Криворожский государственный педагогический университет
Кафедра информатики и прикладной математики

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА НА ЯЗЫКЕ ЛИСП

Лабораторный практикум

Составители:
И.А. Теплицкий
С.А. Семериков

Кривой Рог
2004

СОДЕРЖАНИЕ

Лабораторная работа №1. **Основные понятия Лиспа. Функции и предикаты. Управляющие структуры** 3

Лабораторная работа №2. **Рекурсия. Функционалы. Массивы. Макросы. Работа с файлами** 26

Лабораторная работа №3. **Создание интерпретатора Лиспа на Лиспе** 49

Лабораторная работа №4. **Минимальная система компьютерной алгебры** 58

Лабораторная работа №5. **Создание простейшей экспертной системы** 66

Лабораторная работа №6. **Минимальный интерпретатор языка Пролог**... 80

Лабораторная работа №1.

Основные понятия Лиспа. Функции и предикаты. Управляющие структуры.

1. Введение. Основные понятия Лиспа

1.1 ЛИСП - язык функционального программирования.

Язык ЛИСП (LISP) был разработан в 1958 году американским ученым Джоном Маккарти как функциональный язык, предназначенный для обработки списков. (LISt Processing).

В основу языка положен серьезный математический аппарат:

- лямбда-исчисление Черча
- алгебра списочных структур
- теория рекурсивных функций

Долгое время язык использовался узким кругом исследователей. Широкое распространение язык получил в конце 70-х - начале 80-х годов с появлением необходимой мощности вычислительных машин и соответствующего круга задач. В настоящем - Лисп одно из главных инструментальных средств систем искусственного интеллекта. Он принят как один из двух основных ЯП для министерства обороны США и постепенно вытесняет второй ЯП - АДА.

Система AutoCAD разработана на Лиспе.

1.2 Основные особенности Лиспа.

Представление программы и данных производится одинаково - через списки. Это позволяет программе обрабатывать другие программы и даже саму себя.

Лисп, как правило, является интерпретирующим языком, также как BASIC.

Лисп безтиповый язык, это значит, что символы не связываются по умолчанию с каким-либо типом.

Лисп имеет необычный синтаксис. Из-за большего числа скобок LISP расшифровывают как Lots of Idiotic Silly Parentheses.

Программы, написанные на Лиспе во много раз короче, чем написанные на процедурных языках.

1.3 ЛИСП. Элементарные понятия.

Основу ЛИСПа составляют символьные выражения, которые называются S-выражениями и образуют область определения для функциональных программ.

S-выражение (Symbolic expression) - основная структура данных в ЛИСПе.

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

Атомы бывают двух типов - символьные и числовые.

Символьные атомы - последовательность букв и цифр, при этом должен быть по крайней мере один символ отличающий его от числа.

ДЖОН AB13 B54 10A

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

Символьный атом рассматривается как неделимое целое.

К символьным атомам применяется только одна операция: сравнение.

Числовые атомы - обыкновенные числа (константы).

В ЛИСПЕ список это последовательность элементов (list).

Элементами являются или атомы или списки.

Списки заключаются в скобки, элементы списка разделяются пробелами.

(банан) 1 атом

(б а н а н) 5 атомов

(a b (c d) e) 4 элемента

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

(+ 2 3) 3 атома

(((((первый) 2) второй) 4) 5) 2 элемента

Список, в котором нет ни одного элемента, называется пустым списком и обозначается "()" или символом NIL.

NIL - это и список и атом одновременно.

Пустой список играет такую же важную роль в работе со списками, что и ноль в арифметике.

Пустой список может быть элементом других списков.

(NIL) ;список состоящий из атома NIL

(()) ;то же самое, что и (NIL)

((())) ;- " -((NIL))

(NIL ()) ;список из двух других списков

NIL обозначает кроме этого, в логических выражениях логическую константу "ложь" (false).

Логическое "да"(true) задается символом "T".

Атомы и списки - это символьные выражения или S-выражения.

2. Функции. Базовые функции

2.1 Понятие функции

В математике функция отображает одно множество в другое. Записывается: Y = F (x)

Для x из множества определения (Domain) ставится в соответствие Y из множества значений (range) функции F.

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

Функция без аргументов имеет постоянное значение.

Функция в общем случае задает отображение из нескольких множеств в множество значений.

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

f (x)

g (x , y)

h (x , g (y , z))

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

(f x)

(g x y)

(h x (g y z))

(+ x y)

(- x y)

(* x (+ x z))

Достоинства :

– упрощается синтаксический анализ выражений, так как по первому символу

текущего выражения система уже знает, с какой структурой имеет дело.

– появляется возможность записывать функции в виде списков, т.е. данные (списки) и программа (списки) представляются единым образом.

2.2 Диалог с интерпретатором ЛИСПА

Транслятор Лиспа работает как правило в режиме интерпретатора.

Read-eval-print цикл

```
loop { read
evaluate
print)
```

В Лиспе сразу читается , затем вычисляется (evaluate) значение функции и выдается значение.

Пример:

```
( + 2 3 )
        5
```

В вводимую функцию могут входить функциональные подвыражения :

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

2.3 Блокировка QUOTE

В некоторых случаях не требуется вычисления значений выражений, а требуются само выражение. Если прямо ввести * (+ 2 3) , то 5 получится как значение. Но можно понимать (+ 2 3) не как функцию, а как список. S-выражения, которые не надо вычислять, помечают для интерпретатора апострофом " ' " (quote).

QUOTE - специальная функция с одним аргументом, которая возвращает в качестве значения этот аргумент.

```
' ( + 2 3 )
        ( + 2 3 )
```

Или

```
' y
        y
( quote ( + 2 3 ) )
        ( + 2 3 )
( quote y )
        y
```

Вместо апострофа можно использовать функцию QUOTE.

```
' ( a b ' ( c d ) )
        (a b ( quote c d ) )
```

Апостроф автоматически преобразуется в QUOTE.

```
( quote ' y )
        ( QUOTE Y )
'' y
        ( QUOTE Y )
( QUOTE QUOTE )
        QUOTE
```

Перед константами не надо ставить апостроф, так как число и его значение совпадают.

```
' 3.17
        3.17
( + ' 2 3 )
```

5
t
T
' t
T
' nil
NIL

2.4 Функция EVAL

Функция EVAL обеспечивает дополнительный вызов интерпретатора Лиспа. При этом вызов может производится внутри вычисляемого S-выражения. Функция EVAL позволяет снять блокировку QUOTE.

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

quote и eval действуют во взаимно противоположенных направлениях и аннулируют эффект друг друга.

Примеры:

```
( setq x ' ( a b c ) )
        ( a b c )
' x
        x
x
        ( a b c )
( eval ' x )
        ( a b c )
```

EVAL - это универсальная функция Лиспа, которая может вычислить любое правильно составленное s-выражение.

2.5 Использование символов в качестве переменных

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

```
t
        T
1.6
        1.6
```

Если попытаться вычислить символ, то система выдает ошибку.

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

Для Лиспа есть отличие:

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

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

Для связывания символов используется три функции:

SET
SETQ
SETF

Функция SET связывает символ со значением, предварительно вычисляя значения

аргументов.

В качестве значения функция SET возвращает значение второго аргумента.

Если перед первым аргументом нет апострофа, то значение будет присвоено значению этого аргумента.

(set 'd ' (x y z))
(x y z)
(set a ' e)
e
(set ' a ' b)
b
a
b
b
e

На значение символа можно сослаться записав его без апострофа.

Функция SETQ аналогична SET, но не вычисляет значение первого аргумента. Буква q на блокировку.

(setq m ' k)
k
m
k

Обобщенная функция SETF действует аналогично SETQ , но может использоваться для присвоения символу не только значения.

2.6 Базовые функции

В Лиспе для обработки списков, т.е. для разбора, анализа и построения списков существуют базовые функции. Они образуют систему аксиом языка, к которым сводятся символьные вычисления. В этом смысле их можно сравнить с основными арифметическими операциями. Простота базовых функций и их малое число - одно из достоинств Лиспа.

Базовые функции: CAR CDR CONS ATOM EQ. Функции CAR и CDR извлекают информацию из списка, или обеспечивают доступ к элементам списка. CONS объединяет элементы в список. ATOM и EQ проверяют аргументы.

Первый элемент списка - голова. Список без головы - хвост.

Функция CAR возвращает в качестве значения первый элемент списка, т.е. голову.

CAR < список >

(car '((head) tail)) -> (head)

(car (a b))

ошибка - имя несуществующей функции.

car применяется только для списков, т.е. если есть голова списка.

(car nil)
nil
(car ' nil)
nil
(car ' (nil a))
nil

Функция CDR

(cdr ' (a))
nil
(cdr nil)
nil

Так как список из одного элемента, его хвост - пустой список.

Для атомов
(cdr ' kat) ошибка, т.к. не список.
(cdr ' ((a b) (c d)))
((c d))

Имена функций CAR и CDR возникли по историческим причинам. Автор Лиспа реализовывал свою первую систему на машине IBM 605. Для хранения адреса головы списка использовался регистр CAR (content of address registr) Для хранения адреса хвоста списка использовался регистр CDR (content of decrement registr)

В CL можно наряду с CAR и CDR использовать имена FIRST REST.
(FIRST ' (a b c))
a
(FIRST (REST ' (1 2 3 4))
2
(FIRST ' (REST ' (1 2 3 4)))
REST

Рассмотрим (car (cdr ' ((a b) c d)))

Первым выполняется cdr ,а затем car, т.е. в Лиспе первым выполняются внутренние функции, а затем внешние. Исполнение идет "изнутри наружу".

Функция CONS строит новый список из своих аргументов.
cons: s-выражение + список = список
CONS < s-выражение > <список >
Примеры:
(cons ' a ' (b c))
(a b c)
(cons ' (a b) ' (c d))
((a b) c d)

Первый аргумент становится головой второго аргумента, который обязательно является списком.
(cons ' (a b) ' ((a b)))
((a b) (a b))
(cons (+ 1 2) ' (+ 3 4))
(3 + 3 4)
(cons ' (+ 1 2) (+ 3 4))
error
(cons ' (+ 1 2) ' (+ 3 4))
((+ 1 2) + 3 4)

Примеры манипуляции с пустым списком:
(cons ' (a b c) nil)
((a b c))
(cons nil ' (a b c))
(nil a b c)
(cons nil nil)
(nil)
(cons ' a nil)
(a)

Очень полезно, т.к. позволяет превращать элемент в список.

Функции по принципу их использования делятся на группы:

- назначение
- запись
- результат

(cons (car '(голова хвост))
(cdr '(голова хвост)))

(голова хвост)

Селекторы CAR и CDR являются обратными для конструктора CONS .

Список, разбитый с помощью функции CAR и CDR на голову и хвост, можно восстановить функцией CONS.

Комбинируя функции CAR и CDR можно выделить произвольный элемент списка :

(cdr (cdr (car ' ((a b c) d))))
(a b c)
(b c)
(c)

Можно записать проще :

(CDDAR ' ((a b c) d))

Т.е. записывается (C...R список)

A - CAR D - CDR

В CL можно использовать только три буквы, в других реализациях больше.

Функция NTH извлекает n-й элемент из списка.

Форма записи:

NTH < n > <список >

Пример :

Извлекаем седьмой элемент :

(NTH 7 '(1 2 3 4 5 6 7 8 9 10))
7

Функция LIST создает список из s- выражений (списков или атомов). Число аргументов может быть любое.

Примеры:

(list 1 2)
(1 2)
(list ' a ' b (+ 1 2))
(a b 3)
(list ' a ' (b c) ' d)
(a (b c) d)
(list nil)
(nil)
(list ' a)
(a)
(list ' a nil)
(a nil)

Функция LENGTH возвращает в качестве значения длину списка. т.е. число элементов на верхнем уровне

2.7 Арифметические функции

Арифметические функции могут быть использованы с целыми или действительными аргументами

Число аргументов для большинства арифметических функций может быть разным.

(+ x1 x2 ... xn) возвращает x1 + x2 + x3 + ... + xn.

(- x1 x2 ... xn) возвращает x1 - x2 - x3 - ... - xn.

(* y1 y2 ... yn) возвращает y1 x y2 * y3 * ... * yn.

(/ x1 x2 ... xn) возвращает x1/x2/... /xn.

Специальные функции для прибавления и вычитания единицы: (1+ x) и (1- x).

3. Определение функций. Предикаты

3.1 Определение функций

Необходимо поместить два элемента в начало списка, причем эту операцию мы хотели бы выполнять несколько раз с различными элементами.

Например :
(cons ' a (cons ' b ' (c d)))
(a b c d)
или
(cons ' train (cons ' truck ' (bus car boat)))
(train truck bus car boat)
Лучше, если была бы функция:
(cons-two ' a ' b ' (c d))
(a b c d)
(cons-two ' train ' truck ' (bus car boat))
(train truck bus car boat)
Такую функцию можно определить самим и использовать как встроенную.
Чтобы определить функцию, необходимо:

1. Дать имя функции
2. Определить параметры функции
3. Определить ,что должна делать функция

Для задания новых функций в Лиспе используется специальная форма defun.
(defun < имя-функции > < параметры > < тело-функции >)
Для нашего случая:
(defun cons-two (x y oldlist)
(cons x (cons y oldlist))))
Имя функции - символ. Параметры - список аргументов. Тело функции - вычисляемая форма от аргументов.
Значение определения функции defun - имя функции .
(defun cons-two (x y oldlist)
(cons x (cons y oldlist))))
CONS-TWO
Вызов функции:
(< имя-функции > < значения аргументов >)
(cons-two ' a ' b ' (c d))
(a b c d)
Значение функции - значение тела функции при заданных аргументах.
Примеры:
(defun double (num) (* 2 num))
(double 7)
14
Определенную функцию можно использовать как встроенную:
(setq z (double (+ 5 10)))
30
(double z)
60
Еще один пример:
Необходимо элемент поместить new на второе место в списке: (a c d) станет (a new c d)
Назовем функцию insert-second. Два аргумента: item oldlist.
Тело функции:
(cons (car oldlist) (cons item (cdr oldlist)))

Таким образом, определим функцию:

```
( defun insert-second ( item oldlist )
( cons ( car oldlist ) ( cons item ( cdr oldlist ) ) )
( insert-second 'b '( a c d ) )
        ( a b c d )
```

3.2 Передача параметров. Глобальные и локальные переменные

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

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

Например:

```
( defun f ( x ) ( setq x ' new ) ) ; меняет значение x
        f
( setq x ' old )
        old
x
        old
( f x )
        new
```

Еще пример:

```
( defun double ( num ) ( * num 2 )
        double
( setq num 5 )
        5
( double 2 )
        4
num
        5
```

3.3 Свободные переменные

Если в теле функции есть переменные, не входящие в число ее формальных параметров - они называются свободными. Значения свободных переменных остается в силе после ее выполнения.

Например:

```
( defun f1 ( y ) (setq x 3 ) )
        f1
( f1 5 )
        3
x
        3
```

3.4 Расчет сопротивления цепи

До тех пор пока мы не рассмотрели определение функций, мы не могли приступить к написанию программ на ЛИСПЕ. Теперь можно рассмотреть простейшую.

Задача:
Написать программу расчета сопротивления цепи.
r1=r2=r3=10
Последовательное соединение (serial)
R = R1 + R2
функция (s_r R1 R2)
Определение: (defun s_r (R1 R2) (+ R1 R2))
Параллельное соединение (parallel)
R = (R1 * R2) / (R1 + R2)
функция (p_r R1 R2)
Определение: (defun p_r (R1 R2) (/ (* R1 R2) (+ R1 R2)))
Расчет:
(s_r 10 (p_r 10 10))
    15

3.5 Дополнительные функции обработки списков

Функция APPEND объединяет два и более списков в один.
(APPEND < список - 1 > < список - 2 >)
Пример:
(append ' (a b) ' (c))
    (a b c)
APPEND объединяет элементы, не изменяя их.
(append ' (list) ' (' (a b) ' (c)))
    (list (quote (a b)) (quote (c)))
Рассмотрим несколько примеров, что бы показать отличие APPEND LIST CONS .
Примеры:
(list ' (a b) ' (c d))
    ((a b) (c d))
(cons ' (a b) ' (c d))
    ((a b) c d)
(append ' (a b) ' (c d))
    (a b c d)

- cons всегда берет два аргумента и помещает первый в начало второго.
- list берет один или больше аргументов и образует список, помещая аргументы в скобки.
- append образует новый список, убирая скобки вокруг аргументов и помещая их в один список.

Функция REVERSE изменяет порядок элементов в аргументе.
(REVERSE < список >)
Пример:
(reverse ' (a b c))
    (c b a)
Аргументом reverse должен быть список. reverse не меняет порядок в списках более нижнего уровня.
(reverse ' ((a b c) e))
    (e (a b c))
Функция LAST удаляет из списка все элементы кроме последнего.
(LAST < список >)
Пример:
(last ' (a b c))
(c)

3.6 Базовые предикаты

Предикат в Лиспе - это функция, которая определяет обладает ли аргумент определенным свойством, и возвращает в качестве значения T или NIL.

ATOM проверяет , является ли аргумент атомом.

Значение будет T , если атом , и nil в обратном случае.

(ATOM < s - выражение >)

Примеры:

(atom 'x)
t
atom '(a b))
nil
(atom (cdr ' (a b)))
nil
(atom (car ' (a b)))
t

Предикат atom с пустым списком nil:

(atom nil)
t
(atom ( ))
t

Предикат EQ сравнивает два символа и возвращает T, если они одинаковые, и nil в обратном случае.

(EQ < выражение 1 > < выражение 2 >)

Примеры:

(eq ' cat ' cat)
t
(eq ' cat ' dog)
nil
(eq ' cat (car ' (cat dog))
t
(eq t ' t)
t

EQ можно применять к числам, если они представлены одним типом.

(eq 123 123)
t

Предикат "=" сравнивает числа различного типа.

(= < число - 1 > < число - 2 >)

(= 3 3.0)
t
(= 3 0.3F 01)
t

EQL сравнивает и числа и символы.

(EQL arg1 arg2)

Истина только в том случае, если arg1 arg2 эквивалентны по EQ ,или это числа одного и того же типа, имеющие одно и тоже значение.

(EQL < аргумент - 1 > < аргумент - 2 >)

(eql ' a ' a)
t
(eql ' 12 ' 12)
t

EQUAL – самый общий предикат. Сравнивает не только символы, числа и списки.
(EQUAL < аргумент - 1 > < аргумент - 2 >)
(equal ' (a b c) ' (a b c))
t
(equal nil ' (( )))
nil
Предикат NULL проверяет является ли аргумент пустым списком.
(NULL < аргумент >)
(null ' ( ))
T
(null nil)
T
(null t)
nil

3.7 Предикаты типов

Предикат	Действие	T	NIL
atom	аргумент атом?	(atom 'a)	(atom '(a))
symbolp	аргумент символ?	(symbolp 'a)	(symbolp '10)
listp	аргумент список?	(listp '(a))	(listp 'a)
numberp	аргумент число?	(numberp 10)	(numberp 'a)

3.8 Числовые предикаты

Предикат	Действие	T	NIL
zerop	arg = 0	(zerop 0)	(zerop 1)
plusp	arg > 0	(plusp 1)	(plusp -1)
minusp	arg < 0	(minusp -1)	(minusp 1)
=	arg1 = arg2 = arg3 = ...	(= 2 2 2)	(= 1 2 3)
>	arg1 > arg2 > arg3 > ...	(> 3 2 1)	(> 2 3 1)
<	arg1 < arg2 < arg3 < ...	(< 1 2 3)	(< 2 3 1)

4. Логические функции. Управляющие структуры. Простые функции печати. PROGN

4.1 MEMBER

Функция проверяет, находится ли первый аргумент внутри списка, представленного вторым аргументом.

Если элемента в списке нет, MEMBER возвращает nil.

Функция MEMBER имеет два аргумента: первый аргумент - это s-выражение, второй - обязательно список.

(MEMBER < s-выражение > < список >)
(member ' b ' (c d b a))
(b a)

Если элемент в списке есть, то MEMBER возвращает хвост второго аргумента, начинающийся с этого элемента.

Таким образом member в качестве истины возвращает не T , а величину не-NIL.

В Лиспе для предикатов значение не-NIL означает истину.

(member ' c ' (a b (c)))
NIL

т.к. элемент находится не на том уровне.

4.2 Логические функции

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

Функция NOT берет один аргумент и возвращает значение, противоположное значению аргумента. Если аргумент NIL, NOT возвращает T. Если аргумент любое значение не-NIL, NOT возвращает NIL.

NOT имеет один аргумент, который может быть любым s-выражением (не только предикатом).

(NOT < s-выражение >)

Примеры:

```
not ( zerop 1 ) )
        T
( not nil )
        T
( not ' ( a b c ) )
        NIL
```

Логическая функция OR берет один или несколько аргументов. Она выполняет эти аргументы слева направо и возвращает значение первого аргумента, который не NIL. Если все аргументы OR имеют значение NIL, то OR возвращает NIL.

В OR , аналогично NOT, аргументами могут быть любые выражения.

(OR < arg-1 >< arg-2 >< arg-3 > . . .)

Примеры:

```
( or t nil )
        T
* ( or nil nil )
        NIL
* ( or ( atom 1) ( > 3 4 ) '( a b c ) ) )
        ( a b c )
```

Таким образом:

1) OR возвращает значение не-NIL, если по крайней мере один аргумент не NIL.

2) OR используется для выделения первого не пустого элемента в списке.

Логическая функция AND берет один или несколько аргументов. Она выполняет эти аргументы слева направо. Если она встречает аргумент, значение которого NIL, она возвращает NIL, не продолжая вычисления остальных. Если NIL аргументов не встретилось, то возвращается значение последнего аргумента.

