить и начать выполнение отлаживаемой программы, введя с консоли ее имя. На экране наблюдается выполнение программы в замедленном режиме. Выполняемые функции последовательно высвечиваются. В случае ошибки анимация прерывается, место ошибки в тексте программы остается выделенным. Если анимация “затянулась”, ее можно прервать с клавиатуры клавишей BREAK. Скорость анимации регулируется в меню редактора VLISP: Tools (Инструменты) \

Environment Options (Настройки рабочей среды) \ General Options (Общие настройки)\ Diagnostic (Диагностика).

Контроль значений переменных

Необходимость контроля значений переменных возникает в двух случаях: программа прерывается по ошибке, программа выполняется до конца, но дает неверный или сомнительный результат, и требуется проконтролировать алгоритм вычислений. Если вследствие ошибки произошло прерывание программы, можно посмотреть значения всех переменных и результаты выполнения любого выражения в тексте программы до момента прерывания. Это позволит разобраться в причинах, вызвавших ошибку. Сначала с помощью анимации определяется место ошибки. Затем следует установить курсор на символ какой-либо переменной или выделить курсором необходимое выражение и нажать кнопку Watch панели инструментов.

Искусственное прерывание программы

Наиболее сложно устранить ошибку, если программа выполняется без “сбоя”, но выдает неверный результат. В этом случае программу искусственно прерывают и осматривают значения переменных. Для прерывания программы достаточно установить курсор в необходимом месте по тексту программы и указать кнопку – Toggle breakpoint
на панели инструментов VLISP. Для продолжения программы следует нажать кнопку – Continue.

4.2 ОСНОВНЫЕ КОМАНДЫ АВТОКАДА

Выполнить произвольную команду Автокада можно, набрав ее в строке команд. Практически любая команда имеет ряд параметров, которые нужно задать для ее выполнения. Например, окружность можно построить по центру и радиусу, по двум точкам на концах диаметра, по трем точкам окружности и т.д. Кроме этого, любую команду можно выполнить из программы на Автолиспе. Для этого используется встроенная функция Автолиспа COMMAND, которая имеет вид (COMMAND "НАЗВАНИЕ КОМАНДЫ АВТОКАДА" "ПАРАМЕТР 1" "ПАРАМЕТР 2" ...).

Рассмотрим пример выполнения команды построения окружности. После ввода в строке команд команды рисования окружности **CIRCLE** на экране появится строка выбора параметров этой команды: 3P/2P/TTR/<Center point>. 2P означает изображение окружности по двум противоположным точкам диаметра, 3P - по трем точкам на окружности, TTR - по двум касательным и радиусу и, наконец, параметр <Center point> означает выбор центра окружности. Выбор точки выполняется путем ввода ее координат через запятую в строке команд или подведения графического курсора к определенному положению на экране и нажатия клавиши

<Ввод>. Если выбрать параметр 2P, набрав эти символы в строке команд, появятся последовательно запросы на ввод первой и второй точки противоположных концов диаметра. При этом, перемещая графический курсор для выбора второй точки, можно наблюдать за изменением положения изображаемой окружности. Этот принцип “резиновой нити” выполняется для всех команд Автокада. Выполнить построение окружности по двум точкам на концах диаметра с координатами 100,150 и 200,220 из программы на Автолиспе можно строкой

(COMMAND "CIRCLE" "2P" '(100 150) '(200 220))

Если же требуется построение окружности по центральной точке с координатами 100,120 и радиусу 50, то построение выполняется функцией (COMMAND "CIRCLE" '(100 120) 50).

Из приведенных примеров видно, что точка в Автолиспе представлена списком из двух своих координат. Ниже приведен ряд важных команд Автокада с указанием их особенностей при вызове из Автолиспа.

Команда **LINE** предназначена для построения последовательности отрезков, причем началом последующего отрезка является конец предыдущего. Для завершения последовательности построения следует выбрать точку, совпадающую с концом последнего отрезка, или ввести параметр **"C"** для соединения конца последнего отрезка с началом всей последовательности. Соответствующие команды Автолиспа имеют вид

(COMMAND "LINE" '(X1 Y1) ... '(XN YN) "") и

(COMMAND "LINE" '(X1 Y1) ... '(XN YN) "C"),

пустая строка в первой команде означает завершение последовательности построения.

Команды **ELLIPSE, DOUGHNUT, POLIGON** предназначены соответственно для построения эллипса, кольца и правильного n-угольника. Каждая из этих команд имеет небольшое число очевидных параметров, а команда **ARC**, позволяющая рисовать дугу окружности, дает 11 вариантов ее построения.

Например, (command "arc" "c" p1 p2 p3) – построение дуги окружности по центру и двум точкам, где "c" – задает опцию центр, p1 – центр дуги окружности, p2 и p3 – начальная и конечная точки дуги; (command "arc" p4 p5 p6) – построение дуги окружности по трем точкам.

Команда **PLINE** (ПОЛИЛИНИЯ) позволяет рисовать последовательность чередующихся отрезков и дуг, переключая их параметрами **A** (Arc) и **L** (Line). Кроме этого, данная команда позволяет менять тип линий и их ширину. Рассмотрим пример использования этой команды на Автолиспе: (COMMAND "PLINE" '(30 40) "A" "CE" '(20 45) '(40 65) "L" '(100 65) “W" 4 0 '(105 65) ""). Сначала от точки с координатами (30,40) до точки (40,65) проводится дуга (параметр A) с центром в точке с координатами (20,45) (параметр CE). Затем из конечной точки дуги в горизонтальном направлении до точки с координатами (100,65) проводится отрезок прямой (параметр L). Последний небольшой отрезок (100,65)-(105,65) является отрезком переменной

ширины (W) от 4 до 0 и выглядит как “уголок”, а вместе эта пара отрезков изображает “стрелку”. Завершается команда PLINE также как и LINE.

Команда **TEXT** позволяет вывести строку заданной высоты и направления в заданной точке. Например, команда Автолиспа (COMMAND "TEXT" '(100 200) "10" "90" "AUTOCAD") выводит текст AUTOCAD, верхняя левая точка которого имеет координаты (100,200), высота текста равна 10 и угол направления - 90^o (вверх). Другие параметры команды выравнивают текст по центру или правому краю, меняют стиль и шрифт.

Команда **HATCH** выполняет штриховку замкнутых объектов. Обычно в Автолиспе используется следующий формат этой команды (COMMAND "HATCH" "U" "45" "4" "N" '(X1 Y1) ... '(XN YN)). Параметр U означает, что штриховка выполняется прямыми линиями, причем линии расположены под углом 45^o, расстояние между ними равно 4, и линии не перекрещиваются (N). Точки (X1,Y1), ..., (XN YN) должны принадлежать графическим примитивам, образующим замкнутую область. Если Автокад распознает наличие замкнутой области, он выполняет ее штриховку с указанными параметрами.

Команда BHATCH (Boudary Hatch) создает ассоциативную и не ассоциативную штриховку. Ассоциативная штриховка имеет связь с ее границей и изменяется при изменении границы. Она позволяет штриховать область, ограниченную замкнутой кривой, как путем простого указания внутри контура, так и путем выбора объектов. Она автоматически определяет контур и игнорирует любые целые примитивы и их составляющие, которые не являются частью контура.

Команда HATCH создает не ассоциативную штриховку

Примеры

Перед выполнением операции штриховки необходимо выполнить изображение замкнутого контура с помощью команды pline.

Например: `(command "pline" p1 p2 p3 p4 p1 "")`. В данном случае выбор замкнутого контура, подлежащего штриховке, делается с помощью указания одной точки.

`(command "hatch" "jis_wood" "5" "0" p3 p5 "")` – построение штриховки с выбором замкнутых контуров, определяемых точками, принадлежащими этим контурам (рис. 3.9а). "jis_wood " – определяет тип штриховки, "5" – масштаб штриховки, "0" – угол наклона штриховки, p3, p5 – точки, необходимые для выбора объектов, определяющих замкнутые контуры (рис. 2а).

`(command "hatch" "jis_wood" "5" "0" "w" p6 p7 "")` – построение штриховки с выбором замкнутого контура с использованием рамки "w" – задание опции выбора объектов с помощью рамки; p4, p5 – точки, задающие вершины рамки по диагонали (рис. 2б).

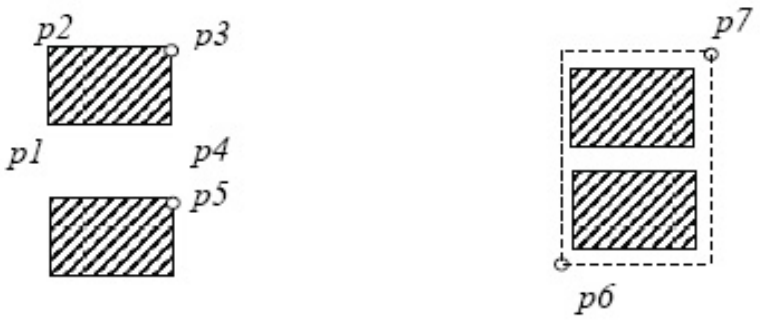


Рис. 2. Варианты штриховки в АвтоКАДе

Вывод штриховки для области, указанной внутренней точкой (`command "bhatch" "p" "ansi31" "0.5" "30" (12,12" "")`. Возможно указание сразу нескольких областей: `(command "bhatch" "p" "ansi31" "0.5" "30" "7,13" "21,16" "")`

При составлении программы, позволяющей наносить **размеры**, необходимо вначале задать значения системных размерных переменных. Нанося размеры на чертеже, необходимо рассчитывать вспомогательные точки, определяющие положение размерных линий и примитивов.

Простановка линейного размера между точками p1 и p2 расположение размерного текста в точке p3 `(command "dimlinear" p1 p2 p3)`

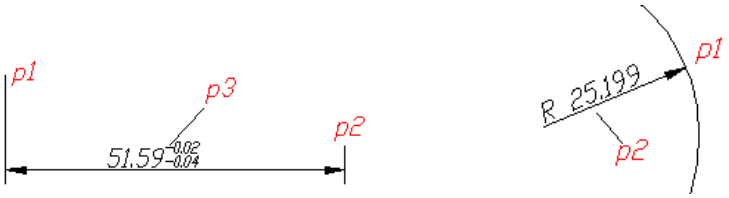


Рис. 3. Вынос размеров в АвтоКАДе

Простановка радиального размера `(command "dimradius" p1 p3)`

Точка на дуге - p1 определяет и расположение размерной линии по направлению к центру окружности. Случай а) - обозначение радиуса внутри окружности - в этом случае размерная линия до центра не доходит. Случай б) - размер ставится вне окружности. Тогда размер идет от центра окружности через точку p1 к указанной точке p2.

Примеры

Пример 1. `(command "dim" "horiz" p1 p2 p3 "" "exit")` – нанесение горизонтального размера. Значение размерного текста определяется длиной между точками p1 и p2 (рис. 4а).

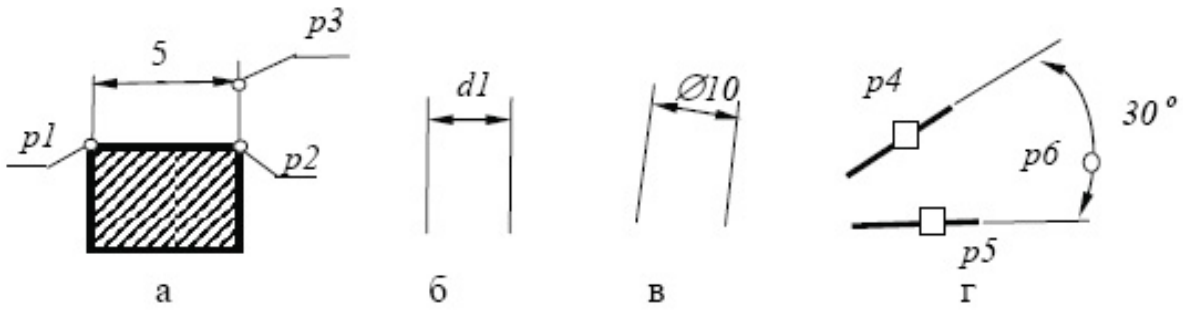


Рис. 4 Примеры формирования размеров

Пример 2. `(command "dim" "horiz" p1 p2 p3 "d1" "exit")` – нанесение горизонтального размера. В качестве размерного текста выступает строковая константа "*d1*" (рис. 4б).

Пример 3. `(command "dim" "aligned " p1 p2 p3 (strcat "%%c" (rtos h 2 0)) "exit")` – нанесение параллельного размера со знаком диаметра (рис. 4в).

Функция strcat предназначена для соединения двух строковых констант. "%%c" – запись кода знака диаметра в файле шрифта. Функция rtos позволяет выполнять преобразование переменной h, имеющей тип данных действительного числа в значение строковой константы.

Пример 4. `(command "dim" "angular" p4 p5 p6 "30" "exit")` – нанесение углового размера, точки p4, p5 указывают две прямые, между которыми проставляется угловой размер (данные прямые должны быть изображены заранее); p6 – указывает положение размерной линии; 30 – задание текста (рис. 4г).

Команда **LIMITS** устанавливает границы изображения. После ввода команды следует набрать новые координаты левого нижнего и правого верхнего угла. При этом заметим, что, во-первых, для того, чтобы увидеть результат действия команды LIMITS, необходимо вслед за ней выполнить команду **ZOOM** (ПОКАЖИ) с параметром **A** (All - Все). Из Автолиспа данная команда выполняется строкой (COMMAND "ZOOM" "A"). Во-вторых, размеры изображения не устанавливаются произвольно. Например, (COMMAND "LIMITS" '(0 0) '(297 210). Команда **ZOOM** позволяет также увеличивать выбранный фрагмент изображения на все поле рисунка, то есть устанавливать границы рисунка соответствующими выделенной области. Как правило, команды LIMITS и ZOOM ALL используются перед началом построения изображений.

Команда **REDRAW** (ОСВЕЖИ) выполняет перерисовку экрана. Когда Автокад создает изображения, на экране появляются различные вспомогательные пометки, помогающие в процессе построения чертежей. Для их удаления и используется в конце построения команда REDRAW, которая позволяет увидеть “очищенный” чертеж.

Команда **GRCLEAR** очищает поле рисунка графического редактора.

Перечисленных команд достаточно для построения простых чертежей. Существует большой набор других команд, для изучения которых необходимо обратиться к специальной литературе по Автокаду или к технической документации.

4.3 ОСОБЕННОСТИ АВТОЛИСПА

Язык Автолисп является диалектом стандартного Лиспа и по своему синтаксису практически совпадает с muLisp, который был рассмотрен в первой главе. Вместе с тем Автолисп имеет ряд дополнительных функций, которые связаны с его применением как языка построения и анализа изображений. Эти особенности и рассмотрены в настоящем разделе. Функция Автолиспа **COMMAND** рассмотрена в предыдущем параграфе.

Прежде всего для построения чертежей необходимы точные расчеты, поэтому вещественная арифметика является важной составной частью Автолиспа. Кроме арифметических функций **“+”, “-”, “*”, “/”,** выполняющих операции над числами с плавающей точкой (например, (/ 1.5E10 4.5E-3)), в Автолиспе есть и элементарные математические функции с вещественными аргументами. Перечислим некоторые из них:

(EXPT A B) - возведение в степень A^B;

(SIN A), (COS A), (ATAN A) - синус, косинус и арктангенс от вещественных аргументов *(Углы выражаются в радианах)* ;

(SQRT A), (EXP A), (LOG A) - квадратный корень, экспонента и натуральный логарифм от вещественных аргументов.

Важным типом данных в Автолиспе являются строки, использующиеся, например, для вывода текста на изображениях. Строки в Автолиспе заключаются в двойные кавычки (например, “”Это строка”) и для работы с ними используется ряд функций, полностью аналогичных соответствующим функциям Си.

Функция **(STRLEN A)** возвращает целое число, равное количеству символов в строке A. Функция **(STRCAT A B)** возвращает строку, образующуюся в результате сцепления строк-аргументов A, B. Наконец, функция **(SUBSTR A N1 N2)** или **(SUBSTR A N1)** выделяет из строки A подстроку, начинающуюся с символа с номером N1. Если аргумент N2 присутствует, возвращается N2 символов, иначе возвращаются все символы до конца строки A.

Перечислим ряд функций Автолиспа для преобразования типов данных. Функция **(FIX A)** преобразует вещественное число A в целое, путем отбрасывания дробной части (аналог TRUNC Паскаля). Функция **(FLOAT A)**, наоборот, используется для преобразования целого числа A в вещественное. Функции **(ATOF A)** и **(ATOI A)** принимают в качестве аргумента строку A и

преобразуют ее соответственно в вещественное и целое число. При этом содержание строки должно допускать такое преобразование. Функция **(RTOS A B C)** служит для преобразования вещественного числа A в строку. Параметр B определяет тип преобразования и может иметь значения от 1 до 5. Единица означает форму числа с плавающей точкой (<мантисса>E<порядок>), а двойка - с фиксированной (<целая часть>.<дробная часть>). Во втором случае параметр C определяет число знаков дробной части. В частности, (RTOS A 2 0) означает перевод в строку, представляющую целую часть числа A.

Большее значение в Автолиспе, чем в других диалектах Лиспа имеют и функции ввода. Обычно при автоматическом построении чертежей программа выдает запрос на ввод числовых значений определенных параметров, после чего выполняется само построение. В Автолиспе для этого имеется ряд функций ввода, имеющих одинаковый формат. Функция **(GETINT A)** выполняет запрос на ввод целого числа и возвращает введенное число. Параметр A представляет собой строку-подсказку и может отсутствовать. Пример команды: (SETQ A (GETINT "ВВЕДИТЕ РАЗМЕР ШАЙБЫ:")). Введенное по подсказке число присваивается переменной A. Аналогичные строки-подсказки имеет функция ввода INPUT языка Бейсика. Ниже указан ряд аналогичных функций ввода Автолиспа, причем единственный аргумент у них имеет тот же самый смысл.

Функции **(GETREAL A)** и **(GETSTRING A)** обеспечивают соответственно ввод вещественного числа и ввод строки символов. Функция **(GETPOINT A)** предназначена для ввода точки. При выполнении команды можно ввести координаты точки в строке команд, а можно выбрать точку графическим курсором. Функция GETPOINT возвращает список координат (X,Y) введенной точки. Функции **(GETDIST A)** и **(GETANGLE A)** выполняют соответственно ввод расстояния между двумя выбранными точками и угла между прямой, проходящей через две выбранные точки, и осью OX.

```
(setq p (getpoint))
(setq p (getpoint "Where? ")) /"Где? "/
```

Реальные задачи построения и обработки чертежей, как правило, проще решаются итерационными, а не рекурсивными программами. Для организации циклов в Автолиспе предназначена функция **WHILE**, имеющая формат (WHILE A B1 B2,...,BN). Аргумент A является условием, при выполнении которого (A<>NIL) последовательно оцениваются аргументы B1, ..., BN. Затем снова проверяется условие A. Когда A становится равным NIL, функция WHILE завершает работу и возвращает в качестве своего значения последнюю оценку BN.

Функции-предикаты, AND, OR, IF, COND, SETQ, DEFUN полностью соответствуют

соответствующим функциям muLisp, в частности, функция DEFUN может иметь вид (DEFUN A B C1, ..., CN), что обычно используется для записи итерационных процедур.

Отметим также несколько функций для работы с точками изображения. Функция (**POLAR A B C)** используется для перехода на чертеже от одной точки к другой. Аргумент A представляет собой точку (то есть является списком координат X,Y), B - вещественное число, определяющее величину угла в радианах, и C - вещественное число, задающее расстояние. Функция POLAR возвращает новую точку (список координат), находящуюся на расстоянии C от точки A под углом B к этой точке относительно оси OX против часовой стрелки. Заметим, что в Автолиспе имеется встроенная константа **PI** (число π). Функция **(DISTANCE A B)** вычисляет расстояние между точками A и B, а функция **(ANGLE A B)** - величину угла в радианах между прямой, проходящей через точки A и B, и осью OX. Наконец, функция **(INTERS A1 A2 B1 B2 P)** находит точку пересечения прямых, проходящих через точки A1, A2 и B1, B2. Если аргумент P (признак) не равен NIL, точка пересечения ищется только внутри отрезков (A1, A2) и (B1,B2), в противном случае определяется точка пересечения прямых линий бесконечной длины.

4.4 СИСТЕМНЫЕ ПЕРЕМЕННЫЕ AUTOCAD:

Единицы измерения, как и многие другие параметры, определяются значениями системных переменных Автокада. Системная переменная - ячейка памяти, содержащая определенное значение и имеющая неизменное имя. Значения системных переменных задают различные режимы работы команд Автокада.

К системным переменным нельзя обращаться напрямую, как к обычным переменным AutoLISPa. Для доступа к системным переменным в AutoLISPe имеются две функции:

```
(getvar "имя")
```

Функция getvar возвращает значение системной переменной с именем "имя", заданным как текстовая строка. Например, системная переменная "LASTPOINT" содержит координаты текущей точки. Для их использования в программе следует использовать функцию getvar в виде:

```
(getvar "LASTPOINT")
```

Если в ходе отрисовки полилинии следующую точку удобнее рассчитать от предыдущей при помощи функции polar, необязательно записывать все промежуточные точки в переменные. Можно использовать функцию getvar, например:

```
(command "pline" ( LIST ( + A 10 ) ( - B 20))
```

```
(command polar ( getvar "LASTPOINT" ) 0 40 ) "")
```

В приведенном примере координаты начальной точки рассчитываются. Чтобы не записывать эту точку в отдельную переменную, следующая точка, координаты которой рассчитывается при помощи функции polar, использует в качестве опорной координаты текущей (т.е. начальной) точки, всегда записываемые в виде списка в системную переменную "LASTPOINT".

```
(setvar "имя" значение )
```

Функция setvar меняет значение соответствующей системной переменной. Осторожно! Хорошенько подумайте, прежде чем менять значение системной переменной. Эти значения записываются в файл чертежа. Часть системных переменных (например, переменная, содержащая номер версии Автокада) доступна только для чтения и их значения нельзя изменить.

OSMODE (Системная переменная)

Определяет какие привязки к объектам включены/отключены. Значение определяется как сумма следующих значений.

0 NONe (отключено)
1 ENDpoint (конточка, конечная точка)
2 MIDpoint (середина)
4 CENter (центр дуги, окружности или дугового сегмента полилинии)
8 NODe (дословный перевод: "узел". Фактически - привязка к примитиву ТОЧКА (POINT))
16 QUAdrant (квадрант. Работает для дуг, окружностей и дуговых сегментах полилиний.)
32 INTersection (пересечение. Работает только на примитивах, имеющих "истинное" пересечение)
64 INSertion (точка вставки. Берет ее из блоков (INSERT), однострочных и многострочных текстов и атрибутов)
128 PERpendicular (перпендикуляр)
256 TANgent (касательная)
512 NEArest (ближайшая. Вычисляется точка примитива, ближайшая к текущему положению курсора)

1024 Clears all object snaps
2048 APParent Intersection (кажущееся пересечение. Если примитивы лежат в разном уровне (значение координаты Z), то фактически они не пересекаются, но привязаться к точке их визуального перекрестия можно. Не всегда работает корректно, если текущий план и система координат не совпадают).
4096 EXTension (продолжение. Лично я ею почему-то не пользуюсь, подробностей работы не знаю совсем)
8192 PARalle (параллельно. Насчет подробностей - то же, что и для 4096)

4.5 АВТОМАТИЧЕСКАЯ МОДЕРНИЗАЦИЯ ЧЕРТЕЖЕЙ СРЕДСТВАМИ АВТОЛИСПА

Внутреннее представление изображений Автокада

Чуть ли не самой главной особенностью Автолиспа является то, что он позволяет осуществлять доступ к графической базе данных (ГБД) Автокада, многократно умножая возможности адаптации последнего к какому-либо типу задач. Попытаемся вкратце показать, как можно использовать возможности Автолиспа для работы непосредственно с объектами чертежа.