(AND < arg-1 >< arg-2 >< arg-3 > . . .)

Примеры:

```
( and 5 nil )
        NIL
( and ' a ' b )
        b
( and ( listp nil ) ( atom nil ) )
        T
```

Таким образом, AND возвращает NIL значение, если хотя бы один из аргументов NIL, в противном случае возвращается значение последнего аргумента. AND используется для выделения пустого элемента в списке.

4.3 Управляющие структуры

В обычных языках программирования существуют средства управления вычислительным процессом: организация разветвлений и циклов.

В Лиспе для этих целей используются управляющие структуры - предложения (clause).

Внешне предложения записываются как вызовы функций: первый элемент предложения - имя; остальные - аргументы.

В результате вычисления предложения получается значение. Отличие от вызова функции в использовании аргументов.

Управляющие структуры делятся на группы. Одна из групп - разветвления вычислений. В нее входят условные предложения: COND IF WHEN UNLESS

Предложение COND является основным средством организации разветвления вычислений.

Структура условного предложения :

```
( COND ( < проверка-1 > < действие-1 > )
( < проверка-2 > < действие-2 > )
............................................................
( < проверка-n > < действие-n > ))
```

В качестве аргументов < проверка > и < действие > могут быть произвольные формы.

Значение COND определяется следующим образом:

1. Выражения < проверка-i >, выполняющие роль предикатов вычисляются последовательно, слева направо, до тех пор, пока не встретится выражение, значением которого не является NIL.

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

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

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

Последнюю строку можно записать: (t ' atom)))

Пример: (функция проверяет тип аргумента)

```
( defun classify ( arg )
( cond
( ( null arg ) nil )
( ( list arg ) 'list )
( ( numberp arg ) 'number )
( t 'atom ) )

( classify 'a )
        atom
( classify 5 )
        number
```

Еще один пример:

```
( defun double1 ( num )
( cond
( ( numberp num ) ( * num 2 )
( t ' не-число ) )
```

Эта функция гарантировано удваивает число, отбрасывая не числовые аргументы.

В COND могут отсутствовать результирующие выражения для предикатов, а так же присутствовать несколько действий.

```
( COND
( < проверка-1 > )
( < проверка-2 > < действие-2 > )
(< проверка-3 > < дейст.-31 > < дейст.-32 > < дейст.-33 >))
```

Если нет действия - результат значение предиката. Если не одно действие - результат значение последнего аргумента.

COND наиболее общее условное предложение. Часто пользуются более простыми условными предложениями.

(IF < условие > < то форма > < иначе форма >)

Пример:

(if (atom x) 'atom 'not - atom)

Условные предложения WHEN и UNLESS являются частными случаями условного предложения IF:

Если условие соблюдается, то выполняются формы.

(WHEN < условие >
< форма-1 > < форма-2 > < форма-3 > ...)

Если условие не соблюдается, то выполняются формы.

(UNLESS < условие >
< форма-1 > < форма-2 > < форма-3 > ...)

Любую логическую функцию можно заменить COND-выражением и наоборот.

Пример:

car-функция с проверкой:

```
( defun gcar ( l )
( cond
( ( listp l ) ( car l ) )
( t nil ) ) )
```

то же через логические функции:

```
( defun gcar1 ( l )
( and
( listp l ) ( car l ) ) )

(gcar '(a b))
        a
(gcar 'a)
        nil
```

4.4 Ввод и вывод информации

До сих пор в определяемых функциях ввод и вывод результатов осуществлялись в процессе диалога с интерпретатором.

Интерпретатор читал вводимое пользователем выражение, вычислял его значение и возвращал его пользователю.

Теперь мы рассмотрим специальные функции ввода и вывода Лиспа.

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

Вызов функции осуществляется в виде:

(READ)

функция без аргументов.

Как только интерпретатор встречает READ, вычисления приостанавливаются до тех пор пока пользователь не введет какой-либо символ или выражение.

(READ)

READ не указывает на ожидание информации. Если прочитанное выражение необходимо для дальнейшего использования, то READ должен быть аргументом какой либо формы, которая свяжет полученное выражение:

(setq x ' (read))

(+ 1 2) - вводимое выражение

(+ 1 2) - значение
x
(+ 1 2)
(eval x)
3

Еще один пример:

(defun tr (arg)
(list (+ arg (read)) (read)))

(tr 8)
14
cat
(22 cat)

Функция PRINT - это функция с одним аргументом. Она выводит значение аргумента на монитор, а затем возвращает значение аргумента.

Для вывода выражений можно использовать функцию print.

(PRINT < arg >)

print перед выводом аргумента переходит на новую строку, а после него выводит пробел.

* (print (+ 2 3))
5 - вывод
5 - значение

print и read - псевдофункции, у которых кроме значения есть побочный эффект.

Значение функции - значение ее аргумента.

Побочный эффект - печать этого значения.

Это не значит, что всегда две строки. Только когда print на высшем уровне, чего практически не бывает.

Пример:

(setq row ' (x x x))
(x x x)
(print (cdr row))
(x x) - печать
(x x) - значение
(cons 'o (print (cdr row)))
(x x) - печать
(o x x) - значение

4.5 PROGN, PROG1, PROG2

Функции PROGN, PROG1, PROG2 относятся к управляющим структурам, к группе объединяющей последовательные вычисления.

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

(PROGN < форма-1 > < форма-2 > ...... < форма-n >)
(PROG1 < форма-1 > < форма-2 > ...... < форма-n >)
(PROG2 < форма-1 > < форма-2 > ...... < форма-n >)

У этих предложений переменное число аргументов, которые они последовательно вычисляют:

Пример:

(progn (setq x 2) (setq y (* 3 2)))
6
(prog1 (setq x 2) (setq y (* 3 2)))

2
у
6
В Лиспе часто используется так называемый неявный PROGN, т.е вычисляется последовательность форм, а в качестве значения берется значение последней формы.

При определении функций может использоваться неявный PROGN .

```
( defun
< имя функции >
< список параметров >
< форма1 форма2 .... формаN > )
```

Тело функции состоит из последовательности форм отражающих, последовательность действий. В качестве значения функции принимается значение последней формы.

Пример:

Определим функцию, которая печатает список, вводит два числа, и печатает их сумму.

```
( defun print-sum ( ) ; функция без аргументов
( print ' ( type two number ) )
( print ( + ( read ) ( read ) ) ) )

( print-sum )
        ( type two number ) 3 4
        7
        7
```

5. LET. Циклические предложения

5.1 LET

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

В общем виде LET записывается

(LET ((var1 знач1) (var2 знач2)...) форма1 форма2 ... формаN)

LET вычисляется следующим образом:

1. Локальные переменные var1, var2, ...varM связываются одновременно со знач1, знач2, ..., значM.
2. Вычисляются последовательно аргументы форма1, форма2, формаN.
3. В качестве значения предложения принимается значение последнего аргумента (неявный PROGN).
4. После выхода из предложения связи переменных var1, var2, ...varM ликвидируются.

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

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

```
(defun rectangle (dim)
(let ((len (car dim)) (wid (cadr dim)))
(print (list 'area (* len wid)))
(print (list 'perimeter (* (+ len wid) 2)))))

(rectangle '(4 5))
        (area 20)
```

```
(perimetr 18)
(perimetr 18)
```

Можно сначала рассчитать площадь, т.е. определить

```
(defun rectangle (dim)
(let ((len (car dim)) (wid (cadr dim))
(area (* len wid))
(print ( 'area area))
(print (list 'perimeter (* (+ len wid)))))))
```

Обращение

```
(rectangle '(4 5))
```

даст ошибку, т.к значение area не определено.

Надо использовать предложение LET*, в котором значение переменных задается последовательно.

```
(defun rectangle (dim)
(let* ((len (car dim)) (wid (cadr dim))
(area (* len wid)))
(print (list 'area area))
(print (list 'perimeter (* (+ len wid) 2)))))))
```

5.2 Условный выход из функции: PROG RETURN

Встречаются ситуации, когда из тела функции, представленного последовательностью форм, требуется выйти, не доходя до последней формы. Это можно сделать используя предложения PROG RETURN, которые используются вместе.

Рассмотрим пример.

Необходимо написать функцию которая вводит два значения. Если это числа функция печатает их сумму и разность.

Если хотя бы одно не является числом, печатается nil.

```
(defun s-d ()
(prog (x y); локальные переменные
(print '(type number))
(setq x (read))
(and (not (numberp x)) (return nil))
(print '(type number))
(setq y (read))
(and (not (numberp y)) (return nil))
(print (+ x y))
(print (- x y))))

(s-d)
        (type number)8
        (type number) (1)
        nil
```

return возвращает результат для prog.

Если return не встретился - результат prog будет nil .

```
(s-d)
        (type number)8
        (type number)1
        9
        7
        nil
```

Если локальных переменных нет, записывается (prog ()...)

5.3 Дополнительные функции печати

PRINT печатает значение аргумента без пробела и перевода на другую строку:

```
(progn (print 1) (print 2) (print 3))
        123
```

СТРОКИ - последовательность знаков заключенная в кавычки.

"string"

СТРОКА - специальный тип данных в Лиспе. Это атом, но не может быть переменной. Как у числа значение строки сама строка.

```
"(+ 1 2)"
        "(+ 1 2)"
```

Строки удобно использовать для вывода с помощью оператора PRINC.

PRINC печатает строки без "", без пробела и перевода строки

Пример

```
(progn (setq x 4) (princ " x = ")
(prin1 x) (princ " m "))
        x = 4 m
```

" m ": значение последнего аргумента.

PRINC обеспечивает гибкий вывод.

TERPRI производит перевод строки. Как значение возвращает nil.

```
(progn (setq x 4) (princ "xxx ") (terpri) (princ "xox "))
        xxx
        xox
        " xox"
```

5.4 Циклические предложения

Циклические вычисления в Лиспе выполняются или с помощью итерационных (циклических) предложений или рекурсивно. Познакомимся вначале с циклическими предложениями

Предложение LOOP реализует бесконечный цикл

(LOOP форма1 форма2 .....)

в котором формы вычисляются до тех пор, пока не встретится явный оператор завершения RETURN.

Определим функцию add-integer, которая будет брать один аргумент, являющийся положительным целым, и возвращает сумму всех чисел между 1 и этим числом.

1+2+3+4+ ... +N

1+2+3+4=10

```
(add-integers 4)
        10
(defun add-integers (last)
(let (( count 1) (total 1))
(loop
(cond (( equal count last) (return total)))
(setq count (+ 1 count))
(setq total (+ total count)))))
```

Еще один пример численной итерационной функции. Определим функцию выполняющую умножение двух целых чисел через сложение. Т.е. умножение x на y выполняется сложением x с самим собой y раз.

Например, 3 x 4 это 3 + 3 + 3 + 3 = 12

```
(int-multiply 3 4)
```

```
    12
(defun int-multiply (x y)
(let ((result 0)( count 0))
(loop
(cond (( equal count y) (return result)))
(setq count (+ 1 count))
(setq result (+ result x)))))
```

Из приведенных примеров можно получить общую форму для численной итерации

```
(defun < имя-функции > < список-параметров >
(let (< инициализация переменной индекса >
< инициализация переменной результата >)
(loop
(cond < проверка индекса на выход > (return результат))
< изменение переменной счетчика >
< другие действия в цикле,
включая изменение переменой результата >)))
```

Еще пример. Определим функцию factorial

```
(factorial 5)
    120
1 x 2 x 3 x 4 x 5 = 120
(defun factorial ( num )
(let ((counter 0)( product 1))
(loop
(cond (( equal counter num) (return product)))
(setq counter (+ 1 counter))
(setq product (* counter product )))))
```

Пример,

```
( progn (setq x 0)
(loop (if ( = 3 x) (return 't) (print x))
(setq x (+ 1 x))))
    0
    1
    2
    t
```

Определим функцию, использующую печать и ввод. Функция без аргументов читает серию чисел и возвращает сумму этих чисел, когда пользователь вводит не число. Функция должна печатать " Enter the next number: " перед каждым вводом.

```
( read-sum)
    Enter the next number: 15
    Enter the next number: 30
    Enter the next number: 45
    Enter the next number: stop
    90
(defun read-sum ()
(let ((input) (sum 0))
(loop
(princ "Enter the next number:")
(setq input (read))
(cond (( not (numberp input)) (return sum)))
(setq sum (+ input sum)))))
```

Предположим, что нам необходима функция double-list, принимающая список чисел и возвращает новый список в котором каждое число удвоено.

```
(double-list '(5 15 10 20))
          (10 30 20 40)
(defun double-list ( lis )
(let ((newlist nil))
(loop
(cond (( null lis ) (return newlist)))
(setq newlist (append newlist (list (* 2 (car lis)))))
(setq lis (cdr lis )))))
```

Посмотрим как будет идти вычисление:

	list	newlist
Начальное состояние	(5 15 10 20)	()
итерация 1	(15 10 20)	(10)
итерация 2	(10 20)	(10 30)
итерация 1	(20)	(10 30 20)
итерация 4	()	(10 30 20 40)
результат		(10 30 20 40)

5.5 DO

Это самое общее циклическое предложение.

Общая форма

```
( DO (( var1 знач1 шаг1) ( var2 знач2 шаг2)....)
( условие-окончания форма11 форма12...)
форма21
форма21 ...)
```

1) Вначале локальным переменным var1 ..varn присваиваются начальные значения знач1..значn. Переменным, для которых не заданы начальные значения присваивается nil.

2) Затем проверяется условие окончания, если оно выполняется вычисляются форма11, форма12... В качестве значения берется значение последней формы.

3) Если условие не выполняется, то вычисляются форма21, форма22...

4) На следующем цикле переменным vari присваиваются одновременно новые значения определяемые формами шагi и все повторяется.

Пример

```
( do (( x 1 ( + 1 x)))
(( > x 10) ( print 'end))
( print x))
```

Будет печатать последовательность чисел, в конце end.

Можно сравнить итерационное вычисление с LOOP и DO.

Напишем функцию list-abs, которая берет список чисел и возвращает список абсолютных величин этих чисел.

```
(defun list-abs (lis)
(let ((newlist nil))
(loop
(cond (( null lis ) (return (reverse newlist))))
(setq newlist (cons (abs (car lis)) newlist))
(setq lis (cdr lis )))))


          (list-abs '(-1 2 -4 5))
```

То же, только через DO

```
(defun list-abs (lis)
(do ((oldlist lis (cdr oldlist))
(newlist nil (cons (abs (car oldlist)) newlist)))
```

```
((null oldlist) (reverse newlist)))))
```

Может одновременно изменяться значения нескольких переменных

```
( do (( x 1 (+ 1 x))
( y 1 (+ 2 y))
( z 3)); значение не меняется
(( > x 10) ( print 'end))
(princ " x=") ( prin1 x)
(princ " y=") ( prin1 y)
(princ " z=") ( prin1 z) (terpri))
```

Можно реализовать вложенные циклы

```
( do (( x 1 (+ 1 x)))
(( > x 10))
( do (( y 1 (+ 2 y)))
(( > y 4))
( princ " x= ") ( prin1 x)
( princ " y= ") ( prin1 y)
(terpri) ))
```

Напишем функцию, которая будет читать элементы с клавиатуры и объединять в список. Ввод будет закончен, когда будет введен последний элемент end

```
( defun appen-read ()
( do (( x ( list ( read)) ( append x (list (read)))))
(( equal (last x) '(end))); ???? '(end)
(print x)))

( appen-read)
```

Если есть необходимость, можно использовать DO* аналогично LET*.

5.6 DOTIMES

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

Общая форма

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

здесь var - переменная цикла,

num - форма определяющая число циклов,

форма - return - результат, который должен быть возвращен.

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

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

Последним вычисляется форма-return.

Если форма-return отсутствует, возвращается nil.

Например,

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

Задания:

1. Напишите выражение, вычисляющее среднее арифметическое чисел 23, 5, 43, 17.
2. Напишите последовательность вызовов car, cdr, выделяющих из приведенных списков символ «цель»:
   а) (1 2 цель 3 4)
   б) ((1) (2 цель) (3 (4)))
   в) ((1 (2 (3 4 цель))))
3. Определите функции (null x), (caddr x), (list x1 x2 x3), используя car и cdr.
4. Опишите с помощью предиката > и условного предложения функцию, возвращающую среднее из трех чисел.
5. Запрограммируйте с помощью предложения do итеративную функцию факториал.
6. Определите функцию (произведение x y), вычисляющую произведение двух целых положительных чисел.
7. Определите функцию определения длины списка рекурсивно с помощью cond и итеративно с помощью prog.
8. Определите рекурсивную функцию (фиб x), вычисляющую x-ый элемент ряда Фибоначчи.
9. Определите функцию, добавляющую в конец списка заданное число.
10. Определите функцию читай_фразу, которая вводит фразу на естественном языке, заканчивающуюся вопросительным или восклицательным знаком, и преобразует ее в список.
11. Определите функцию (линия x), печатающую x раз звездочку.
12. Используя функцию (линия x), напишите функцию (прямоугольник x y), заполняющую всю область x на y звездочками.
13. Определите функцию, которая спрашивает у пользователя имя и вежливо отвечает ему.
14. Определите функцию, возвращающую последний элемент списка.
15. Определите функцию, удаляющую последний элемент списка.
16. Определите предикат, проверяющий, является ли аргумент одноуровневым списком.
17. Определите функцию, результатом которой будет первый атом списка.
18. Определите функцию, которая обращает список (а б в) и разбивает его на уровни (((в) б) а).
19. Определите функции, преобразующую список (а б в) к виду (а (б (в))) и наоборот.
20. Определите функцию, удаляющую из списка каждый второй элемент.
21. Определите функцию, разбивающую список (а б в г ...) на пары ((а б) (в г) ...).
22. Определите функцию, которая, чередуя элементы списков (а б ...) и (1 2 ...), образует новый список (а 1 б 2 ...).
23. Напишите программу, вводящую коэффициенты квадратного уравнения, вычисляющую и выводящую его корни.

Лабораторная работа №2.

Рекурсия. Функционалы. Массивы. Макросы. Работа с файлами.

6. Рекурсия

Функция является рекурсивной, если в ее определении содержится вызов самой этой функции.

Рекурсия – основной и самый эффективный способ организации повторяющихся вычислений в функциональном программировании и в Лиспе.

ПРИМЕР. Определим функцию MEMBER

```
(defun MEMBER (item list)
        (cond ((null list) nil)
        ((eql (car list) item) list)
        (t (MEMBER item (cdr list)))))
```

6.1 Численная рекурсия

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

Например (sumall 9) должно вернуть 45

Это можно решить циклически; мы решим рекурсивно.

Отметим два факта:

1. Если n=0, сумма чисел между 0 и n равна 0.
2. Если n>0, сумма чисел между 0 и n равна n плюс сумма чисел между 0 и n-1.

Эти два факта переводятся непосредственно в определение функции:

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

Производится проверка n : равно нулю или нет.

Если значение n=0, то функция возвращает 0 .

В противном случае , функция вызывает сама себя для вычисления суммы чисел между 0 и n-1 и и добавляет n к этой сумме.

6.2 Как работает рекурсивная функция

Посмотрим как работает рекурсивная функция. Проследим за несколькими вызовами.

Как работает (sumall 0) все ясно. Функция возвращает 0. Эта ветвь в cond называется терминальной (terminating) завершающей, так как функция дает значение без рекурсивного вызова.

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

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

Если посмотреть,

(sumall 3) вызывает (sumall 2)

(sumall 1) вызывает (sumall 0).

После того как (sumall 0) вернет 0,

(sumall 1) вернет 1,

(sumall 2) вернет 3,

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

Выполним трассировку этого вызова:

(trace sumall)

```
(sumall)
(sumall 9)
Entering: SUMALL, Argument list: (9)
 Entering: SUMALL, Argument list: (8)
  Entering: SUMALL, Argument list: (7)
   Entering: SUMALL, Argument list: (6)
    Entering: SUMALL, Argument list: (5)
     Entering: SUMALL, Argument list: (4)
      Entering: SUMALL, Argument list: (3)
       Entering: SUMALL, Argument list: (2)
        Entering: SUMALL, Argument list: (1)
         Entering: SUMALL, Argument list: (0)
         Exiting: SUMALL, Value: 0
        Exiting: SUMALL, Value: 1
       Exiting: SUMALL, Value: 3
      Exiting: SUMALL, Value: 6
     Exiting: SUMALL, Value: 10
    Exiting: SUMALL, Value: 15
   Exiting: SUMALL, Value: 21
  Exiting: SUMALL, Value: 28
 Exiting: SUMALL, Value: 36
Exiting: SUMALL, Value: 45
45
(untrace sumall)
```

Этот простой случай иллюстрирует несколько правил в записи рекурсивной функции.

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

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

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

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

6.3 Как писать рекурсивные функции

При написании рекурсивных функций мы должны планировать терминальные и рекурсивные ветви.

Таблица. Рекурсивная функция SUMALL

* Шаг 1. Завершение (Терминальная ветвь)
  n=0 - аргумент
  (sumall 0) = 0 - значение
* Шаг 2. Рекурсивная ветвь
  Рекурсивные отношения между (sumall n) и (sumall(- n 1)
  o 2a. Примеры рекурсии

(sumall n)	(sumall(- n 1)
(sumall 5)=15	(sumall 4)=10
(sumall 1)=1	(sumall 0)=0

o 2b. Характеристическое рекурсивное отношение
(sumall n) может быть получена из (sumall (- n 1) прибавлением N

1. Планирование терминальной ветви.

При написании рекурсивной функции мы должны решить, когда функция может вернуть значение без рекурсивного вызова.

2. Планирование рекурсивной ветви.

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

Таким образом мы должны решить:

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

2. Кроме этого необходимо построить форму, называемую рекурсивным отношением, которая связывает правильное значение текущего вызова со значением рекурсивного вызова.

В нашем случае это (sumall n) и (sumall (- n 1)).

Иногда просто найти это отношение, а если не получается надо выполнить следующую последовательность шагов.

a. Определить значение некоторого простого вызова функции и ее соответствующего рекурсивного вызова.

b. Определить соотношение между парой этих функций.

Пример. Определим функцию power. Она берет два численных аргумента m и n вычисляет значение m в степени n.

(power 2 3) возвращает 8

Вначале составим рекурсивную таблицу.

* Шаг 1. Завершение (Терминальная ветвь)
n=0 - аргумент
(power 2 0) = 1 - значение
* Шаг 2. Рекурсивная ветвь
Рекурсивные отношения между
(power m n) и (power m (- n 1))
o 2a. Примеры рекурсии
(power 5 3) =125 (power 5 2) = 25
(power 3 1)=3 (power 3 0)=1
o 2b. Характеристическое рекурсивное отношение
(power m n) может быть получена из (power m (- n 1) умножением на m

Текст функции.

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

6.4 CDR рекурсия

Рассмотрена рекурсивная обработка чисел. Когда информация представлена в виде списка, то появляется необходимость рекурсивной обработки списков. Основная рекурсия над списками – CDR рекурсия.

Логика и структура CDR рекурсии сходна с численной рекурсией.

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

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

Составим рекурсивную таблицу для (list-sum lis)

* Шаг 1. Завершение (Терминальная ветвь)
(list-sum nil) = 0 - значение
* Шаг 2. Рекурсивная ветвь

Рекурсивные отношения между
(list-sum lis) и (list-sum (cdr lis))
o 2a. Примеры рекурсии
(list-sum '(2 5 3)) =10 (list-sum '(5 3)) = 8
(list-sum '(3)) =3 (list-sum nil)=0
o 2b. Характеристическое рекурсивное отношение (list-sum lis) может быть получена из (list-sum (cdr lis)) сложением с (car lis).

Текст функции.

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

6.5 Несколько терминальных ветвей

Мы рассмотрели случай рекурсии с одной терминальной и одной рекурсивной ветвью. Однако в определении рекурсивной функции может быть несколько терминальных ветвей. Две терминальные ветви будут в том случае, когда ведется поиск цели в последовательности значений и мы желаем получить результат, как только цель найдена.

* Ветвь 1. Цель найдена и надо вернуть ответ
* Ветвь 2. Цель не найдена и нет больше элементов.

Пример. Написать функцию greaternum.

Она имеет два аргумента: список чисел и заданное число. Функция возвращает первое число в списке превышающее заданное. Если этого числа нет - возвращается заданное число.

Программа.

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

Порядок ветвей в рекурсивном определении существенен.

6.6 Несколько рекурсивных ветвей

Несколько рекурсивных ветвей может понадобиться, если функция обрабатывает все элементы в структуре, но использует некоторые элементы отлично от других.

В этом случае составляются два рекурсионных отношения.

Пример. Напишите функцию negnums, которая получает список чисел и возвращает список, который содержит только отрицательные числа (0 положителен).

(negnums '(-1 5 -6 0 2)) возвращает (-1 -2)

* Шаг 1. Завершение (Терминальная ветвь)
  (negnums nil) = nil
* Шаг 2. Рекурсивная ветвь
  Рекурсивные отношения между (negnums l) и (negnums (cdr l))
  o 1. (car l) < 0
  + 2a. Примеры рекурсии
    (negnums '(-5 3)) =(-5)
    (negnums '(3)) = nil
    (negnums '(-5 3 -6 0)) =(-5 -6)
    (negnums '(3 -6 0)) =(-6)
  + 2b. Характеристическое рекурсивное отношение (negnums l) может быть получена из (negnums (cdr l)) (cons (car l) (negnums (cdr l))
  o 2. (car l) >= 0
  + 2a. Примеры рекурсии

```
(negnums '(1 -5 3 -6 0 )) =(-5 -6)
(trace negnums) '(- 5 3 -6 0 )) = (-5 -6)
```

+ 2b. Характеристическое рекурсивное отношение (negnums l) может быть получена из (negnums (cdr l))

```
(negnums l) = (negnums (cdr l)
```

Программа

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

6.7 Общая форма

Общая форма определения рекурсивной функции

```
(defun <имя> <параметры>
(cond (терминальная ветвь1)
(терминальная ветвь2)
..................
(терминальная ветвьn)
(рекурсивная ветвь1)
(рекурсивная ветвь2)
...................
(рекурсивная ветвьn)))
```

7. Поиск на ЛИСПЕ. Функционалы. Свойства символов

7.1 Алгоритм поиска на Лиспе (функциональный подход к задаче о фермере, волке, козе и капусте)

Фермер (Farmer), волк (Wolf), козел (Goat) и капуста (Cabbidge) находятся на одном берегу. Надо перебраться на другой берег на лодке.

* Лодка перевозит только двоих.
* Нельзя оставлять на одном берегу козу и капусту, козу и волка.

Главная проблема в формировании алгоритма - найти эффективное представление структурой данных Лиспа информации о задаче.

Процесс перевозки может быть представлен последовательностью состояний. Состояние представляется списком из четырех элементов, каждый из которых отражает размещение объектов F, W, G, C:

(e w e w) - F, G на восточном берегу (e - east);

F W G C - W, C на западном берегу (w - west).

Определим две функции:

конструктор - make-state, которая берет в качестве аргументов размещение F, W, G, C и возвращает состояние

```
(defun make-state (f w g c) (list f w g c))
```

и четыре функции доступа, каждая из которых берет состояние и возвращает размещение.

```
(defun farmer-side (state) (nth 0 state))
(defun wolf-side (state) (nth 1 state))
(defun goat-side (state) (nth 2 state))
(defun cabbage-side (state) (nth 3 state))
```

Оставшаяся программа основывается на этих функциях доступа и конструкторах. В частности, они используются для реализации четырех возможных действий фермера:

- перевоз через реку или самого себя или W, G, C.

Эти функции используют четыре функции доступа для разбиения состояния на его компоненты.

Функция opposite (определена позже) определяет новое размещение объектов, которые пересекли реку, а make-state собирает их в новое состояние.

Например, функция farmer-takes-self может быть определена:

```
(defun farmer-take-self (state)
        (safe (make-state (opposite (farmer-side state))
        (wolf-side state)
        (goat-side state)
        (cabbage-side state))))
```

Отметим, что эта функция возвращает новое состояние, независимо от того, безопасно оно или нет. Однако могут быть опасные состояния: например, когда W, G или G, C находятся на одном берегу.

Программа должна найти в качестве решения только безопасные состояния. Проверка на опасные состояния должна производиться на разных стадиях программы. В нашем случае это можно сделать в функциях движения.

Реализуем это, используя функцию safe, которая имеет следующее поведение:

```
        (safe '(w w w w)) ; состояние безопасно, возвращается без изменений
                (w w w w)
```

Safe используется в каждой функции перемещения для фильтрования опасных состояний. Таким образом, любое перемещение, которое ведет к опасному состоянию будет возвращать nil вместо состояния. Алгоритм формирования пути может проверять этот nil и использовать его для избежания этого состояния.

Используя safe, будем иметь:

```
(defun farmer-take-wolf (state)
        (cond ((equal (farmer-side state) (wolf-side state))
        (safe (make-state (opposite (farmer-side state))
        (opposite(wolf-side state))
        (goat-side state)
        (cabbage-side state))))
        (t nil)))
```

Оставшиеся функции движения определяются аналогично, но включают условный тест для определения, находятся ли фермер и предполагаемый пассажир на одной и той же стороне реки. Если нет, то перемещение не может быть сделано, то есть пассажир и фермер не находятся на одном берегу, эта функция будет возвращать nil.

```
(defun farmer-take-goat (state)
        (cond ((equal (farmer-side state) (goat-side state))
        (safe (make-state (opposite (farmer-side state))
        (wolf-side state)
        (opposite(goat-side state))
        (cabbage-side state))))
        (t nil)))
(defun farmer-take-cabbage (state)
        (cond ((equal (farmer-side state) (cabbage-side state))
        (safe (make-state (opposite (farmer-side state))
        (wolf-side state)
        (goat-side state)
        (opposite(cabbage-side state)))))
        (t nil)))
```

Теперь можно определить функцию opposite, которая возвращает другую сторону.

```
(defun opposite (side)
        (cond ((equal side 'e) 'w)
```

((equal side 'w) 'e)))

Safe определена с использованием cond для проверки двух опасных состояний: F находится на противоположном берегу от W, G, и от G и C.

Если состояние безопасно, оно будет возвращено без изменений.

```
(defun safe (state)
        (cond ((and (equal (goat-side state) (wolf-side state))
        (not (equal (farmer-side state) (wolf-side state)))) nil)
        ((and (equal (goat-side state) (cabbage-side state))
        (not (equal (farmer-side state) (goat-side state)))) nil)
        (t state)))
(defun path (state goal)
        (cond ((equal state goal))
        (t (or (path (farmer-takes-self state) goal)))
        (path (farmer-take-wolf state) goal)
        (path (farmer-take-goat state) goal)
        (path (farmer-take-cabbage state) goal))
```

Эта версия функции path является простым переводом и содержит несколько ошибок, которые надо исправить. В частности, отметим использование формы OR для управления выполнением ее аргументов.

Напомним, что OR выполняет свои аргументы до тех пор, пока один из них не вернет не-nil величину. Когда это случается, OR завершается без выполнения других аргументов и возвращает это не-nil, как результат.

Таким образом, OR используется не только как логический оператор, но также обеспечивает способ управления поиском пути. OR используется здесь вместо cond, потому что величина, которая тестируется, и величина, которая возвращается в случае не-nil, одна и та же.

Одна проблема с этим определением заключается в том, что функция перемещения может вернуть значение nil, если перемещение не может быть сделано, когда оно ведет не к безопасному состоянию, чтобы предотвратить path от попытки генерировать дочерние состояния от состояния nil, она (path), должна сначала проверять, является ли текущее состояние nil, если да, то path должна вернуть nil.

Другая проблема, которая возникает в реализации path, заключается в возможности возникновения петель в пространстве состояний. Если данную реализацию path запустить, фермер будет ездить взад-вперед между двумя берегами бесконечно, то есть алгоритм приведет к бесконечным переходам между двумя одинаковыми состояниями.

Чтобы предотвратить это, в path надо ввести третий параметр, been-list, список всех состояний, которые уже были достигнуты. Каждый раз, когда path вызывается рекурсивно с новым дочерним состоянием, состояние-родитель должно быть добавлено в been-list. Вставляя в path предикат member, можно проверять, что текущее состояние не является элементом been-list, то есть здесь поиск уже побывал. Это выполняется проверкой текущего состояния, на присутствие в been-list перед генерацией его наследников.

Path теперь определяется:

```
(defun path (state goal been-list)
        (cond ((null state) nil)
        ((equal state goal) (reverse (cons state been-list)))
        ((not (member-lis state been-list ))
        (or (path (farmer-take-self state) goal (cons state been-list))
        (path (farmer-take-wolf state) goal (cons state been-list))
        (path (farmer-take-goat state) goal (cons state been-list))
        (path (farmer-take-cabbage state) goal (cons state been-list))))))
```

Функция member используется другая, так как проверяется принадлежность элемента списку списков member-equal.

```
(defun member-lis (x lis)
        (cond ((null lis) nil)
        ((equal x (car lis)) t)
        (t (member-lis x (cdr lis)))))
```

Вместо того, чтобы вернуть t, можно вернуть список состояний, которые были пройдены до достижения цели. Так как цель не содержится в списке, она может быть вставлена, как последний элемент.

Перед тем, как вернуть список, его надо перевернуть, используя reverse. Окончательно, чтобы параметр been-list скрыть от пользователя, может быть написана вызывающая функция, которая имеет два аргумента - начальное и конечное состояния и вызывает path с пустым списком been-list = nil.

```
(defun solve-fwgc (state goal) (path state goal nil))
```

Для решения можно использовать

```
(solve-fvgc '(w w w w) '(e e e e))
```

ответ:

```
1) w w w w    5) w e w w
2) e w e w    6) e e w e
3) w w e w    7) w e w e
4) e e e w    8) e e e e
```

7.2 Функционалы

До сих пор мы рассматривали функции, в качестве аргументов, которые использовали данные.

Например

(add1 3) добавляет к аргументу 1

4

```
(defun add1 (x) (+ 1 x))
```

Однако, в качестве аргумента функции можно указывать и функцию.

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

Различие между понятиями "данные" и "функция", определяются не на основе их структуры, а в зависимости от использования.

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

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

Важный класс функционалов, используемых в Лиспе – отображающие функционалы (MAP функционалы)

MAP функционалы - функции, которые некоторым образом отображают (map) список в новый список.

MAPCAR один из основных отображающих функционалов.

(MAPCAR f '(x1 x2 x3 ... xN))

MAPCAR функционал имеет два аргумента. Первый аргумент - функция, а второй - аргумент список.

Когда MAPCAR выполняется, функция определенная первым аргументом применяется к каждому элементу, списка, определенному вторым аргументом и результат помещает (отображает) в новый список.

Пример,

```
(mapcar 'add1 '( 1 2 3))
        (2 3 4)
(MAPCAR f '(x1 x2 x3 ... xN))
```

эквивалентно
(list (f 'x1) (f 'x2) .... (f 'xN))
Можно использовать в функциях
(defun list-add1 (lis) (mapcar 'add1 lis))
(list-add1 '(1 2 3))
(2 3 4)
В качестве аргумента для MAPCAR может быть использовано значение символа
(setq x '(a b (d)))
(setq y 'atom)
(mapcar y x)
(t t nil)
MAPCAR может обрабатывать больше списков, если в функциональном аргументе несколько аргументов.
Например
(defun addlist (l1 l2) (mapcar '+ l1 L2))
(addlist '(1 2 3) '(1 2 3))
(2 4 6)
т.е.
(list (+ 1 1) (+ 2 2) (+ 3 3))
Если списки разной длины, то длина результата будет равна длине наименьшего.

7.3 Лямбда-выражения

Структура MAP функций ограничивает формы отображаемых функций.
Так, если мы желаем получить список с элементами
x * x + 1
мы должны определить функцию
(defun f1 (x) (+ 1 (* x x)))
(mapcar 'f1 '(1 2 3))
Таким образом определяется специальная функция, которая используется только в MAPCAR.
Аналогично происходит с add1.
Более эффективно в этом случае использовать, т.н. лямбда0выражения:
(mapcar '(lambda (x) (+ 1 (* x x))) '(1 2 3))
сравни (defun f1 (x) (+ 1 (* x x)))
(mapcar '(lambda (x) (+ 1 x)) '(1 2 3))
Т.о., лямбда-выражения позволяют определять функцию внутри другой функции.
Лямбда-выражения определяют функцию, не имеющую имени.
Общая форма:
(lambda (параметры) <тело функции>)

7.4 Свойства символов

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

aqe	28
occupation	lawyer
salary	90
children	Bill Alice Susan
свойство	значение

Список свойств в этом случае выглядит

(aqe 28 occupation lawyer salary 90 children (Bill Alice Susan))

Узнать свойство атома можно используя функцию:

(GET <символ> <свойство>) возвращает значение

```
( get 'Mary 'age)
        28
( get 'Mary 'children)
        ( Bill Alice Susan))
( get 'Mary 'hobby)
        nil
```

Чтобы задать свойство, необходимо использовать обобщенную функцию присвоения setf

```
( setf ( get <символ> <свойство>) <значение>)
( setf ( get 'Mary 'salary) 90)
        90
```

Сначала свойство задается, а затем извлекается. Мы поступили наоборот, хотя в Лиспе присутствует функция putprop:

(putprop <символ> <значение> <свойство>)

<свойство> - нечисловой атом;

<значение> - любое выражение ;

Можно определить

```
(defun putprop ( atom value property)
(setf (get atom property) value))
```

и использовать при работе со списками.

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

При внесении нового свойства, оно помещается вначале списка свойств.

```
(putprop 'Mary 'cinema 'hobby)
        ( hobby cinema .....)
```

Замена значения свойства производится повторным присвоением.

Например,

```
(putprop 'mary 29 'age)
(get 'mary 'age)
```

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

```
(putprop 'mary (+ 1 (get 'mary 'age)) 'age)
```

Удаление свойства и его значения производится функцией

```
(remprop <символ> <свойство>)
(remprop 'Mary 'age)
        T
```

SYMBOL-PLIST даст информацию о списке свойств

```
( SYMBOL-PLIST 'Mary)
        (aqe 28 occupation lawyer salary 90 children ( Bill Alice Susan))
```

8. Внутреннее представление списков. Применяющие функционалы

8.1 Внутреннее представление списков

Каждый атом занимает ячейку. Списки, являются совокупностью атомов, и связываются вместе специальными элементами памяти, называемыми списочными ячейками или cons-ячейки.

Каждая списочная ячейка состоит из двух частей, полей. Первое поле - CAR, второе CDR.

Поля типа списочная ячейка содержат указатели на другие ячейки, или nil.

Список из одного атома, представляется, как (a.nil). NIL указывает на конец списка. Вместо nil пишут - \. Список получается как операция (cons 'a nil).

Каждому элементу списка соответствует списочная ячейка.

Любой список можно записать в точечной нотации.

(a) <=> (a.nil)

(a b c) <=> (a.(b.(c.nil)))

Выражение, представленное в точечной нотации можно привести к списочной, если cdr поле является списком.

(a.(b c)) <=> (a b c)

(a.(b.c)) <=> (a b.c)

Результатом действия функции car будет значение левого поля первой списочной ячейки

(car '(a (b c) d)

a

Результатом действия функции cdr будет значение правого поля первой списочной ячейки.

(cdr '(a (b c) d)

((b c) d)

CONS создает новую списочную ячейку, car поле которой указывает на первый элемент, а cdr на второй

(cons 'x '(a b c))

или

LIST

(list 'a '(b c)) (a (b c))

Получается следующим образом:

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

Рассмотрим выражение

(setq y '(a b c))

Переменная Y будет иметь значение '(a b c)

Использование переменной в функции обеспечивает доступ к структуре

(setq x (cons 'd y))

CONS не изменяя структуры, увеличивает список.

Если в функции присвоения список задается явно, то под него отводятся новые списочные ячейки. т.е.

(setq z '(a b c))

Переменная z будет иметь значение '(a b c)

EQ проверяет физическое равенство списков, и EQUAL – логическое, т.е. для EQ необходимо, чтобы списки имели одинаковую структуру, а для EQUAL одинаковые элементы. (Структура списка определяется списочными ячейками).

(setq lis1 '(a b))

(setq lis2 '(a b))

Другая структура списка

(setq lis3 lis1)

(equal lis1 lis2)

t

(equal lis1 lis3)

t

(eql lis1 lis2)

nil

(eql lis1 lis3)

```
t
(eq lis1 lis2)
    nil
(eq lis1 lis3)
    t
```

Однако для отдельного атома это не выполняется, т.е. новые ячейки не отводятся.

```
(setq m 'abc)
(setq n 'abc)
(eq m n)
    t
```

В результате вычислений в памяти могут возникнуть структуры, на которые нельзя ссылаться. Это происходит, когда вычисляемая структура не сохраняется с помощью setq, или когда теряются ссылки на старое значение.

Например

```
(setq l1 '((a) b c))
(setq l1 (cdr l1))
```

Для повторного использования ставшей мусором памяти в Лисп-системах предусмотрен специальный сборщик мусора, (garbage collector) GC, который автоматически запускается когда в памяти остается мало места.

Сборщик мусора перебирает все ячейки и собирает ставшие мусором ячейки в область свободной памяти для использования.

8.2 Обработка списков без разрушения. Append

Обычные функции производят операции над списками, не внося изменений в указатели списочных ячеек. В том случае когда, возникает такая необходимость, создается копия списочной ячейки с новым содержанием полей.

Пример, для функции APPEND рассмотрим следующую последовательность действий.

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

APPEND создает копии всех списочных ячеек для каждого элемента во всех аргументов, исключая последний аргумент, в то время как cons создает только одну списочную ячейку.

Если складывают два списка в 1000 и 1 элемент, то будет создано 1000 копий списочных ячеек, вместо того чтобы исправить один указатель.

8.3 Разрушающие функции

В Лиспе есть несколько функций, которые изменяют содержимое указателей, вместо того чтобы создавать новые списочные ячейки. Эти функции называют разрушающими , т.к. указатель заменяется и исходная структура разрушается.

Функция nconc позволяет соединить два списка изменением указателя.

```
(setq new (nconc first second))
```

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

Как побочный эффект

```
both
```

(a b c d)

first

(a b c d) – список first разрушен.

Другие две функции изменяют структуру своих аргументов.

rplaca -"replace the car" имеет два аргумента , причем первый должен быть списком. Функция заменяет car первого аргумента на второй. Т.е. указатель car первой списочной ячейки меняется на указатель ко второму аргументу.

Аналогично

rplacd - "replace the cdr", но заменяется cdr указатель.

Рассмотрим следующую последовательность действий:

(setq lis1 '(a b c))

(setq lis2 '(a b c))

Проведем следующие действия

(rplaca lis1 'd)

(d b c)

(rplacd lis2 '(e f))

(a e f)

Можно представить через setf

(rplaca x y) <=> (setf (car x) y)

(rplacd x y) <=> (setf (cdr x) y)

Можно использовать для замены элементов.

Например, рассмотрим функцию, replace-item, которая имеет три элемента: первый элемент должен быть списком. Функция разрушающе замещает первое местоположение второго аргумента в списке на третий аргумент.