Любой создаваемый в Автокаде чертеж состоит из примитивов, геометрическое описание которых хранится в специальном формате (формате Автокада) в файле чертежа (расширение .dwg). При загрузке чертежа Автокад заполняет графическую базу данных: записывает системные Настройки, создает список объектов и вносит в ГБД геометрическое описание этих объектов, присваивая каждому примитиву уникальное имя. В сеансе редактирования каждый примитив (отрезок, дуга, окружность и т.п.) имеет свое имя, по которому его распознает сам Автокад. Для оперирования примитивами необходимо в программе на Автолиспе сначала найти имя примитива в базе данных Автокада, чтобы потом изменять геометрические характеристики примитива. Попробуем извлечь это имя из ГБД при помощи Автолиспа.

Каждый примитив Автокада представляет собой достаточно сложную совокупность данных, организованную в виде иерархического списка. Графические примитивы могут быть *простыми*, такими как линии, окружности, дуги окружностей, строки текста (изображаемые

соответственно командами LINE, CIRCLE, ARC, TEXT), и *составными*. Составными примитивами являются, например, полилинии, создаваемые командой PLINE. Составные примитивы делятся на *субпримитивы*. Для полилиний субпримитивами являются вершины. Каждый примитив или субпримитив представлен в Автокаде списком вида (A_1 A_2 ... A_N), причем каждый элемент данного списка является или точечной парой: $A_i=(B_i . C_i)$ (раздел 3.5), или списком: $A_i=(B_i C_{i1} ... C_{ik})$. В обоих случаях B_i является числовым кодом типа данных, записанных в списке или точечной паре. Первый элемент исходного списка (A_1) является точечной парой **(-1 . <внутреннее имя примитива>)**, в которой записано буквенно-цифровое имя примитива, присвоенное ему системой Автокад. Вторым элементом списка также является точечная пара: **(0 . "тип примитива")**, элемент "тип примитива" при этом может соответствовать примитивам, таким как "LINE", "CIRCLE", "ARC", а также субпримитивам, например, тип "VERTEX" - вершины полилиний. Последующие элементы списка представляют собой различные атрибуты примитива, если их значения отличаются от значений по умолчанию. Набор атрибутов зависит от версии Автокада, и они распознаются по своему числовому коду. Например, пара (62 . A) определяет цвет примитива в зависимости от значения A.

Такие числовые коды называтся кодами формата DXF (Drawing exchange Format - формат обмена рисунками). Каждый подсписок всегда имеет две части. Первая (голова посписка) - код DXF, вторая (хвост подсписка) - данные.. Код 8 говорит о том, что следующее за ним число - номер слоя. Код 10 - начальная точка примитива, код 11 - конечная и т.п. Наконец, для дуги окружности (тип "ARC") 10 - кодирует координаты центра дуги, 40 - радиус дуги, 50 и 51 - соответственно начальный и конечный углы дуги в радианах.

В случае составных примитивов каждый субпримитив кодируется аналогично, например, для вершин (VERTEX) полилиний число 10 кодирует их координаты.

Отметим, что набор кодов DXF различен для примитивов разных типов.

Получение информации о примитивах

Нарисуйте отрезок. (команда line). Для того, чтобы указать на примитив, используются различные способы. В частности, можно указать последний нарисованный элемент.

Взятие имени последнего созданного примитива: `(entlast)`

Имя примитива необходимо сохранить в переменной с помощью оператора присваивания setq. Введите в командной строке:

Command: `(setq ENAME (entlast))`

Автокад возвращает: `<Entity name: 60000018>`

Тем самым мы присвоили переменной ENAME имя последнего примитива (в данном случае отрезка). Имена примитивов в Автокаде - шестнадцатеричные величины; имя примитива

может быть, например, таким: 60000A14. Используя это имя, вы можете при помощи функции ENTGET получить доступ к данным, связанным с примитивом:

Доступ к данным: `(entget имя)`

Данные о примитиве выдаются списком, поэтому должны быть записаны в переменную, которую потом можно анализировать:

```
(setq EDATA (ENTGET ENAME))
```

В результате выполнения команды вы получите сообщение, содержащее в своей структуре всю информацию о примитиве. Начиная от его имени и заканчивая слоем и цветом:

```
((-1.<Имя примитива: 60000020>)
      (0."LINE") (8."O") (10 1.0 2.0 0.0) (11 6.0 6.0 0.0))
```

Заметим, что не все примитивы столь легко читаемы.

Если цвет или тип линии не указываются, значит они соответствуют типам линии и цвету для заданного слоя.

Команды Автолиспа для анализа и обработки изображений

В системе Автокад имеется ряд функций, обеспечивающих доступ к внутреннему представлению графического примитива, а также выполняющих модификации примитивов путем изменения представляющих их списков.

Функция **ENTNEXT** возвращает *внутреннее программное имя* примитива или субпримитива изображения. Ее формат: (ENTNEXT P1) или (ENTNEXT). Если аргумент P1 отсутствует, то возвращается первый из примитивов чертежа. При наличии P1 функция возвращает примитив, следующий за P1 во внутреннем представлении изображения. Если такой отсутствует, ENTNEXT возвращает NIL. Если в качестве P1 указан составной примитив, то ENTNEXT возвращает первый субпримитив P1, затем последующий и т.д. Функции ENTNEXT аналогична функциям FINDFIRST и FINDNEXT системы Турбо-Паскаль, которые используются для поиска файлов.

Функция **(ENTGET P1)** с аргументом, являющимся внутренним программным именем примитива, возвращает *список его внутреннего представления*. Структура этого списка рассмотрена в предыдущем разделе.

Функция **(ENTMOD S1)** выполняет изменения графического примитива. Аргументом S1 является список внутреннего представления примитива, и предполагается, что над списком были произведены необходимые изменения средствами Автолиспа. На процесс изменения изображений накладываются некоторые ограничения, например, нельзя менять тип примитива. Если ENTMOD применяется к субпримитиву, то видимых изменений на экране не происходит. Для изменения изображения составного примитива, когда над всеми его субпримитивами

произведены необходимые изменения функцией ENTNEXT, необходимо выполнить функцию **(ENTUPD P1)**, аргумент P1 которой является внутренним программным именем составного примитива.

Функция **(ENTDEL P1)** удаляет графические примитивы из изображения. Аргументом функции должно быть программное имя удаляемого примитива. Функция ENTDEL не удаляет субпримитивов. Пока не закончился сеанс работы с Автокадом, удаленный примитив может быть восстановлен той же функцией ENTDEL с аргументом P1 - именем удаленного примитива.

Если требуется получить от пользователя необходимый примитив, используется команда **(ENTSEL <строка-подсказка>).** После появления строки-подсказки выбирается единичный примитив, при этом выбор должен быть сделан указанием на примитив. Возвращается список, первый элемент которого - имя выбранного примитива, а второй - координаты точки, в которой он был указан. Строка-подсказка используется для запроса примитива; если она не указана, то выдается стандартный запрос "Выберите объекты:".

Если изображение не содержит составных примитивов, то для его обработки обычно используется приведенный ниже фрагмент циклической программы.

```
(SETQ P (ENTNEXT))     ;выделение первого примитива из;  изображения P - его имя
(WHILE P               ;пока примитивы не закончатся выполняется цикл их обработки
 (SETQ S (ENTGET P))   ;получения списка свойств примитива P  S - список

    <ПРЕОБРАЗОВАНИЕ СПИСКА S ДЛЯ МОДЕРНИЗАЦИИ ИЗОБРАЖЕНИЯ>

 (ENTMOD S)            ; изменение изображения на экране по модифицированному списку S
 (SETQ P (ENTNEXT P))  ; получение следующего примитива в изображении, если его нет P=NIL
)                      ; конец цикла WHILE
```

Пример модернизации изображения

Для иллюстрации работы функций, приведенных в предыдущем разделе, рассмотрим следующую задачу. Пусть на экране изображены отрезки прямых линий и окружности. Требуется радиусы всех окружностей уменьшить в 2 раза и каждый отрезок также сделать вдвое меньшей длины, сохранив положение его центра.

Для решения задачи определим две вспомогательные рекурсивные функции (GETVAL S K) и (PUTVAL S K V). Аргументами функции GETVAL являются список свойств S некоторого

графического примитива и числовой код K искомого свойства. Функция GETVAL возвращает значение искомого свойства. Если заданный числовой код не найден, функция возвращает NIL. Аргументами PUTVAL являются список свойств S, числовой код K и значение свойства V. Функция находит свойство по коду K и заменяет его значение на V, возвращая при этом измененный список. Если свойство не найдено, возвращается NIL.

Функция GETVAL последовательно просматривает головы списка свойств примитива ((K_1 V_1) (K_2 V_2) ... (K_N V_N)) и при совпадении K_i с аргументом K выдает значение V_i. Заметим, что для получения элемента V_i из точечной пары (K_i . V_i) используется функция CDR, а не CADR, как â случае списка (K_i V_i) (раздел 3.5). Если же свойство представлено списком (K_i V_{i1} ... V_{ik}), то нужно также использовать CDR для получения всего списка значений.

```
(DEFUN GETVAL (S K)
(COND
  ((NULL S) NIL)
  ((EQ K (CAAR S)) (CDAR S))
  (T (GETVAL (CDR S) K))
))
```

Функция PUTVAL определяется аналогично.

```
(DEFUN PUTVAL (S K V)
(COND
  ((NULL S) NIL)
  ((EQ K (CAAR S)) (CONS (CONS K V)(CDR S)))
  (T (CONS (CAR S) (PUTVAL (CDR S) K V)))
))
```

Теперь можно определить функцию, выполняющую преобразование чертежа. Определяемая функция M2 последовательно просматривает все примитивы чертежа, и если встретилась окружность, то значение ее радиуса в списке свойств уменьшается в 2 раза (числовой код свойства - 40). Если встретился отрезок с координатами концов (X1, Y1) и (X2, Y2), то, очевидно, координаты “укороченного” пополам относительно центра отрезка будут равны (X1N=(X1+XS)/2, Y1N=(Y1+YS)/2) и (X2N=(X2+XS)/2, Y2N=(Y2+YS)/2), где (XS, YS) - координаты центра отрезка: (XS=(X1+X2)/2, YS=(Y1+Y2)/2). Просмотр выполняется по схеме, приведенной в предыдущем разделе.

Процедура CHANGECIRCLE меняет список S для окружности, уменьшая радиус в 2 раза и изображает изменения на экране.

```
(DEFUN  CHANGECIRCLE (S) ; Аргумент S - список свойств окр.
```

```
 (SETQ R (GETVAL S 40))        ; извлечение радиуса
 (SETQ R (/ R 2))              ; уменьшение радиуса в 2 раза
 (SETQ S (PUTVAL S 40 R))      ; вставка нового значения радиуса
                                 ; в список S
 (ENTMOD S)                    ; модернизация изображения на экране
)
; Процедура CHANGELINE меняет список S для отрезка, уменьшая
; его в 2 раза относительно центра, и отображает изменения
(DEFUN CHANGELINE (S)
 (SETQ A1 (GETVAL S 10))              ; извлечение списка координат
                                    ;          начальной точки
 (SETQ A2 (GETVAL S 11))              ; извлечение списка координат
                                        ; конечной точки
 (SETQ XS (/(+(CAR A1)(CAR A2))2))    ; вычисление координат
 (SETQ YS (/(+(CADR A1)(CADR A2))2)) ; середины отрезка
 (SETQ X1N (/(+(CAR A1)XS)2))   ;         вычисление новых координат
 (SETQ Y1N (/(+(CADR A1)YS)2)) ; начальной точки
 (SETQ X2N (/(+(CAR A2)XS)2))   ;         вычисление новых координат
 (SETQ Y2N (/(+(CADR A2)YS)2)) ; конечной точки
  ; замена в списках (X Y Z) , координат X,Y новыми значениями
 (SETQ A1 (CONS X1N(CONS Y1N (CDDR A1))))
 (SETQ A2 (CONS X2N(CONS Y2N (CDDR A2))))
  ; установка новых значений координат начальной точки
 (SETQ S (PUTVAL S 10 A1))
  ; установка новых значений координат конечной точки
 (SETQ S (PUTVAL S 11 A2))
 (ENTMOD S)          ; отображение модифицированного отрезка
)
; Функция M2 организует цикл просмотра всех примитивов и если их
; тип оказывается LINE или CIRCLE, выполняется вызов
; соответствующих процедур по их модификации
(DEFUN M2 NIL
 (SETQ P (ENTNEXT))       ; выбор первого примитива
 (WHILE P                 ; цикл перебора примитивов
  (SETQ S (ENTGET P))     ; извлечение списка свойств примитива
```

```
 (SETQ A (GETVAL S 0))   ; извлечение названия типа примитива
 (IF (EQ A "CIRCLE")     ; проверка, что примитив - окружность
  (CHANGECIRCLE S)       ; вызов процедуры модификации
                         ; окружности
  NIL                    ; ELSE для IF
 )                       ; конец IF
 (IF (EQ A "LINE")       ; проверка, что примитив - отрезок
  (CHANGELINE S)         ; вызов процедуры модификации отрезка
  NIL                    ; ELSE для IF
 )                       ; конец IF
 (SETQ P (ENTNEXT P))    ; извлечение следующего примитива
 )                       ; конец WHILE
)                        ; конец M2
```

4.6 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какова отличительная особенность способа хранения чертежей в Автокаде?
2. В чем отличие растрового способа хранения изображений от векторного?
3. Как вызвать команду Автокада из функции на Автолиспе?
4. Что такое графический примитив? Приведите примеры графических примитивов.
5. Для чего используется команда LIMITS?
6. Перечислите известные вам функции ввода языка Автолисп.
7. В каком виде представляются чертежи “внутри” системы Автокад?
8. Что такое “субпримитив”?
9. Каково назначение функций ENTNEXT, ENTGET, ENTDEL, ENTMOD, ENTUPD?

4.7 УПРАЖНЕНИЯ

1 Написать функцию SQUARES для построения чертежа, изображенного на рис. 8. Стороны наибольшего квадрата имеют длина L и параллельны координатным осям. Вершинами внутренних квадратов являются середины сторон больших квадратов. Общее количество квадратов задается пользователем. На рис. 8 изображен результат построения для N=4.

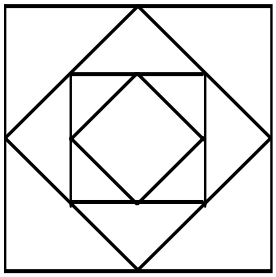


Рис. 8. Результат работы функции SQUARES

Запишем программу на Автолиспе для построения данного чертежа. Пусть некоторая точка Q1 имеет координаты (X1 Y1), а точка Q2 - координаты (X2 Y2). Тогда, очевидно, точка QS, являющаяся серединой отрезка, соединяющего точки Q1 и Q2, имеет координаты ((X1+X2)/2 (Y1+Y2)/2). Координаты точки QS легко вычислить на Лиспе, если учесть, что функция (CAR Q1) возвращает голову списка (X1 Y1), то есть координату X1, а функция (CADR Q1) извлекает второй элемент списка - координату Y1.

```
(DEFUN SQUARES (N)        ; N-количество квадратов
  (COMMAND "LIMITS" '(0 0) '(320 200)); уст. границ изображения
  (COMMAND "ZOOM" "A")    ;показ новых границ в граф. редакторе
  (SETQ P1(GETPOINT "Введите левую нижнюю точку квадрата"))
  ; точки P1,P2,P3,P4 являются вершинами квадрата
  ; величина L является длиной исходного квадрата
  (SETQ L (GETINT "Введите длину стороны квадрата"))
  (SETQ P2 (POLAR P1 0 L))          ;вправо от P1 на величину L
  (SETQ P3 (POLAR P2 (/ PI 2) L))   ; вверх от P2 на L
  (SETQ P4 (POLAR P3 (- PI) L))     ; влево от P3 на L
  (COMMAND "LINE" P1 P2 P3 P4 "C") ; изображение квадрата
  (SETQ I 1)                ; начальное значение счетчика I=1
  (WHILE (< I N)            ; условие окончания цикла:
                            ; I=N все квадраты постр.
    (SETQ P1B P1)           ; сохранение значений вершин
    (SETQ P2B P2)           ; предыдущего квадрата
    (SETQ P3B P3)
    (SETQ P4B P4)
    ; вычисление координаты X середины отрезка P1B,P2B
    (SETQ XS (/(+(CAR P1B)(CAR P2B))2))
    ; вычисление координаты Y середины отрезка P1B,P2B
    (SETQ YS (/(+(CADR P1B)(CADR P2B))2))
    ; построение точки P1 - середины отрезка P1B,P2B
```

```
    (SETQ P1 (LIST XS YS))
    ; построение середины отрезка P2B,P3B
    (SETQ XS (/(+(CAR P2B)(CAR P3B))2))
    (SETQ YS (/(+(CADR P2B)(CADR P3B))2))
    (SETQ P2 (LIST XS YS))
    ; построение середины отрезка P3B,P4B
    (SETQ XS (/(+(CAR P3B)(CAR P4B))2))
    (SETQ YS (/(+(CADR P3B)(CADR P4B))2))
    (SETQ P3 (LIST XS YS))
    ; построение середины отрезка P4B,P1B
    (SETQ XS (/(+(CAR P4B)(CAR P1B))2))
    (SETQ YS (/(+(CADR P4B)(CADR P1B))2))
    (SETQ P4 (LIST XS YS))
    (COMMAND "LINE" P1 P2 P3 P4 "C"); построение нового
                                    ; квадрата
    (SETQ I (+ I 1))                ; увеличение счетчика
                                    ; нарисованных квадратов
  )                                 ; конец цикла WHILE
  (COMMAND "REDRAW")                ; очистка вспомогательных
                                    ; элементов
)                                   ; конец функции SQUARES
```

Если приведенная программа находится в файле RIS.LSP, то для получения изображения, приведенного на рис. 8, достаточно в строке команд Автокада набрать команду (LOAD "RIS"), выполняющую загрузку модуля, и команду (SQUARES 4), выполняющую построение чертежа с 4 квадратами.

2 Пусть на экране изображены отрезки прямых линий и окружности. Требуется радиусы всех окружностей уменьшить в 2 раза и каждый отрезок также сделать вдвое меньшей длины, сохранив положение его центра.

Для решения задачи определим две вспомогательные рекурсивные функции (GETVAL S K) и (PUTVAL S K V). Аргументами функции GETVAL являются список свойств S некоторого графического примитива и числовой код K искомого свойства. Функция GETVAL возвращает значение искомого свойства. Если заданный числовой код не найден, функция возвращает NIL. Аргументами PUTVAL являются список свойств S, числовой код K и значение свойства V. Функция находит свойство по коду K и заменяет его значение на V, возвращая при этом измененный список. Если свойство не найдено, возвращается NIL.

Функция GETVAL последовательно просматривает головы списка свойств примитива ((K_1 V_1) (K_2 V_2) ... (K_N V_N)) и при совпадении K_i с аргументом K выдает значение V_i. Заметим, что для получения элемента V_i из точечной пары (K_i . V_i) используется функция CDR, а не CADR, как в случае списка (K_i V_i) (раздел 3.5). Если же свойство представлено списком (K_i V_{i1} ... V_{ik}), то нужно также использовать CDR для получения всего списка значений.

```
    (DEFUN GETVAL (S K)
    (COND
```

```
    ((NULL S) NIL)
    ((EQ K (CAAR S)) (CDAR S))
    (T (GETVAL (CDR S) K))
  ))
```

Функция PUTVAL определяется аналогично.

```
(DEFUN PUTVAL (S K V)
(COND
  ((NULL S) NIL)
  ((EQ K (CAAR S)) (CONS (CONS K V)(CDR S)))
  (T (CONS (CAR S) (PUTVAL (CDR S) K V)))
))
```

Теперь можно определить функцию, выполняющую преобразование чертежа. Определяемая функция M2 последовательно просматривает все примитивы чертежа, и если встретилась окружность, то значение ее радиуса в списке свойств уменьшается в 2 раза (числовой код свойства - 40). Если встретился отрезок с координатами концов (X1,Y1) и (X2, Y2), то, очевидно, координаты “укороченного” пополам относительно центра отрезка будут равны (X1N=(X1+XS)/2, Y1N=(Y1+YS)/2) и (X2N=(X2+XS)/2, Y2N=(Y2+YS)/2), где (XS, YS) - координаты центра отрезка: (XS=(X1+X2)/2, YS=(Y1+Y2)/2). Просмотр выполняется по схеме, приведенной в предыдущем разделе.