(defun replace-iteme (lis old new)

(rplaca (member old lis) new).

Разрушающие функции необходимо использовать при работе с большими списками, чтобы не увеличивать расход памяти, например использовать nconc вместо append.

Однако использование разрушающих функций приводит к побочным эффектам.

Можно получить бесконечные списки:

(setq v1 '(a b c))

(a b c)

(setq v2 v1)

(a b c)

(setq v2 (nconc v1 v2))

(a b c a b c....)

Поэтому использование разрушающих функций требует осторожности.

8.4 Применяющие функционалы

Одним из видов функционалов, используемых в Лиспе, являются применяющие функционалы. Они применяют функциональный аргумент к его параметрам.

Так как применяющие функционалы вычисляют значение функции, в этом смысле они аналогичны функции EVAL, вычисляющей значение выражения.

Предположим, мы хотим объединить в один список несколько вложенных списков, т.е. из ((a b c) (d e f) (k l)) получить (a b c d e f k l).

Для этой задачи используем функцию apply. APPLY имеет два аргумента: имя функции и список, и применяет названную функцию к элементам списка, как к аргументам функции.

Определим функцию, которая рассчитывает среднее списка чисел

(defun list-mean (lis)

(/ (apply '+ '(lis) (length lis)))

```
(list-mean '(1 2 3 4))
        2.5
```

Часто apply используют вместе с mapcar.

Мы хотим найти общее число элемент в списках

```
(countall '((a b c) (d e f) (k l)))
(defun countall (lis)
(apply '+ (mapcar 'length lis)))
```

Можно определить более сложную функцию countatom, которая считает элементы на в любом списке.

Например

```
(countatom '(a (a (b) c) (d) e (f g)))
        8
(defun countatom (lis)
(cond ((null lis) 0)
((atom lis) 1)
(t (apply '+ (mapcar 'countatom lis)))))
```

Построить функцию list-last, образующую список из хвостов списков.

Например

(list-last '((a b) (b c) (c d))) возвращает (b c d)

```
(defun list-last (lis)
(apply 'append (mapcar 'last lis)))
```

APPEND работает медленно и оставляет много мусора.

Можно это сделать через nconc:

```
(defun list-last (lis)
(apply 'nconc (mapcar 'last lis)))
```

В Лиспе имеется отображающий функционал mapcan

```
(mapcan fn x1 x2 ... xN) <=>
(apply 'nconc (mapcar fn x1 x2 ... xN))
```

Т.е. mapcan объединяет результаты в один список, используя функцию nconc.

```
(defun list-last (lis)
(mapcan 'last lis))
```

Применяющий функционал FUNCALL аналогичен APPLY, но аргументы он принимает, не в списке, а по отдельности:

```
(funcall fn x1 x2 ... xN) <=> (fn x1 x2 ... xN)
```

Здесь fn - функция с n аргументами.

Например,

```
(funcall '+ 1 2) <=> * (+ 1 2)
        3
(funcall (car '(+ - / *)) 1 2)
        3
```

Рассмотрим использование funcall для построения функции map2, которая действует аналогично mapcar, но берет в качестве аргументов два элемента из списка, а не один.

Например:

(map2 'list '(A Christie V Nabokov K Vonnegut))

дает ((A Christie) (V Nabokov) (K Vonnegut))

Эта функция имеет вид:

```
 (defun map2 (f2 lst)
(if (null lst)
nil
(cons (funcall f2 (car lst) (cadr lst))
(map2 f2 (cddr lst)))))
```

9. Массивы. Макросы. Пример программы на Лиспе – Дифференцирование выражений

9.1 Массивы

Для хранения большого количества данных в Лиспе используются массивы.

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

Для определения массива заданной размерности используется функция make-array (make-array <размерность>) поддерживаются только одномерные массивы-векторы.

Пример:

```
(setq data (make-array 10))
# (0 0 0 0 0 0 0 0 0 0)
```

data - имя массива,

0 - начальное наполнение.

Доступ производится с помощью функции aref. AREF имеет два аргумента - имя массива и индекс и возвращает значение ячейки

```
(aref <имя> <индекс>)
(aref data 8)
        0
```

так как там записан 0.

Особенности: первый аргумент не блокируется, первая ячейка имеет номер 0.

Поместить данные в массив можно ,используя функцию setf

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

aref – вызывает значение ячейки, функция setf помещает значение.

Рассмотрим массив testdata

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

Можно (setq testdata (vector 18 'dog '(a b) 0)) (aref d 1)

В результате получим

```
        testdata 0 1 2 3
        18 dog (a b) 0
```

Можно использовать эти данные

```
(cons (aref testdata 1) (list (aref testdata 3)
(aref testdata 2)))
(dog 0 ( a b))
(aref testdata (aref testdata 3))
        18
```

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

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

```
(defun array-list (arnam len)
(do (( i 0 ( + 1 i))
( result nil (append result (list (aref arnam i)))))
(( equal i len) result)))
(array-list testdata 4)
( 18 dog (a b) 0)
```

(array-length <имя>) возвращает длину массива.

(array-length testdata) возвращает длину массива 4.

9.2 Обратная блокировка

Обычная блокировка не позволяет вычислять выражения
'(a b c)
(a b c)
Иногда возникает необходимость вычислить только часть выражения.
Если
(setq b '(x y z))
и мы запишем
`(a ,b c) ,то b - будет вычислено и мы получим
(a (x y z) c)
` - обратная верхняя запятая обозначает блокировку, которая может частично сниматься запятой.
Обратная блокировка может использоваться при печати:
(setq n 'john)
(setq m 'Robert)
(print `(boys ,n and ,m))
(boys john and roberts)
Наиболее часто обратная блокировка используется в мощном средстве Лиспа – макросах.

9.3 Макросы

Это специальное средство, позволяющее формировать вычисляемое выражение в ходе выполнения программы.
Рассмотрим например функцию blancs, которая производит n переводов каретки (пропускает n линий)
(defun blancs (n)
(do ((count n (- count 1)))
((zerop count) nil)
(terpri)))
(blancs 4) пропустит четыре строки.
Будем писать программу, которая позволит выполнять некоторое действие n раз
Для blancs:
(do-times '(terpri) 4)
или
(do-times '(print 'hello) 3)
Напечатает hello три раза
Можно через eval определить
(defun do-times (operation n)
(do ((count n (- count 1)))
((zerop count) nil)
(eval operation)))
Это можно сделать также через специальную форму : defmacro
(defmacro do-times (operation n)
` (do ((count ,n (- count 1)))
((zerop count) nil)
,operation))
Как видно, форма макро похожа на определение функции, но есть отличия:
1. Аргументы макро не вычисляются перед их использованием.

Тогда обращение записывается:

(do-times (print 'hello) 3)

2. Когда макро вызывается, Лисп сначала вычисляет тело макро, а затем выполняет получившееся выражение.

Например,после обращения

(do-times (print 'hello) 3)

Получим

(do ((count 3 (- count 1)))
((zerop count) nil)
(print 'hello))

Этот список потом вычисляется.

Таким образом при вызове макро сначала вычисляется тело (этап называется расширением) и формируется выражение.

На втором этапе вычисляется полученное выражение, и полученное значение возвращается как результат.

Вызывая macro с разными аргументами получим разные результаты. Если мы вызываем:

(do-times (print count) 10)

После вычисления тела получим:

(do ((count 10 (- count 1)))
((zerop count) nil)
(print count))

Печатается числа от 10 до 1.

Можно определить функцию обратной печати чисел

(defun print-number (n)
(do-times (print count) n))
(print-number 5)

При разработке макро необходимо выполнить три шага:

1. Написать пример функции, которую должна формировать макро,

2. Выделить общие части для нескольких функций и переменные. Переменные части обозначить переменными, выделить запятыми и вынести в аргументы. Постоянные части записать напрямую.

3. Определить макро, которое реализует этот вызов.

Пример. Надо определить макро term-search, которая будет просматривать список и выделять первый элемент удовлетворяющий заданному условию.

Шаг1. Сформулируем пример. Запишем тело для поиска чисел:

```
   (setq l '(s d 3 d))    (setq item 5)
     (do (( tail  l (cdr tail)))
          ((null tail) nil)
          ( cond ((numberp (car tail)) (return (car tail)))))
```

Шаг2. Выделяем общие части список lis - l и предикат predicate - number.

Шаг3.Формируем макро

```
      (defmacro term-search ( predicate lis)
      ` (do (( tail , lis (cdr tail)))
           ((null tail) nil)
  (cond ((,predicate (car tail)) (return (car tail))))))
```

9.4 Пример программы на Лиспе – Дифференцирование выражений

Напишем программу дифференцирования алгебраических выражений. Для наглядности ограничимся алгебраическими выражениями в следующей форме:

(+ x y) (* x y)

Сложение и умножение можно свободно комбинировать.

Например,

(*(+ a (* a b)) c) Программируя непосредственно получаем

```
(defun diff0 ( l x)
   (cond (( atom l)
           (if (eq l x) 1  ;l=1
                0));l=константа
        (( eq (first l) '+)
          (list '+
                (diff0  (second l) x)
                (diff0  (third  l) x)))
        (( eq (first l) '*)
          (list '+
            (list '*
                (diff0  (second l) x)
                (third  l))
            (list '*
                (diff0  (third l) x)
                (second  l))))
        (t l)))
```

Используем

```
 (diff0 '(+ x (* 3 x)) 'x)
        ( + 1 (+ (* 0 x) (* 1 3))) = 4
 (diff0 '(- x (* 3 x)) 'x)
        (- x (* 3 x))  А почему?
 (diff0 '(* x ( + x 1)) 'x)
        (+ (* 1 ( x 1))(*(1 0) x))
```

Вычисляемые выражения не упрощаются, хотя это не трудно сделать.

Эта программа неудобна, так как трудно расширять, приходится все группировать в один cond. Она не является модульной.

Мы получим более удобное решение, если для каждого действия определим свою дифференцирующую функцию и через свойство diff свяжем эту функцию с символом, обозначающим действие.

Упростим запись самой дифференцирующей функции:

```
(defun dif1 (l x)
      (cond (( atom l)
             (if (eq l x) 1 0))
            ( t (funcall (get (first  l) 'diff) (cdr l) x))))
```

Функции дифференцирования становятся значениями свойства 'diff:

```
(setf (get '+ 'diff) 'dif+) (setf (get '* 'diff) 'dif*)
```

Таким образом извлекаем действие из данных. Сами функции записываются:

```
 (defun dif* (l x)
      (list '+ (list '* (dif1 (first l) x)
                    (second l))
               (list '* (dif1 (second l) x)
                    (first l))))
 (defun dif+ (l x)
      (list '+ (dif1 (first l) x) (dif1 (second l) x)))
```

Благодаря модульности можно дополнить для

```
(defun dif- (l x)
      (list '- (dif1 (first l) x) (dif1 second l) x)))
```

Таким образом, первоначальное управление вычислительным процессом, связанное

со структурой программы, мы преобразовали в динамическое управление основанное на данных.

Можно использовать макросы. Определим макрос defdif, с помощью которого определяются дифференцирующие функции для новых действий:

```
 (defmacro defdif (act args body)
       `(setf (get ',act 'diff)
       '(lambda,args,body)))
```

Тогда дифф. функции задаются:

```
(defdif + (l x)
       (list '+ (dif1 (first l) x) (dif1 (second l) x)))
(defdif * (l x)
       (list '+ (list '* (dif1 (first l) x)
                       (second l))
                (list '* (dif1 (second l) x)
                       (first l))))
(dif1 '(+ x x) 'x)
(defdif - (l x)
       (list  '- (dif1 (first l) x)
                (dif1 (second l) x)))
  (dif1 '(+ x (* 3 x)) 'x)
 (dif1 '(- x (* 3 x)) 'x)
(dif1 '(* x ( - x 1)) 'x)
```

Дополним программу несколькими функциями, обеспечивающими ввод информации:

Чтение дифф. списка

```
(defun read-list () (princ " diff-list ? ") (setq l (read)))
```

Пусть продолжает вычисления до тех пор пока не будет введено не d.

```
(defun d ()  (princ "enter command : d -diff;q - quit") (terpri)
         (if (eq  (read) 'd)   (progn (read-list)
                              (print  (dif1 l 'x ))
                              (terpri) (d))
                              'end))
```

Вызов программы теперь (d)

Можно сразу загружать программу и начать ее выполнение, для этого используют функцию load

(load <файл>)

10. Параметры функций. Чтение и запись информации в файлы

10.1 Задание параметров при определении функций

При определении функции можно использовать механизм ключевых слов для того чтобы при вызове функций аргументы трактовать по разному.

С помощью ключевых слов в лямбда-списке можно выделить

* необязательные аргументы (optional)
* параметр, связываемый с хвостом списка аргументов изменяющейся длины (rest)
* ключевые параметры (key)

Ключевые слова начинаются с символа & и их записывают перед соответствующими параметрами в лямбда-списке.

Действие ключевого слова распространяется до следующего ключевого слова.

Параметры, перечисленные в лямбда-списке до первого ключевого слова, являются обязательными.

Вы можете определить необязательные аргументы для вашей функции. Любой

аргумент после символа &optional необязательный:

```
( Defun bar ( x &optional y) ( if y x 0 ))
        bar
( Defun baaz ( &optional ( x 3 ) ( z 10 )) ( + x z) )
        BAAZ
( bar 5 )
        0
( bar 5 t)
        5
( Baaz 5 )
        15
( Baaz 5 6 )
        11
(Baaz)
        13
```

Можно вызывать функцию bar или с одним или с двумя аргументами. Если она вызвана с одним аргументом, x будет связано со значением этого аргумента и незаданный аргумент y будет связан с nil; если она вызвана с двумя аргументами, x и y будут связаны со значениями первого и второго аргумента, соответственно.

Функция baaz имеет два необязательных аргумента. Кроме этого она определяет недостающие значения для каждого из них: если пользователь определит только один аргумент, z будет связано с 10 вместо nil, и если пользователь не определит никаких аргументов, x будет связана с 3 и z с 10.

Такое определение значений называется определение по умолчанию.

Вы можете задавать вашу функцию принимающей любое число аргументов, заканчивая список аргументов &rest параметром.

LISP будет собирать все аргументы не попавшие в обязательные параметры в список и связывать &rest параметр с этим списком.

Итак:

```
( Defun foo ( x &rest y) y)
        FOO
( Foo 3 )
        NIL
( Foo 4 5 6 )
        (5 6)
( defun fn ( x &optional y &rest z))
    (list x y z))
        fn
(fn 'a)
        (A NIL NIL)
( a b (c d))
```

Можно задать функции другой вид необязательного аргумента называемого аргументом ключевого слова. Пользователь может задавать эти аргументы в последующем в любом порядке, потому что они маркированы ключевыми словами.

Символы t и nil называются константами-символами, потому что они при выполнении дают сами себя.

Существует целый класс таких символов, которые называются ключевыми словами; любой символ, чье имя начинается с двоеточия является ключевым словом.

(Ниже приведены некоторые использования ключевых слов).

Примеры:

```
:this-is-a-keyword
        :THIS-IS-A-KEYWORD
```

```
:so-is-this
        :SO-IS-THIS
:me-too
        :ME-TOO
( Defun foo ( &key x y) ( cons x y) )
        FOO
( Foo :x 5 :y 3 )
        (5 . 3)
( Foo :y 3 :x 5 )
        (5 . 3)
( Foo :y 3 )
        ( NIL. 3 )
(Foo)
        (NIL)
```

&key параметр может иметь также значение по умолчанию:

```
( Defun foo ( &key ( x 5 )) x )
        FOO
( Foo :x 7 )
        7
(Foo)
        5
(defun test ( x &optional (y 3) (z 4) &rest  a)
(cons z ( list x a y)))
(test 1 2 3)
(test 1)
(test 3 4 5)
(test 3 2 1 1 2 3)
```

10.2 Входные и выходные потоки

При вводе и выводе информации в Лиспе используется понятие потоков - stream

Для потока определены имя, операции открытия open операции закрытия close, направления output и input

10.3 Определение выходных и входных потоков

Для открытия файла для записи задается его имя, производится операция open и указывается направление output:

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

Зададим

(setq s 'e)

Можно вывести это значение в файл

(princ s our-output-stream) ;

Можно занести список

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

Чтобы правильно закрыть поток, необходимо в конец поместить

(terpri our-output-stream)

Затем файл закрывается

(close our-output-stream)

Можно посмотреть информацию в файле.

Откроем файл для чтения:

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

Прочитае информацию
(read our-input-stream)
Закроем файл
(close our-input-stream)

10.4 Чтение символов из файла

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

Сформируем файл
Пусть
(setq s "---+++")
(setq p "+++---")
Определим поток вывода
(setq our-output-stream (open "picture.spl" :direction :output))
(princ s our-output-stream) ; записываем первую стороку
(terpri our-output-stream) ; заканчиваем ее
(princ p our-output-stream) ; записываем вторую строку
(terpri our-output-stream) ; заканчиваем файл
Тепрь файл закрывается
close our-output-stream)
В файле теперь находится
---+++
+++---
Для чтения символов из файла будем использовать функцию
(READ-CHAR <входной поток>)

Данная функция позволяет читать печатные символы (CHAR) из файла. В качестве значения получается десятичное представление кода символа.

Используем эту функцию для посимвольного ввода информации из файла для ее последующего анализа.

Определим
(setq our-input-stream (open "picture.spl" :direction :input))
Для чтения символа используем
(read-char our-input-stream)
Будем получать последовательность значений
43
43
43
45
45
45
10 и т.д.
Для восстановления содежимого файла применяется перекодировка
(setq x (read-char our-input-stream)
Содержимое x можно показать

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

Можно представить информацию без искажений, если использовать цикл

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

```
(( = x 45) (prin1 '-))
 (( = x 10 ) (terpri))))))
```

После вывода имеем

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

Закрытие входного потока

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

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

Задания:

1. Напишите рекурсивную функцию для вычисления произвольной целой степени числа.
2. Определите функцию last, возвращающую последний элемент списка.
3. Определите функцию удаляющую последний элемент списка.
4. Определите предикат, проверяющий, является ли аргумент одноуровневым списком.
5. Определите функцию, возвращающую первый атом списка.
6. Определите функцию, вычисляющую количество атомов в списке.
7. Создайте генератор последовательностей вида (а), (б а), (а б а), (б а б а), ...
8. Создайте программу чтения произвольного текстового файла.
9. Изучите графические возможности данной реализации Лиспа и создайте функции для рисования точек, линий, прямоугольников, окружностей, эллипсов.

Лабораторная работа №3.

Создание интерпретатора Лиспа на Лиспе

1. Интерпретатор Лиспа на Лиспе

Интерпретатор Лиспа сравнительно легко запрограммировать на самом Лиспе. Но определение какого-нибудь языка на нем самом – это не такая уж очевидная задала. Хорошим пояснением к этому будет история зарождения подобной идеи.

Маккарти рассказывает, что во время его размышлений о семантике лисповской функции EVAL Стивен Рассел предложил запрограммировать се тем же способом, что и другие функции Лиспа:

«*... Этот EVAL был написан и опубликован в нашей статье, когда Стив Рассел сказал: "Послушайте, почему мне не запрограммировать этот EVAL и вы получите интерпретатор", на что я сказал ему: "Ну, ну, вы путаете теорию с практикой, наш EVAL предназначен для чтения, а не для вычислений." Но он на этом не остановился и сделал интерпретатор. Таким образом, С. Рассел оттранслировал EVAL из моей статьи в машинный код 704, отладил его и объявил результат в качестве интерпретатора Лиспа, чем он, конечно, и являлся...*»

После того как идея была подана, ее осуществление не вызвало уже особых затруднений, так как в Лиспе данные и программы представляются одинаково. Используемые при этом основные средства – это блокировка вычислений, при помощи которой можно при необходимости запретить вычисления; базовые функции; условные операторы и механизмы рекурсии и определения функции.

Рассмотрим ниже простой интерпретатор Лиспа, запрограммированный на нем самом. Он в некоторой степени отличается как от первоисточника, так и от интерпретаторов реальных систем. Тем не менее он дает довольно ясное представление о ядре интерпретатора и его работе, а также показывает, как легко и изящно семантика Лиспа поддается определению и программированию и на других языках.

При необходимости интерпретатор можно расширять, добавляя в систему новые ветви и определения функций. Используя ту же технику, можно также реализовать новые языки программирования высокого уровня для различных приложений. В реализации интерпретатора будем использовать чисто функциональное программирование.

2. Примитивы интерпретатора

Пусть наш интерпретатор называется EVAL1 в отличие от интерпретирующего его системного интерпретатора EVAL. Соответственно назовем и интерпретируемые нами базовые функции: CAR1, CDR1, CONS1, EQUAL1 и ATOM1. Кроме того, будем использовать соответствующее предложению LAMBDA предложение LAMBDA1 и формы COND1 и QUOTE1. T1, NIL1 и числа будут константами системы.

Во время вычисления предполагается, что вызываемые пользователем функции определены предложением LAMBDA1, которое является свойством FN имени функции.

3. Универсальная функция EVAL1

Предположим, что связи переменных хранятся в ассоциативном списке СВЯЗИ. Вычисление форм происходит путем интерпретации их структур и выполнения соответствующих действий. Универсальную функцию EVAL1 можно определить приведенным ниже условным предложением:

```
(defun eval1 (форма &optional (св_зи nil))
  (cond
    ((atom форма) ; форма атомарна_
      (cond
```

```
        ((eq форма 't1) 't1)
        ((eq форма 'nil1) 'nil1)
        ((numberp форма) форма)
        ((cadr (assoc форма св_зи)))
        (t (format t "~%У атома нет св_зи: ~S" форма))
      );!!
    );!!
    ;; голова формы - атом
    ((atom (car форма))
      (cond
        (  (eq (car форма) 'quote1) (cadr форма)  )
        (  (eq (car форма) 'cond1) (eval-cond (cdr форма) св_зи)  )
        ;; вызов функции
        (  (get (car форма) 'fn)
             (apply1
                (get (car форма) 'fn)
                (eval-список (cdr форма) св_зи)
                 св_зи
             )
        )
        ;; встроенна_ функци_
        (t (apply1
              (car форма)
              (eval-список (cdr форма) св_зи)
               св_зи
           )
        )
      )
    )
    ;; LAMBDA1-предложение
    (t (apply1
          (car форма)
          (eval-список (cdr форма) св_зи)
          св_зи
       )
    )
  )
)
```

Из определения видно, что если ФОРМА – просто переменная (т.е. не константа T1, NIL1 или число), то ее значение получают с помощью функции ASSOC из представленного в форме ассоциативного списка вычислительного контекста СВЯЗИ.

Функция EVAL-COND определяет семантику условного оператора COND1:

```
(defun eval-cond (ветви контекст)
  (cond
    (  (null ветви) 'nil1  )
    (
      (not (eq (eval1 (caar ветви) контекст) 'nil1))
      (eval1 (cadar ветви) контекст)
    )
    (t (eval-cond (cdr ветви) контекст))
  )
)
```

Если форма – это список, первый элемент которого – имя функции, то выражение можно вычислить, получив LAMBDA1-предложение, соответствующее имени функции, как свойство под именем FN с помощью функции GET н применив его при помощи функции APPLY1. Форма может быть непосредственно LAMBDA1-вызовом с фактическими параметрами:

((**LAMBDA1** *фор-параметры тело*) *факт-параметры*)

Эту ситуацию обрабатывает последняя ветвь EVAL1.

4. Основная часть интерпретатора: APPLY1

Функция APPLY1 отвечает за применение функции к значениям своих аргументов. С ее помощью определены также базовые функции интерпретатора: CAR1, CDR1, CONS1, ATOM1, EQUAL1 и LAMBDA1-предложение:

```
(defun apply1 (функци_ аргументы св_зи)
  (cond
    (
      (atom функци_) ; базовые функции
      (cond
        (
          (eq функци_ 'car1)
          (cond
            ((eq (car аргументы) 'nil1) 'nil1)
            ;; Голова NIL1 - это NIL1
            (t (caar аргументы))
          )
        )
        (
          (eq функци_ 'cdr1)
          (cond
            ((eq (car аргументы) 'nil1) 'nil1)
            ;; Хвост NIL1 - это NIL1
            ((null (cdar аргументы)) 'nil1)
            (t (cdar аргументы))
          )
        )
        (
          (eq функци_ 'cons1)
          (cond
            (
              (eq (cadr аргументы) 'nil1)
              (list (car аргументы))
            )
            (t (cons (car аргументы) (cadr аргументы)))
          )
        )
        (
          (eq функци_ 'atom1)
          (cond
            (  (atom (car аргументы)) 't1)
            (t 'nil1)
          )
        )
        (
          (eq функци_ 'equal1)
          (cond
            ( (equal (car аргументы) (cadr аргументы))  't1)
            (t 'nil1)
          )
        )
        (t (format t "~%Неизвестна_ функци_: ~S" функци_))
      )
    )
    (
      (eq (car функци_) 'lambda1)
      ;; л_мбда-вызов
      (eval1 (caddr функци_)
             (создай-св_зи (cadr функци_) аргументы св_зи)
      )
    )
    (t (format t "~%Это не л_мбда-вызов: ~S" функци_))
  )
)
```

Функция ФУНКЦИЯ должна быть либо известной транслятору базовой функцией, либо более сложным вычислением, определенным через выражение LAMBDA1. Функцией СОЗДАЙ-СВЯЗИ в окружение добавляются связи формальных параметров с фактическими:

```
;;; Окружение новыми парами им_-значение
(defun создай-св_зи (формальные фактические окружение)
  (cond
    ((null формальные) окружение)
    (t (acons
          (car формальные)
          (car фактические)
          (создай-св_зи
             (cdr формальные)
             (cdr фактические)
             окружение
          )
       )
    )
  )
)


(defun acons (x y a)
   (cons (cons x (cons y nil)) a)
)
```

Вторым аргументом APPLY1 – АРГУМЕНТЫ – является список значений аргументов вызываемой функции, который вычисляется функцией EVAL-СПИСОК.

```
(defun eval-список (невычисленные св_зи)
  (cond
    ((null невычисленные) nil)
    (t (cons
          (eval1 (car невычисленные) св_зи)
          (eval-список (cdr невычисленные) св_зи)
        )
    )
  )
)
```

Заметим, что интерпретатор Лиспа образуется в основном двумя взаимно-рекурсивными функциями EVAL1 и APPLY1, через которые семантика языка получила формальное определение. Первоначально Маккарти определил интерпретатор в метанотации Лиспа, при использовании которой элегантность определения языка особенно заметна:

```
1 evalquole[fn;x] = apply[fn;x;NIL]

2 apply[fn;x;a] = [atom[fn] -> [eq[fn,CAR]  —> caar[x];
3                               eq[fn;CDR]  -> cdar[x];
4                               eq[fn;CONS] -> cons[car{x];cadr[x]];
5                               eq[fn;ATOM] -> atom[car[x]];
6                               eq[fn,EQ]   -> eq[car[x],cadr{x]];
7                               T           -> apply[eval[fn;a];x;a]];
8       eq[car[fn];LAMBDA] -> eval[caddr[fn];pairlis[cadr[fn];x;a]];
9       eq[car[fn];LABEL]  ->
                  apply[caddr[fn];x;cons[cons[cadr[fn];caddr[fn]];a]];
10      eq[car[fn];FUNARG] -> apply[cadr[fn];x;caddr[fn]]]

11 eval[e;a]=[atom[e] -> cdr[assoc[e;a]];
12            atom[car[e]] -> [eq[car[e];QUOTE] -> cadr[e];
13                             eq[car[e];COND] -> evcon[cdr[e];a];
14                             eq[car[e];FUNCTION] ->
                                         list[FUNARG;cadr[e];a];
15                             T -> apply[car[e];evlis[cdr[e];a;a]];
16            T -> apply[car[e],evlis[cdr[e];a];a]];

17 pairlis[x;y;a]=[null[x] -> a;
```

```
                   T -> cons[cons[car[x];car[y]];pairlis[cdr[x];cdr[y];a]]]

18 assoc[x;a]=[eq[caar[a];x] -> car[a];
               T             -> assoc[x;cdr[a]]]

19 evcon[c;a]=[eval[caar[c];a] -> eval[cadar[c],a];
               T               -> evcon[cdr[c];a]]

20 evlis[m;a]=[null[m] -> NIL;
               T       -> cons[eval[car[m];a];evlis[cdr[m];a]]]
```

5. Примеры вычислений

Теперь можно опробовать интерпретатор:

```
> (eval1 't) ; T не является константой нашего интерпретатора

У атома нет св_зи: t
nil
> (eval1 'nil1)
nil1
> (eval1 '(конс quote1 a) (quote1 (b c))))

Неизвестна_ функци_: конс
nil
> (setf (get 'конс 'fn)  ; определение функции
      '(lambda1 (голова хвост)
          (cons1 голова хвост)))
(lambda1 (голова хвост) (cons1 голова хвост))

> (trace eval1 apply1)  ; включить трассировку
(apply1 eval1)
> (eval1 '(конс (quote1 a) (quote1 b c))))
  Entering: EVAL1, Argument list: ((конс (quote1 a) (quote1 (b c))))
   Entering: EVAL1, Argument list: ((quote1 a) nil)
   Exiting: EVAL1, Value: a
   Entering: EVAL1, Argument list: ((quote1 (b c)) nil)
   Exiting: EVAL1, Value: (b c)
   Entering: APPLY1, Argument list: ((lambda1 (голова хвост) (cons1 голова хвост
)) (a (b c)) nil)
    Entering: EVAL1, Argument list: ((cons1 голова хвост) ((голова a) (хвост (b
c))))
     Entering: EVAL1, Argument list: (голова ((голова a) (хвост (b c))))
     Exiting: EVAL1, Value: a
     Entering: EVAL1, Argument list: (хвост ((голова a) (хвост (b c))))
     Exiting: EVAL1, Value: (b c)
     Entering: APPLY1, Argument list: (cons1 (a (b c)) ((голова a) (хвост (b c))
))
     Exiting: APPLY1, Value: (a b c)
    Exiting: EVAL1, Value: (a b c)
   Exiting: APPLY1, Value: (a b c)
  Exiting: EVAL1, Value: (a b c)
(a b c)
> (setf (get 'member1 'fn)  ; другое определение
    '(lambda1 (элемент список)
       (cond1
         ((equal1 список nil1) nil1)
         ((equal1 элемент (car1 список)) список)
         (t1 (member1 элемент (cdr1 список))))))
(lambda1 (элемент список) (cond1 ((equal1 список nil1) nil1) ((equal1 элемент (c
ar1 список)) список) (t1 (member1 элемент (cdr1 список)))))
> (untrace eval1 apply1)  ; снятие трассировки
nil
> (eval1 '(member1 (quote1 b) (quote1 (a b c))))
```

```
(b c)
> (eval1 '(member1 (quote1 d) (quote1 (a b c))))
nil1
```

6. Печать результатов – структурная печать

В Лисп-систему в ее редактор в качестве вспомогательного средства обычно входит *структурная печать* (prettyprinter). С ее помощью функцию или произвольный список можно вывести красивой форме, с определенными логикой отступами и проверить, правильно ли расставлены скобки. Однако можно отказаться от предусмотренной структурной печатью формы вывода, например, при работе с реализованным на Лиспе языком или способом представления данных более высокого уровня. В этом случае придется заново программировать вывод.

В качестве примера формы в мы далее запрограммируем небольшую функцию структурной печати PPRINT1 (в Коммой Лиспе эта функция называется PPRINT). Структурная печать рекурсивно выводит списки так, что одно- и двухэлементные списки печатаются в том как они есть, а более длинные преобразуются к удобной для чтения форме:

```
(defun pprint1 (l &optional (отступ 0))
  (cond
    ((atom l)           ; Вывод атома
      (установи отступ)
      (prin1 l)
    )
    ((< (length l) 3)   ; Короткий список просто выводится
      (установи отступ)
      (prin1 l)
    )
    (t
      (terpri)          ; Длинный список преобразуется
      (установи отступ)
      (princ "(")
      (prin1 (car l))
      (pp-хвост (cdr l) отступ)
    )
  )
)

;;; Хвост выводится с отступом
(defun pp-хвост (l отступ)
  (cond
    ((null l) (princ ")") t)
    (t
      (terpri)
      (установи отступ)
      (pprint1 (car l) (+ отступ 3))
      (pp-хвост (cdr l) отступ)
    )
  )
)

;; выводит n пробелов
(defun установи (n)
  (cond
    ((= n 0) t)
    (t
      (princ " ")
      (установи (- n 1))
    )
  )
)
```

Пример использования:

```
> (pprint1 '(list (car x) (cons (car y) (cdr z))))
 (list
   (car x)
   (cons
         (car y)
         (cdr z)))
t
> (pprint1 '(cond ((null x) nil) (t (cons (car x) (cdr y)))))
 (cond
   ((null x) nil)
   (t (cons (car x) (cdr y))))
t
```

7. Программирование диалога

С помощью функции PPRINT1, READ и PRINx можно запрограммировать цикл диалога (top level) нашего интерпретатора Лиспа EVAL1. В приведенном ниже определении в качестве приглашения интерпретатора будем использовать символ "<-". Кроме этого, признаком выводимых результатов будет символ "->".

```
(defun интерпретатор nil
  (progn
    (princ "Наш Лисп:")
    (terpri)
    (terpri)
    (princ "<-")
    (do ((выражение (read) (read)))
       ((eq выражение 'конец) 'до-свидани_)
       (princ "->")
       (pprint1 (eval1 выражение) 3)
       (terpri)
       (princ "<-")
    )
  )
)
```

Пример использования:

```
> (интерпретатор)
Наш Лисп:

<- 5
->    5
<- t
->
У атома нет св_зи: t   nil
<- t1
->    t1
<- (quote1 ухо-селёдки)
->    ухо-селёдки
<- (cons1 (quote1 a) (quote1 (b c)))
->
   (a
         b
         c)
<- конец
до-свидани_
>
```

Задание: используя в качестве имени функции «функция1» и ранее определенные функции quote1, cond1, car1, cdr1, cons1, atom1, equal1, lambda1, определите функции:

1. last
2. first
3. second
4. third
5. rest
6. nth
7. length
8. append
9. reverse
10. last
11. null
12. listp
13. numberp
14. symbolp
15. zerop
16. plusp
17. minusp
18. consp
19. endp
20. integerp
21. floatp
22. rationalp
23. member
24. not
25. rest
26. acons
27. 1+
28. 1-
29. mod
30. abs
31. signum
32. sin
33. cos
34. tan
35. asin
36. acos
37. expt
38. exp
39. sqrt
40. numerator
41. denominator
42. evenp
43. oddp

Для определения номеров заданий необходимо выбрать в первом столбце нижеследующей таблицы свой номер по списку группы и выполнить задания из второго столбца.

Вариант	*Задания*
1.	1, 4, 7, 9, 14, 16, 17, 20, 24, 27, 30, 31, 32, 34, 42
2.	3, 4, 5, 8, 9, 11, 12, 13, 14, 23, 24, 27, 29, 38, 42
3.	1, 3, 9, 10, 13, 16, 20, 21, 22, 26, 36, 38, 40, 41, 43
4.	1, 3, 4, 6, 7, 9, 15, 17, 21, 23, 35, 36, 37, 39, 42
5.	5, 6, 13, 20, 21, 24, 26, 27, 28, 32, 36, 37, 39, 42, 43
6.	7, 13, 14, 22, 25, 26, 28, 29, 31, 33, 36, 37, 40, 41, 42
7.	4, 5, 6, 8, 10, 12, 16, 21, 23, 24, 29, 31, 34, 38, 39
8.	1, 4, 8, 10, 14, 21, 23, 25, 32, 33, 36, 39, 40, 41, 42
9.	3, 5, 6, 13, 14, 15, 17, 18, 21, 28, 29, 31, 35, 37, 42
10.	1, 2, 3, 5, 6, 7, 18, 22, 25, 26, 31, 36, 40, 41, 42
11.	3, 4, 10, 15, 17, 21, 24, 25, 28, 35, 36, 37, 40, 42, 43
12.	2, 3, 6, 15, 16, 20, 21, 25, 26, 29, 31, 35, 39, 41, 42
13.	1, 3, 7, 9, 12, 13, 14, 18, 19, 21, 23, 30, 32, 36, 41
14.	2, 9, 11, 12, 13, 17, 19, 20, 21, 26, 28, 35, 38, 39, 40
15.	2, 3, 5, 7, 9, 16, 21, 23, 25, 31, 32, 34, 35, 37, 43
16.	1, 6, 11, 17, 18, 25, 26, 27, 28, 30, 32, 35, 36, 38, 40
17.	2, 8, 9, 13, 20, 22, 24, 25, 27, 29, 33, 35, 36, 40, 42
18.	2, 3, 6, 8, 11, 14, 18, 20, 21, 23, 25, 27, 28, 37, 43
19.	2, 5, 8, 9, 11, 12, 18, 23, 25, 29, 33, 34, 37, 40, 41
20.	1, 4, 5, 8, 9, 12, 14, 15, 22, 26, 30, 34, 35, 36, 43
21.	7, 8, 9, 10, 14, 16, 17, 22, 27, 28, 31, 34, 35, 39, 40
22.	2, 5, 6, 7, 14, 18, 20, 23, 24, 25, 26, 28, 34, 36, 38
23.	2, 4, 17, 21, 23, 25, 27, 30, 31, 33, 34, 40, 41, 42, 43
24.	1, 3, 8, 10, 11, 14, 22, 26, 29, 31, 33, 36, 38, 39, 40
25.	1, 2, 4, 6, 7, 13, 21, 23, 24, 29, 33, 37, 38, 42, 43
26.	1, 3, 5, 6, 10, 13, 20, 23, 24, 26, 27, 31, 32, 33, 41
27.	1, 2, 5, 10, 11, 15, 16, 18, 26, 27, 33, 34, 37, 42, 43
28.	1, 2, 6, 7, 8, 13, 14, 15, 16, 17, 20, 28, 31, 39, 40
29.	2, 3, 4, 9, 11, 12, 14, 15, 17, 19, 24, 31, 38, 39, 43
30.	5, 8, 12, 13, 15, 16, 19, 26, 27, 29, 33, 36, 38, 39, 41
31.	2, 6, 7, 8, 14, 15, 17, 18, 24, 28, 35, 36, 37, 39, 41
32.	2, 8, 9, 11, 12, 13, 14, 18, 28, 30, 32, 33, 35, 38, 43
33.	1, 5, 7, 9, 12, 13, 15, 16, 19, 22, 26, 31, 33, 34, 43
34.	5, 6, 7, 9, 13, 17, 20, 23, 27, 28, 33, 36, 39, 40, 42

Лабораторная работа №4.

Минимальная система компьютерной алгебры

1. Миксима – символьный вычислитель

Максима – это известная алгебраическая система, разработка которой началась в Массачусетском технологическом институте в 60-х годах в рамках проекта МАС Вначале исследование символьной и алгебраической обработки математических выражений (symbolic and algebraic computing, SAC) била связано с искусственным интеллектом, так как к предлагаемым ею хорошо определенным примерам были применимы общие способы решения проблем, однако довольно скоро она превратилась в отдельную самостоятельную область исследований, относящуюся сейчас больше к математике, чем к искусственному интеллекту.

Миксима – это небольшая программа символьной математики, похожая на Максиму. Она может обрабатывать предложения как в численном, так и в символьном виде. Миксима умеет распознавать, дифференцировать, упрощать выражения и выводить их без лишних скобок. Диалог с Миксимой происходит путем ввода выражения или операторов присваивания, которые предполагается вычислить или упростить их запись. Например:

```
<= 2 + 3 * 4 !      ; арифметическое действие
=> 14               ; значение
<= a := 2 * b !     ; символьное действие
=> 2 * b            ; символьное значение
<= b := 3 !         ; присваивание
=> 3                ; значение
<= a !              ; имя выражения
=> 6                ; значение выражения
<= f := x + 3 * x ! ; присваивание выражения
=> x + 3 * x        ; значение
<= f d x !          ; дифференцирование
=> 4                ; производная
```

Далее структура и работа программы, а также необходимые формы представления и алгоритмы будут рассмотрены более детально.

2. Действия и их порядок

Для простоты предположим, что вводимые для Миксимы символы отделяются друг от друга пробелами. Миксима считывает выражение, вычисляет его значение или преобразовывает его и выводит ответ. Пока что при программировании ограничимся приведенными ниже действиями:

```
(setq *действи_* '(/ * - + := :: d))
```

Здесь +, *, – и + соответствуют нормальным арифметическим действиям, := – это присваивание и d – дифференцирование. Допустим также, что несколько операторов можно писать на одной строке, разделяя их двойным двоеточием. *Порядок* выполнения действий такой же, как и порядок их следования в списке *ДЕЙСТВИ_*. Например, в выражении

$x := a * b / c$

следующий порядок действий;

$x := (a * (b / c))$

В остальных случаях вычисления осуществляются слева направо. Например;

$a - b - c = (a - b) - c$

Для простоты ограничимся бинарными операциями (с двумя аргументами). Тогда отрицательное символьное значение, например $-a$, будет записываться в виде $0-a$.

3. Чтение выражения с преобразованием в списочную форму

Определим прежде всего функцию ЧИТАЙ, которая считывает вводимое

пользователем выражение, оканчивающееся восклицательным знаком, и преобразует его в список:

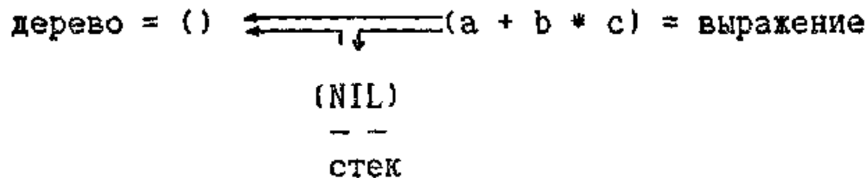
```
(defun читай ()
  (let ((x nil) (res nil))
    (loop
      (setq x (read))
      (when (eq x '!) (return res))
      (setq res (append res (list x)))
    )
  )
)

> (читай)
a b c !
(a b c)
```

4. Преобразование выражения в форму дерева

Язык выражений, рекурсивно построенных из бинарных операций – это простой контекстно-свободный язык. Анализ его выражений может быть осуществлен, например, с помощью следующего довольно простого алгоритма, называемого методом *операторного предшествования*.

Алгоритм основан на использовании стека и представим в виде схемы, состоящей из анализируемого выражения ВЫРАЖЕНИЕ, стека операций СТЕК и дерева ДЕРЕВО, являющегося результатом анализа. Вначале ДЕРЕВО пусто, а стек – список, содержащий пустой список:



Основная идея алгоритма состоит в следующем. Из предложения последовательно друг за другом выбираются символы и помещаются либо прямо в дерево, либо в стек на то время, пока другие символы не будут помещены либо в дерево, либо в стек над этим символом.