Процедура CHANGECIRCLE меняет список S для окружности, уменьшая радиус в 2 раза, и изображает изменения на экране.

```
(DEFUN  CHANGECIRCLE (S) ; Аргумент S - список свойств окр.
  (SETQ R (GETVAL S 40)) ; извлечение радиуса
  (SETQ R (/ R 2))       ; уменьшение радиуса в 2 раза
  (SETQ S (PUTVAL S 40 R)); вставка нового значения радиуса
                         ; в список S
  (ENTMOD S)             ; модернизация изображения на экране
)
; Процедура CHANGELINE меняет список S для отрезка, уменьшая
; его в 2 раза относительно центра, и отображает изменения
(DEFUN CHANGELINE (S)
(SETQ A1 (GETVAL S 10))  ; извлечение списка координат
                         ; начальной точки
  (SETQ A2 (GETVAL S 11)); извлечение списка координат
                         ; конечной точки
  (SETQ XS (/(+(CAR A1)(CAR A2))2))     ; вычисление координат
  (SETQ YS (/(+(CADR A1)(CADR A2))2))   ; середины отрезка
  (SETQ X1N (/(+(CAR A1)XS)2)); вычисление новых координат
  (SETQ Y1N (/(+(CADR A1)YS)2))         ; начальной точки
  (SETQ X2N (/(+(CAR A2)XS)2)); вычисление новых координат
  (SETQ Y2N (/(+(CADR A2)YS)2)); конечной точки
  ; замена в списках (X Y Z) координат X,Y новыми значениями
  (SETQ A1 (CONS X1N(CONS Y1N (CDDR A1))))
  (SETQ A2 (CONS X2N(CONS Y2N (CDDR A2))))
  ; установка новых значений координат начальной точки
  (SETQ S (PUTVAL S 10 A1))
  ; установка новых значений координат конечной точки
```

```
  (SETQ S (PUTVAL S 11 A2))
  (ENTMOD S)         ; отображение модифицированного отрезка
)
; Функция M2 организует цикл просмотра всех примитивов и если
; их тип оказывается LINE или CIRCLE, выполняется вызов
; соответствующих процедур по их модификации
(DEFUN M2 NIL
 (SETQ P (ENTNEXT))       ; выбор первого примитива
 (WHILE P                 ; цикл перебора примитивов
  (SETQ S (ENTGET P))     ; извлечение списка свойств примитива
  (SETQ A (GETVAL S 0))   ; извлечение названия типа примитива
  (IF (EQ A "CIRCLE")     ; проверка, что примитив - окружность
    (CHANGECIRCLE S)      ; вызов процедуры модификации
                          ; окружности
    NIL                   ; ELSE для IF
   )                      ; конец IF
   (IF (EQ A "LINE")      ; проверка, что примитив - отрезок
    (CHANGELINE S)        ; вызов процедуры модификации отрезка
    NIL                   ; ELSE для IF
   )                      ; конец IF
  (SETQ P (ENTNEXT P))    ; извлечение следующего примитива
  )                       ; конец WHILE
)                         ; конец M2
```

4.8 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

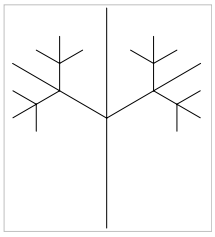
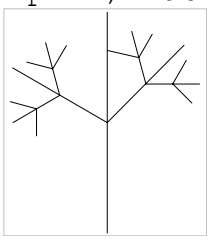
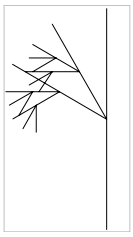
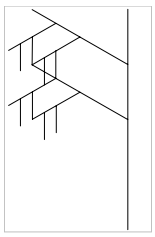
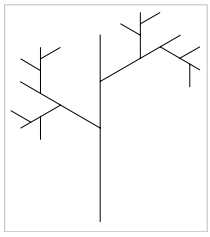
1. Функция рисования правильного многоугольника по длине стороны L, центру P и количеству сторон N выглядит так:

   а)
```
(defun mng (p n l)
   (setq a1 (/ (* 2 PI) n))
   (setq r (/ l (* 2 (sin (/ a1 2)))))
   (setq i 0)
   (command "pline")
   (while (< i n)
      (setq p1 (polar p1 (* a1 i) r))
      (setq i (+ i 1))
      (command p1)
   )
   (command "c")
)
```

b)
```
(defun mng (p n l)
   (setq a1 (/ (* 2 PI) n))
   (setq r (/ l (* 2 (sin (/ a1 2)))))
   (setq p1 (polar p 0 r))
   (setq i 1)
   (while (<= i n)
      (setq p2 (polar p (* a1 i) r))
      (setq i (+ i 1))
      (command "LINE" p1 p2 "")
   (setq p1 p2)
   )
)
```
c)
```
(defun mng (p n l)
   (setq a1 (/ PI n))
   (setq r (/ l (* 2 (sin a1))))
   (setq i 0)
   (command "pline")
   (while (< i n)
      (setq p1 (polar p (* a1 i) r))
      (setq i (+ i 1))
      (command p1)
   )
   (command "c")
)
```
d)
```
(defun mng (p n l)
   (setq a1 (/ (* 2 PI) n))
   (setq r (/ l (* 2 (sin (/ a1 2)))))
   (setq i 0)
   (command "pline")
   (while (< i n)
      (setq p1 (polar p (* a1 i) r))
      (setq i (+ i 1))
      (command p1)
   )
   (command "c")
)
```
e)
```
(defun mng (p n l)
   (setq a1 (/ 360 n))
   (setq r (/ l (* 2 (sin a1))))
   (setq i 0)
   (command "line")
   (while (< i n)
      (setq p1 (polar p (* a1 i) r))
      (setq i (+ i 1))
      (command p1)
   )
   (command "c")
)
```

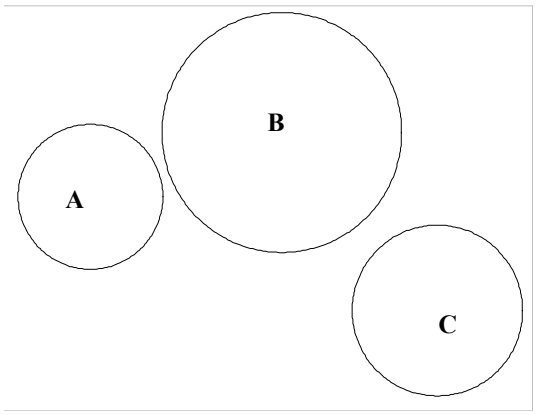
2. Какую картинку нарисует приведенная ниже функция FUN2, будучи вызванной указанным образом:

```
(defun fun2 (a b c d)
  (setq e (polar a b c))
  (command "line" a e "")
  (if (> d 1) (fun1 (polar a b (* 0.5 c)) (+ b (/ PI 3))
                (/ c 2) (- d 1)) nil )
  (if (> d 1) (fun1 (polar a b (* 0.75 c)) (- b (/ PI 3))
                (/ c 2) (- d 1)) nil )
)
(fun1 '(150 2) (/ pi 2) 195 4))
```

a)  b)  c)  d)  e) 

3. Что сделает функция FUN3, если чертеж содержит 3 окружности (см. рисунок), а функции GETVAL и PUTVAL определены так же, как в упражнении 2?

```
(defun fun3 ()
  (setq mr 0)
  (setq p (entnext))
  (while p
   (setq s (entget p))
   (setq r (getval s 40))
   (if (< mr r) (setq mr r) nil)
   (if (< mr r) (setq mpp (getval s 10)) nil)
   (setq p (entnext p))
  )
  (setq p (entnext))
  (while p
   (setq s (entget p))
   (setq  s(putval s 10 mpp))
   (entmod s)
   (setq p (entnext p))
  )
)
```



a) перенесет центры всех окружностей в центр окружности A

b) перенесет центры всех окружностей в центр окружности B

c) перенесет центры всех окружностей в центр окружности C

d) оставит чертеж без изменений

e) вызовет ошибку

4. Если функции GETVAL и PUTVAL определены, как в упражнении 2, и P – внутренне имя окружности, то для ее перемещения вдоль оси Y на величину радиуса необходимо выполнить следующий код:

a)
```
(SETQ A (ENTGET P))
(SETQ B (GETVAL A 40))
(SETQ C (GETVAL A 10))
(SETQ C (CONS (CAR C) (CONS (+ (CADR C) B) (CDDR C))))
(PUTVAL A 10 C)
(ENTMOD A)
```

b)
```
(SETQ A (ENTGET P))
(SETQ B (GETVAL A 40))
(SETQ C (GETVAL A 10))
(SETQ C (CONS (CAR C) (+ (CDR C) B)))
(PUTVAL A 10 C)
(ENTUPD A)
```

c)
```
(SETQ A (GETVAL P 40))
(SETQ B (GETVAL P 10))
(SETQ B (CONS (CAR B) (+ (CDR B) A)))
(PUTVAL P 10 C)
(ENTMOD P)
```

d)
```
(SETQ A (ENTGET P))
(SETQ B (GETVAL A 40))
(SETQ C (GETVAL A 10))
(SETQ C (CONS (CAR C) (+ (CADR C) (CAR B))))
(PUTVAL A 40 C)
(ENTMOD A)
```

e)
```
(SETQ A (ENTGET P))
(SETQ B (GETVAL A 40))
(SETQ C (GETVAL A B))
(SETQ C (CONS (CAR C) (+ (CADR C) (CAR B))))
(PUTVAL A B C)
(ENTMOD A)
```

Ответы: 1.b,d;2.e;3.e;4.a.

4.9 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №5

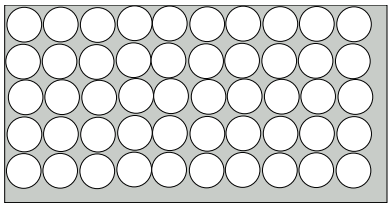
ПОСТРОЕНИЯ ПАРАМЕТРИЧЕСКИХ ЧЕРТЕЖЕЙ В СРЕДЕ AUTOCAD

Задание: С использованием языка AutoLISP, построить чертеж

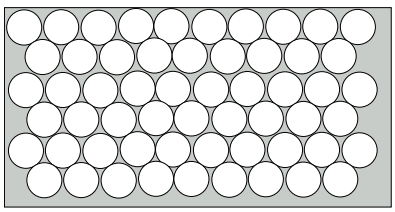
Код программ и тестовые примеры привести в отчете. Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

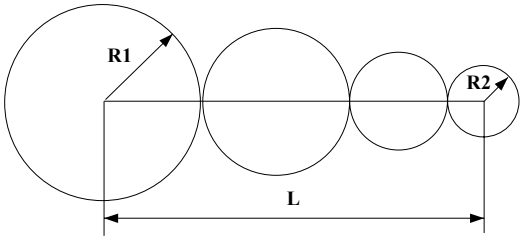
1. В прямоугольнике заданного размера разместить столько окружностей радиуса r, сколько поместится (см. рисунок).



2. В прямоугольнике заданного размера разместить столько окружностей радиуса r, сколько поместится (см. рисунок).

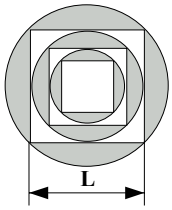


3. По заданному натуральному числу N, радиусу наибольшей окружности R1 и радиусу наименьшей окружности R2 построить приведенный рисунок (радиусы соседних окружностей отличаются на одинаковую величину). Указать расстояние между центрами крайних окружностей.

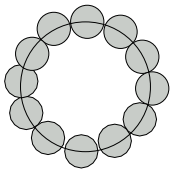


4. Разбить окружность на N равных частей и соединить каждую точку с каждой.

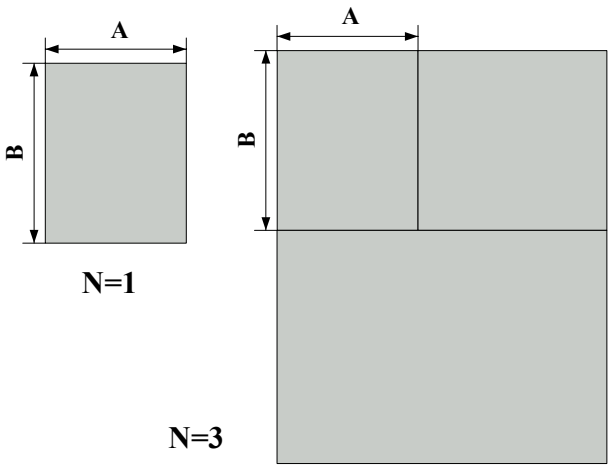
5. По начальному радиусу и числу окружностей N построить рисунок:



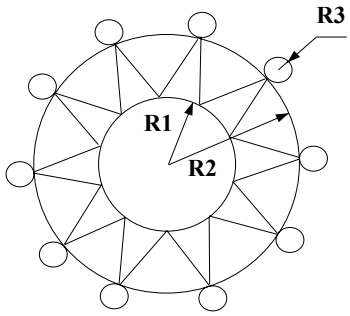
6. По радиусу R и натуральному числу N построить рисунок из N соприкасающихся окружностей радиуса R, причем центры этих окружностей должны лежать на окружности с заданным центром.



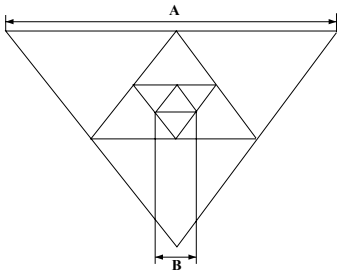
7. Нарисовать иллюстрацию к правилу “золотого сечения”” $\frac{A}{B} = \frac{B}{A+B}$ заданное N раз. Длина A меньшей стороны исходного прямоугольника задана. В качестве меньшей стороны нового прямоугольника выбирается большая сторона предыдущего прямоугольника. Верхний левый угол всех прямоугольников совпадает.



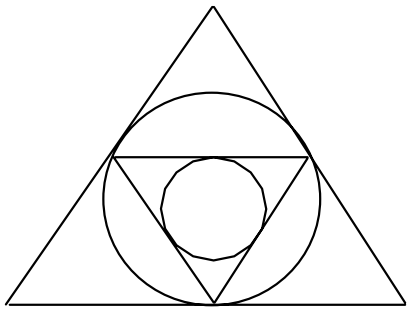
8. Нарисовать “правильную звездочку”” с окружностями на концах по трем радиусам. Задано R1, R2, R3 и количество зубцов N.



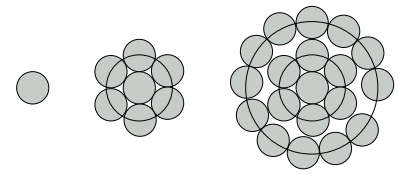
9. По заданной стороне L и натуральному числу N построить рисунок из N вписанных равносторонних треугольников. Вынести размеры горизонтальных сторон наибольшего и наименьшего из треугольников. Размерные линии не должны пересекаться.



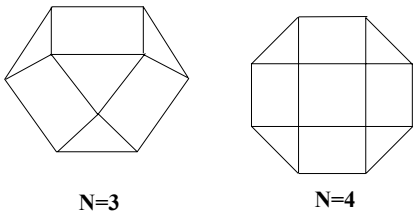
10. По заданной стороне большого треугольника L и числу окружностей N построить приведенный рисунок. Все треугольники равносторонние. На рисунке приведен пример для N=2.



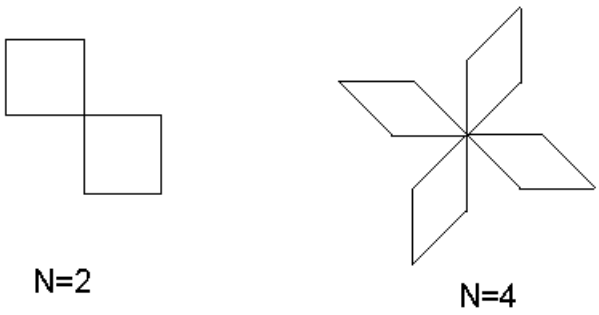
11. По радиусу R и натуральному числу N построить рисунок из N концентрических окружностей, состоящих из окружностей радиуса R. Маленькие окружности не должны пересекаться, радиус больших окружностей кратен радиусу маленьких окружностей.



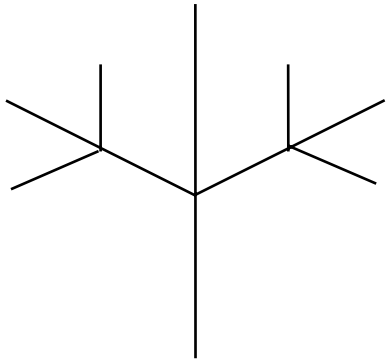
12. По заданному числу N и длине L построить плоскую проекцию 3-мерного тела, у которого основания параллельны и являются правильными многоугольниками (с N и 2N сторонами). Длина всех сторон оснований равна L. N боковых граней тела являются прямоугольниками, оставшиеся N - равнобедренными треугольниками.



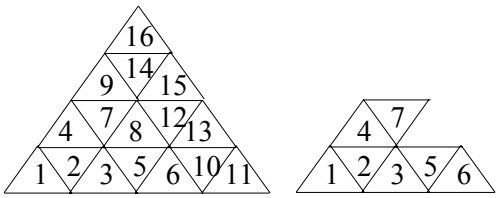
13. По заданному числу лепестков N и размеру стороны L построить цветок, приведенный на рисунке. Углы в ромбах зависят от N.



14. По заданному количеству лучей N, построить снежинку, состоящую из веточек формы приведенной на рисунке (длинна ствола L задается, угол ветвления α зависит от N)



15. По заданному числу N, крайней левой точке A и стороне маленького треугольника L построить N равносторонних треугольников, располагая их так, как показано на рисунке:



4.9 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №6

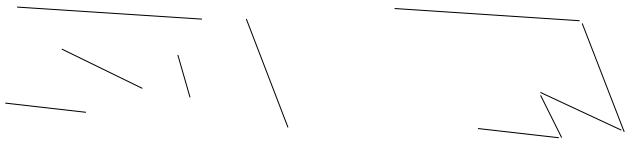
МОДИФИКАЦИЯ ЧЕРТЕЖЕЙ В СРЕДЕ AUTOCAD

Задание: Создать произвольное изображение отдельной программой. Изображение должно содержать помимо примитивов, необходимых для работы, еще какие-либо объекты. С использованием языка AutoLISP, модифицировать чертеж, созданный в среде AutoCAD

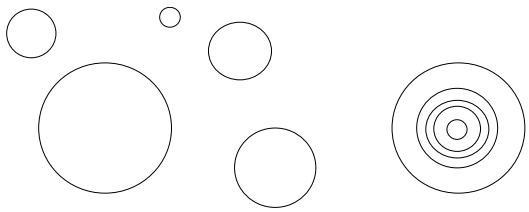
Код программ и тестовые примеры привести в отчете. Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

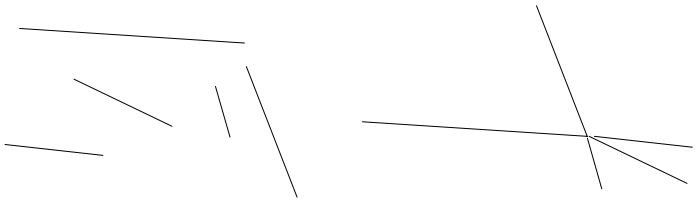
1. Из произвольно размещенных на плоскости отрезков составить ломаную, соединяя отрезки в порядке рисования.



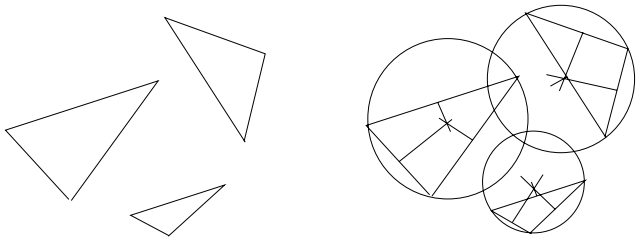
2. Из произвольных окружностей выбрать максимальную и разместить все остальные в ее центре.



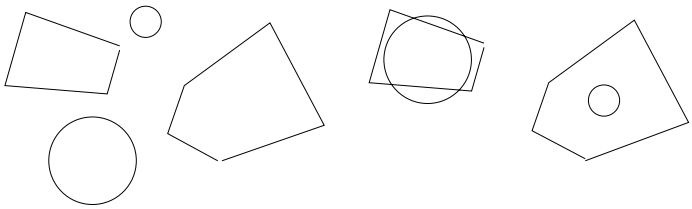
3. Произвольно размещенные на плоскости отрезки стянуть в одну заданную точку.



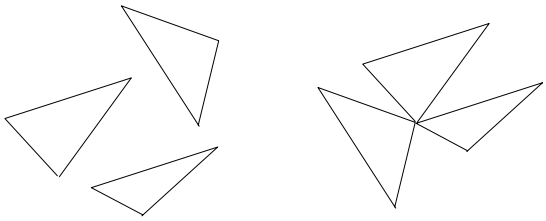
4. Описать окружности вокруг заданных треугольников.



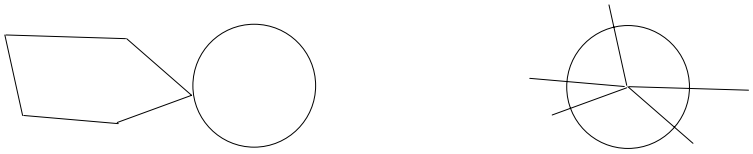
5. Даны произвольные четырехугольники и окружности. Поместить окружности в центр четырехугольников в порядке рисования.



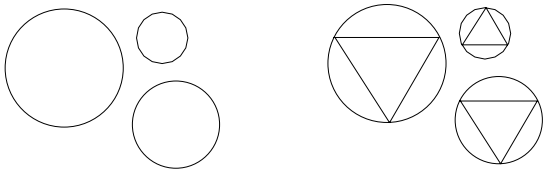
6. Стянуть произвольно размещенные треугольники в заданную точку первой вершиной в порядке рисования.



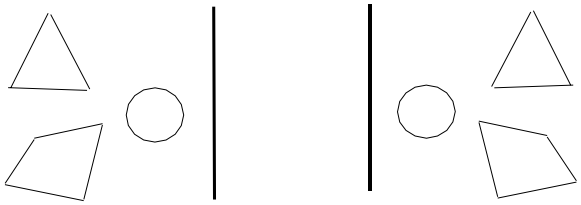
7. Перерисовать заданный многоугольник в центр окружности, заменив его совокупностью отрезков.



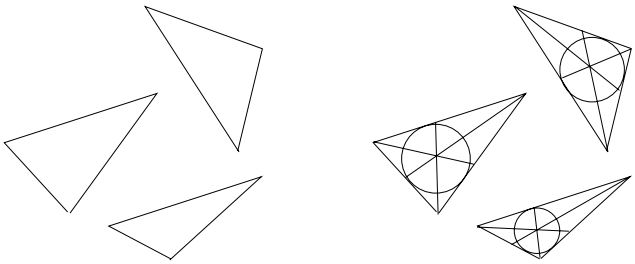
8. Вписать в окружности радиуса меньшего r равносторонние треугольники прямо, а в окружности радиуса большего r - вверх ногами.



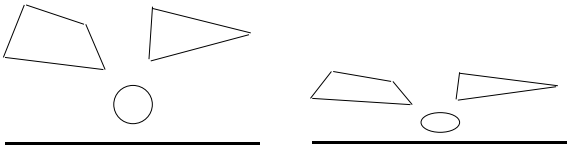
9. Построить зеркальное отображение рисунка, состоящего из многоугольников и окружностей, относительно заданной прямой.



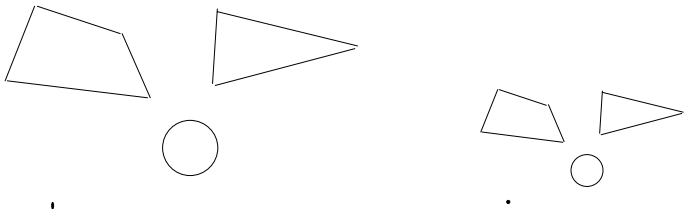
10. В данные треугольники вписать окружности.



11. Рисунок, состоящий из окружностей и многоугольников, сжать по оси Y. Координату Y оси сжатия запросить у пользователя.



12. Выполнить преобразование гомотетии. Центр гомотетии и коэффициент запросить у пользователя.



13. Повернуть чертеж на заданный угол относительно заданной точки.

14. Для чертежа, состоящего из окружностей и многоугольников, вынести размерные линии на заданное расстояние от фигур.

5 ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА ПРОЛОГ

5.1 СТРУКТУРЫ ДАННЫХ

Большинство языков программирования описывают последовательность действий, которые необходимо совершить для получения результата (процедурное программирование – языки Си, Паскаль), или правила работы с определенным объектом (объектно-ориентированное программирование – Си++, Delphi). Язык Пролог относится к другой категории языков. Программа на Прологе не задает последовательность действий, а лишь описывает имеющуюся информацию и задает вопрос. Интерпретатор Пролога, делая логические выводы из имеющейся информации, ищет ответ на этот вопрос, или в терминах Пролога доказывает цель. Такой подход называется **декларативным подходом к программированию**. Он оказывается очень удобным для решения многих задач.

Информация, составляющая Пролог-программу, представляется в виде **фактов** и **правил**, которые вместе образуют **базу данных**. **Вопрос**, заданный к базе данных, называется **целью**.

Вопросы и правила Пролога отражают сложные логические связи между объектами базы, и интерпретатор Пролога при доказательстве цели способен извлекать скрытую информацию путем логических построений. Математически язык Пролог эквивалентен логической теории предикатов первого порядка, из которой строятся формальные доказательства математических теорем. Название языка образовано из двух слов: **Pro**gramming in **log**ic (программирование логики), отражающих существо языка.