Если считанный символ – это величина (константа или переменная), то его помещают прямо в дерево. Если это – операция, то ее помещают или в дерево, или в стек в соответствии с тем, выше или ниже ее приоритет, чем приоритет операции, находящейся в верхней ячейке стека (или NIL). Когда все символы прочитаны до конца, все находящиеся после этого в стеке операции переносятся в дерево и на этом анализ заканчивается. Стек здесь выступает в роли комнаты ожидания, а которой операции с более низким приоритетом ждут, пока не пройдут более высокоприоритетные операции. Более точно можно сформулировать алгоритм следующим образом:

Алгоритм разбора:

1. Если предложение пусто, то перейти к п. 6, иначе взять следующий символ,
2. Если символ является переменной или константой, то перенести его в дерево и перейти к п. 1.
3. Если символ – это операция, и стек пуст, то поместить символ в стек и перейти к п. 1.
4. Если приоритет операции выше приоритета верхней операции, уже находящейся в стеке, то поместить ее в стек и перейти к п. 1.
5. Если приоритет операции меньше или равен приоритету верхней операции в стеке, то поместить операцию из стека и дерево и перейти к п. 3.
6. Если стек пуст, то дерево разбора готово и алгоритм на этом заканчивается.
7. Переместить символ из верхушки стека в дерево и перейти к п. 6.

Алгоритм преобразует выражения в так называемую *обратную польскую* (reverse

Polish) *запись*, в которой символ операции следует непосредственно за аргументами (операндами), например;

$(a + b) \rightarrow (a\ b\ +)$

$(a + b * c) \rightarrow (a\ b\ c\ *\ +)$

$(a * b + c) \rightarrow (a\ b\ *\ c\ +)$

Если в выражении встречается подвыражение, то его можно анализировать, рекурсивно вызывая тот же алгоритм:

$((a + b) * c) \rightarrow (a\ b\ +\ c\ *)$

По мере перемещения символов в стек или в дерево над ними можно было бы выполнять различные действия, касающиеся их формы или порядка. Перемещая, например, в дерево символ операции, можно выполнять и саму операцию, если только, конечно, у находящихся в дереве символов-аргументов имеются значения, для которых определена и выполнима данная операция. Можно также определить и некоторую форму записи или язык, к которому преобразуется выражение в случае, если выполнение операции еще невозможно.

5. Представление выражения в форме дерева

С точки зрения символьной обработки предпочтительнее преобразовывать выражения в форму дерева, в которой легко обрабатывать все подвыражения целиком. Так и будем поступать: всегда, когда в дерево перемещается символ операции, будем вносить изменение, преобразующее два верхних элемента дерева и символ операции в дерево в префиксной форме:

(... *a* + *b*) → ; два верхних элемента дерева и символ добавляемой операции

(... (+ *a* *b*)) ; преобразуется в дерево в префиксной форме

Это преобразование введем в реализующую наш алгоритм разбора функцию АНАЛИЗИРУЙ в виде функции ПРЕОБРАЗУЙ.

```
(defun анализируй (дерево стек выражение)
  (cond
    (
      (null выражение)
      (if (car стек)
        (преобразуй дерево стек выражение)
        (car дерево)
      )
    )
    ((atom выражение) выражение)
    ((atom (car выражение))
      (cond
        (
          (not (member (car выражение) *действи_*))
          (анализируй (cons (car выражение) дерево) стек (cdr выражение))
        )
        (
          (старше (car выражение) (car стек))
          (анализируй дерево (cons (car выражение) стек) (cdr выражение))
        )
        (t (преобразуй дерево стек выражение));???
      )
    )
    (t (анализируй
         (cons (анализируй nil '(nil) (car выражение)) дерево)
         стек (cdr выражение)
       )
    )
  )
)
```

```
(defun преобразуй (дерево стек выражение)
  (анализируй
    (cons (list (car стек)
                (cadr дерево)
                (car дерево)
          ) (cddr дерево)
     )
     (cdr стек)
     выражение
  )
)
```

Определять старшинство операций P и Q будет предикат СТАРШЕ, который возвращает значение T, если P старше, т.е. стоит в списке *ДЕЙСТВИЯ* раньше Q, а иначе возвращает NIL:

```
(defun старше (p q)
  (or
    (null q)
    (member q (member p *действи_*))
  )
)
```

Теперь можно попробовать преобразовать выражение в форму дерева:

```
> (анализируй nil '(nil) '(a + b * c))
(+ a (* b c))
```

6. Порядок обхода дерева

Выражения, представляющие деревья, могут быть получены с помощью трех различных *порядков обхода*, позволяющих систематически обойти все узлы дерева. Возможными порядками обхода являются:

1. Прямой порядок (preorder).
   Префиксная запись (prefix): (+ a (* b c)).
   1. Обработать узел.
   2. Обойти левое поддерево.
   3. Обойти правое поддерево.
2. Промежуточный порядок (inorder).
   Инфиксная запись (infix): (a + (b * c)).
   1. Обойти левое поддерево.
   2. Обработать узел.
   3. Обойти правое поддерево.
3. Обратный порядок (postorder).
   Постфиксная запись (postfix, suffix): (a (b c *) +).
   1. Обойти левое поддерево.
   2. Обойти правое поддерево.
   3. Обработать узел.

Первый и третий способы названы *польской записью* в честь их изобретателя польского математика Л. Лукасевича. Их преимущество перед вторым способом состоит в том, что для них скобки вообще не нужны. Порядок выполнения операций и так ясен на основании порядка следования символов. Польскую запись часто используют в машинных программах, трансляторах и интерпретаторах.

7. Интерпретация и вычисление выражений

Рассмотрим в качестве примера вычисление выражений в постфиксной или *обратной польской записи*. Вычисления можно проводить, используя простой стековый алгоритм по следующей схеме:

Стек = `()` <———————————— `(2 3 4 * +)` = *Дерево*

1. Из дерева считывается символ и перемещается в стек.
2. Если считанный символ – это символ операции, то операция выполняется над двумя верхними элементами стека и все три символа заменяются результатом
3. Перейти к п. 1.

Например:

Стек	*Действие*	*Дерево*
`()`	←	`(2 3 4 * +)`
`(2)`	←	`(3 4 * +)`
`(2 3)`	←	`(4 * +)`
`(2 3 4)`	←	`(* +)`
`(2 3 4 *)`	применение	`(+)`
`(2 (3 4 *))`	*	`(+)`
`(2 12)`	←	`()`
`(2 12 +)`	применение	`()`
`(2 12 +)`	+	`()`
`(14)`	←	`()`

Определим теперь функцию ВЫЧИСЛИ, которая действует по тому же принципу, но вместо списочной формы использует дерево в префиксной форме со всеми расставленными скобками. Значение выражения получается применением операции, указанной в узле, к поддеревьям, значения которых уже предварительно вычислены рекурсивно по тому же алгоритму.

```
(defun вычисли (x &aux значение)
  (cond
    ((numberp x) x)
    (
      (atom x)
      (if (setq значение (get x 'значение))
        (вычисли значение)
        x
      )
    )
    (t (примени (first x)
         (mapcar (function вычисли) (cdr x))
       )
    )
  )
)
```

Значения переменных будем хранить в списке свойств символов под именем ЗНАЧЕНИЕ.

Функция ПРИМЕНИ выполняет операцию из выражения над поддеревьями:

```
(defun примени (операци_ args &aux (op (get операци_ 'fn)))
  (if op
    (apply op args)
    (list op (вычисли (first args)) (вычисли (second args)))
  )
)
```

Чтобы ее можно было осуществить, операция должна быть определена как функция, определение которой хранится под признаком FN в списке свойств символа, являющегося именем операции. В нашем случае для обозначения операции можно использовать те же символы (+, –, * и /), что и в самом Лиспе, но вкладывая в них немного измененный смысл. Это необходимо, например, во время упрощения записи выражения или чтобы возвратить операцию в символьном виде, когда выражение невозможно вычислить. Если операция неизвестна, то ПРИМЕНИ возвращает ее в форме списка, аргументы которого все-таки вычислены.

После этого надо только определить операцию. Используем макрос, упрощающий запись функций в список свойств:

```
(defmacro defдействие (действие args тело)
```

```
  `(setf
    (get ',действие 'fn)
    '(lambda ,args ,тело)
  )
)
```

8. Упрощение выражений

С целью упрощения арифметических выражений определим для операций правила упрощения. Будем учитывать только элементарные упрощения, связанные с константами 0 и 1 или с равенством аргументов. Функции упрощения для различных операций можно определить следующим образом:

```
(defдействие + (x y)
  (cond
    ((and (numberp x) (zerop x)) y)
    ((and (numberp y) (zerop y)) x)
    ((and (numberp x) (numberp y))  (+ x y))
    (t `(+ ,x ,y))
  )
)

(defдействие - (x y)
  (cond
    ((and (numberp y) (zerop y)) x)
    ((and (numberp x) (numberp y))  (- x y))
    (t `(- ,x ,y))
  )
)

(defдействие * (x y)
  (cond
    ((equal x 1) y)
    ((equal y 1) x)
    ((or (and (numberp x) (zerop x)) (and (numberp y) (zerop y))) 0)
    ((and (numberp x) (numberp y)) (* x y))
    (t `(* ,x ,y))
  )
)

(defдействие / (x y)
  (cond
    ((and (numberp x) (zerop x)) 0)
    ((and (numberp y) (zerop y)) 'inf)
    ((equal x y) 1)
    ((and (numberp x) (numberp y)) (/ x y))
    (t `(/ ,x ,y))
  )
)
```

Семантика команды двойное двоеточие, предназначенной для разделения операторов, и команды присваивания несколько отличается от семантики арифметические операций, но и для нее применим использованный выше прием:

```
(defдействие :: (x y) ; вычисли выражения x y
  y                   ; и y и верни значение y
)                     ; в качестве результата

(defдействие := (x y)
  (if (symbolp x)
    (setf (get x 'значение) y)
    (error "~%~S нельзя присвоить значение" x)
  )
)
```

Операцию дифференцирования d можно определить так:

```
(defдействие d (l x)
  (вычисли (дифференцируй l x))
)
```

Функцию дифференцирования выражения ДИФФЕРЕНЦИРУЙ предлагаем определить самостоятельно, предполагая, что первый параметр функции – дифференцируемое выражение в префиксной форме, второй – переменная, по которой осуществляется дифференцирование.

9. Снятие скобок и вывод

Нам нужна еще программа вывода результатов, которая преобразует выражение в префиксной форме, полученное в результате, в инфиксную форму и опускает в нем ненужные скобки. Лишними будут скобки, не изменяющие порядка выполнения операций:

```
(defun сн_ть-скобки (x)
  (if (atom x)
    x
    (append
      (сн_ть-у-оператора (first x) (second x))
      (list (first x))
      (сн_ть-у-оператора (first x) (third x))
    )
  )
)

(defun сн_ть-у-оператора (оператор выражение
                          &aux (x (сн_ть-скобки выражение)))
  (if (or (atom x) (старше оператор (second x)) )
    (list x)
    x
  )
)
```

10. Диалог с Миксимой

Наконец необходима еще программа, поддерживающая диалог между пользователем и Миксимой:

```
(defun миксима (&aux выражение)
  (princ "Миксима: ")
  (loop
    (terpri)
    (princ "<= ")
    (setq выражение (читай))
    (when (equal выражение '(конец))
      (return)
    )
    (princ "=> ")
    (prin1 (сн_ть-скобки (вычисли (анализируй nil '(nil) выражение))))
  )
)

>(миксима)
Миксима:
<= a := 4 :: b := c + a * e !
=> (c + 4 * e)
<= ...
```

Задания:

1. Измените функцию ВЫЧИСЛИ таким образом, чтобы при выполнении операции присвоения левая часть выражения (до знака :=) не вычислялась – это позволит переопределять значения переменных.
2. Реализуйте операцию возведения в степень ^ и дифференцирование по ней.
3. Реализуйте операцию деления нацело div.
4. Реализуйте операцию нахождения остатка от деления mod.
5. Реализуйте операцию «ПОРАЗРЯДНОЕ И» and.
6. Реализуйте операцию «ПОРАЗРЯДНОЕ ИЛИ» or.
7. Реализуйте операцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» xor.
8. Реализуйте операцию «АРИФМЕТИЧЕСКИЙ СДВИГ ВЛЕВО» shl.
9. Реализуйте операцию «АРИФМЕТИЧЕСКИЙ СДВИГ ВПРАВО» shr.

Для определения номеров заданий необходимо выбрать в первом столбце нижеследующей таблицы свой номер по списку группы и выполнить задания из второго столбца.

Вариант	*Задания*
1.	1, 2, 3
2.	1, 2, 4
3.	1, 2, 5
4.	1, 2, 6
5.	1, 2, 7
6.	1, 2, 8
7.	1, 2, 9
8.	1, 2, 3
9.	1, 2, 4
10.	1, 2, 5
11.	1, 2, 6
12.	1, 2, 7
13.	1, 2, 8
14.	1, 2, 9
15.	1, 2, 3
16.	1, 2, 4
17.	1, 2, 5
18.	1, 2, 6
19.	1, 2, 7
20.	1, 2, 8
21.	1, 2, 9
22.	1, 2, 3
23.	1, 2, 4
24.	1, 2, 5
25.	1, 2, 6
26.	1, 2, 7
27.	1, 2, 8
28.	1, 2, 9
29.	1, 2, 4
30.	1, 2, 6
31.	1, 2, 7
32.	1, 2, 9

Лабораторная работа №5.
Создание простейшей экспертной системы

1. Структура экспертной системы

Экспертная система (*система обработки знаний*) – это программная система, знания и умения которой сравнимы с умением и знаниями специалистов в какой-нибудь специальной области знаний. Экспертные системы вместе с системами обработки естественных языков являются наиболее важными в коммерческом плане областями использования искусственного интеллекта. Многие системы естественного языка можно считать и экспертными системами. Они являются экспертами в области лингвистики и, возможно, общего языкознания. Многие алгебраические системы также считаются экспертными системами.

В рамках исследований искусственного интеллекта созданы многочисленные экспертные системы для разных областей знания, таких, например, как медицинская диагностика и обследование пациентов, генные и молекулярные исследования, составление конфигураций вычислительных машин, образование, поиск неисправностей в устройствах и системах и многие другие практические приложения. В этой работе мы запрограммируем небольшую экспертную систему, которую назовем Дарвин.

Типичная экспертная система состоит из двух главных частей: *машины вывода* и *базы знаний* (knowledge base). База знаний содержит данные и знания из области применения; способ решения проблем, используемый в машине вывода, не привязан к *данным из предметной области*. Кроме того, в систему обязательно входит какой-нибудь язык взаимодействия человека с машиной, посредством которого пользователь-специалист ведет диалог с системой, а также в нее входят средства, при помощи которых *инженер знаний* (knowledge engineer) и *эксперт*(*ы*) поддерживают базу знаний.

2. Представление знаний

Для представления знаний используют различные формализмы и языки представления данных. Наиболее часто встречается представление знаний с помощью правил типа ЕСЛИ-ТО. В системе Дарвин правила, описывающие принятие решения, можно задавать в форме, похожей на естественный язык:

(**ЕСЛИ** условие-1
    И условие-2
    ...
    И условие-i
    ТО вывод-1
    И вывод-2
    ...
    И вывод-j)

Условия и выводы – это простые предложения естественного языка. Например:

```
(ЕСЛИ на лампочку подано напряжение
      И лампочка не горит
      ТО лампочка, вероятно, перегорела)
(ЕСЛИ читатель перестал понимать
      И читатель хочет учиться
      И читатель еще может читать
      ТО читателю нужно начать все сначала
      И читателю следует быть повнимательней)
```

3. Машина вывода

Машина вывода – это универсальный механизм (программа или аппарат), который с

помощью правил базы знаний строит новые выводы, задает дополнительные вопросы и так далее до тех пор, пока не придет к какому-нибудь приемлемому конечному результату или ответу. Исходные данные и выводы, полученные в результате принятия решений, в дальнейшем будем называть известными системе *фактами*.

В более крупных системам база знаний содержит помимо правил знания и других типов: факты об объектах проблемной области и их свойства, данные об обстоятельствах и событиях, иерархии понятий, метаданные и т.д.

Работа экспертных систем основывается в первую очередь на обширной базе знаний. Работа машин вывода часто довольно проста н прямолинейна. Рассмотрим более подробно структуру машины принятия решений программы Дарвин.

5. Факты и правила

В системе Дарвин факты представляются просто в виде списков символов. Например, следующий список мог бы описывать наши знания о каком-нибудь животном:

```
(
  (животное имеет шерсть)
  (животное полосатое)
  (животное жвачное)
)
```

База знаний системы Дарвин образуется из списков, подобных ранее описанным правилами ЕСЛИ-ТО. Например:

```
(setq ПРАВИЛО12
       '(ЕСЛИ животное жвачное
         И животное полосатое
         ТО животное зебра)
)
```

Чтобы правила можно было свести к фактам, их условия и выводы необходимо представлять списками фактов. Для этого предложения, являющиеся правилами, можно преобразовать в структуры, которые состоят из имени правила, условий и выводов, представленных в виде списка фактов:

```
(defstruct правило имя условия выводы)
```

Соответствующую предыдущему правилу ПРАВИЛО12 структуру можно теперь создать при помощи вызова:

```
(make-правило
  :имя     'ПРАВИЛО12
  :условия '(
               (животное жвачное)
               (животное полосатое)
             )
  :выводы  '(
               (животное зебра)
             )
)
```

Если условия правила являются элементами какого-нибудь списка фактов, то применение правила к этому списку фактов можно реализовать путем добавления в список выводов из правила. Например, применив ПРАВИЛО12 к ранее представленному списку данных, можно сделать заключение: животное, о котором идет речь – это зебра.

6. Правила вывода базы знаний

Знания системы Дарвин о животных содержатся в базе знаний, содержащей следующие 14 правил:

```
(setq ПРАВИЛО1
  '(ЕСЛИ животное имеет шерсть
    ТО животное млекопитающее)
)
```

```
(setq ПРАВИЛО2
  '(ЕСЛИ животное кормит детеньшей молоком
    ТО животное млекопитающее)
)
(setq ПРАВИЛО3
  '(ЕСЛИ животное имеет перь_
    ТО животное птица)
)
(setq ПРАВИЛО4
  '(ЕСЛИ животное умеет летать
    И животное несет _йца
    ТО животное птица)
)
(setq ПРАВИЛО5
  '(ЕСЛИ животное ест м_со
    ТО животное хищник)
)
(setq ПРАВИЛО6
  '(ЕСЛИ животное имеет острые зубы
    И животное имеет когти
    И глаза животного посажены пр_мо
    ТО животное хищник)
)
(setq ПРАВИЛО7
  '(ЕСЛИ животное млекопитающее
    И животное имеет копыта
    ТО животное жвачное)
)
(setq ПРАВИЛО8
  '(ЕСЛИ животное млекопитающее
    И животное жует жвачку
    ТО животное жвачное)
)
(setq ПРАВИЛО9
  '(ЕСЛИ животное млекопитающее
    И животное хищник
    И животное желто-коричневое
    И животное имеет темные п_тна
    ТО животное гепард)
)
(setq ПРАВИЛО10
  '(ЕСЛИ животное млекопитающее
    И животное хищник
    И животное желто-коричневое
    И животное полосатое
    ТО животное тигр)
)
(setq ПРАВИЛО11
  '(ЕСЛИ животное жвачное
    И животное длинношеее
    И животное длинноногое
    И животное имеет тёмные п_тна
    ТО животное жираф)
)
(setq ПРАВИЛО12
   '(ЕСЛИ животное жвачное
     И животное полосатое
     ТО животное зебра)
)
(setq ПРАВИЛО13
  '(ЕСЛИ животное птица
    И животное не умеет летать
    И животное длинношеее
    И животное длинноногое
```

```
    И животное черно-белое
    ТО животное страус)
)
(setq ПРАВИЛО14
  '(ЕСЛИ животное птица
    И животное не умеет летать
    И животное плавает
    И животное черно-белое
    ТО животное пингвин)
)
```

Описанные выше правила легко понять программисту и создателю системы, и поэтому они не требуют описания. Следующая функция АНАЛИЗИРУЙ берет список правил, анализирует каждое правило и преобразует его к ранее описанной структуре (ПРАВИЛО), полями которой являются: имя правила, условия и выводы:

```
;Функция для добавления элемента в конец списка
(defun присоедини (x y)
  (append x (list y))
)

;Анализирует правила и составляет из них список
(defun анализируй (правила)
  (mapcar (function анализируй-правило) правила)
)

;Преобразует правило в форме предложения в структуру
(defun анализируй-правило (им_)
  (let ((правило (eval им_)))
    (make-правило
      :им_ им_
      :услови_ (услови_ правило)
      :выводы (выводы правило)
    )
  )
)

;Возвращает условия в виде списка
(defun услови_ (предложение)
  (предложение-и (cdr предложение) nil nil)
)

;Возвращает выводы в виде списка
(defun выводы (предложение)
  (предложение-и (cdr (member 'то предложение)) nil nil)
)

;Выделяет предложения, разделяемые символом И
(defun предложение-и (предложение часть результат)
  (cond
    (
      (null предложение) ;; условие окончания
      (присоедини результат часть)
    )
    (
      (eq (car предложение) 'то) ;; условие окончания
      (присоедини результат часть)
    )
    (
      (eq (car предложение) 'и) ;; новое предложение
      (предложение-и
        (cdr предложение)
        nil
        (присоедини результат часть)
      )
```

```
      )
      (
        t ; следующее слово
        (предложение-и
          (cdr предложение)
          (присоедини часть  (car предложение))
          результат
        )
      )
    )
  )
```

Правила в виде записей легко обрабатывать, и функциям более высокого уровня не нужно знать об анализе данных или деталях представления правил. Впоследствии это облегчает возможную модификацию программ и их дальнейшее развитие. Если бы правила представлялись, например, в виде списков, состоящих из условий и выводов, то на более высоком уровне пришлось бы иметь информацию о порядке следования элементов в списке. Теперь же на части правил можно указывать логическим именем.

Правила системы Дарвин хранятся в списке *БАЗА-ЗНАНИЙ*:

```
(setq *база-знаний*
  '(ПРАВИЛО1 ПРАВИЛО2 ПРАВИЛО3 ПРАВИЛО4 ПРАВИЛО5
    ПРАВИЛО6 ПРАВИЛО7 ПРАВИЛО8 ПРАВИЛО9 ПРАВИЛО10
    ПРАВИЛО11 ПРАВИЛО12 ПРАВИЛО13 ПРАВИЛО14)
)
```

Проанализированные варианты правил сохраним в переменной *ПРАВИЛА* следующей командой:

```
(setq *правила* (анализируй *база-знаний*))
```

Значением переменной *ПРАВИЛА* будет список, состоящий из структур.

Предположим, что система Дарвин сохраняет известные ей факты в списке *ФАКТЫ*, состоящем из элементов данных. В этом случае применимость правила можно проверить функцией ПРОВЕРЬ-ПРАВИЛО и выводы правила можно при необходимости добавить к фактам функцией ДОБАВЬ-ВЫВОДЫ:

```
;Проверяет, применимо ли правило
(defun проверь-правило (правило)
  (подмножество (правило-услови_ правило) *факты*)
)

;Проверяет, является ли множество подмножеством
(defun подмножество (подмножество множество)
  (equal
    подмножество
    (intersection подмножество множество)
  )
)

(defmacro mypush (v l)
      `(setf ,l (cons ,v ,l))
)


;Расширяет список фактов выводами правила
(defun добавь-выводы (правило)
  (do ((выводы (правило-выводы правило) (cdr выводы)))
    ((null выводы) *факты*)
    (if (member (car выводы) *факты*)
      nil
      (progn
        (format t "Согласно правилу ~S : " (правило-им_ правило))
        (выведи-элементы (car выводы))
        (mypush (car выводы) *факты*)
      )
    )
```

```
  )
)

;Выводит элементы списка
(defun выведи-элементы (список)
  (dotimes (i (length список))
    (princ (nth i список))
    (princ " ")
  )
  (terpri)
  t
)
```

7. Стратегия обратного вывода

Машина вывода применяет правила к известным в текущий момент системе фактам для нахождения новых фактов до тех пор, пока в результате применения не будет получен искомый результат. Если у системы недостаточно информации для дальнейшего поиска решения, то она запрашивает у пользователя дополнительную информацию и сохраняет ее в своем списке фактов, а затем пытается еще раз применить к своим правилам новые дополнительные факты и т.д.

В исследованиях по искусственному интеллекту созданы различные способы нахождения решения и выполнения действия. В системе Дарвин мы применим *обратный вывод* (backward chaining), в котором решение пытаются найти, идя в обратном направлении от конечного результата.

8. Работа системы Дарвин

Систему Дарвин на самом верхнем уровне можно представить как распознавателя животных, который пытается на основе своих правил доказать некоторую гипотезу.

Представим возможные конечные результаты в виде списка *ГИПОТЕЗЫ*:

```
(setq *гипотезы*
  '(
    (животное пингвин)
    (животное страус)
    (животное зебра)
    (животное жираф)
    (животное тигр)
    (животное гепард)
  )
)
```

Все эти гипотезы встречаются в части вывода некоторых правил. При обратном выводе условия этих правил можно интерпретировать как новые гипотезы, если вывод является конечным результатом, и так далее. Так система порождает гипотезы более низкого уровня до тех пор, пока не обнаружит, что новых правил для порождения гипотез больше нет. Тогда система запрашивает условия правила непосредственно у пользователя, которые он на этом уровне уже, вероятно, сможет задать сам и не будучи специалистом, затем система с помощью новой информации пытается продвинуться дальше в своих выводах:

```
(setq *факты* nil)
(setq *запросы* nil)

;;; Глобальные динамические переменнне
(defvar *гипотезы*)  ; значение определено выше
(defvar *правила*)   ; значение определено выше
(defvar *факты*)     ; значение определено выше
(defvar *запросы*)   ; значение определено выше

;;; Главная программа Дарвин
```

```
(defun Дарвин ()  ; Экспертная система для распознавания животных
  (setq *факты* nil)
  (setq *запросы* nil)
  (знаток-зверей *гипотезы*)
)

;Пытается проверить какую-нибудь гипотезу
(defun знаток-зверей (гипотезы)
  (cond
    (
      (null гипотезы)
      "Не могу доказать никакую из известных мне гипотез"
    )
    (
      (докажи (car гипотезы))
      (car гипотезы)    ; результат
    )
    (
      t
      (знаток-зверей (cdr гипотезы)) ; новая попытка
    )
  )
)
```

Доказательство гипотезы или утверждения, можно выполнить при помощи следующей процедуры:

Процедура Докажи:

Дано: Доказываемая гипотеза.

Значение: Значение функции – истина, если данную гипотезу можно доказать

Действия:

1. Если гипотеза уже есть в списке фактов, значит она доказана.
2. Если это не так, то соберем все правила, с помощью которых можно было бы доказать гипотезу, т.е. те правила, в части вывода которых есть проверяемая гипотеза. Если какое-нибудь из этих правил можно прямо применить к списку фактов, то, значит, гипотеза доказана.

   Если никакое из вобранных правил нельзя применить, то попытаемся доказать применимость какого-либо правила, доказав все условия правила, взяв их как новые гипотезы и рекурсивно применив процедуру Докажи.
3. Если описанные действия не дают результата, то спросим у пользователя, верна ли гипотеза (если только это уже не спрашивалось), и если она верна, то присоединим утверждение к списку фактов. Присоединим утверждение к списку *ЗАПРОСЫ* независимо от ответа, чтобы потом это же самое не пришлось повторно спрашивать.

Эту рекурсивную процедуру можно непосредственно программировать. Все три ветви вышеописанной процедуры помечены в определении соответствующими комментариями:

```
(defun докажи (гипотеза)
  (let ((прав nil))
    (cond
      ((member гипотеза *факты*) t)            ; ветвь 1
      (
        (setq прав (возможные гипотеза))       ; ветвь 2
        (if (пр_мо гипотеза прав)
          t
          (рекурсивно гипотеза прав)
        )
      )
      (
        t                                      ; ветвь 3
        (ветка3 гипотеза)
      )
```

```
    )
  )
)

(defun ветка3 (гипотеза)
  (cond
    ((member гипотеза *запросы*) nil)
    (
      (and
        (princ "Это правда, что: ")
        (выведи-элементы гипотеза)
        (eq (read) 'Да)
      )
      (setq *факты* (union (list гипотеза) *факты*))
      t
    )
    (
      t
      (mypush гипотеза *запросы*)
      nil
    )
  )
)

(defun возможные (гипотеза)
  (let ((result nil))
    (dotimes (i (length *правила*) result)
      (when (member
                гипотеза (правило-выводы (nth i *правила*))
                :test (function equal)
                )
          (setq result (append result (list (nth i *правила*))))
      )
    )
  )
)

;Проверяет, можно ли доказать гипотезу непосредственно
;при помощи какого-нибудь правила
(defun пр_мо (гипотеза возможные)
  (cond
    ((null возможные) nil)
    ((null *факты*) nil)
    (
      (проверь-правило (car возможные))
      (добавь-выводы (car возможные))
    )
    (
      t
      (пр_мо гипотеза (cdr возможные))
    )
  )
)

;Рекурсивно проверяет гипотезу
(defun рекурсивно (гипотеза возможные)
  (cond
    ((null возможные) nil)
    (
      (проверь-непр_мо (правило-услови_ (car возможные)))
      (добавь-выводы (car возможные))
    )
    (
      t
```

```
        (рекурсивно гипотеза (cdr возможные))
      )
    )
  )

;Рекурсивно проверяет все условия
(defun проверь-непр_мо (услови_)
  (every (function докажи) услови_)
)
```

На основании своих правил программа Дарвин может задавать пользователю лишь разумные с точки зрения гипотезы вопросы. Однако первую гипотезу приходится выбирать наобум.

9. Примеры запросов

Теперь немного побеседуем с программой Дарвин. В первом примере пользователь видит перед собой пингвина, во втором – зебру, но животных он не знает. С помощью программы Дарвин он может определить виды этих животных, давая системе простые ответы на ее вопросы:

```
> (Дарвин)
Это правда, что: животное имеет перь_
Да
Согласно правилу правило3 : животное птица
Это правда, что: животное не умеет летать
Да
Это правда, что: животное плавает
Да
Это правда, что: животное черно-белое
Да
Согласно правилу правило14 : животное пингвин
(животное пингвин)
> (Дарвин)
Это правда, что: животное имеет перь_
Нет
Это правда, что: животное умеет летать
Нет
Это правда, что: животное имеет шерсть
Да
Согласно правилу правило1 : животное млекопитающее
Это правда, что: животное имеет копыта
Да
Согласно правилу правило7 : животное жвачное
Это правда, что: животное полосатое
Да
Согласно правилу правило12 : животное зебра
(животное зебра)
```

Задания:

1. Дополните экспертную систему меню, содержащим следующие пункты:
   1. Произвести экспертную оценку
   2. Вывести информацию о ...
   3. Объяснить последний вывод
   4. Завершение работы

   По первому пункту должна запускаться машина вывода. По второму пункту должен вводиться запрос о предмете и выводиться все правила и гипотезы, его содержащие. По третьему пункту необходимо в удобной форме распечатать цепочку выводов последнего утверждения (гипотезы) в виде последовательностей «Так как ..., можно сделать вывод, что ...». Выбор четвертого пункта должен приводить к завершению работы системы.
2. Создайте экспертную систему, решающую, какую формулу применять при решении задач из школьного курса физики (разделы кинематика и динамика). В качестве гипотезы задайте формулы, правила постройте на основе следующего фрагмента базы знаний:

```
багатозначний(формула)
дозвзн(розділ)=кінематика,динаміка
дозвзн(рух_тіла)=прямолінійний,обертальний
дозвзн(швидкість)=постійна,рівнозмінна
дозвзн(врахування_причин)=не_враховуються,враховуються
дозвзн(сила)=сила_тертя,сила_тяжіння,сила_пружності,декілька_сил
дозвзн(зад_сил)=задані,не_задані
питання(врахування_причин)=Чи враховуються причини руху?
питання(рух_тіла)=Який характер руху тіла?
питання(швидкість)=Який характер швидкості?
питання(сила)=Яка сила розглядається в задачі?
питання(зад_сил)=Чи задані сили?

правило1: якщо
                врахування_причин=не_враховуються
              то
              розділ=кінематика.
правило2: якщо
              врахування_причин=враховуються
             то
          розділ=динаміка.


правило3: якщо
                 розділ=кінематика і
                 рух_тіла=прямолінійний і
                 швидкість=постійна
                 то
                  формула= X=X0+V*t і
                  формула= V=S/t.


правило4: якщо
                 розділ=кінематика і
                 рух_тіла=прямолінійний і
                 швидкість=рівнозмінна
                 то
                 формула= a=(V-V0)/t і
                 формула= V=V0+a*t і
                 формула= X=X0+V0*t+A*t^2/2
   bmp=pryskor.
правило5: якщо
                 розділ=кінематика і
                 рух_тіла=обертальний
                то
                формула= w=f/t і
                формула= V=w*r і
```

```
                формула= a=V^2/r.
правило0: якщо
        розділ=кінематика
  то
      підрозділ=невизн.

правило6: якщо
              розділ=динаміка і
             зад_сил=задані
            то
           підрозділ=рух_під_дією_сил.

правило7: якщо
            розділ=динаміка і
              зад_сил=не_задані
        то
             підрозділ=закони_ньютона.

правило8: якщо
           розділ=динаміка і
          підрозділ=закони_ньютона
          то
          формула= F=M*a і
          формула=  F1=-F2.
правило9: якщо
      розділ=динаміка і
      підрозділ=рух_під_дією_сил і
             сила=сила_тертя
       то
          формула= F=-k*N.
правило10: якщо
             розділ=динаміка і
              підрозділ=рух_під_дією_сил і
            сила=сила_тяжіння
            то
            формула= F=G*M1*M2/r^2.
правило11: якщо
              розділ=динаміка і
              підрозділ=рух_під_дією_сил і
              сила=сила_пружності
               то
               формула= F=-k*X.

правило12: якщо
               розділ=динаміка і
                підрозділ=рух_під_дією_сил і
               сила=декілька_сил
                то
                формула= m*a=F1+F2+  +FN.
кінець
```

3. Создайте экспертную систему «Классификатор растений» на основе следующего фрагмента базы знаний:

   Если класс голосемянные и форма листа чешуеобразная, то семейство - кипарисовые.
   Если класс голосемянные и форма листа иглоподобная и конфигурация хаотическая, то семейство - сосновые.
   Если класс голосемянные и форма листа иглоподобная и конфигурация - 2 ровных ряда и серебристая полоса, то семейство - сосновые.
   Если класс голосемянные и форма листа иглоподобная и конфигурация - 2 ровных ряда и серебристой полосы нет, то семейство - болотный кипарис.
   Если тип - деревья и форма листа широкая и плоская, то класс - покрытосемянные.
   Если тип - деревья и неверно, что форма листа широкая и плоская, то класс -

```
голосемянные.
Если стебель зеленый, то тип - травянистые.
Если стебель древесный и положение стелящееся, то тип - лианы.
Если стебель древесный и положение прямостоящее и один основной ствол, то
тип - деревья.
Если стебель древесный и неверно, что положение прямостоящее и один основной
ствол, то тип - кустарниковые.
```

4. Создайте экспертную систему «Прогнозирование наводнений» на основе следующего фрагмента базы знаний:

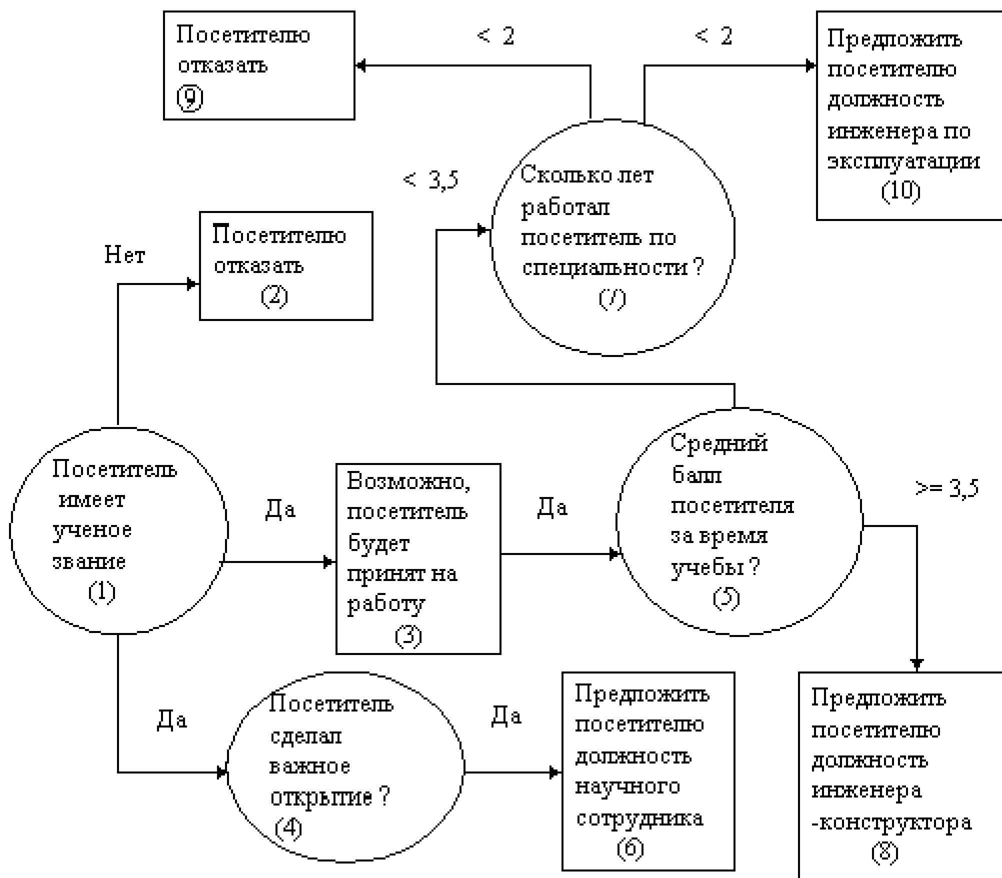
```
1 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = ОБИЛЬНЫЙ,
   ТО VILLAGE = ЭВАКУИРОВАТЬ
2 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = НЕ СИЛЬНЫЙ И
   SNOW = МНОГО,
   TEMPERATURE = ВЫСОКАЯ,
   ТО VILLAGE = ЭВАКУИРОВАТЬ
3 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = НЕ СИЛЬНЫЙ И
   SNOW = МНОГО,
   TEMPERATURE = СРЕДНЯЯ И
   RAIN = УМЕРЕННЫЙ,
   ТО VILLAGE = УСИЛИТЬ ВНИМАНИЕ
4 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = НЕТ И
   SNOW = МНОГО,
   TEMPERATURE = СРЕДНЯЯ И
   RAIN = СЛАБЫЙ,
   ТО VILLAGE = НЕ БЕСПОКОИТЬСЯ
5 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = НЕ СИЛЬНЫЙ И
   SNOW = МАЛО,
   ТО VILLAGE = НЕ БЕСПОКОИТЬСЯ
6 ЕСЛИ WATER-LEVEL = НЕ ВЫСОКИЙ И
   RAIN = СИЛЬНЫЙ И
   SNOW = МНОГО,
   TEMPERATURE = ВЫСОКАЯ,
   ТО VILLAGE = УСИЛИТЬ ВНИМАНИЕ
7 ЕСЛИ WATER-LEVEL = НЕ ВЫСОКИЙ И
   RAIN = СИЛЬНЫЙ И
   SNOW = МНОГО,
   TEMPERATURE = СРЕДНЯЯ,
   ТО VILLAGE = НЕ БЕСПОКОИТЬСЯ
8 ЕСЛИ WATER-LEVEL = НЕ ВЫСОКИЙ И
   RAIN = СИЛЬНЫЙ И
   SNOW = МАЛО,
   ТО VILLAGE = НЕ БЕСПОКОИТЬСЯ
9 ЕСЛИ WATER-LEVEL = ВЫСОКИЙ И
   RAIN = НЕСИЛЬНЫЙ,
   ТО VILLAGE = НЕ БЕСПОКОИТЬСЯ
```

5. Создайте экспертную систему «Классификатор птиц», различающую орла, сокола и воробья, используя в качестве принципов классификации имя объекта, длину клюва, окрас оперенья, размах крыльев, высоту полета, аэродинамические принципы.
6. Создайте мини-систему для прогнозирования работы фондовой биржи, используя следующий фрагмент базы знаний:

```
1 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ = ПАДАЮТ,
     ТО УРОВЕНЬ ЦЕН НА БИРЖЕ = РАСТЕТ
2 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ = РАСТУТ,
     ТО УРОВЕНЬ ЦЕН НА БИРЖЕ = ПАДАЕТ
3 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА = ПАДАЕТ,
     ТО ПРОЦЕНТНЫЕ СТАВКИ = РАСТУТ
4 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА = РАСТЕТ,
```

```
ТО ПРОЦЕНТНЫЕ СТАВКИ = ПАДАЮТ
```

7. Создайте экспертную систему «Прием на работу», реализующую следующую схему:



Для определения номеров заданий необходимо выбрать в первом столбце нижеследующей таблицы свой номер по списку группы и выполнить задания из второго столбца.

Вариант	*Задания*
1.	1, 2, 3
2.	1, 2, 4
3.	1, 2, 5
4.	1, 2, 6
5.	1, 2, 7
6.	1, 2, 3
7.	1, 2, 4
8.	1, 2, 5
9.	1, 2, 6
10.	1, 2, 7
11.	1, 2, 3
12.	1, 2, 4
13.	1, 2, 5
14.	1, 2, 6
15.	1, 2, 7
16.	1, 2, 3
17.	1, 2, 4
18.	1, 2, 5
19.	1, 2, 6
20.	1, 2, 7
21.	1, 2, 3
22.	1, 2, 4
23.	1, 2, 5
24.	1, 2, 6
25.	1, 2, 7
26.	1, 2, 3
27.	1, 2, 4
28.	1, 2, 5
29.	1, 2, 6
30.	1, 2, 7
31.	1, 2, 3
32.	1, 2, 4

Лабораторная работа №6.

Минимальный интерпретатор языка Пролог

Язык Пролог – язык *логического программирования*. Теоретической основой Пролога является исчисление предикатов первого порядка. Пролог относится к декларативным языкам. Программирование на языке Пролог состоит из следующих этапов: 1) объявления некоторых фактов об объектах и отношениях между ними; 2) определения некоторых правил об объектах и отношениях между ними; 3) формулировка вопросов об объектах и отношениях.

Правила, факты и цели представляют собой *хорновские дизъюнкты* (клозы). *Правила* представляются в виде:

$B{:}\text{-}\ A_1, A_2, ..., A_n\ (n \geq 1)$,

где A_i, $i=1, 2, ..., n$ – условия, B – заключение. Подобная запись правила равносильна импликации вида:

$A_1\ \&\ A_2\ \&\ ...\ \&\ A_n \rightarrow B$.

Факты представляются в виде клозов, в которых присутствуют только заключения:

B:-

Клозы, в которых присутствуют только условия, вида:

$:\text{-}\ A_1, A_2, ..., A_n$

интерпретируются как целевые утверждения (*цели*), представленные в форме отрицания.