Факты.

Факт – это то, что известно наверняка, т.е. заведомо правильно. Факты Пролога устанавливают определенную смысловую связь между некоторыми понятиями.

Рассмотрим примеры записи фактов на Прологе.

```
студент(иванов).           ; Иванов  является студентом.
нравится(коля,лена).       ; Коле нравится Лена.
дарить(витя,лена,цветы).   ; Витя подарил Лене цветы.
дарить(коля,лена,книга(дюма,три_мушкетера).
; Коля подарил Лене книгу автора Дюма с названием
; Три_мушкетера.
сотрудник(петров,30).      ; Петров является сотрудником
; фирмы, его возраст - 30 лет.
```

На данных примерах видны особенности задания фактов в языке Пролог. Структура факта имеет вид: $p(a_1,a_2,...,a_n)$., где p называется **предикатом**, а $a_1,a_2,...,a_n$ - аргументами предиката. Задание факта завершается точкой.

Предикат по смыслу определяет связь между некоторыми объектами. Синтаксически предикат представляет собой идентификатор, т.е. последовательность букв, цифр и символов подчеркивания, начинающуюся с буквы. *При этом первым символом предиката должна быть малая буква.* Такие идентификаторы в Прологе называются символьными атомами или константами.

Аргументы предикатов могут любой возможный в Прологе тип:

- константы (числа, строки),
- символьные атомы (идентификаторы, начинающиеся с маленькой буквы),
- переменные (идентификаторы, начинающиеся с заглавной буквы или символа подчеркивания),
- списки,
- структуры.

Структуры имеют форму предиката. Примером такого аргумента является *книга(дюма, три_мушкетера)*. Списки и переменные рассмотрены ниже в соответствующих разделах.

В различных диалектах Пролога возможны другие типы данных, в частности, файловые и строковые переменные, существует и разделение чисел на целые и вещественные. Некоторые особенности аргументов рассмотрены в разделе о программировании на Турбо-Прологе.

Вопросы

Стандартный Пролог представляет собой диалоговую систему, поэтому после определения некоторой базы из фактов, можно задать к ней ряд вопросов. Рассмотрим пример:

```
студент(иванов).
нравится(коля,лена).
; заданы несколько фактов
; а теперь задаем вопросы
?-студент(иванов).
; верно ли, что Иванов - студент?
ДА
; ответ Пролога
?- студент(сидоров).
; верно ли, что Сидоров - студент?
НЕТ
?-нравится(коля,лена).
; верно ли, что Коле нравится Лена?
ДА
```

Таким образом, символы "?-" определяют признак вопроса, на который Пролог-система пытается получить ответ. Сам вопрос в Прологе называется еще **целевым утверждением** или **целью**. Процесс поиска ответа называется **согласованием целевого утверждения**, или **доказательством цели**.

Существует ряд правил, по которым производится доказательство целевого утверждения. Для доказательства необходимо, чтобы совпали имена предикатов и были согласованы все аргументы.

Если аргументами являются числа или символьные атомы, то они согласованы только тогда, когда совпадают. Из приведенных примеров видно, что Пролог отвечает на поставленный вопрос положительно только в том случае, если в базе обнаруживается требуемая информация.

Более сложные вопросы можно задавать, если использовать переменные Пролога.

Переменные

Вопросы из предыдущего примера требуют ответа да/нет. Пусть нам требуется узнать подробности. Для этого нам придется задать вопрос с переменными.

Рассмотрим пример задания таких вопросов.

```
студент(иванов).
нравится(коля,лена).
нравится(лена,лена).       ; Лена нравится сама себе
дарить(коля,лена,книга(дюма,три_мушкетера).
?- студент(Кто).           ; Кто является студентом?
Кто=иванов ;
НЕТ
?-нравится(Кому, лена).    ; Кому нравится Лена?
Кому=коля ;
Кому=лена ;
НЕТ
?-нравится(Кому,Кто).      ; Кто кому нравится?
Кому=коля, Кто=лена ;
Кому=лена, Кто=лена ;
НЕТ
?-дарить(коля,лена,Что).
; Что подарил Коля Лене?
Что= книга(дюма,три_мушкетера ;
НЕТ
?-дарить(коля,лена,книга(Автор,Название).
; Какую книгу (какого автора и с каким
; названием) подарил Коля Лене?
Автор=дюма, Название= три_мушкетера ;
НЕТ
```

Из приведенных примеров видно, что интерпретатор Пролога возвращает интересующие нас подробности через значения переменных. Обратите внимание, что после получения первого ответа на вопрос, необходимо ввести символ ";" (точка с запятой), который в стандартном

Прологе означает продолжение поиска. После вывода всех возможных решений, ответом Пролога является "НЕТ", что в данном случае означает лишь, что все возможные решения исчерпаны. Ввод символа, отличного от точки с запятой, прекращает дальнейший поиск решений. Некоторые диалекты Пролога, в частности Турбо-Пролог, при задании вопроса сразу выводят все возможные ответы.

Синтаксически переменные определяются как идентификаторы, начинающиеся с заглавной буквы. Кроме этого, переменными являются идентификаторы, начинающиеся с символа подчеркивания, например, "_кто", "_x" и т.д. При согласовании аргументов, являющихся переменными, возможно несколько случаев, рассмотрение которых позволяет полностью сформулировать правила согласования цели с утверждением базы.

- Переменные величины могут иметь значение, в этом случае, они называются **связанными**. Переменные, которым не присвоено никакое значение, называются **свободными**.
- Перед началом согласования цели и утверждения базы все переменные свободны.
- Если свободная переменная согласовывается с числом, символьным атомом или структурой, то этой переменной присваивается соответствующее значение, и она при дальнейшем согласовании становится связанной.
- При согласовании связанной переменной она заменяется своим значением.
- Если одна переменная свободная, а другая связанная, то свободной переменной присваивается значение связанной.
- При согласовании связанной переменной с константой выполняется проверка на равенство. Таким образом, согласование переменных объединяет вместе как операцию присваивания (':=' в Паскале), так и операцию сравнения ('=' в Паскале).
- Если при согласовании обе переменные были свободными, то они успешно согласуются, оставаясь при этом свободными, но становятся **сцепленными**. Если при дальнейшем согласовании одна из сцепленных переменных получит какое-либо значение, то вторая сцепленная переменная получит такое же значение.

Рассмотренные варианты исчерпывают возможные случаи согласования предикатов.

Рассмотрим еще несколько примеров.

```
нравится(коля,лена).
нравится(лена,лена).
?-нравится(X,X).
X=лена ;
НЕТ
```

Данный вопрос можно сформулировать так: "Кто нравится сам себе?" Действительно, при согласовании цели с первым фактом базы, сначала переменная X принимает значение 'коля', а

при согласовании переменной X в качестве второго аргумента, значение 'коля' не согласуется с константой 'лена'. Для второго утверждения согласование происходит успешно, что и приводит к указанному ответу.

Рассмотрим теперь пример, когда переменные используются в фактах базы. Сформулируем факт, означающий, что каждый человек нравится сам себе.

```
нравится(Человек,Человек).
```

Зададим теперь вопрос:

```
?-нравится(оля,Кто). ; Кто нравится Оле?
Кто=оля
```

При согласовании сначала переменная Человек становится связанной со значением 'оля' (первый аргумент вопроса), затем значение связанной переменной Человек ('оля') согласуется со свободной переменной Кто. Переменная Кто в результате принимает значение 'оля', которое и выводится в качестве результата.

Другой вопрос:

```
?-нравится(Кому, оля). ; Кому нравится Оля?
Кому=оля
```

В данном вопросе сначала свободные переменные Кому и Человек становятся сцепленными, не получая при этом значений. При согласовании вторых аргументов переменная Человек получает значение 'оля' и через сцепление передает это значение переменной Кому. Значение переменной Кому и является результатом согласования.

Что произойдет если задать вопрос:

```
?-нравится(X,Y).
```

По результатам согласования переменные X и Y сцепляются с переменной Человек, но после согласований остаются свободными. Такое согласование с неопределенными значениями приводит в различным реакциям в разных системах Пролога. Обычно выводятся следующее сообщение:

```
X=_, Y=_
```

Переменная величина, обозначаемая одним символом подчеркивания, называется **анонимной** и имеет при согласовании две особенности. Во-первых, анонимная переменная согласуется с любым аргументом, оставаясь при этом свободной. Иначе говоря, две анонимные переменные в предикате считаются различными. Во-вторых, значения анонимных переменных не выводятся в качестве результата вопроса. Рассмотрим пример:

```
родители(мария,николай,анна).
; Мария и Николай являются родителями Анны.
родители(ольга,сергей,дмитрий)
; Ольга - мать Дмитрия, а Сергей - его отец.
?-родители(_,Отец,_).
Отец=николай ;
Отец=сергей ;
НЕТ
```

Данный вопрос можно сформулировать следующим образом: кто является чьим-либо отцом? В вопросе не имеет значения, какие у него дети и кто является матерью. Заметим, что анонимную переменную в данном случае нельзя заменить обычной.

```
?-родители(X,Отец,X).
```

При согласовании первого аргумента переменная X становится связанной со значением 'мария' и поэтому не может согласоваться с третьим аргументом факта из базы, ни одного решения в этом случае получено не будет.

С другой стороны, вопрос ?-родители(X,Отец,Y). является корректным, но приводит к выдаче значений переменных X и Y, что не является необходимым.

```
?-родители(X,Отец,Y).
X=мария,Отец=николай,Y=анна ;
X=ольга,Отец=сергей,Y=дмитрий ;
НЕТ
```

Конъюнкции

Пусть дана следующая база, состоящая из фактов:

```
нравится(коля,лена).
нравится(витя,лена).
нравится(лена,витя).
нравится(лена,лена).
```

Как задать вопрос, определяющий людей, которые взаимно нравятся друг другу? Оказывается, для этого необходимо использовать следующее выражение:

```
?-нравится(X,Y),нравится(Y,X).
X=витя,Y=лена ;
X=лена, Y=витя ;
X=лена, Y=лена ;
НЕТ
```

Запятая, расположенная между предикатами означает операцию *конъюнкции* (логическое И), т.е. в вопросе необходимо согласовать оба целевых утверждения. Рассмотрим принцип такого совместного согласования целей на простом примере.

Дана база:

```
p1(a).  ; 1
p1(b).  ; 2
p2(a).  ; 3
p2(c).  ; 4
```

Зададим к ней вопрос:

```
?-p1(X), p2(X).
```

Процесс согласования выполняется по следующей схеме:

1. Предикат p1(X) согласуется с первым утверждением базы, при этом переменная X связывается со значением a.

2. Значение переменной передается второму предикату, таким образом, целью становится утверждение p2(a).

3. Предикат p2(a) согласуется с 3 утверждением базы и поскольку процесс доказательства завершен, выводится результат X=a.

4. При продолжении процесса доказательства он начинается с согласования предиката p2(a), причем согласование начинается с четвертого утверждения (следующего за согласованным на предыдущем шаге). Больше целевой предикат не может быть согласован и выполняется возвращение обратно к первому предикату.

5. Начинается повторное доказательство первого предиката. При этом переменная X становится свободной, т.е. целевой предикат снова принимает вид p1(X). Сам процесс повторного доказательства начинается со второго утверждения базы (следующего за утверждением для первого предиката, согласованным на предыдущем шаге).

6. Предикат p1(X) согласуется со вторым утверждением, в результате X принимает значение b.

7. Снова начинается доказательство второго утверждения, которое теперь принимает вид p2(b). Доказательство начинается с первого утверждения базы. Поскольку данный целевой предикат не может быть согласован с утверждениями базы, происходит возвращение к первому предикату.

8. Снова переменная X становится свободной, целевой предикат принимает вид: p1(X). Процесс согласования цели начинается с третьего утверждения (следующего за предыдущим согласованным *для этого предиката*). Поскольку предикат не может быть согласован с оставшимися утверждениями базы, процесс доказательства завершается.

Таким образом, при движении вперед процесс доказательства начинается с первого утверждения базы, при этом последующему предикату передаются все связи и сцепления переменных предыдущих предикатов. При возвращении назад (откате цели) продолжается процесс доказательства предыдущего предиката, причем доказательство выполняется, начиная с утверждения, следующего за согласованным для данного предиката на предыдущем шаге. Связи и сцепления старого согласования целевого предиката при этом разрушаются.

Правила

Если факты описывают то, что верно всегда, то правила задают связи, которые выполняются при определенном условии, т.е. правила Пролога соответствуют условиям вида: ЕСЛИ... ТО... .

Рассмотрим пример. Пусть Лене нравятся все люди, которым нравится музыка. Это утверждение можно записать в следующем виде:

```
нравится(лена,Человек):-нравится(Человек,музыка).
```

Знак ':-' означает правило. В некоторых вариантах Пролога, в частности, в Турбо-Прологе возможно использование для обозначения правила слова '*if*' (если):

```
нравится(лена,Человек) if нравится(Человек,музыка).
```

Пусть имеется следующая база:

```
нравится(лена,витя).
нравится(андрей,музыка).
нравится(лена,Человек) :- нравится(Человек,музыка).
```

Зададим вопрос:

```
?-нравится(лена,Кто).
Кто=витя ;
Кто=андрей ;
НЕТ
```

Первый ответ Пролога получен из прямого факта, имеющегося в базе данных, а второй является логическим следствием приведенного правила базы.

При согласовании целевого утверждения с правилом, оно прежде всего должно быть согласовано с левой частью правила. При удачном согласовании правая часть становится целевым утверждением, в которое передаются связи и сцепления переменных по результату согласования левой части. Далее, если удалось доказать правую часть правила, то все установленные связи и сцепления переменных передаются в левую часть и все правило считается доказанным.

В приведенном примере сначала целевое утверждение нравится(лена,Кто) согласуется с правилом из базы, при этом переменные Кто и Человек становятся сцепленными, но не получают значений. После этого целевым утверждением становится: нравится(Человек,музыка). Это утверждение можно доказать, согласовав его со вторым утверждением базы. Переменная Человек принимает значение 'андрей'. Это значение передается в левую часть правила и через сцепление – переменной Кто исходного вопроса. Это значение переменной и выводится в качестве результата.

Поскольку в правой части правил, как и в вопросах, возможны конъюнкции, и предикаты могут согласовываться как с фактами, так и с правилами, процесс доказательства может быть весьма сложным и приводить к неожиданным результатам.

5.2 ПРИМЕРЫ ЗАПИСИ ПРЕДИКАТОВ НА ПРОЛОГЕ

Рассмотрим примеры определения ряда новых предикатов через заданные предикаты. Рассмотрим следующие родственные отношения:

```
отец(X,Y)            ; X является отцом Y
мать(X,Y)            ; X является матерью Y
женщина(X)           ; X является женщиной
```

```
различны(X,Y)          ; X и Y различные.
```

Последний предикат определяется из встроенного предиката, определенного в соответствующем разделе.

Требуется из данных отношений определить новые родственные связи:

```
родители(X,Y,Z )       ; X,Y являются родителями Z,
                       ; причем X-мать, а Y-отец.
сестра(X,Y)            ; X является сестрой Y
дедушка(X,Y)           ; X является дедушкой Y
предок(X,Y)            ; X является предком Y
```

Предикат 'родители' определяется очевидным образом:

```
родители(X,Y,Z) :- мать(X,Z),отец(Y,Z).
```

Таким образом предикат 'родители' согласован, если имеются прямые сведения о родителях в виде факта, или сведения о матери и отце некоторого человека.

Определим теперь предикат 'сестра(X,Y)'. Очевидно, что X должна быть женщиной, иначе X может оказаться братом, и, кроме того, у X и Y должны быть общие родители. Таким образом, получаем:

```
сестра(X,Y) :- женщина(X), родители(M,F,X),родители(M,F,Y).
```

Рассмотрим пример. Пусть имеются следующие утверждения базы:

```
женщина(анна).
женщина(елена).
отец(алексей,елена).
мать(ольга,елена).
родители(алексей,ольга,анна).
родители(X,Y,Z) :- мать(X,Z),отец(Y,Z).
сестра(X,Y) :-женщина(X),родители(M,F,X),родители(M,F,Y).
```

Зададим теперь вопрос: Кто кому является сестрами?

```
?-сестра(Кто,Кому).
Кто=анна, Кому=анна ;
Кто=анна, Кому=елена ;
НЕТ
```

Итак, кроме ожидаемого ответа получился и неожиданный: Анна оказалась сестрой самой себе! Действительно, полученный результат не противоречит введенному определению, т.к. Анна является женщиной и у ней с самой собой общие родители. Обратите также внимание, что поиск родителей Анны выполнен по факту с предикатом 'родители', а для Елены Прологу пришлось доказывать, что Алексей и Ольга являются ее родителями через факты об отцовстве и материнстве. Для исправления предиката необходимо добавить предикат 'различны', что приводит к корректному определению:

```
сестра(X,Y) :-  женщина(X), родители(M,F,X),
  родители(M,F,Y), различны(X,Y).
```

Для определения предиката 'дедушка' используем правило, что дедушка – это тот человек, который является отцом другого человека (а не матерью, иначе получится бабушка), а этот человек в свою очередь является родителем внука дедушки (здесь уже может быть и мать).

```
дедушка(X,Y) :- отец(X,Z),отец(Z,Y).
дедушка(X,Y) :- отец(X,Z),мать(Z,Y).
```

Таким образом, определение предиката 'дедушка' состоит из двух правил. Если определить предикат родитель(X,Y), означающий, что X является родителем Y, то возможно определение через одно правило.

```
дедушка(X,Y) :- отец(X,Z),родитель(Z,Y).
```

Наиболее трудным для понимания является предикат 'предок'. Он определяется через предикат родитель(X,Y), но необходимо при этом получать в качестве решения даже самых отдаленных предков.

Например, пусть дана база:

```
родитель(алексей,иван).
родитель(иван,николай).
родитель(николай,александр).
```

При задании вопроса, кто кому является предком, необходимо получить все варианты ответов, в частности, что Алексей – предок Ивана, Николая и Александра, Иван – предок Николая и Александра, а Николай является предком только Александра.

Предикат 'предок' можно легко определить, если использовать рекурсию. Тогда определение предка становится не сложнее определения дедушки: X является предком Y, если X является родителем некоторого Z, который является предком Y. При таком определении 'отдаленность' предков на каждом шаге уменьшается на единицу, и становится возможным перебрать всех предков данного человека. Единственное отличие данного предиката от предиката 'дедушка' состоит в том, что для рекурсивного определения требуется указание предельного случая, когда уменьшать отдаленность уже не нужно. Таким предельным случаем является вариант, когда X является просто родителем Y. Предельный случай рекурсии часто добавляется в базу данных в виде факта. Итак окончательно получаем определение предиката 'предок'.

```
предок(X,Y) :- родитель(X,Y).
предок(X,Y) :- родитель(X,Z), предок(Z,Y).
```

Характер доказательства рекурсивных предикатов часто бывает весьма сложным, попробуйте, например, определить, в каком порядке будут получены все ответы для приведенной выше базы данных.

5.3 ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ ТУРБО-ПРОЛОГА.

Турбо-Пролог имеет ряд специфических черт, отличающих его от стандартного Пролога, причем эти его особенности имеют как достоинства, так и недостатки. Прежде всего, Турбо-Пролог является компилятором, поэтому процесс решения состоит во вводе базы и запуске программы на выполнение после компиляции. При запуске программы на экране появляется

подсказка-приглашение к вопросу: *GOAL*: (ЦЕЛЬ) для ввода вопроса к базе. Сам вопрос вводится без символов '?-'. Все полученные решения выводятся сразу без дополнительного ввода точки с запятой.

Интегрируемая среда системы Турбо-Пролог совпадает с аналогичной средой Турбо-Паскаля версии 5.5 и позволяет редактировать, загружать, сохранять базы на диск, отмечать в тексте ошибки компиляции и выполнять другие аналогичные действия, выбираемые из меню.

Компилятор Турбо-Пролога является одним из наиболее быстрых, но он при этом накладывает ряд ограничений на возможные типы предикатов. Основное его отличие от стандартного Пролога состоит в том, что Турбо-Пролог является *типизированным* вариантом Пролога, т.е. требует предварительного описания предикатов. Это и позволило создать эффективный процесс поиска решений.

Основными разделами создаваемой базы данных Турбо-Пролога являются *DOMAINS, PREDICATES, CLAUSES* и *GOAL*. Различия между заглавными и малыми буквами отсутствует, как в языке Паскаль.

Раздел DOMAINS (области данных) служит для определения новых типов аргументов. Аналогом данного раздела является раздел TYPE Турбо-Паскаля. Встроенными типами являются: *symbol, integer, real, string, char, file.*

symbol – символьный атом, т.е. идентификатор, задаваемый последовательностью латинских букв, цифр и символов подчеркивания, начинающаяся с малой латинской буквы,

integer – целые числа от -32768 до 32767.

real – вещественные числа от -1E-307 до 1E308.

string – строки, т.е. последовательность любых символов, заключенных в двойные кавычки, длиной не более 250 знаков. Пример: "Today", "yes".

char – любые одиночные символы, заключенные в одиночные апострофы. Пример: 'a', '\13', 'я'.

file – допустимое в ДОС имя файла. Пример: data.txt.

Используя встроенные типы данных, можно определить типы данных, отвечающие характеру задачи. Пример:

```
DOMAINS
 man, woman = symbol
 number = integer
```

Раздел PRIDICATES служит для описания предикатов. В этом разделе должны быть перечислены все используемые в программе предикаты с указанием типов их аргументов. Пример:

```
DOMAINS
 man, woman = symbol
 number = integer
```

```
PREDICATES
 husband(man)
 wife(woman)
 child(symbol)
 student(symbol, number)
```

Обратите внимание, что в определении типов аргументов предикатов могут использоваться как встроенные, так и определенные типы данных.

Раздел GOAL определяет целевое утверждение и является обязательным, если выполняется компиляция с созданием исполняемого (EXE) файла. Если выполнение происходит из интегрированной оболочки среды Турбо-Пролог, то данный раздел можно опустить, т.к. запрос целевого утверждения появится в специальном окне.

Наконец, раздел CLAUSES является ключевым, в нем задаются все утверждения базы (факты и правила). Ниже приведен пример базы и вопроса к ней для предиката like (нравится).

```
DOMAINS
 person=symbol
PREDICATES
 like(person,person)
CLAUSES
 like(alex,olga).
 like(olga,alex).
 like(victor,music).
 like(olga,X) :- like(X,music).
GOAL
 like(olga,Who).
```

Выдача ответов и принцип согласования цели полностью соответствует стандартному Прологу.

5.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое предикат?
2. Из чего состоит база данных Пролога?
3. Чем различаются факты и правила?
4. Какие типы данных в Пологе вы знаете?
5. Для чего используются вопросы?
6. Как отличить переменные величины от констант при записи программы на Прологе?
7. Что называется доказательством цели?
8. Какие переменные называются свободными? Сцепленными? Связанными?
9. Как происходит согласование цели с базой?
10. Что такое анонимная переменная? Зачем она нужна?
11. Как работает предикат конъюнкции?
12. Какие разделы имеет программа на Турбо-Прологе? Каково их назначение?

5.5 УПРАЖНЕНИЯ

1 Пусть в базе имеется факт

p(a(V,Z),Z,Z,V).

и задан вопрос

?- p(X,Y,c,b(x)).

Что получится в результате согласования?

Рассмотрим последовательность согласования всех 4 аргументов.

1. Первый аргумент вопроса X согласуется с аргументом a(V,Z). Переменные V и Z остаются свободными.
2. Переменная Y сцепляется с переменной Z.
3. Переменная Z из факта связывается и получает значение c, далее через сцепление переменных Y и Z Y также получает значение c. Кроме этого, X теперь равно a(V,c).
4. Переменная V связывается со значением b(X). В результате X равен a(b(x),c). Итак, ответ системы:

```
X=a(b(x),c), Y=c
```

2 В базе находятся следующие факты

p(a,a).

p(b,c).

и задан вопрос

?-p(X,Y),p(X,Z).

Найти, какие будут получены решения.

Рассмотрим шаги согласования.

1. Предикат p(X,Y) согласуется с фактом p(a,a). В результате переменная X принимает значение a и Y также принимает значение a.
2. Целевым предикатом теперь становится второй операнд конъюнкции p(a,Z), при этом X заменилось на свое значение Z.
3. Целевой предикат p(a,Z) начинает согласовываться с началом базы и согласуется с первым предикатом. В результате получается первое решение

   ```
   X=a, Y=a, Z=a
   ```
4. Предикат p(a,Z) начинает согласовываться со следующим предикатом базы p(b,c). Поскольку согласование невозможно, выполняется откат назад.
5. Предикат p(X,Y) с вновь свободными переменными начинает согласовываться с предикатом p(b,c). Согласование успешно, X=b, Y=c.

6. Целевым предикатом вновь становится второй операнд конъюнкции в виде p(b,Z). Предикат начинает согласовываться с начала базы, с первым предикатом не согласуется, но согласуется со вторым. В результате Z=c, и получается второе решение `X=b, Y=c, Z=c`
7. Предикат p(b,Z) не может быть согласован с другими фактами базы, и выполняется откат назад. Аналогично предикат p(X,Y) также не может быть согласован с фактами базы, поскольку они закончились. На этом процесс решения завершен.

Итак, Пролог система выведет ровно два решения:

```
X=a, Y=a, Z=a
X=b, Y=c, Z=c
```

5.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. В базе имеется 2 приведенных ниже факта. К базе задается вопрос. Что выведет Пролог-система?

```
p(X,a).
p(Z,b).
?- p(X,Y),p(X,X)
```

   a) `X=a, Y=a`
      `X=b, Y=a`
      `X=a, Y=b`
      `X=b, Y=b`
   b) `X=a, Y=a`
      `X=b, Y=b`
   c) `X=a, Y=a`
      `X=b, Y=b`
      `X=a, Y=b`
      `X=b, Y=a`
   d) `X=a,b, Y=a,b`
   e) `X=a, Y=b`
      `X=b, Y=a`

2. В базе имеется один факт. К ней задается вопрос. Что получится в результате согласования?

```
p(a(V,Z),Z,a(b,Q),V,Q).
?-p(X,Z,X,Y,c).
```

   a) `X=a(V,Z), Y=V, Z=c`
   b) `X=a(b,c), Y=b, Z=_`
   c) `X=a(b,c), Y=b, Z=c`
   d) `X=a(b,c), Y=V, Z=Z`
   e) `X=a(b,c), Y=c, Z=b`

3. Каков результат согласования приведенной ниже цели для данной базы:

```
p(a,a).
p(a,b).
p(X,a) :- p(X,b).
?-p(X,Y)
```

a) X=a, Y=a
b) X=a, Y=a
   X=a, Y=b
c) X=a, Y=a
   X=a, Y=b
   X=a, Y=a
d) X=a, Y=a
   X=a, Y=a
   X=a, Y=b
e) X=a, Y=b
   X=a, Y=a
   X=a, Y=a

4. Каков результат согласования для приведенной ниже базы и вопроса?

```
p(X,X,c,d).
p(_,_,a,b).
?-p(Z,V,Z,V).
```

a) Z=c, V=d
b) Z=a, V=b
c) Z=_, V=_
d) Z=a, V=b
   Z=c, V=d
e) Z=c, V=d
   Z=a, V=b
   Z=_, V=_

Ответ: 1.a; 2.e; 3.c, 4.b.

5.7 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ №7

ПОСТРОЕНИЕ БАЗЫ ДАННЫХ В СРЕДЕ TURBO-PROLOG

Задание: Написать базу, которая будет выдавать все возможные подстановки: то есть если цель содержит только переменные, то выдать все сочетания, которые дадут истину (в ограниченном диапазоне). Если же, заданы только значения, то болжен быть ответ Yes или No. Также все промежуточные варианты - часть переменные, часть значения.

Код программ и тестовые примеры привести в отчете. Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

1 N*a=K*c

2 x^n=z

3 k! + n! =a

4 Числа Армстронга (153 = 1^3 + 5^3 + 3^3)

5 Сумма 2-х чисел, которые делятся на 3 или 7 или на оба сразу.

6 N=x^2+v^2+z^2

7 Проверить входит ли в число A цифра P

8 Является ли число P делителем числа A

9 Два числа - один делитель

10 Число, которое делится на 3, но не делится на 6.

11 Число сочетаний из N по K

12 Геометрическая прогрессия A n = K* A n-1

13 1^n+2^(n-1)+3^(n-2)+..+n^1=K

14 1+2^2+3^3+..+n^n=K

15 1+2!+3!+...+n!=a

16 3-х значное число, у которого сумма 2-х цифр равна третьей

17 Сумма цифр 3-значного числа равна их произведению

18 Числа Фибоначи

19 Простое - непростое число.

20 Совершенное - несовершенное

6 РЕКУРСИВНОЕ ПРОГРАММИРОВАНИЕ НА ПРОЛОГЕ

6.1 ВСТРОЕННЫЕ ПРЕДИКАТЫ ПРОЛОГА

Встроенные предикаты используются для выполнения арифметических и логических действий, сравнения чисел, управления процессом вывода, доступом к элементам базы данных и ряда других действий. Некоторые предикаты нарушают при этом стандартную форму записи, при которой сначала находится имя предиката, а затем в скобках перечисляются аргументы. Такие предикаты в стандартном Прологе называются операторами и рассматриваются в разделе 8.2. В данном разделе приведены основные встроенные предикаты стандартного Пролога и в конце каждого раздела приводятся особенности встроенных предикатов Турбо-Пролога.

Предикаты сравнения на равенство

Предикатами сравнения являются предикат равенства X=Y, предикат неравенства X\=Y и предикат строгого равенства X==Y. Все предикаты сравнения располагаются между своими аргументами.

Предикат равенства согласует аргументы X и Y по правилам согласования целевого утверждения Пролога: если их значения являются предикатами, то должны совпадать имена этих предикатов и согласовываться их аргументы; если одна переменная свободна, то она связывается со значением другой переменной; если обе переменные свободны, то они становятся сцепленными; и, наконец, константы и значения связанных переменных сравниваются обычным образом.

Заметим одну особенность встроенных предикатов. Пусть имеется целевое утверждение:

```
?- p1, X=Y, p2.
```

Тогда, при удачном согласовании p1 и X=Y, происходит переход к предикату p2. Когда процесс доказательства p2 завершается, выполняется обратное движение к предикату равенства, *но предикат X=Y повторно не согласуется с базой!* В противном случае было бы невозможно выйти из бесконечного удачного согласования. Поэтому происходит дальнейший откат к цели p1 и лишь повторное ее успешное согласование позволяет снова проверять равенство X=Y.

Предикат X\=Y является противоположным по смыслу предикату равенства. Он согласуется с базой, если не согласуются его аргументы, при этом между аргументами не устанавливается никаких связей.

Предикат строгого равенства X==Y действует аналогично обычному предикату равенства за исключением случая, когда аргументами являются свободные переменные. В этом случае предикат согласуется с базой только тогда, когда переменные уже были сцеплены ранее, т.е. предикат строгого равенства никогда не производит сцепления.

Используя предикат неравенства, можно определить предикат 'сестра' из раздела 6.2 следующим образом:

```
сестра(X,Y) :- женщина(X), родители(M,F,X),
  родители(M,F,Y), X\=Y.
```

В Турбо-Прологе используются предикат равенства '=', предикат неравенства имеет вид '<>', а предикат '==' отсутствует.

Предикаты сравнения чисел

Предикатами сравнения чисел являются X=Y, X\=Y, X<Y, X>Y, X<=Y, X>=Y, выполняющие соответственно операции 'равно', 'не равно', 'меньше', 'больше', 'меньше или равно', 'больше или равно'. Аргументами этих предикатов, за исключением равенства и неравенства, должны быть числа, и предикат, находящийся в целевом утверждении, согласуется с базой, если выполняется соответствующее условие. Заметим, что предикаты сравнения располагаются между своими аргументами.

Рассмотрим пример определения предиката с использованием предикатов сравнения. Пусть имеется предикат правил_страной(Человек, Начало, Конец), определяющий, что данный Человек управлял государством от года Начало до года Конец. Определим предикат правитель(Человек, Год), определяющий, что Человек правил в заданном Году. Для этого необходимо, чтобы выбранный Год находился в интервале [Начало, Конец]. Определение предиката 'правитель' имеет вид:

```
правитель(Человек, Год) :-
    правил_страной(Человек, Начало, Конец),
    Год>=Начало, Год<=Конец.
```

Теперь рассмотрим базу:

```
правил_страной(ленин, 1917, 1924).
правил_страной(сталин, 1924, 1953).
правил_страной(хрущев, 1953, 1964).
правил_страной(брежнев, 1964, 1982).
правитель(Человек, Год) :-
    правил_страной(Человек, Начало, Конец),
    Год>=Начало, Год<=Конец.
```

Для данной базы можно определить, кто был правителем в заданном году.

```
?-правитель(Кто,1923).
Кто=ленин
?-правитель(Кто,1964).
Кто=хрущев ;
```

```
Кто=брежнев ;
НЕТ
```

В Турбо-Прологе используются предикаты сравнения '>', '>=', '<', '<=', смысл которых аналогичен стандартному Прологу.

Арифметические предикаты

В стандартном Прологе используются следующие арифметические предикаты: X+Y, X-Y, X*Y, X/Y. Кроме этого, имеется предикат, выполняющий отрицание: -X. В целевых утверждениях возможно использование произвольного сочетания предикатов с обычными приоритетами. Важной особенностью арифметических предикатов стандартного Пролога состоит в том, что сами они не вычисляют значений числовых выражений. Например, возможно утверждение базы:

```
2+3*4.
```

и вопрос к ней:

```
?-X+Y*Z.
X=2, Y=3, Z=4
```

Для вычисления числовых значений арифметических выражений в стандартном Прологе используется встроенный предикат is: X is Y. Аргумент X должен быть свободной переменной, связанной переменной с числовым значением или числом, а аргумент Y представляет собой арифметическое выражение. Предикат is согласуется с базой, если согласуются аргументы X и Y. Пример:

```
p(2,3,4).
?-p(X,Y,Z), T is X+Y*Z.
X=2,Y=3,Z=4,T=14
```

Рассмотрим пример определения предиката с использованием арифметики. Имеются предикаты население(Страна,Значение) и предикат площадь(Страна,Значение). Определим новый предикат плотность_населения(Страна,Значение).

```
плотность_населения(Страна,X) :-
  население(Страна,Y), площадь(Страна, Z),
  X is Y/Z.
```

Пример базы:

```
население(сша, 203).
население(индия, 548).
население(бразилия, 108).
площадь(сша,8).
площадь(индия,3).
площадь(бразилия,8).
```

```
плотность_населения(Страна,X) :-
  население(Страна,Y), площадь(Страна, Z),
  X is Y/Z.
```

Теперь можно задать различные вопросы для построенного предиката.

```
?-плотность(сша,X).
; Какова плотность населения США?
X=25
?- плотность(X,186).
; У какой страны плотность равна 186?
X=индия
?-плотность(X,Y).     ; Какая плотность
; у имеющихся стран в базе данных?
X=сша, Y=25 ;
X=индия, Y=186 ;
X=бразилия, Y=18 ;
НЕТ
```

Данный пример показывает, что часто построенные предикаты обладают свойством, позволяющим вместо произвольного аргумента использовать как константу, так и переменную, что позволяет задавать всевозможные варианты вопросов. Заметим, что при изменении "направления вопроса" не нужно изменять ни базу данных, ни тем более интерпретатор Пролога. Это одно из преимуществ декларативного подхода.

В Турбо-Прологе предикат is отсутствует и арифметические предикаты '+', '-', '*', '/' вычисляются сразу, поэтому их аргументами должны быть числа или переменные с числовыми значениями.

Предикат отрицания NOT

Предикат not(X), согласуется с базой данных в том и только в том случае, когда X не согласуется с базой.

Использование данного предиката имеет особенность, состоящую в том, что двойное отрицание утверждения X не эквивалентно самому X.

Например, пусть целевой предикат имеет вид p(Z),p1(Z), и первый предикат согласуется с утверждением p(a). Тогда переменная Z становится связанной и второй целевой предикат принимает вид p1(a). С другой стороны, если использовать целевое утверждение not(not(p(Z)),p1(Z), то сначала p(Z) согласуется с утверждением p(a), переменная Z становится связанной, затем предикат not приводит к несогласованию с утверждением, при этом Z становится свободной, затем повторный not снова приводит к согласованию с утверждением, но при этом переменная Z уже не связывается, и второй целевой предикат принимает вид: p1(Z) со свободной переменной.

Таким образом, двойной not согласуется в том случае, если выражение p(Z) может быть *в принципе* согласовано с базой, но конкретный вариант согласования не учитывается.

Дизъюнкция

Дизъюнкция целевых утверждений обозначается ';' (точка с запятой) и представляет логическую связку ИЛИ. При согласовании целевого утверждения X;Y сначала делается попытка согласовать предикат X, при удачном согласовании все целевое утверждение X;Y считается согласованным. Если предикат X доказать не удается, то начинает согласовываться предикат Y. Решения, найденные для предиката Y также считаются решениями всего утверждения X;Y.

Например, определим предикат 'человек', при этом будем считать X человеком в том случае, если в базе имеются сведения, что у X имеется мать. Исключение сделаем для Адама и Евы. Итак, получаем правило определения человека:

```
человек(X) :- X=адам; X=ева; мать(_,X).
```

Однако легко заметить, что это определение легко преобразовать в три независимые утверждения:

```
человек(адам).
человек(ева).
человек(X):- мать(_,X).
```

Если предикаты не содержат **отсечений**, которые рассматриваются в следующем разделе, такое преобразование можно выполнить всегда, и при этом программа принимает значительно более читаемый вид.

Предикат отсечения

Предикат отсечения обозначается символом '!' (восклицательный знак) и меняет ход согласования целевых утверждений. Косвенным аналогом данного предиката является оператор *goto* таких языков программирования, как Паскаль или Си. Оператор goto значительно затрудняет понимание программы и часто приводит к трудно исправимым ошибкам, поэтому в течение длительного времени с ним велась активная борьба в книгах и журналах по программированию. Вместе с тем, при аккуратном использовании оператор goto иногда позволяет программисту более просто решить задачу. Аналогично обстоят дела с оператором отсечения в Прологе. Несмотря на то, что его применение приводит часто к трудно предсказуемым действиям, он позволяет резко сократить перебор вариантов и упростить программу.

Рассмотрим принцип работы предиката '!' на следующем примере. Даны утверждения базы:

```
p(Z) :- p1(Z),p2(Z),!,p3(Z).
p(Z) :- p4(Z).
```

К данной базе задан вопрос:

```
?-p(X).
```

Сначала p(X) согласуется с первым утверждением базы, и правая часть p1(Z),p2(Z),!,p3(Z) становится целевым утверждением. Теперь начинают согласовываться предикаты p1(Z) и p2(Z) по обычным правилам, и если согласование хотя бы одного из них прошло неудачно, то происходит возвращение к исходному предикату p(Z), который согласуется с новым утверждением p(Z):- p4(Z), и целевым предикатом становится p4(Z).

Предикат отсечения начинает работать при удачном согласовании p1(Z) и p2(Z). В этот момент происходит *переход через отсечение слева направо*, и полученное решение для p1(Z) и p2(Z) *замораживается.* Предикат p3(Z) согласуется по обычным правилам, и все полученные решения, которых может не быть, выводятся.

После окончания согласования p3(Z) выполняется откат к предыдущей цели, т.е. *происходит переход через отсечение справа налево.* При этом прекращается поиск новых решений для p1(Z) и p2(Z) и кроме этого, *прекращается также согласование и исходного утверждения p(Z)*, поэтому в данном случае p4(Z) не становится целевым утверждением. Согласование p(Z) по второму правилу произойдет только в том случае, если не будет перехода через отсечение слава направо, т.е. если цель p1(Z),p2(Z) не удастся согласовать ни разу.

Другими словами, без использования отсечения база выглядела бы следующим образом:

```
p(Z) :- p1(Z),p2(Z),p3(Z).
p(Z) :- not(p1(Z),p2(Z)),p4(Z).
```

Последний вариант обладает одним неоспоримым преимуществом перед вариантом, использующим отсечение: в нем можно изменять порядок правил в базе без потери смысла. Если поменять порядок строк в исходной базе, то предикат p4(Z) будет согласовываться всегда, независимо от результата согласования p1(Z),p2(Z):

```
p(Z) :- p4(Z).
p(Z) :- p1(Z),p2(Z),!,p3(Z).
```

Рассмотрим примеры использования предиката отсечения.

Пусть имеется база данных читателей библиотеки с информацией о несданных книгах, описываемой предикатом книга_не_сдана(Читатель, Книга). Читателям библиотеки доступны основные услуги, и тем читателям, у которых нет несданных книг, доступны и дополнительные услуги. Итак, имеем факты:

```
читатель(иванов).
читатель(петров).
книга_не_сдана(петров, математический_анализ).
книга_не_сдана(петров, линейная_алгебра).
основные_услуги(использование_каталога).
основные_услуги(получение_консультации).
```

```
дополнительные_услуги(взять_книги).
дополнительные_услуги(заказать_книгу_мба).
```

Добавим к этим фактам правила для предиката виды_услуг(Читатель,Услуга), определяющие, какие услуги доступны для читателей.

```
виды_услуг(Читатель,Услуга) :-
  читатель(Читатель),
  книга_не_сдана (Читатель, Книга), ! ,
  основные_услуги(Услуга).
виды_услуг(Читатель,Услуга) :-
  читатель(Читатель),
  основные_услуги(Услуга).
виды_услуг(Читатель,Услуга) :-
  читатель(Читатель),
  дополнительные_услуги (Услуга).
```

Определим теперь, какие услуги доступны для Иванова и Петрова.

```
?-виды_услуг(иванов, Услуга).
```

Целевой предикат не согласуется с первым правилом, т.к. в базе нет информации, что у Иванова есть несданные книги, поэтому он последовательно согласуется со вторым и третьим предикатом, в результате чего выводятся решения:

```
Услуга= использование_каталога ;
Услуга= получение_консультации ;
Услуга= взять_книги ;
Услуга= заказать_книгу_мба ;
НЕТ
```

Для Петрова согласование целевого утверждения ?-виды_услуг(петров, Услуга) выполняется иначе. Предикат согласуется с первым утверждением, т.к. у Петрова есть несданная книга. Далее целевым предикатом становится основные_услуги (Услуга) после перехода через отсечение слева направо, в результате чего получаются следующие решения:

```
Услуга= использование_каталога ;
Услуга= получение_консультации ;
НЕТ
```

Теперь выполняется переход справа налево через отсечение, и, поскольку согласования были заморожены, поиск новых решений прекращается. Исходный предикат ?-виды_услуг(петров, Услуга) не согласуется ни со вторым, ни с третьим правилом базы. Заметим, что при отсутствии отсечения процесс согласования выполняется для всех трех утверждений и, кроме этого, предикат основные_услуги(Услуга) согласуется дважды, т.к. у Петрова две несданные книги. Таким образом, убрав отсечение, получим следующий набор решений:

```
Услуга= использование_каталога ;
Услуга= получение_консультации ;
Услуга= использование_каталога ;
Услуга= получение_консультации ;
Услуга= использование_каталога ;
Услуга= получение_консультации ;
Услуга= взять_книги ;
```

```
Услуга= заказать_книгу_мба ;
НЕТ
```

Предикат отсечения иногда используется вместе с предикатом fail, который никогда не согласуется с базой данных. Сам по себе предикат fail не имеет смысла, но вместе с отсечением он исключает тупиковые ветви, сокращая перебор вариантов поиска решений.

Рассмотрим следующий пример. Требуется определить предикат, вычисляющий величину налога в зависимости от дохода. Прежде чем определять величину налога для физического лица необходимо убедиться, что это лицо не является иностранцем, т.к. с иностранцев налог не берется. Поэтому начальные правила определения предиката величина_налога(Человек, Доход, Налог) имеют вид:

```
величина_налога(Человек, Налог) :-
  иностранец(Человек), !, fail.
величина_налога(Человек, Налог) :-
  доход(Человек, Доход), Доход<10000,
  Налог is Доход*0.12.
```

Если теперь имеются факты:

```
доход(петров,5000).
доход(иванов,8000).
иностранец(иванов).
```

то для Петрова получается решение:

```
?-величина_налога(петров, Налог).
Налог=600 ;
НЕТ
```

Если задать такой же вопрос для Иванова, его то при согласовании выполняется предикат иностранец(иванов), и при переходе через отсечение справа налево для предиката величина_налога(петров, Налог) не происходит поиска новых решений. Если убрать отсечение, то после предиката fail, происходит возвращение к предикату величина_налога(петров, Налог), который согласуется со вторым утверждением базы, в результате чего появляется решение:

```
Налог=960
```

Предикаты определения типов

В Прологе имеется ряд предикатов, которые определяют, какой тип имеет данный аргумент в момент его проверки.

Предикат **var**(X) согласуется с базой данных, если X является несвязанной переменной. Если переменной присвоена некоторая структура, у которой не определены некоторые из переменных, то var(X) также согласуется с базой.

Предикат **atom**(X) согласуется с базой данных, если его аргумент является символьным атомом.

Предикат **integer**(X) согласуется с базой, когда аргумент X является целым числом.

Предикат **atomic**(X) согласуется с базой, когда аргумент является символьным атомом или числом.

В Турбо-Прологе для определения типа аргумента используются предикаты **free**(X) и **bound**(X). Предикат free(X) согласуется с базой, когда X – несвязанная переменная. Предикат bound(X) согласуется, когда X – связанная переменная.

Примеры предикатов определения типов приведены в последующих разделах, пока рассмотрим простейший пример. Определим предикат plus1(Number1, Number2), который увеличивает первое число на 1. Смысл постановки задачи состоит в том, что определяемый предикат должен выполнять действие в обе стороны:

```
?-plus1(5,X).
X=6
?-plus1(X,9).
X=8
?-plus1(3,4)
YES
```

Такие универсальные предикаты всегда предпочтительнее предикатов, выполняющие действия в одном направлении. Ниже приведена программа на Турбо-Прологе, определяющая предикат plus1. В ее определении используется предикат bound.

```
PREDICATES
plus1(integer,integer).
CLAUSES
plus1(Number1,Number2) :- bound(Number1),
  Number2 =Number1+1.
plus1(Number1, Number2) :- bound(Number2),
  Number1 =Number2-1.
```

Предикаты ввода-вывода и работы с базой данных

Поскольку Пролог является диалоговой системой, в большинстве программ нет необходимости использовать дополнительные предикаты ввода-вывода. Кроме того, эти предикаты сильно зависят от конкретной реализации Пролога. В данном разделе приведем лишь некоторые предикаты для справки. Рассмотренные здесь предикаты не обязательно использовать при выполнении заданий.