В Прологе имеется мощный встроенный механизм логического вывода с поиском и возвратом. Для доказательства целевых утверждений в Прологе используется семантическая линейная резолюция. Основой построения интерпретатора Пролога является процедурная интерпретация правил. Учитывая это обстоятельство, вместо термина «логический вывод» используется термин «вычисление». Вместо термина «доказательство цели» иногда используется термин «согласование целевого утверждения с базой данных». Назовем конъюнкцию целей, которую следует доказать на рассматриваемом шаге выполнения программы, *резольвентой*. Вначале резольвента совпадает с начальной целью (вопросом). Если после удачной унификации очередной цели эту цель заменить телом выбранного правила, то получим резольвенту для очередного шага выполнения программы. Операция, связанная с заменой цели G телом того же правила из программы P, заголовок которого совпадает с данной целью, называют *редукцией*. При этом заменяемая цель при редукции называется снятой, а добавляемые цели – производными.

Терм A называется *примером* (или конкретизацией) терма B, если существует подстановка #, такая, что A=#[B]. Терм #[B] получается одновременной заменой всех вхождений переменных в B на их образы относительно #. Множество термов $\{T_1, T_2, ..., T_n\}$ *унифицируемо*, если существует подстановка # такая, что $\#[T_1]=\#[T_2]=...=\#[T_n]$. В этом случае # называют *унификатором* для $\{T_1, T_2, ..., T_n\}$, т.к. при ее применении это множество «склеивается» в один элемент. *Наиболее общий* (или простейший) *унификатор* (НОУ) L для множества $\{T_i\}$ обладает тем свойством, что если # – есть любой унификатор для $\{E_i\}$, дающий $\#[E_i]$, то существует некоторая подстановка #' такая, что $\#[E_i]=\#'{*}L[E_i]$, где * – операция композиции подстановок. Алгоритм унификации будет рассмотрен ниже при изучении рекурсивной функции унификации `unify`, написанной на языке Лисп.

Чтобы избежать путаницы при использовании переменных с одинаковыми именами в разных правилах, условимся переименовывать переменные всякий раз, когда предложение выбирается для выполнения редукции. Среди новых имен не должно быть имен, ранее использованных при вычислении.

Алгоритм работы интерпретатора в первом приближении можно описать так.

Исходные данные. Логическая программа P и вопрос G..

Результат. Выражение Q[G], если этот пример выводится из P, или сообщение о неудаче, если вывод невозможен.

Алгоритм.

1. В качестве начальной резольвенты принять входную цель G.
2. Если резольвента не пуста, то взять в качестве текущей самую левую цель G из резольвенты. Выполнить п. 4.
3. Если резольвента пуста, то полученный пример является ответом на вопрос. Выполнить п. 16.
4. Найти в программе правило, у которого функтор заголовка совпадает с функтором текущей цели.
5. Если правило в программе найдено, то выполнить п. 10.
6. Если правило в программе отсутствует, то выполнить п. 15.
7. Найти в программе очередное альтернативное правило.
8. Если такое правило есть, то выполнить п. 10.
9. Если альтернативных правил больше нет, то выполнить п. 13.
10. Выполнить унификацию выбранного правила и текущей цели.
11. Если унификация закончилась успешно, то осуществить редукцию текущей цели. Выполнить п. 2.
12. Если унификация закончилась неудачей, то выполнить п. 7.
13. Если цель предыдущего шага является начальной целью, то выполнить п. 15.
14. 14.Если цель предыдущего шага не является начальной целью, то заменить текущую цель в резольвенте целью предыдущего шага и выполнить п. 7.
15. Вывести сообщение о неудаче.
16. Закончить выполнение алгоритма.

Интерпретатор, реализующий приведенный алгоритм, должен остановиться после обнаружения первого примера заданной цели. В реальных интерпретаторах предусматривается режим повторного запуска, обеспечивающий получение других примеров цели, если они существуют.

Ниже рассматривается интерпретатор языка Пролог, написанный на языке Lisp. Функции `rename_variables`, `print-bindings`, `try-each`, `unify`, `value` являются рекурсивными, поскольку в определении этих функций содержится вызов самих этих функций. Кроме того, функции `prove` и `try-each` являются взаиморекурсивными, т.к. они вызывают друг друга.

Запуск интерпретатора осуществляется следующим образом:

```
(prolog db)
```

где `db` – база данных. База данных представляется как список, состоящий из фактов и правил. Факты и правила в свою очередь представляются в списочной форме. Переменные в правилах представляются в виде двухэлементного списка вида:

```
(? <имя переменной>)
```

Порядок следования головы и составляющих тела правила в списке сохраняются. Например, факт, определяемый на Прологе как

```
father(madelga, ernest)
```

представляется в виде списка `(father madelga ernest)`, а правило

```
grandparent(X, Y):- parent(X, Z), parent(Z, Y).
```

в виде списка

```
((grandparent(? X)(? Y))
(parent (? X) (? Z))(parent (? Z) (? Y)))
```

Среди прочих основных структур данных можно выделить ассоциативный список `environment`, в котором сохраняются связи переменных с их значениями (иначе, подстановка). С помощью численного параметра `level` отличаются связи одноименных переменных из различных правил.

Основной цикл интерпретатора Пролога реализуется функцией верхнего уровня `prolog`, которой в качестве параметра `database` передается Пролог-программа. Функция определена следующим образом:

```
(defun prolog (database &aux goal)
      (do () ((not (progn (princ "Query?") (setq goal (read)))))
            (prove (list (rename-variables goal '(0)))
```

```
                          '((bottom-of-environment))
                          database
                          1)))
```

Функция выводит строку «Query?» и ожидает ввода пользователем целевого утверждения `goal`. Если ввести пустой список, то интерпретатор завершает свою работу. Введенное целевое утверждение передается функции `prove` для доказательства. Перед этим, однако, переменные в целевом утверждении переименовываются и им присваивается уровень 0. Список `(bottom-of-environment)` выступает в качестве маркера дна стека связей переменных.

Функция `prove` определяется следующим образом:

```
(defun prove (list-of-goals environment database level)
     (cond ((null list-of-goals)
            (print-bindings environment environment)
            (not (y-or-n-p "More?")))
           (t (try-each database database
                        (cdr list-of-goals) (car list-of-goals)
                        environment level))))
```

Функция `prove` получает следующие параметры: `list-of-goal` – резольвенту (в виде списка целей); `environment` – связи переменных; `database` – Пролог-программу в списочной форме; `level` – уровень.

Функция `prove` проверяет список целей и, если данный список пуст, то считается, что целевое утверждение успешно доказано, выводятся связи переменных, вызывается функция `y-or-n-p`, которая запрашивает, продолжать или нет поиск альтернативных решений при доказательстве целевого утверждения. Если список целей `list-of-goal` не пуст, то вызывается функция `try-each` для доказательства целей из этого списка, причем первая цель из этого списка делается текущей.

Текст функции try-each представлен ниже:

```
(defun try-each (database-left database goals-left goal
                environment level
                &aux assertion new-enviroment)
      (cond ((null database-left) nil)
            (t (setq assertion
                     (rename-variables (car database-left)
                                       (list level)))
               (setq new-environment
                     (unify goal (car assertion) environment))
               (cond ((null new-environment)
                      (try-each (cdr database-left) database
                                goals-left goal
                                environment level))
                     ((prove (append (cdr assertion) goals-left)
                             new-environment
                             database
                             (+ 1 level)))
                     (t (try-each (cdr database-left) database
                                  goals-left goal
                                  environment level))))))
```

Функция `try-each` получает следующие параметры: `database-left` – остаток Пролог-программы, используемой в процессе доказательства; `database` – всю Пролог-программу; `goals-left` – еще не доказанные цели из резольвенты; `goal` – цель, доказываемую в настоящий момент; `environment` – связи переменных; `level` – уровень.

Функция `try-each` проверяет список `database-left`. Если он пуст, то функция возвращает `NIL` – признак неудачи при доказательстве. В противном случае она выбирает очередное правило (`assertion`) из остатка Пролог-программы, переименовывает в нем переменные с помощью функции `rename-variables` и пытается выполнить унификацию головы выбранного правила (`car assertion`) и текущей цели (`goal`) с помощью функции `unify`. Если унификация прошла неуспешно (список `new-environment` пуст), то из остатка

Пролог-программы выбрасывается текущее «неунифицируемое» правило и вновь вызывается функция `try-each` в попытке доказать ту же цель `goal`. Если унификация прошла успешно (список `new-environment` содержит новые, появившиеся при унификации связи переменных), то осуществляется редукция текущей цели `goal` путем занесения предикатов тела текущего правила `assertion` в список недоказанных целей `goal-left` и вызов функции `prove` для доказательства новой резольвенты `goal-left`. Если доказательство этой резольвенты прошло неуспешно, то из остатка Пролог-программы выбрасывается текущее правило и делается попытка доказать прежнюю резольвенту `goal` на этом остатке Пролог-программы.

Функция `unify` предназначена для унификации двух термов – `x` и `y`. В качестве третьего параметра функция получает список `environment`, который представляет текущий НОУ. Функция возвращает НОУ или `NIL` (`NIL` – если термы неунифицируемы или НОУ является пустым списком). На языке Lisp функция `unify` представляется следующим образом:

```
(defun unify (x y environment &aux new-environment)
   (setq x (value x environment))
   (setq y (value y environment))
   (cond ((variable-p x) (cons (list x y) environment))
         ((variable-p y) (cons (list y x) environment))
         ((or (atom x) (atom y))
              (cond ((equal x y) environment)
                    (t nil)))
         (t (setq new-environment (unify (car x) (car y)
                                          environment))
            (cond (new-environment (unify (cdr x) (cdr y)
                                          new-environment))
                  (t nil)))))
```

Алгоритм функции `unify(x,y,environment)` – «унифицировать» – может быть записан в следующем виде:

1. Попытаться присвоить `x` с помощью функции `value` значение `x`.
2. Попытаться присвоить `y` с помощью функции `value` значение `y`.
3. Если `x` – переменная, то добавить пару `(x, y)` к строящемуся НОУ (`environment`). Возврат `environment`.
4. Если `y` – переменная, то добавить пару `(y, x)` к строящемуся НОУ (`environment`). Возврат `environment`.
5. Если `x` или `y` является атомом, то
   Если `x=y`, то Возврат `environment`
   Иначе Возврат `NIL`
6. Если условия п. п. 3, 4, 5 не выполняются, то
   Делать
     Путем унификации первых элементов списков `x` и `y` получить их НОУ – `new-environment`
     Если `new-environment` не пуст (унификация прошла успешно),
     то Делать
       Унифицировать хвосты списков `x` и `y` с учетом унификации голов списков `x` и `y` и достроением НОУ `new-environment`
       Возврат НОУ, построенного в результате унификации хвостов
     Конец
     Иначе Возврат `NIL`
   Конец

Следует заметить, что НОУ представляет собой список пар вида:

`(<переменная> <замещающий терм>).`

Функция `unify` не проверяет вхождение переменной в замещающий терм. Однако ситуация вхождения переменной в замещающий терм является ошибочной.

Функция value предназначена для определения значения переменной. Параметрами функции являются: x – переменная (в виде списка, начинающегося символом ?); environment – связи переменных. Функция возвращает значение переменной, если с переменной связано значение, или имя переменной, если с переменной не связано значение в списке environment. В случае, если x – не переменная, то возвращается само невычисленное x. Текст функции представлен ниже:

```
(defun value (x environment &aux binding)
      (cond ((variable-p x)
             (setq binding (assoc x environment :test #'equal))
             (cond ((null binding) x)
                   (t (value (cadr binding) environment))))
            (t x)))
```

Следует заметить, что даже неконкретизированная переменная может иметь значение – имя другой переменной, иными словами, ссылку на другую переменную. Таким образом, переменные в ассоциативном списке environment могут «сцепляться», т.е. образовывать некий логический список. Функция value, получив в качестве начала данной логической цепи переменную x путем рекурсивных вызовов «раскручивает» данную цепочку и возвращает в качестве своего значения элемент-конец цепочки.

Функция variable-p определяет, является ли ее единственный входной параметр x переменной. Текст данной функции имеет следующий вид:

```
(defun variable-p (x)
      (and x (listp x) (eq (car x) '?)))
```

В соответствии с описанием тела функции, x является переменной, если x является списком и первый элемент этого списка есть символ «?».

Встречающиеся в термах переменные «переименовываются» в уникальные имена функцией rename-variables, текст которой представлен ниже:

```
(defun rename-variables (term list-of-level)
   (cond ((variable-p term) (append term list-of-level))
         ((atom term) term)
         (t (cons (rename-variables (car term) list-of-level)
                  (rename-variables (cdr term) list-of-level)))))
```

Входными данными функции являются: term – терм в списочной форме; list-of-level – уровень. Функция возвращает терм с «переименованными» переменными. Переименование переменной сводится к добавлению к переменной численного значения уровня, что позволяет в других функциях отличать связи одноименных переменных из различных правил. Пример работы функции rename-variables:

```
> (rename-variables '((parent (? x) (? y))) '(0))
(parent (? x 0) (? y 0))
```

Функция print-binding выводит значения связей переменных из списка environment-left, являющегося первым параметром функции. На языке Lisp данная функция определяется как:

```
(defun print-bindings (environment-left environment)
   (cond ((cdr environment-left)
          (cond ((= 0 (nth 2 (caar environment-left)))
                 (prin1 (cadr (caar environment-left)))
                 (princ " = ")
                 (print (value (caar environment-left)
                                           environment))))
          (print-bindings (cdr environment-left) environment))))
```

Функция работает следующим образом. Вначале осуществляется проверка на пустоту хвостовой части печатаемого остатка списка environment-left. Если она не пуста, то проверяется уровень переменной, стоящей первой в списке пар environment-left (переменная определяется предложением caar environment-left). Если он равен нулю, то печатаются имя переменной (стоит на втором месте в списке, представляющей переменную – (cadr(caar environment))), знак «=» и значение переменной, определяемое с помощью функции value. Если хвостовая часть environment-left пуста, то рекурсивно вызывается

функция `print-bindings` по сути дела с пустым списком, что приводит на следующем этапе рекурсии к возврату вторым оператором `cond` значения `NIL` и, таким образом, «затуханию» рекурсии.

Функция `y-or-n-p` предназначена для ввода ответа, подтверждающего (`y`) или не подтверждающего (ответ, отличный от `y`) поиск альтернативного решения Пролог-интерпретатором. Текст функции представлен ниже:

```
 (defun y-or-n-p (prompt)
       (princ prompt)
       (eq (read) 'y))
```

Множество предложений трактуется как программа. Эти предложения образуют множество правил, в котором все использованные отношения перечислены или определены через другие отношения или рекурсивно через самих себя. Для такого множества хорновских предложений можно определить процедурную интерпретацию.

Программа, написанная на Прологе, образуется просто из множества хорновских предложений:

```
(setq программа
  '(((дедушка (? z) (? y))    ; правила
     (родитель (? x) (? z))
     (отец (? z) (? y)))
    ((родитель (? x) (? y))
     (отец (? x) (? y)))
    ((родитель (? x) (? y))
     (мать (? x) (? y)))
    ((отец marta mauno))     ; факты
    ((отец mauno juho))
    ((мать marta leena))
    ((мать mauno anna))))
```

Программа запускается путем передачи ей в качестве вопроса и для принятия решения некоторого предиката, например:

```
 (prolog программа)
Query?(родитель marta (? x))
x = mauno
More?y
x = leena
More?y
Query?
```

Задания:

Модифицировать представленный выше Пролог-интерпретатор таким образом, чтобы реализовались следующие конструкции языка Пролог:

1) анонимные переменные. В языке Пролог анонимные переменные обозначаются через символ _. Анонимные переменные индивидуальны, но доступ к ним исключен.

2) предикат `fail` (отказ), который вызывает при выполнении правила неудачу.

3) арифметические операции: сложение, вычитание, умножение, деление. Операторы могут быть заданы в префиксной форме и представлены в виде термов.

4) операции сравнения: больше, больше или равно, меньше, меньше или равно, равно, не равно. Операторы могут быть заданы в префиксной форме и представлены в виде термов.

5) предикат `cut` (отсечение), используемый для управления обратной трассировкой. Обычно в языке Пролог данный оператор обозначается символом «!». Его действие состоит в том, что переменные, появляющиеся в текущем правиле слева от `cut`, после конкретизации становятся неизменяемыми. Иными словами, запрещается использовать все альтернативные выводы конъюнкции целей, находящихся левее от `cut`.

6) предикат `repeat`, предназначенный для порождения множественных решений в процессе возврата. Этот предикат генерирует бесконечное число циклов повторения поиска с возвратом. Предикат всегда истинен. Всякий раз, как до него доходит перебор, порождается новая ветвь вычислений. Поведение предиката полностью соответствует следующему Пролог-определению:

```
repeat.
repeat:- repeat.
```

7) предикаты `retract` и `assert`, которые соответственно удаляют и добавляют хорновские клозы в базу данных.

8) предикат `call`. Целевое утверждение `call(X)` считается согласованным с базой данных, если попытка доказать согласованность `X` завершается успехом.

9) предикат `not`. Целевое утверждение `not(X)` считается согласованным с базой данных, если попытка доказать согласованность `X` заканчивается неудачей. Целевое утверждение `not(X)` считается несогласованным, если попытка доказать согласованность `X` успешно завершается.

10) дизъюнкция целей. В Прологе для определения дизъюнкции целевых утверждений используется функтор «`;`». Если `X` и `Y` – целевые утверждения, то целевое утверждение `X; Y` согласуется с базой данных, если согласуется `X` или `Y`. В случае предложенного выше Лисп-интерпретатора Пролога необходимо внести коррективы в представление Пролог-программы. Можно ввести дополнительные списки, в которые включаются конъюнкции целей. При этом дизъюнкция будет представлять собой список конъюнкций. Например, прологовское правило

```
F:- (A,B);C.
```

можно записать в виде списка `(F (A B) (C))`.

11) операторы ввода и вывода термов. Для вывода в языке Пролог используется предикат `write`. Если значением переменной `X` является терм, то появление цели `write(X)` вызовет печать этого терма на дисплее. В случае, если переменная `X` неконкретизирована, будет напечатан некоторый числовой уникальный идентификатор, начинающийся сознака «_». Для ввода термов в языке Пролог используется предикат `read`. Если переменная `X` не конкретизирована, то целевое утверждение `read(X)` приведет к вводу следующего терма и этот терм будет присвоен в качестве значения переменной `X`. Если в момент рассмотрения целевого утверждения `read(X)` его аргумент конкретизирован, то попытка доказать согласованность этого целевого утверждения с базой данных вызовет чтение следующего терма и попытку сопоставления его с аргументом, заданным в `read`. Согласованность цели с базой данных зависит от результатов этого сопоставления.

12) предикат `trace`. Выполнение предиката `trace` заключается в установлении режима полной трассировки. Трассировка описывается в терминах четырех видов происходящих событий: CALL (ВЫЗОВ), EXIT (ВЫХОД), REDO (ПЕРЕДЕЛКА), FAIL

(НЕУДАЧА). Событие CALL фиксирует начало попытки Пролога согласовать цель с базой данных. Событие EXIT фиксирует момент, когда некоторая цель только что согласована с базой данных. Событие REDO фиксирует момент, когда система возвращается к цели, пытаясь повторно согласовать ее с базой данных. Событие FAIL фиксирует момент, когда попытка согласовать цель с базой данных заканчивается неудачно.

13) предикат `functor`. Предикат `functor` определен таким образом, что `functor(T,F,N)` означает, что `T` – это терм с функтором `F`, имеющим `N` аргументов.

14) предикат `arg`, используемый для доступа к конкретному аргументу терма. Предикат `arg` определен таким образом, что `arg(N,T,A)` означает, что `N`-й аргумент `T` есть `A`. Предикат `arg` всегда должен использоваться с конкретизированными первым и вторым аргументами.

15) возможность работы со списками. Для представления списков в Прологе обычно используется скобочная форма записи. Она представляет собой заключенную в квадратные скобки последовательность элементов списка, разделенных запятыми. Например, в Прологе допустимы следующие списки:

`[]` – пустой список;

`[a,b,[c,d,e]]` – вложенные списки;

`[a,V1,b,[X,Y]]` – список, включающий переменные.

Работа со списками основана на расщеплении их на голову и хвост. В Прологе введена специальная форма для представления списка с головой X и хвостом Y. Это записывается как `[X|Y]`, где для разделения `X` и `Y` используется вертикальная черта `|`. При реализации работы со списками следует модифицировать форму представления Пролог-программы, предложенную выше. При этом необходимо ввести специальные маркеры для идентификации списков (возможно, маркеры начала, конца, головы, хвоста).

16) предикат «`=..`», используемый для сборки или разборки терма. Целевое утверждение `X=..L` означает, что `L` есть список, состоящий из функтора терма `X`, за которым следуют аргументы `X`. Для реализации данного предиката должны быть реализованы средства работы со списками.

17) модифицировать функцию `unify` таким образом, чтобы она проверяла вхождение переменной в заменяемый терм и в случае вхождения выдавала сообщение об ошибке.

Для определения номеров заданий необходимо выбрать в первом столбце нижеследующей таблицы свой номер по списку группы и выполнить задания из второго столбца.

Вариант	*Задания*
1.	1
2.	2
3.	3
4.	4
5.	5
6.	6
7.	7
8.	8
9.	9
10.	10
11.	11
12.	12
13.	13
14.	14
15.	15
16.	16
17.	17
18.	1
19.	2
20.	3
21.	4
22.	5
23.	6
24.	7
25.	8
26.	9
27.	10
28.	11
29.	12
30.	13
31.	14
32.	15