Предикат **get**(X) позволяет осуществлять ввод по одному символу, а **put**(X) – выводить символ. Ввод и вывод одного утверждения Пролога выполняется соответственно предикатами read(X) и write(X), X – должен быть предикатом Пролога.

Предикат consult(X) позволяет добавить к базе данных утверждения Пролога из файла. Аргумент X является символьной строкой с именем этого файла.

Наконец, предикаты **asserta**(X) и **assertz**(X) добавляют предикат X Пролога к текущей базе данных. Первый предикат добавляет утверждение в начало базы, а второй предикат – в конец базы. Предикат X может быть прочитан из файла или построен средствами языка.

6.2 ПРИМЕРЫ ЗАПИСИ ПРОГРАММ НА ПРОЛОГЕ

В данном разделе приведен пример программы вычисления факториала. На Паскале или Си вычисление факториала можно выполнять циклически (итерационно) или рекурсивно. Большинство предикатов Пролога по своему смыслу рекурсивны, но возможно также ими имитировать итерационный процесс. В заключение приведен предикат, вычисляющий факториал "в обе стороны".

Рекурсивно факториал от натурального числа n можно определить следующим образом: 0!=1, n!=n*(n-1)!, при n>0. Не составляет труда записать это определение на Си:

```
int f (int n)
{
 if (n==0)  return 1;
 else   return (n*f(n-1));
}
```

Так же легко определить факториал соответствующим предикатом Пролога. На Турбо-Прологе соответствующее определение имеет вид:

```
PREDICATES
 f(integer,integer).
CLAUSES
 f(0,1).
 f(N,P) :- N1=N-1, f(N1,P1), P=N*P1.
```

Обратите внимание, что при записи программы на Си результат возвращался как значение вычисленной функции. Предикат Пролога не возвращает никакого значения, поэтому значение возвращается через его аргументы. Таким образом, предикаты Пролога действуют как процедуры, получая входные значения через одни аргументы и возвращая результат через другие.

На Си можно легко записать итерационный алгоритм вычисления факториала, для этого нужно ввести переменную, выполняющую роль счетчика сомножителей:

```
int f (int n)
{
 int k,p;
 for(k=0, p=1; k<=n; k++)
  p=p*k;
 return p;
}
```

На Прологе данный итерационный процесс моделируется введением предиката f1 с дополнительными аргументами, имеющими тот же смысл, что и переменные в приведенной выше функции на Си.

```
PREDICATES
 f(integer,integer).
 f1(integer,integer,integer,integer).
CLAUSES
 f(N,P) :- f1(N,0,1,P).
 f1(N,N,P,P) :-!.
 f1(N,K,P,P2):- K1=K+1, P1=P*K1, f1(N,K1,P1,P2).
```

Обратите внимание на передачу результата и проверку условия предиката f1(N,N,P,P), а также на необходимость использования отсечения, без которого процесс согласования продолжился бы после найденного решения, что привело бы к зацикливанию программы.

Пусть имеется определенный предикат f(N,P), вычисляющий P=N!. Определим теперь предикат ff(N,P), действующий в обе стороны.

```
?-ff(4,X).
X=24
?- ff(X,720).
X=6
?-ff(X,25).
NO
?-ff(3,6).
YES
?-ff(4,23).
NO
```

```
PREDICATES
 f(integer,integer).
 ff(integer,integer).
 ff1(integer,integer,integer,integer).
CLAUSES
f(0,1).
f(N,P):-N1=N-1, f(N1,P1), P=N*P1.
/* прямое вычисление */
ff(X,Y) :- bound(X),f(X,Y).
/* обратное вычисление итерационная процедура */
ff(X,Y) :- bound(Y),ff1(X,1,1,Y).
/* удачный поиск, текущее значение факториала (3-й аргумент)
совпадает с требуемым (4-й аргумент), передача результата X и
прекращение дальнейшей работы */
ff1(X,X,Y,Y) :- !.
/* неудачный поиск, прекращение дальнейшей работы */
ff1(X,X1,Y1,Y) :- Y1>Y,!,fail.
/* продолжение итерационного процесса вычисления очередного
значения */
ff1(X,K,P,Y) :- K1=K+1, P1=P*K1, ff1(X,K1,P1,Y).
```

6.3 СПИСКИ

Определение и свойства списков

Списки являются основной структурой данных на Прологе для моделирования последовательности элементов, которые в языках программирования Паскаль или Си обычно организованы в виде массивов. Синтаксис списков на Прологе отличается от синтаксиса списков в Лиспе. Ниже приведены примеры записи списков на Прологе.

Пролог	**Лисп**
[1,2,3]	(1 2 3)
[[a]]	((a))
[[a],b,[c,d]]	((a) b (c d))
[]	() или nil.

Списки могут выступать в качестве аргументов предикатов и согласуются друг с другом при совпадении элементов.

Пример:

```
p([[1,2],3,4]).
?-p(X,3,Y).
X=[1,2], Y=4;
НЕТ
```

Для решения серьезных задач необходимо иметь мощный инструмент для построения списков и их обработки. Таким инструментом являются *частная и общая рекурсия,* а список перед обработкой развивается на *голову* и *хвост*. Напомним, что головой списка является первый элемент списка, а хвостом – список без первого элемента. Разбиение списка на голову и хвост выполняется в Прологе с помощью конструкции [X|Y], где X – голова списка, Y – хвост списка.

Пример:

```
p([[1,2],3,[4]]).
?-p([X|Y]).
X=[1,2], Y=[3,[4]];
НЕТ
```

Предиката, аналогичного функции *cons* языка Лисп, в Прологе нет, но его легко определить с помощью одного факта:

```
cons(X, Y, [X|Y]).
```

Данный предикат позволяет как собирать список из головы и хвоста, так и разбивать список на голову и хвост.

```
cons(X, Y, [X|Y]).
```

```
?- cons([1], [2,3], Z).
Z=[[1],[2,3]];
НЕТ
?-cons(X,Y,[1,2,3]).
X=1, Y=[2,3];
НЕТ
```

В Турбо-Прологе имеется ряд ограничений возможного вида списков. Одноуровневый список из символьных атомов можно определить следующим образом:

```
DOMAINS
listatom=symbol*
PREDICATES
p(listatom).
CLAUSES
p([a,b,c]).
```

В Domains определяется новый тип данных – список из символьных атомов (признаком списка является символ *), а в Predicates – предикат с аргументом из такого списка. Аналогично можно определить одноуровневый список из целых чисел:

```
DOMAINS
listinteger=integer*
```

Однако списки с произвольным уровнем вложенности в Турбо-Прологе недопустимы. Вместе с тем можно определить одноуровневый список из элементов любого типа, например, список из одноуровневых списков:

```
DOMAINS
list=integer*
list1=list*
PREDICATES
p(list1).
```

Теперь можно определить следующее утверждение для определенного выше предиката p: p([[1,2], [3], [4,5]]). Но утверждение p([[1,2], 3, [4]]) уже некорректно, т.к. в нем имеется элемент, не являющийся списком. Аналогично, некорректным является факт p([[1,2],[[3]],[4]]) из-за второго элемента, который представляет собой список вложенности два.

Работа со списками произвольной структуры

Хотя в Турбо-Прологе отсутствует возможность введения списков произвольной структуры, в комплект поставки Турбо-Пролога входит интерпретатор Pie-Prolog, в достаточной степени приближенный к стандарту Пролога. В частности, в нем нет никаких ограничений на списки. Интерпретатор Pie-Prolog написан на Турбо-Прологе и запускается выполнением файла pie.pro. Интерпретатор имеет встроенный редактор, вызываемый командой edit, который в свою очередь позволяет загружать в него внешние файлы (F3) и сохранять их (F2).

6.4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие операторы сравнения на равенство существуют в стандартном Прологе?
2. Какие другие операторы сравнения предусмотрены в языке Пролог?
3. Каковы особенности арифметических операторов в стандартном Прологе?
4. Как работает предикат NOT?
5. Что такое дизъюнкция? Как записывается оператор дизъюнкции на Прологе?
6. Как работает предикат отсечения?
7. Почему не рекомендуется частое использование предиката отсечения?
8. Какие предикаты определения типов вы знаете?
9. Как записываются списки на Прологе?
10. Какова особенность записи списков на Турбо-Прологе по сравнению со стандартным Прологом?

6.5 УПРАЖНЕНИЯ

1 Определить на стандартном Прологе предикат, вычисляющий сумму элементов одноуровневого числового списка

Пусть предикат, который необходимо определить, имеет вид: sum(X,Y), где X – числовой список, Y – сумма его элементов. Для его программирования используем принцип хвостовой рекурсии. Предельный случай рекурсии представляет собой факт, а общий случай – правило.

Рекурсивное определение суммы элементов состоит в том, что сумма элементов пустого списка равна нулю, а непустого – сумме элементов хвоста с добавлением головы.

```
sum([],0).
sum([X1|X2,Y) :- sum(X2,Y1), Y is Y1+X1.
```

Проверим, как работает наш предикат:

```
?-sum([1,2,4],X).
X=7;
НЕТ
```

Для записи данного предиката на Турбо-Прологе следует лишь заменить Y is Y1+X1 на Y=Y1+X1.

2 Определить на стандартном Прологе member(X,Y), который согласуется с базой, если элемент X содержится в списке Y.

Данный предикат легко определить, используя принцип хвостовой (частной) рекурсии: элемент содержится в списке, если он совпадает с головой, в противном случае данный элемент содержится в списке в том и только в том случае, если он содержится в хвосте списка. Данное

рекурсивное определение позволяет записать соответствующие утверждения Пролога, причем не требуется проверки на предельный случай, т.е. на пустой список.

```
member(X,[X|_]).
member(X,[Y|Z]) :- member(X,Z).
```

Проверим работоспособность написанного предиката:

```
?-member(a,[b,c,a,z]).
ДА
?-member(x, [b,c,a,z]).
НЕТ
```

Введенный предикат обладает интересным свойством, состоящим в том, что в качестве первого аргумента можно использовать переменную величину.

```
?-member(X,[ b,c,a,z]).
X=b;
X=c;
X=a;
X=z;
НЕТ
```

Представим себе, что предикат member является частью целевого утверждения вида:

```
..., member(X,[a,b,c,d]), p(X), ...
```

Тогда предикат p последовательно начинает согласовываться с базой с аргументами p(a), p(b), p(c), p(d). Таким образом, предикат member позволяет организовать перебор по заданному множеству элементов.

3 Построить предикат append(X,Y,Z), соединяющий два списка вместе аналогично соответствующей функции Лиспа:

?-append([1,2,3], [a,b,c], X).

X=[1,2,3,a,b,c]

Рекурсию проведем по первому списку. В предельном случае, когда первый список пуст, его присоединение ко второму списку не изменяет второй список. В рекурсивной ветви программы хвост первого списка присоединяется ко второму по рекурсивному предположению, и голова первого списка переносится в результат.

```
append([],X,X).
append([X|Y],Z,[X|P]) :- append(Y,Z,P).
?-append([1,2],[3,4,5],Z).
Z=[1,2,3,4,5]
```

Предикат append позволяет задавать не только прямые вопросы о соединении списков, но и обратные!

```
?- append([1,2,3],X,[1,2,3,4,5]).
X=[4,5]
?- append(X,[3,4,5],[1,2,3,4,5]).
```

```
X=[1,2]
```

Боле того, можно задать вопрос: из каких подсписков состоит исходный список?

```
?- append(X,Y,[1,2,3,4]).
X=[], Y=[1,2,3,4];
X=[1], Y=[2,3,4];
X=[1,2], Y=[3,4];
X=[1,2,3], Y=[4];
X=[1,2,3,4], Y=[];
НЕТ
```

4 Написать предикат, суммирующий все числовые атомы произвольного списка: sumatom (X,Y), где X – список, Y – сумма его атомов.

Для определения данного предиката удобно использовать принцип общей рекурсии. Хотя рекурсивное определение предиката аналогично его определению в Лиспе, для компактной записи следует использовать особенности Пролога.

Как всегда при общей рекурсии необходимо рассматривать два предельных случая: пустой список и числовой атом. Обработка случая атома становится простой, если соответствующее утверждение поместить после рекурсивной ветви, в которой выполняется отсечение и сложение атомов головы и хвоста. Согласование с последней ветвью будет происходить только тогда, когда аргумент не удается разбить на голову и хвост, т.е. он не является списком.

Запишем вышеизложенное на Прологе:

```
sumatom([],0).
sumatom([X|Y],Z) :-  !,sumatom(X,Z1),    sumatom(Y,Z2),
                     Z is Z1+Z2.
sumatom(X,X).
?- sumatom([[1,2],3,[[4]]],X).
X=10
```

5 Определить предикат memq(X,Y), проверяющий, содержится ли выражение X в списке Y.

В отличие от предиката member, голова списка Y может быть списком, в котором также следует искать X. Таким образом, рекурсивное определение предиката memq очень простое: X ищется в голове *или* в хвосте, а в предельном случае, когда Y становится атомом, он должен совпасть с X.

```
memq(X,X).
memq(X,[Y|Z]) :- memq(X,Y).
memq(X,[Y|Z]) :- memq(X,Z).
```

Проверка работоспособности:

```
?- memq(a,[s,z,[a,b],x]).
ДА
?-memq(a,[s,z,[y,b],x]).
НЕТ
```

6.6 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Что выведет интерпретатор Пролога для приведенных базы и цели?

```
pred(a1).
pred(a2).
main(X) :- asserta(pred(b1)), assertz(pred(b2)), pred(X).
?-main(A)
```

   a) A=a1
      A=b1
      A=a2
      A=b2
   b) A=b1
      A=a1
      A=a2
      A=b2
   c) A=a1
      A=a2
      A=b1
      A=b2
   d) A=a1
      A=a2
   e) A=b1
      A=b2
      A=a1
      A=a2

2. Каково будет значение X для приведенных базы и цели?

```
pred([],[]).
pred([A|B],[D|C]):-pred(A,C),pred(B,D).
pred([A|B],[A|C]):-not(pred(A,_)),pred(B,C).
?-pred([a,b,[c,d,[[e]]]],X).
```

   a) X=[a,b,[],c,d,[],[],e]
   b) X=[a,b,[c,d,[[e]]]]
   c) X=[[[[e]],d,c],b,a]]
   d) X=[a,b,c,d,e]
   e) X=[e,d,c,b,a]

3. Какая из приведенных ниже программ на Турбо-Прологе обеспечит указанный результат согласования?

```
Goal: pred([a,b,c],[1,2,3,4,5],X)
X=[["a"],["b","b"],["c","c","c"]]
```

a)
```
domains
 list1 = symbol*
 list2 = integer*
 list3 = list1*
predicates
 sprd(symbol,integer,list1)
 pred(list1,list2,list3)
clauses
 pred([],[],[]).
 pred([A|B],[C|D],[E|F]):-sprd(A,C,E),pred(B,D,F).
 sprd(_,0,[]).
 sprd(A,B,[A|C]):-B>0,D=B-1,sprd(A,D,C).
```

b)
```
domains
 list1 = symbol*
 list2 = integer*
 list3 = list1*
predicates
 sprd(symbol,integer,list1)
 pred(list1,list2,list3)
clauses
 pred([],_,[]).
 pred(_,[],[]).
 pred([A|B],[C|D],[E|F]):-sprd(A,C,E),pred(B,D,F).
 sprd(_,0,[]).
 sprd(A,B,[A|C]):-B>0,D=B-1,sprd(A,D,C).
```

c)
```
domains
 list1 = symbol*
 list2 = integer*
predicates
 sprd(symbol,integer,list1)
 pred(list1,list2,list1)
clauses
 pred([],[],[]).
 pred([A|B],[C|D],[E|F]):-sprd(A,C,E),pred(B,D,F).
 sprd(_,0,[]).
 sprd(A,B,[A|C]):-B>0,D=B-1,sprd(A,D,C).
```

d)
```
domains
 list1 = symbol*
 list2 = integer*
 list3 = list1*
predicates
 sprd(symbol,integer,list1)
 pred(list1,list2,list3)
clauses
 pred([],_,[]):-!.
 pred(_,[],[]):-!.
 pred([A|B],[C|D],G):-sprd(A,C,E),pred(B,D,F),
                      G=[E|F].
 sprd(_,0,[]).
 sprd(A,B,[A|C]):-B>0,D=B-1,sprd(A,D,C).
```

e)
```
domains
 list1 = symbol*
 list2 = integer*
 list3 = list1*
predicates
 pred(list1,list2,list3)
clauses
 pred([],[],[]).
 pred(_,0,[]).
 pred(A,B,[A|C]):-D=B-1,pred(A,D,C).
 pred([A|B],[C|D],[E|F]):-pred(A,C,E),pred(B,D,F).
```

4. Какая из приведенных ниже программ эквивалентна данной?

```
main(A,0,A).
main(A,B,[C]):-B>0,D is B-1,main(A,D,C).
```

a)
```
main(A,B,C) :- pred(A,B,0,[],C).
pred(_,A,A,B,B).
pred(A,B,C,D,E) :- C<B, F=C+1,pred(A,B,F,[A|D],E).
```

b)
```
main(A,B,C) :- pred(A,B,0,[],C).
pred(_,A,A,B,B).
pred(A,B,C,[A|D],E) :- C<B,F is C-1,pred(A,B,F,D,E).
```

c)
```
main(A,B,C) :- pred(A,B,1,[],C).
pred(_,A,A,B,B).
pred(A,B,C,D,E) :- C<B,F is C+1,pred(A,B,F,[A|D],E).
```

d)
```
main(A,B,C) :- pred(A,B,0,[],C).
pred(_,A,A,B,B).
pred(A,B,C,D,E) :- C<B,F is C+1,pred(A,B,F,[A|D],E).
```

e)
```
main(A,B,C) :- pred(A,B,0,[],C).
pred(_,A,B,A,B).
pred(A,B,C,D,E) :- C<B,F = C-1,pred(A,B,F,[A|D],E).
```

Ответы: 1.b; 2.a; 3.b,d; 4.d.

6.7 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ № 8

РЕКУРСИВНАЯ ОБРАБОТКА СПИСКОВ НА ПРОЛОГЕ

Задание: Написать базу, для рекурсивной обработки списков в среде Turbo-Prolog.
Код программ и тестовые примеры привести в отчете. Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

Варианты заданий взять те же, что и для лаюораторной работы № 2.

7 ДРУГИЕ ВОЗМОЖНОСТИ ПРОГРАММИРОВАНИЯ НА ПРОЛОГЕ

7.1 ОТЛАДКА ПРОГРАММ НА ПРОЛОГЕ

Основным средством отладки в Прологе является трассировка, т.е. слежение за процессом согласования целей. Трассировка позволяет наблюдать за тем, какие правила базы используются для доказательства целей и какие значения имеют связанные переменные.

В стандартном Прологе трассировка включается предикатом trace, а выключается предикатом notrace. При включенной трассировке процесс согласования предиката с базой связывается с четырьмя возможными событиями, которые выводятся на экран в момент их выполнения.

Событие **CALL** происходит, когда целевой предикат начинает согласовываться с началом базы.

Событие **EXIT** выполняется в момент удачного согласования целевого предиката с одним из утверждений базы.

Событие **FAIL** означает, что целевой предикат не согласуется с базой и выполняется откат назад к предыдущей цели.

Событие **REDO** означает, что целевой предикат начинает повторно согласовываться с базой данных.

Для каждого события при трассировке выводится целевой предикат, в котором связанные переменные заменены их значениями. Подобный способ отладки является эффективным для нахождения ошибок и понимания принципов согласования цели с базой данных.

Рассмотрим на конкретном примере выполнение трассировки. Дана следующая база:

```
нравится(витя, лена).
нравится(коля, лена).
нравится(лена, витя).
нравится(лена, лена).
```

Зададим теперь вопрос к базе: "кто нравится кому взаимно?" и включим трассировку.

```
?- trace.
?- нравится (X, Y), нравится(Y, X).
CALL: нравится(X,Y)
```

; начинается поиск соответствия первого целевого утверждения.

```
EXIT: нравится(витя, лена)
```

; соответствие найдено.

```
CALL: нравится(лена, витя)
```

; целевым становится второе утверждение, причем переменные связаны.

```
    EXIT: нравится(лена, витя)
```
; соответствие для второго найдено, и в результате получено решение.

```
    X=витя, Y=лена
```
; продолжение поиска

```
    FAIL: нравится(лена, витя)
```
; больше решений для второго предиката нет при данных значениях

; переменных

```
    REDO: нравится(X, Y)
```
; повторное согласование первого предиката, переменные вновь свободны

; и согласование начинается со второго утверждения базы

```
    EXIT: нравится(коля, лена).
```
; найдено второе решение

```
    CALL: нравится(лена, коля)
```
; начался поиск решения для второго предиката при заданных значениях

; переменных, причем поиск начинается с начала базы (движение вперед

; по целевым предикатам).

```
    FAIL: нравится(лена, коля)
```
; неудача, т.к. отсутствуют сведения, что Лене нравится Коля.

```
    REDO: нравится(X, Y)
```
; снова возвращение к первому предикату, причем теперь согласование

; начинается с третьего факта базы (откат назад).

```
    EXIT: нравится(лена, витя)
```
; очередное решение найдено.

```
    CALL: нравится(витя, лена)
```
; снова движение вперед ко второму предикату.

```
    EXIT: нравится(витя, лена)
```
; соответствие найдено и в результате получено второе решение.

```
    X=витя, Y=лена ;
    FAIL: нравится(витя, лена)
```
; при повторном поиске решений для второго предиката появляется данное

; событие, т.к. нет второго экземпляра данного факта в базе.

```
    REDO: нравится(X, Y)
```
; снова возвращение к первому предикату, согласование начинается

; с четвертого факта базы.

```
    EXIT: нравится(лена, лена)
```
; получено соответствие

```
    CALL: нравится(лена, лена)
```
; поиск решения по всей базе.

```
    EXIT: нравится (лена, лена)
```
; соответствие найдено и выводится третье решение.

```
    X=лена, Y=лена ;
```

```
    FAIL: нравится (лена, лена)
```
; больше таких данных нет.

```
    REDO: нравится(X, Y)
```
; теперь делается попытка согласования первого предиката с базой, но база

; уже закончилась!

```
    FAIL: нравится(X, Y)
```
; неудача согласования первого целевого утверждения и всей конъюнкции.

В Турбо-Прологе имеется возможность выполнять аналогичную трассировку программ. Трассировка включается выбором пункта меню: Options → Compiler directives→ Trace. Трассировка выводит события в окне трассировки и, кроме этого, в окне редактирования курсором отмечается утверждение базы, с которым в данный момент происходит согласование целевого предиката, что создает дополнительные преимущества анализа работы программы.

В Турбо-Прологе возможно включить трассировку на отдельные предикаты. Для этого используется команда trace с указанием имени предиката. Например,

trace like

7.2 ОПЕРАТОРЫ ПРОЛОГА

Строго говоря, в Прологе существует только один сложный тип данных – это предикаты. Вместе с тем, в Прологе имеется возможность записывать арифметические выражения вида a+b*c, которые при вычислении дают правильные приоритеты операций, между предикатами ставятся конъюнкции (','), с помощью вертикальной черты строятся списки и т.д. Оказывается, все конструкции, отличные от предикатов представляют собой **операторы**, которые могут размещаться между своими аргументами.

Операторы Пролога используются лишь для наглядной записи выражений и перед выполнением программы преобразуются в предикатную форму, обработка которой выполняется по стандартным правилам согласования. Рассмотрим примеры.

В Прологе символы арифметических операций '+','-','*','/' представляют собой предикаты с двумя аргументами (например, +(a,b)) и преобразуются к соответствующему виду:

```
a+b   → +(a,b)
a+b*c → +(a,*(b,c)).
```

Каждый оператор характеризуется своим именем, позицией, приоритетом и ассоциативностью. Приоритет операторов представляет собой целое положительное число и определяет порядок преобразования операторов в предикатную форму при отсутствии скобок. Чем ниже приоритет оператора, тем раньше он должен быть преобразован. В стандартном Прологе приоритет умножения равен 21, а приоритет сложения – 31, поэтому умножение

переводится раньше сложения (при арифметическом вычислении оно вычисляется раньше). Если переставить приоритеты умножения и сложения, то выражение a+b*c преобразуется к виду *(+(a,b),c).

Позиция оператора может быть инфиксной, префиксной и постфиксной. Инфиксные операторы располагаются *между двумя своими аргументами*. Примерами таких операторов являются арифметические операторы ('+','-','*','/'), операторы сравнения ('>','<','>=','=<'), конъюнкция, дизъюнкция (',', ';'), оператор правила (':-') и другие. Префиксные операторы располагаются *перед своим аргументом*. Примерами префиксных операторов являются not (not a → not(a)), оператор смены знака (унарный минус '-', -a → -(a)) оператор вопроса ('?-', ?-a → ?-(a)) и ряд других. Постфиксные операторы располагаются *после своего единственного аргумента*. Встроенные постфиксные операторы в Прологе отсутствуют, но в математике они встречаются и их можно определить в программе Пролога. Примером постфиксного оператора в математике является факториал a! → !(a).

Приоритет и позиция предикатов однозначно определяет порядок преобразования произвольного выражения с операторами различных приоритетов в предикатную форму. Последнее свойство операторов – ассоциативность – позволяет однозначно преобразовывать выражения с операторами одинакового приоритета. Пусть дано выражение: a+b+c, оно может быть преобразовано в предикатную форму двумя способами: +(a,+(b,c)) (т.е., если бы имелись скобки a+(b+c)) и +(+(a,b),c) (т.е. из вида (a+b)+c).

Ассоциативность позволяет определить однозначный порядок преобразования выражения без скобок. Для инфиксных операторов ассоциативность записывается в одной из 4 форм: xfx, xfy, yfx, yfy. f обозначает оператор, а символы x и y – его аргументы. Аргумент y означает, что на соответствующем месте слева или справа от оператора f может находиться оператор одинакового приоритета с f, а аргумент x означает, что соответствующий аргумент f может содержать лишь операторы меньшего приоритета.

Рассмотрим пример. Оператор '+' определен в стандартном Прологе как инфиксный оператор вида: yfx. Пусть дано выражение a+b+c. Это выражение можно интерпретировать как a+(b+c) и (a+b)+c. В первой интерпретации первый оператор '+' выражения a+b+c имеет слева аргумент a, справа – аргумент b+c. Но символ x выражения yfx означает, что справа у оператора не может быть другого оператора одинакового с f приоритета, поэтому первая интерпретация исключается. При второй интерпретации второй оператор '+' имеет слева аргумент a+b, а справа – аргумент c. Это допускается видом оператора yfx. Таким образом, задание ассоциативности приводит к однозначной интерпретации выражения a+b+c → +(+(a,b),c). Если же требуется выражение вида +(a,+(b,c)), то необходимо явно использовать скобки: a+(b+c).

Встроенная операция конъюнкция является инфиксным оператором вида xfy, поэтому выражение a,b,c переводится к виду: ,(a,(b,c)).

Наконец, оператор '>' является инфиксным оператором xfx, и, следовательно, выражение a>b>c не допускается, т.к. при любой его интерпретации с одной стороны от знака '>' оказывается оператор того же приоритета.

Префиксные и постфиксные операторы определяются аналогично. Выражение fx для префиксного оператора означает невозможность иметь справа оператор того же приоритета, а fy допускает наличие такого оператора. Например, оператор вопроса к базе имеет вид fx и поэтому невозможно выражение ?- ?- b.

Для постфиксных операторов соответствующие обозначения имеют вид: xf и yf.

Кроме встроенных операторов, предикат op позволяет определить дополнительные операторы, необходимые для решения задачи. Предикат op имеет три аргумента:

```
op(<приоритет>, <позиция/ассоциативность>,<имя>).
```

Пусть мы хотим определить операцию степень a^b. Ясно, что она должна выполняться раньше умножения, поэтому ее приоритет должен быть меньше приоритета умножения. С другой стороны, выражение a^b^c лучше интерпретировать как a^(b^c), поскольку (a^b)^c равно a^(b*c). Поэтому ассоциативность оператора определяется выражением xfy. Ниже приводится цель Пролога, определяющая оператор степени:

```
?-op(10,xfy,'^').
```

В Прологе имеется встроенный предикат вывода выражения X: **display**(X), который осуществляет вывод X в предикатной форме в отличие от предиката **write**(X), выводящей выражение в "естественной" форме. Используя предикат display можно увидеть внутреннее строение произвольного утверждения Пролога.

```
?-op(10,xfy,'^').
?-display(a*b^c^d*e).
*(*(a, ^(b,^(c,d) ) ), e)
```

Списки Пролога также строятся специальным предикатом '.' (точка) по следующему правилу: [X|Y] → .(X,Y). Например:

```
[a,b,c] → .(a, .(b, .(c,[])))
[[a],b] → .( .(a,[]), .(b,[]))
```

Приведем некоторые встроенные операторы Пролога.

```
?-op(255,xfx,':-')
?-op(255,fx,'?-')
?-op(254,xfy,';')
?-op(254,xfy,',')
?-op(60,fx,'not)
?-op(51,xfy,'.')
?-op(40,xfx,is)
```

```
?-op(40,xfx,'=')
?-op(40,xfx,'\=')
?-op(40,xfx,'<')
?-op(40,xfx,'=<')
?-op(40,xfx,'>=')
?-op(40,xfx,'>')
?-op(31,yfx,'-')
?-op(31,yfx,'+')
?-op(21,yfx,'/')
?-op(21,yfx,'*')
?-op(11,yfx,mod).
```

В заключение раздела приведем пример программы, выполняющую решение алгебраических неравенств.

Данная программа способна, например, доказать, что если e>0, c>0, b>0, a>0 и a>e, то b*(a+c)/e>b. Запишем все свойства неравенства '>' в виде правил базы, затем дополним базы фактами из условия задачи и зададим вопрос доказательства неравенства.

Сначала переопределим оператор '>' , т.к. встроенный оператор сравнивает числа и не работает с математическими выражениями.

```
?-op(40,xfx,'>').
```

Приоритет оператора '>' больше приоритетов арифметических операторов, поэтому выражения, содержащие арифметические операторы и '>' переводятся в "правильный" вид. Например, a*b>c → >(*(a,b),c). Теперь запишем ряд свойств неравенства '>' в виде правил Пролога.

1. Если A>0 и B>C*A, то B/A>C. На Прологе имеем:

   B/A>C :- A>0, B>C*A.

2. Если B>0 и A>C то. B*A>B*C

   A>C :- B>0, B*A>B>C.

3. Если A>B и C>0, то A+C>B.

   A+C>B :- A>B, C>0.

Этих трех общих правил достаточно для доказательства приведенного неравенства. Добавим в базу факты:

```
c>0.
e>0.
b>0.
a>0.
a>e.
```

Зададим теперь вопрос и включим трассировку для наблюдения за ходом решения задачи:

```
?-trace.
?- b*(a+c)/e>b.
CALL: b*(a+c)/e>b
EXIT: b*(a+c)/e>b
```

; целевое утверждение согласовалось с первым правилом базы и теперь

; целью является конъюнкция левой части правила со значениями

; переменных: e>0, b*(a+c)>b*e.

```
    CALL: e>0 ;
    EXIT: e>0
```

; первая конъюнкция доказывается нахождением в базе соответствующего факта.

```
    CALL: b*(a+c)>b*e ;
    EXIT: b*(a+c)>b*e
```

; вторая конъюнкция согласуется со вторым правилом базы и целью теперь

; является конъюнкция: b>0, a+c>e.

```
    CALL: b>0
    EXIT: b>0
```

; утверждение доказывается нахождением факта в базе.

```
    CALL: a+c>e
    EXIT: a+c>e
```

; цель согласуется с третьем правилом базы и левая часть правила

; приобретает вид: a>e, c>0.

```
    CALL: a>e
    EXIT: a>e
    CALL: c>0
    EXIT: c>0
```

; эти утверждения находятся среди фактов базы.

```
    ДА
```

; ответ Пролога, теорема доказана.

Данная задача является ярким примером *декларативного программирования*, т.е. метода написания программ, порядок выполнения действий определяют не команды языка, а исходные данные задачи. В приведенном примере сначала строится база, в которую заносятся все известные правила, касающиеся свойств неравенств. Их может быть большое количество, порядка нескольких сотен или тысяч. При этом не определяется порядок их просмотра, т.е. все свойства неравенств просто сваливаются в общую кучу без всякой структуры. Далее к базе задается вопрос, который и определяет ход решения задачи, какие утверждения будут доказываться и в каком порядке. Для некоторых задач декларативный подход к программированию оказывается очень мощным методом.

7.3 СТРУКТУРЫ

Еще одним сложным типом данных в Прологе являются структуры. Фактически структура - это тот же предикат, но используемый только в качестве аргумента других предикатов.

В стандартном Прологе синтаксис структур полностью совпадает с синтаксисом предикатов. В Турбо-Прологе для использования структур необходимо описать их в разделе DOMAINS.

Для описания структуры используются два идентификатора: имя определяемого типа и имя предиката. Например, опишем структуру PAIR, состоящую из символа и целого числа:

```
DOMAINS
 pair_type = pair(symbol, integer)
```

При определении новых типов на основе уже определенных структур используется имя типа. Например, список из структур PAIR описывается так:

```
 pair_list = pair_type*
```

В отличие от языков Си и Паскаль, элементы структуры в Прологе не имеют имен. Для доступа к ним используется предикат структуры. Например, определим предикат для форматированного вывода списка пар символ-число в виде строки:

```
PREDICATES
 print_list(pair_list).
 print_pair(pair_type)
CLAUSES
 print_pair(pair(A,B)) :-
                write("Символ ",A,", значение ",B), nl.
 print_list([]).     ; терминальная ветвь рекурсии
 ; рекурсивная ветвь
 print_list([A|B]):-print_pair(A), print_list(B).
```

Обратите внимание, что имя типа PAIR_TYPE использовалось при определении нового типа и описании аргументов предиката PRINT_PAIR в разделе PREDICATES. Для доступа к элементам структуры из правил раздела CLAUSES используется имя предиката PAIR.

Проверим, как будет работать этот предикат:

```
Goal: print_list([pair(a,2),pair(b,3),pair(cde,10)])
Символ a, значение 2
Символ b, значение 3
Символ cde, значение 10
Yes
```

Хотя в большинстве случаев без использования структур можно обойтись, часто предикат с аргументом-структурой выглядит более понятно, чем предикат с множеством аргументов более простых типов.

7.4 ПРОГРАММА ДЛЯ РЕШЕНИЯ АРИФМЕТИЧЕСКИХ РЕБУСОВ

В качестве примера «большой» программы на Прологе рассмотрим программу для решения арифметических ребусов. В ней мы будем использовать практически все средства языка Пролог, описанные в предыдущих главах. Рекомендуем внимательно прочитать

приведенное ниже описание, оно поможет вам закрепить ваши знания и послужит хорошей иллюстрацией для обсуждавшихся в данном пособии приемов программирования.

Арифметические ребусы и способы их решения

Арифметический ребус – это правильное арифметическое выражение, в котором цифры заменены буквами. Решить такой ребус значит выяснить, какой цифре соответствует каждая буква. Например, решением ребуса

```
  send
+ more
------
 money
```

является

```
  9567
+ 1085
------
 10652
```

т.е. s=9, e=5, n=6, d=7, m=1, o=0, y=2.

Для построения ребусов можно использовать любые арифметические операции, но мы ограничимся суммой двух слагаемых.

Простейший способ программирования решения таких ребусов на Прологе – это для заданного набора букв построить все возможные наборы соответствующих цифр и проверять, является ли такой набор решением ребуса. Этот способ безусловно даст все правильные ответы, но программа, написанная таким способом будет работать очень долго.

Мы запрограммируем другой способ, которым обычно пользуется человек при решении таких ребусов. Человек строит множество всех букв и множество всех допустимых цифр. Затем, начиная с младших разрядов, решающий устанавливает соответствие букв и цифр так, чтобы не получилось противоречия. Если противоречие все-таки возникло, производится откат на 1 шаг назад. Таким образом, большинство неправильных комбинаций отсеивается сразу.

Типы, необходимые для решения ребуса на Турбо-Прологе

Пусть слагаемые представлены списками символов, причем для удобства эти списки перевернуты, т.е. младшие разряды слагаемых находятся в списках слева. Для нашего примеры такие списки будут выглядеть следующим образом:

```
[d,n,e,s], [e,r,o,m], [n,y,e,n,o,m]
```

Первоначально множество допустимых (незанятых) цифр – это все цифры от 0 до 9, т.е. список чисел [0,1,2,3,4,5,6,7,8,9].

Соответствия символов и цифр будем хранить в виде списка структур pair = pr(symbol, integer). Первоначально список соответствий пуст.

Итак, для программирования решения ребуса на Турбо-Прологе нам необходимо определить типы для списка символов, списка целых чисел, структуры для описания соответствия и списка таких структур:

```
DOMAINS
  pair  = pr(symbol, integer) ; структура для соответствия
  listp = pair*
  list  = symbol*
  list2 = list*
  int   = integer ;просто короткий псевдоним
  listn = int*
```

Предикат getpair для поиска соответствий

В процессе поиска решения ребуса нам необходимо каждому символу ставить в соответствие число. При этом возможны два случая: соответствие уже было установлено ранее или символу необходимо присвоить какое-либо число из списка свободных чисел. Для выполнения такой операции будет служить предикат getpair.

```
PREDICATES
 getpair(symbol,int,listp,listn,listp,listn)
```

Опишем аргументы этого предиката. Первый аргумент – это символ, второй – число, которое ему соответствует. Следующие два аргумента содержат список соответствий (список структур pair) и список свободных чисел. Последние два – список соответствий и свободных чисел *после* назначения символу числа. Таким образом, если символу уже было поставлено в соответствие какое-либо число, то последние 4 аргумента попарно совпадают, т.к. соответствия не изменяются; в противном случае в список соответствий добавляется пара из первых двух аргументов, а из списка свободных чисел удаляется назначенное число.

Заметим, что при вызове предиката getpair 1-ый, 3-ий и 4-ый аргументы должны быть связанными, а остальные – свободными.

Запишем правила для предиката getpair.

Сначала рассмотрим случай, когда данный символ уже имеет соответствие. В этом случае в списке, заданном 3-им аргументом, должна быть пара, содержащая наш символ. Второй элемент этой пары содержит число, которое надо вернуть через 2-ой аргумент.

Для описания этого случая нам потребуется 2 правила. Первое описывает случай, когда наш символ содержится в голове списка соответствий. При этом согласование предиката необходимо прекратить, иначе в последующих правилах нашему символу могут быть назначены новые (дополнительные) числа, что недопустимо.

```
getpair(S,C,[pr(S,A)|Soot],Fre,
    [pr(S,A)|Soot],Fre):-!,A=C.
```

Если наш символ не содержится в голове списка соответствий, он может содержаться в его хвосте, поэтому необходимо рекурсивно перебрать все элементы этого списка. Обратите внимание, что голова списка соответствий не теряется, а переносится в 5-ый аргумент.

```
getpair(S,C,[A|Soot],Fre,[A|SootN],FreeN):-
  getpair(S,C,Soot,Fre,SootN,FreeN).
```

Если наш символ не найден в списке соответствий, т.е. 3-ий аргумент является пустым списком, то необходимо создать новое соответствие, используя числа из списка свободных (4-ый аргумент). При этом необходимо перебрать все возможные варианты. В приведенных ниже двух правилах используется та же идея, что и в предикате member из пункта 6.2.

```
getpair(S,C,[],[C|FreeN],[pr(S,C)],FreeN).
getpair(S,C,[],[A|Fre],SootN,[A|FreeN]):-
    getpair(S,C,[],Fre,SootN,FreeN).
```

Предикат sumlist для решения ребуса

При сложении чисел «столбиком» может возникать перенос единицы из младшего разряда, который нам необходимо учитывать. По окончании сложения значение переноса должно быть равным нулю. Кроме того, слагаемые и сумма могут быть неодинаковой длины, и это мы тоже должны учесть.

Предикат, который будет выполнять сложение списков символов и строить список соответствий, назовем sumlist:

```
PREDICATES
 sumlist(list, list, list, int,
    listp, listn,
    listp, listn)
```

Поясним значения аргументов этого предиката. Первые 3 аргумента являются списками символов, соответствующими трем слагаемым и сумме. 4-ый аргумент – значение переноса из младшего разряда. При первоначальном вызове предиката sumlist и в терминальной ветви значение этого параметра должно равняться 0, на промежуточных стадиях оно может принимать значения 0 или 1. Если бы вместо сложения использовалась операция умножения, значение этого параметра лежало бы в диапазоне от 0 до 9.

Следующие 2 параметра содержат уже построенный список соответствий и список еще неиспользованных чисел. При первоначальном запуске предиката список соответствий пуст, а список свободных чисел содержит все числа от 0 до 9. В терминальной ветви список соответствий является решением ребуса.

Последние 2 параметра – это решение ребуса (список соответствий) и список неиспользованных чисел. При первоначальном вызове предиката эти параметры должны быть переменными.

Теперь мы можем начать писать правила для предиката sumlist.

В предельном случае списки для слагаемых и суммы пусты, а значение списка соответствий и списка свободных чисел необходимо передать в качестве результата через два последних аргумента. В случае пустых списков согласование предиката необходимо прекратить, но полученный набор соответствий будет решением только в том случае, если значение переноса равно 0. Таким образом, терминальная ветвь предиката sumlist описывается правилом:

```
sumlist([],[],[],Per,A,B,A,B):-!,Per=0.
```

Следующее правило описывает случай, когда длина суммы больше длин слагаемых, как в нашем примере. В этом случае старший разряд суммы должен совпадать с разрядом переноса. Для проверки этого совпадения используем предикат getpair, определенный в предыдущем пункте.

Поскольку длина суммы не может превосходить длину самого длинного слагаемого больше, чем на 1, если оба слагаемых представлены пустыми списками, то и хвост суммы должен быть пустым.

```
sumlist([],[],[B|Bx],Per,Soot,Fre,SNew,FNew):-
                  !,Bx=[],getpair(B,Per,Soot,Fre,SNew,FNew).
```

Для удобства будем считать, что второе слагаемое всегда длиннее первого. Если это не так, то просто поменяем их местами:

```
sumlist(A,[],B,Per,Soot,Fre,Soot1,Free1):-
            !, sumlist([],A,B,Per,Soot,Fre,Soot1,Free1).
```

Теперь рассмотрим случай, когда одно слагаемое уже закончилось, а второе слагаемое и сумма еще содержат разряды. Кроме того, значение переноса может быть ненулевым.

Поступим так же, как и при сложении «столбиком»: сложим очередной разряд слагаемого и разряд переноса, младшую цифру результата (остаток от деления на 10) занесем в сумму, а старшую используем в качестве переноса при сложении оставшихся цифр.

Для того чтобы сложить младший разряд слагаемого с переносом, необходимо поставить в соответствие символу из головы списка-слагаемого некоторое число. Для этого используем предикат getpair. Получив очередной разряд суммы, с помощью того же предиката сверим его с головой из списка-суммы. Если это сравнение оказалось удачным, рекурсивно продолжим суммировать оставшиеся разряды. Обратите внимание, что результаты работы предиката getpair не теряются, а накапливаются, передаваясь между вызовами с помощью дополнительных переменных.

```
sumlist([],[A1|Ax1],[B|Bx],Per,St,Fre,SN,FN):-
  !,getpair(A1,Ar1,St,Fre,Soot1,Free1),
  B1= Ar1+Per,
  Br=B1 mod 10, PNew=(B1-Br)/10,
  getpair(B,Br,Soot1,Free1,Soot3,Free3),
  sumlist([],Ax1,Bx,PNew,Soot3,Free3,SN,FN).
```

Теперь осталось записать последнее правило, соответствующее случаю, когда оба слагаемых содержат разряды. Оно абсолютно аналогично предыдущему.

```
sumlist([A1|Ax1],[A2|Ax2],[B|Bx],Pr,S,F,SN,FN):-
  getpair(A1,Ar1,S,F,Soot1,Free1),
  getpair(A2,Ar2,Soot1,Free1, Soot2, Free2),
  B1= Ar1+Ar2+Pr,
  Br=B1 mod 10, PN=(B1-Br)/10,
  getpair(B,Br,Soot2,Free2, Soot3, Free3),
  sumlist(Ax1,Ax2,Bx,PN,Soot3,Free3,SN,FN).
```

Последний штрих

В принципе, все, что нужно для решения ребуса, мы написали, однако пользоваться предикатом sumlist довольно неудобно: нужно переворачивать списки, задавать начальные значения параметров. В данном пункте мы напишем дополнительные предикаты и цель, позволяющие даже неквалифицированному пользователю работать с нашей программой. Программа будет запрашивать ввод исходных данных и выводить результат в понятном виде. Диалог с системой будет выглядеть так:

```
Ребусо-решатель
Введите первое слагаемое: send
Введите второе слагаемое: more
Введите сумму: money
 9567
 1085
-----
10652
```

Во-первых, данные будут вводиться не в виде списков, а в виде целых символов. Для разбора слова на отдельные символы будем использовать предикат name. Используемые в нем стандартные предикаты Турбо-Пролога frontchar и str_char служат соответственно для выделения первого символа (char) идентификатора (symbol) и преобразования char к symbol.

```
name("",[]).
name(A,[B|C]):-frontchar(A,E,D),name(D,C),str_char(B,E).
```

Теперь полученный список символов нужно развернуть. Предикат слияния списков рассматривался в пункте 6.2.

```
revs([],[]).
revs([A|B],C):-revs(B,C1), appends(C1,[A],C).
```

К полученным спискам уже можно применять наш предикат sum_list. Результатом его работы будет список соответствий. Теперь, используя список соответствий и наши перевернутые списки символов, получим списки чисел в правильном порядке. Для этого напишем предикат revn, который использует уже знакомый нам предикат getpair. Из-за ограничений Турбо-Пролога на типы мы не можем использовать для слияния списков чисел предикат, разработанный для слияния символьных списков, хотя они отличаются только определением в секции PREDICATES.

```
revn([],[],_).
revn([A|B],C,X):-revn(B,C1,X),
  getpair(A,Ar,X,[],X,[]),appendn(C1,[Ar],C)
```

Теперь объединим эти предикаты в один предикат rebus, принимающий три введенных значения типа symbol и возвращающий три списка чисел. Этот же предикат проверяет, чтобы старшие разряды результата были ненулевыми.

```
rebus(A,B,C,A2,B2,C2):-
  name(A,A1), name(B,B1), name(C,C1),
  revs(A1,A3),revs(B1,B3),revs(C1,C3),
  All=[0,1,2,3,4,5,6,7,8,9],
  sumlist(A3,B3,C3,0,[],All,X,_),
  revn(A3,A2,X),revn(B3,B2,X),revn(C3,C2,X),
  A2=[A21|_],A21<>0,B2=[B21|_],B21<>0,
  C2=[C21|_],C21<>0.
```

Осталось организовать ввод-вывод. Для того, чтобы красиво напечатать результат, напишем предикат r1. Он использует предикаты len (вычисление длины списка), writex (вывод числа и лидирующих пробелов), tab (рисование горизонтальной черты заданной длины) и стандартный предикат nl для перевода строки. Вспомогательные предикаты приведены в полном листинге программы .

```
r1(A,B,C,Ax,Bx,Cx):-rebus(A,B,C,Ax,Bx,Cx),
  len(Ax,Al),len(Bx,Bl),len(Cx,Cl),
  Ac=Cl-Al,Bc=Cl-Bl,
  writex(Ax,Ac),writex(Bx,Bc),tab(Cl),
  writex(Cx,0),nl.
```

Цель содержит запросы на ввод данных и вызов предиката r1. Предикат fail, завершающий цель, обеспечивает получение всех возможных решений нашего ребуса.

Оттестируем нашу программу:

```
Ребусо-решатель
Введите первое слагаемое: donald
Введите второе слагаемое: gerald
Введите сумму: robert
```

```
526485
197485
------
723970

Ребусо-решатель
Введите первое слагаемое: aa
Введите второе слагаемое: ab
Введите сумму: cdc
 66
 65
---
131

 77
 74
---
151

 88
 83
---
171
```

Полный текст программы.

```
DOMAINS
 pair  = pr(symbol, integer)
 listp = pair*
 list  = symbol*
 list2 = list*
 int   = integer
 listn = int*
PREDICATES
 name(symbol,list)
 getpair(symbol,int,listp,listn,listp,listn)
 rebus(symbol,symbol,symbol,listn,listn,listn)
 writex(listn,int)
 revn(list, listn, listp)
 revs(list, list)
 appends(list,list,list)
 appendn(listn,listn,listn)
 sumlist(list,list,list,int,listp,listn,
  listp,listn)
 len(listn,int)
 tab(int)
 r1(symbol,symbol,symbol,listn,listn,listn)
CLAUSES
 len([],0).
 len([_|A],B):-len(A,C),B=C+1.
 tab(0):-nl,!.
 tab(A):-write("-"),B=A-1,tab(B).
 name("",[]).
```

```
name(A,[B|C]):-frontchar(A,E,D),name(D,C),
 str_char(B,E).
getpair(S,C,[pr(S,A)|St],Fr,[pr(S,A)|St],Fr):-
 !,A=C.
getpair(S,C,[A|Soot],Fre,[A|SootN],FreeN):-
 getpair(S,C,Soot,Fre,SootN,FreeN).
getpair(S,C,[],[C|FreeN],[pr(S,C)],FreeN).
getpair(S,C,[],[A|Fre],SootN,[A|FreeN]):-
 getpair(S,C,[],Fre,SootN,FreeN).
revs([],[]).
revs([A|B],C):-revs(B,C1), appends(C1,[A],C).
revn([],[],_).
revn([A|B],C,X):-revn(B,C1,X),
 getpair(A,Ar,X,[],X,[]),appendn(C1,[Ar],C)
appends([],X,X).
appends([A|B],X,[A|C]):-appends(B,X,C).
appendn([],X,X).
appendn([A|B],X,[A|C]):-appendn(B,X,C).
sumlist([],[],[],Per,A,B,A,B):-!,Per=0.
sumlist([],[],[B|Bx],Per,St,Fre,SNew,FNew):-!,
 Bx=[],getpair(B,Per,St,Fre, SNew, FNew).
sumlist(A,[],B,Per,Soot,Fre,Soot1,Free1):-!,
 sumlist([],A,B,Per,Soot,Fre,Soot1,Free1).
sumlist([],[A1|Ax1],[B|Bx],Pr,St,Fr,SN,FN):-!,
 getpair(A1,Ar1,St,Fr,Soot1,Free1),
 B1= Ar1+Pr,
 Br=B1 mod 10, PNew=(B1-Br)/10,
 getpair(B,Br,Soot1,Free1,Soot3,Free3),
 sumlist([],Ax1,Bx,PNew,Soot3,Free3,SN,FN).
sumlist([A1|Ax1],[A2|Ax2],[B|Bx],P,S,F,SN,FN):-
 getpair(A1,Ar1,S,F,Soot1,Free1),
 getpair(A2,Ar2,Soot1,Free1, Soot2, Free2),
 B1= Ar1+Ar2+P,
 Br=B1 mod 10,PN=(B1-Br)/10,
 getpair(B,Br,Soot2,Free2,Soot3,Free3),
 sumlist(Ax1,Ax2,Bx,PN,Soot3,Free3,SN,FN).
writex(A,X):-
 X>0,!,X1=X-1,write(" "),writex(A,X1).
writex([],_):-nl.
writex([A|B],C):-write(A),writex(B,C).
rebus(A,B,C,A2,B2,C2):-
 name(A,A1), name(B,B1), name(C,C1),
 revs(A1,A3),revs(B1,B3),revs(C1,C3),
 All=[0,1,2,3,4,5,6,7,8,9],
 sumlist(A3,B3,C3,0,[],All,X,_),
 revn(A3,A2,X),revn(B3,B2,X),revn(C3,C2,X),
 A2=[A21|_],A21<>0,B2=[B21|_],B21<>0,
 C2=[C21|_],C21<>0.
r1(A,B,C,Ax,Bx,Cx):-rebus(A,B,C,Ax,Bx,Cx),
 len(Ax,Al),len(Bx,Bl),len(Cx,Cl),
 Ac=Cl-Al,Bc=Cl-Bl,
 writex(Ax,Ac),writex(Bx,Bc),tab(Cl),
 writex(Cx,0),nl.
```

```
GOAL
 write("Ребусо-решатель"),nl,
 write("Введите первое слагаемое: "), readln(A),
 write("Введите второе слагаемое: "),readln(B),
 write("Введите сумму: "),readln(C),
 r1(A,B,C,X1,_,_),fail.
```

7.5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое трассировка? Для чего она предназначена?
2. Какие события отслеживает Пролог-система?
3. Как включается трассировка в стандартном Прологе?
4. Каковы дополнительные возможности трассировки в Турбо-Прологе?
5. Чем отличаются операторы от предикатов?
6. Какие типы операторов вы знаете?
7. Что такое ассоциативность оператора?
8. Как определяются новые операторы на Прологе?
9. Как определяются структуры на Турбо-Прологе?
10. Зачем при определении структур используются 2 идентификатора? Каково их назначение?

7.6 УПРАЖНЕНИЯ

1 Каким образом преобразуется в предикатную форму выражение (a+b)*c*(d+e/f)?

Выражения в скобках имеют максимальный приоритет, поэтому сначала рассмотрим выражения (a+b) и (d+e/f). Первое из них содержит всего один оператор и однозначно преобразуется в форму +(a,b).

Вторая скобка содержит два оператора. Возможны два варианта преобразования: +(d,/(e,f)) и /(+(d,e),f). Приоритет оператора '/' выше (его числовое значение меньше), чем приоритет оператора '+', и ни один оператор не может иметь аргумент с приоритетом выше его собственного, поэтому вариант /(+(d,e),f) невозможен.

Вне скобок находятся два оператора умножения, следовательно, возможны два варианта преобразования: *(*(+(a,b),c),+(d,/(e,f))) и *(+(a,b),*(c,+(d,/(e,f)))). Однако, поскольку ассоциативность оператора умножения имеет вид yfx, правый аргумент этого оператора не может быть одного с ним приоритета, т.е. вариант *(+(a,b),*(c,+(d,/(e,f)))) недопустим.

Итак, единственно правильный вариант преобразования исходного выражения в предикатную форму выглядит следующим образом:

```
*(*(+(a,b),c),+(d,/(e,f)))
```

2 Какие строки нужно добавить к разделу DOMAINS, чтобы определить список структур, состоящих из одноуровневых списков символов и целого числа, используемого в качестве счетчика?

Сначала определим тип для списка символов:

```
list_symb = symbol*
```

Теперь можно определять саму структуру. Для этого нам понадобятся 2 идентификатора: имя типа и имя предиката структуры. В скобках после имени предиката укажем типы элементов структуры: list_symb и integer:

```
struc_type = my_struc(list_symb, integer)
```

Теперь осталось определить список таких структур. В качестве базового типа для списка используется имя типа.

```
struc_list = struct_type*
```

Таким образом, в раздел необходимо добавить 3 строки:

```
list_symb = symbol*
struc_type = my_struc(list_symb, integer)
struc_list = struct_type*
```

7.7 ТЕСТЫ ДЛЯ САМОКОНТРОЛЯ

1. Как преобразуется в предикатную форму выражение a+b*c*d/e+f?

   a) `+(a,/(*(b,c,d),e),f)`
   b) `+(a,+(/(*(*(b,c),d),e)),f)`
   c) `+(+(a,/(*(*(b,c),d),e)),f)`
   d) `+(+(/(*(*(b,c),d),e),a),f)`
   e) `+(+(a,*(*(b,c),/(d,e)),f)`

2. Необходимо описать структуру STUD_TYPE записи студента, состоящую из двух строк (имени и фамилии) и списка целочисленных оценок. Какие строки необходимо написать в разделе DOMAINS?

   a) `stud_type = stud(string,string,integer*)`
   b) `stud_type = stud(string,string,list_i)`
      `list_i = integer *`
   c) `list_i = integer *`
      `stud_type = stud(string,string,list_i)`
   d) `stud_type = stud(char*,char*,integer*)`
   e) `student = stud(string,string,integer)`
      `stud_type = student*`

3. Какая из следующих строк определяет оператор факториала так, чтобы можно было вычислить выражение вида 5!+3! как (5)!+(3)!, а запись 3!! считалась ошибочной?

   a) `op(10,xf,'!').`

b) `op(50,xf,'!').`
c) `op(50,yf,'!').`
d) `op(10,yf,'!').`
e) `op(10,fy,'!').`

4. Известно, что:

```
x>0.
b>0.
a>c.
```

Какие правила следует добавить в базу, чтобы Пролог-система могла доказать следующее утверждение:

```
?-(a+b)*c>c*x+x
```

a)
```
A>B+X:-X>0,A>B.
A*X>B*X:-X>0,A>B.
```
b)
```
A>B+X:-X>0,A>B.
A*X>B:-X>B,A>1.
```
c)
```
A*X>B*X:-X>0,A>B.
A+B*C>A+B:-C>1.
```
d)
```
A+B*C>A+B:-C>1.
A*X>B:-X>B,A>1.
```
e)
```
A+C>X+C:-A>X,C>1.
A*X>B:-X>B,A>1.
```

Ответ: 1.c; 2.b,c; 3.a; 4.a.

7.8 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ № 9

РЕШЕНИЕ ЛОГИЧЕСКИХ ЗАДАЧ НА ПРОЛОГЕ

Задание: Написать базу для решения логической задачи. Предикаты базы должны максимально приближены к формулировке задачи.

Код программ и тестовые примеры привести в отчете. Быть готовым пояснить любой вопрос по тексту кода.

Варианты:

1. На одном заводе работали четыре друга: слесарь, токарь, фрезеровщик и сварщик. Их фамилии Борисов, Иванов, Петров и Семенов. У слесаря нет ни братьев, ни сестер, он — самый младший из друзей. Семенов, женатый на сестре Борисова, самый старший. Петров младше сварщика, но старшёе токаря. Назовите фамилии и профессии друзей.
2. Поезд Москва-Ленинград

В поезде Москва-Ленинград едут пассажиры Иванов, Петров и Сидоров. Такие же фамилии имеют машинист, кочегар и кондуктор поездной бригады. Известно, что

1) пассажир Иванов живет в Москве;
2) кондуктор живет на полпути от Москвы до Ленинграда;
3) пассажир, однофамилец кондуктора, живет в Ленинграде;
4) тот пассажир, который живет в одном городе с кондуктором,зарабатывает в месяц 300 долларов;
5) пассажир Петров зарабатывает в месяц 200 долларов;
6) Сидоров (из бригады) недавно выиграл у кочегара партию на бильярде.

Как фамилия машиниста?

3. Учителя

Учителя Альтман (А), Брендель (В) и Клаузнер (С) преподают в одном классе математику (М), физику (Ф), химию (Х), биологию (Б), немецкий язык (Н) и историю (И). Каждый учитель ведет по 2 предмета. Учитель химии живет в одном доме с учителем математики. Альтман - самый молодой из трех преподавателей. Учитель математики часто играет в шахматы с Клаузнером. учитель физики старше учителя биологии, но младше Бренделя. Тот из трех учителей, кто старше двух других, живет дальше всех от школы.

Какие предметы преподает каждый из трех учителей?

4. Кто есть кто

1. Смит, Джонс и Робинсон работают в одной поездной бригаде машинистом, кондуктором и кочегаром. Профессии их названы не обязательно в том же порядке, что и фамилии. В поезде, который обслуживает бригада, едут трое пассажиров с теми же фамилиями. В дальнейшем каждого пассажира мы будем почтительно называть "мистер" (м-р).
2. М-р Робинсон живет в Лос-Анжелесе.
3. Кондуктор живет в Омахе.
4. М-р Джонс давно позабыл всю алгебру, которой его учили в колледже.
5. Пассажир - однофамилец кондуктора живет в Чикаго.
6. Кондуктор и один из пассажиров, известный специалист по математической физике, ходят в одну церковь.
7. Смит всегда выигрывает у кочегара, когда им случается встречаться за партией в бильярд.

Как фамилия машиниста?

5. Кто есть кто?

На одном вечере среди гостей оказалось пять офицеров: пехотинец, артиллерист, лётчик, связист и сапер. Один из них был капитаном, трое - майорами и один - подполковник. Также известно, что:

1. У Яноша такое же звание, как и у сапера и ещё одного офицера, который служит в другом роде войск;
2. Офицер связист и Ференц - неразлучные друзья;
3. На днях офицер-лётчик вместе с Белой и Лайошем побывал у кого-то в гостях;
4. Недавно у артиллериста перестал работать радиоприёмник и он попросил Лайоша помочь связисту устранить неисправность;
5. Ференц чуть было не стал лётчиком, но потом по совету своего друга сапера избрал

другой род войск;
6. Янош по званию старше Лайоша, а Бела - старше Ференца;
7. Пятый офицер, Андраш, накануне вечера был в гостях у Лайоша.
Определите имя каждого офицера, его звание и род войск, в котором он служит.

6. Кондратьев, Давыдов, Федоров и Петров живут на нашей улице. Один из них — столяр, другой — маляр, третий — водопроводчик, четвертый повар.
Недавно маляр хотел попросить своего знакомого столяра сделать кое-что для своей квартиры, но ему сказали, что столяр работает в доме водопроводчика.
Известно также, что Петров никогда не слышал о Давыдове. Федоров играет иногда с маляром иногда с поваром в карты, а со столяром в шахматы. Столяр не знаком с Петровым, вот с Давыдовым часто ездит на рыбалку.
Кто чем занимается?

7. Корнеев, Докшин, Мареев и Скобелев — жители нашего города. Их профессии — пекарь, врач, инженер и милиционер. Корнеев и Докшин — соседи и всегда на работу ездят вместе. Докшин старше Мареева. Корнеев регулярно обыгрывает Скобелева в пинг-понг. Пекарь на работу всегда ходит пешком. Милиционер не живет рядом с врачом. Инженер и милиционер встречались единственный раз, когда милиционер оштрафовал инженера за нарушение правил уличного движения. Милиционер старше врача и инженера. Определите, кто чем занимается.

8. Борисов, Кириллов, Данин и Савин — инженеры. Один из них — автомеханик, другой — химик, третий — строитель, четвертый — радиотехник.
Борисов, который обыгрывает в шахматы Данина, но проигрывает Савину, бегает на лыжах лучше того инженера, который моложе его, и ходит в театр вдвое чаще, чем тот инженер, который старше Кириллова.
Химик, который посещает театр вдвое чаще, чем автомеханик, не является ни самым молодым, ни самым пожилым из этой четверки.
Строитель, который на лыжах бегает хуже, чем радиотехник, как правило, проигрывает в шахматных сражениях автомеханику.
Самый пожилой из инженеров лучше всех играет в шахматы и чаще всех бывает в театре, а самый молодой лучше всех ходит на лыжах.
Назовите профессии каждого из этой четверки инженеров, если известно, что ни в спорте, ни в приверженности к театру среди них нет двух одинаковых.

9. Студенты
Дина, Соня, Коля, Рома и Миша учатся в институте. Их фамилии — Бойченко, Карпенко, Лысенко, Савченко и Шевченко. Мать Ромы умерла. Родители Дины никогда не встречались с родителями Коли, Студенты Шевченко и Бойченко играют в одной баскетбольной команде.
Услышав, что родители Карпенко собираются поехать за город, мать Шевченко пришла к матери Карпенко и попросила, чтобы та отпустила своего сына к ним на вечер, но оказалось, что отец Коли уже договорился с родителями Карпенко и пригласил их сына к Коле.
Отец и мать Лысенко — хорошие друзья родителей Бойченко. Все четверо очень довольны, что их дети собираются пожениться.
Установите имя и фамилию каждого из молодых людей и девушек.

10. Семья Семеновых

В семье Семеновых пять человек: муж, жена, их сын, сестра мужа и отец жены. Все они работают. Один — инженер, другой — юрист, третий — слесарь, четвертый — экономист, пятый — учитель. Вот что еще известно о них.

Юрист и учитель не кровные родственники.

Слесарь — хороший спортсмен. Он пошел по стопам экономиста и играет в футбол за сборную завода.

Инженер старше жены своего брата, но моложе, чем учитель.

Экономист старше, чем слесарь.

Назовите профессии каждого члена семьи Семеновых.

11. Преподаватели

В педагогическом институте Аркадьева, Бабанова, Корсакова, Дашков, Ильин и Флеров преподают экономическую географию, английский язык, французский язык, немецкий язык, историю, математику.

Преподаватель немецкого языка и преподаватель математики в студенческие годы занимались художественной гимнастикой.

Ильин старше Флерова, но стаж работы у него меньше, чем у преподавателя экономической географии.

Будучи студентками, Аркадьева и Бабанова учились вместе в одном университете. Все остальные окончили педагогический институт.

Преподаватель французского языка – дочь Флерова.

Преподаватель английского языка — самый старший из всех по возрасту и по стажу работы. Он работает в этом институте с тех пор, как окончил его. Преподаватели математики и истории— его бывшие студенты.

Аркадьева старше преподавателя немецкого языка.

Назовите, кто какой предмет преподает?

12. Утверждение проектов-2

Согласно договоренности порядок утверждения нового проекта, в разработке которого участвуют учреждения A, B и C, таков:

1.если в утверждении сначала принимают участие учреждения A и B, то должно присоединиться к ним и учреждение C.

2. Если утверждение происходит сначала в учреждениях B и C, присоединяется и учреждение A.

Спрашивается, возможны ли такие случаи при утверждении проекта, когда принимали бы в нем участие только учреждения A и C без учреждения B (при сохранении вышеупомянутой договоренности о порядке утверждения проектов)?

13. Комитеты

В уставе одного клуба записаны следующие правила:

1) финансовый комитет должен быть избран из состава общего комитета;

2) никто не может быть одновременно членом и общего и библиотечного комитетов, если только он не состоит также в финансовом комитете;

3) никто из членов библиотечного комитета не может быть в финансовом комитете.

Определите:

1) может ли член библиотечного комитета быть также и членом общего комитета;

2) может ли член общего комитета не быть членом финансового комитета.

14. При составлении расписания на пятницу были высказаны пожелания, чтобы информатика была первым или вторым уроком, физика - первым или третьим, история - вторым или третьим. Можно ли удовлетворить одновременно все высказанные пожелания?

15. Обсуждая конструкцию нового трёхмоторного самолёта, трое конструкторов поочередно высказали следующие предположения:
1) при отказе второго двигателя надо приземляться, а при отказе третьего можно продолжать полёт;
2) при отказе первого двигателя лететь можно, или при отказе третьего двигателя лететь нельзя;
3) при отказе третьего двигателя лететь можно, но при отказе хотя бы одного из остальных надо садиться.
Лётные испытания подтвердили правоту каждого из конструкторов. Определите, при отказе какого из двигателей нельзя продолжать полёт.

16. На автоматизированном участке цеха стоят 5 станков, действия которых скоординированы следующим образом. Если работают первый и третий станки, то четвертый не работает при условии, если подключен пятый станок. Если же первый станок подключен без третьего или выключен пятый станок, то четвертый обязательно подключен. Если пятый станок работает вместе со вторым при выключенном первом станке, то включен третий станок. Если выключены второй или пятый станки, то одновременно выключен и четвертый.

1. Мы наблюдаем работу первого и четвертого станков. Что можно сказать о состоянии остальных станков, скрытых за перегородкой?

2. Можно ли в данной системе остановить для ремонта одновременно третий и четвертый станки, оставив хотя бы один из других станков включенным?

17. На этот раз на допрос вызваны четверо подозреваемых в ограблении: *A, B*, *C, D.* Неопровержимыми уликами доказано,что по крайней мере один из них виновен и что никто, кроме *A, B, C, D,* в ограблении не участвовал. Кроме того удалось установить следующее:

1) *A* безусловно не виновен;

2) если *B* виновен, то у него был ровно один сообщник;

3) если C виновен, то у него было ровно два сообщника.

Инспектору Крэгу было особенно важно узнать, виновен или не виновен D,так как D был опасным преступником. К счастью, приведенных выше фактов достаточно, , чтобы установить виновность или невиновность подозреваемого. Виновен или не виновен D?

18. Подсудимых четверо: *A, B,* C, *D*. Установлено следующее.

1) Если *A* и *B* оба виновны, то С был соучастником.
2) Если *A* виновен, то по крайней мере один из обвиняемых *B, C* был соучастником.
3) Если *C* виновен, то *D* был соучастником.
4) Если *A* не виновен, то *D* виновен.

Кто из четырех подсудимых виновен вне всякого сомнения и чья вина остается под сомнением?

19. По обвинению в ограблении перед судом предстали *A*, *B,* и *C.* Установлено следующее:
1) если *A* не виновен или *B* виновен, то С виновен;
2) если *A* не виновен, то *C* виновен.

Можно ли установить виновность каждого из трех подсудимых?

20. Подозреваемые в хищении *A, B, C* были вызваны на допрос. Установлено следующее: 1) никто, кроме *A*, *B* и *C*, в хищении не замешан; 2) *A* никогда не идет “на дело”, по крайней мере без одного соучастника; 3) *C* не виновен. Виновен или не виновен *B ?*

21. На складе было совершено хищение. Преступник (или преступники) вывез награбленное на автомашине. Подозрение пало на трех преступников-рецидивистов А, В и С, которые были допрошены. Установлено следующее:

1) Никто, кроме А, В, С, не был замешан в хищении;

2) С никогда не ходит “на дело” без А (и, возможно, других соучастников);

3) В не умеет водить машину.

Виновен или нет А?

СОДЕРЖАНИЕ