Министерство образования и науки Российской Федерации

Севастопольский государственный университет

Программирование на языке Лисп для систем искусственного интеллекта

Методические указания к лабораторным работам по дисциплине
“Методы и системы искусственного интеллекта”
для студентов дневной и заочной форм обучения
направления 09.03.02 — “Информационные системы и технологии”

Севастополь
2015

УДК 004.8 (075.8)

Программирование на языке Лисп для систем искусственного интеллекта: методические указания к лабораторным работам по дисциплине “Методы и системы искусственного интеллекта” для студентов дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии”/ СевГУ ; сост. **В. Н. Бондарев**. **—** Севастополь : Изд-во СевГУ, 2015. **—** 68с.

Методические указания предназначены для проведения лабораторных занятий по дисциплине “Методы и системы искусственного интеллекта” для студентов дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии”. **Целью методических указаний** является обучение студентов практическим навыкам написания и отладки функциональных программ на языке Лисп при построении интеллектуальных систем различного назначения

Методические указания составлены в соответствии с требованиями программы по дисциплине “Методы и системы искусственного интеллекта” для студентов дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии” и утверждены на заседании кафедры Информационных систем (протокол № от 2015 г.)

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент: Кожаев Е.А., к.т.н., доцент кафедры информационных технологий и компьютерных систем

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Дисциплина «Методы и системы искусственного интеллекта» относится к числу нормативных дисциплин цикла профессиональной подготовки. Целью преподавания дисциплины является обучение студентов основным принципам построения компьютерных систем, способных моделировать интеллектуальную деятельность человека. Программой дисциплины предусматривается изучение следующих разделов:

- основные понятия и определения;
- способы представления задач и поиск решений;
- представление знаний;
- основные модели вывода;
- языки систем искусственного интеллекта — Лисп и Пролог;
- экспертные системы и обработка естественного языка.

Лабораторный практикум по дисциплине состоит из двух частей. Первая часть включает решение задач искусственного интеллекта (ИИ) на языке Лисп, а вторая — на языке Пролог.

Эти языки поддерживают две важные парадигмы программирования, используемые при построении СИИ, — функциональное и логическое программирование. Функциональное программирование реализуется средствами языка Лисп, а логическое — языка Пролог. Важная характеристика этих языков — ориентация не на числовую обработку, а на обработку символов. Поэтому основной структурой данных, поддерживаемой в этих языках, являются списки.

В языке Лисп списки используются и для представления обрабатываемых данных, и для представления самих программ. Пролог базируется на языке исчисления предикатов и также имеет удобные средства для обработки списков. Поддержка этими языками более мощных абстракций данных значительно повышает эффективность труда разработчика систем ИИ (в десять и более раз [1]). По этим причинам значительная часть прототипов программ с элементами ИИ реализуется на языках Лисп или Пролог. И сегодняшний исследователь ощущает себя неполноценным, если он не обладает основательным знакомством как с языком Лисп, так и с языком Пролог.

В настоящих методических указаниях рассматриваются: основные функции языка Лисп; концепция функционального программирования и способы организации циклических вычислительных процессов в языке Лисп; применение списков и функций высших порядков для организации и хранения базы данных (БД) или базы фактов о предметной области; поиск записей в базе фактов с помощью алгоритма сопоставления с образцом и его применение для организации доступа к базе на ограниченном подмножестве естественного языка; способы представления и стратегии поиска решений задач в пространстве состояний.

Варианты заданий студенты выбирают самостоятельно по номеру индивидуального плана (зачетной книжки). Номер варианта для каждой лабораторной работы определяется как остаток от деления двух последних цифр номера плана

(зачетки) на 25. Например, две последние цифры плана 37. Остаток от деления 37 на 25 равен 12. Следовательно, номер варианта задания 12.

Студенты заочной формы обучения выполняют данные лабораторные в два этапа. **На первом этапе**, выполняемом самостоятельно, студенты оформляют решение соответствующих вариантов заданий в виде контрольной работы. Контрольная работа оформляется в тетради или на листах белой бумаги формата А4. Для каждого задания в контрольной работе должно быть приведено: вариант задания; краткие теоретические сведения по сути решаемой задачи; программа на языке Лисп, решающая поставленную задачу; подробное описание программы. Срок сдачи контрольной работы студентами заочниками — 13 неделя семестра.

На втором этапе, выполняемом в течение лабораторно-экзаменационной сессии, студенты в лабораториях кафедры осуществляют ввод и отладку программ на ЭВМ, демонстрируют их выполнение, анализируют полученные результаты и отвечают на вопросы преподавателя.

Лабораторные работы ориентированы на применение языка ANSI Common Lisp. В частности, Steel Bank Common Lisp (SBCL) [2].

1. ЛАБОРАТОРНАЯ РАБОТА № 1 «ИССЛЕДОВАНИЕ БАЗОВЫХ ФУНКЦИЙ ЯЗЫКА ЛИСП»

1.1. Цель работы

Изучение технологии подготовки и выполнения Лисп-программ в выбранной интегрированной среде, исследование свойств базовых функций обработки списков, а также способов описания и вызова нерекурсивных функций в языке программирования Лисп.

1.2. Краткие теоретические сведения

1.2.1. Основные понятия языка Лисп

Одно из базовых понятий языка Лисп — символ. *Символы* обозначают объекты, которыми манипулирует программа, и в этом смысле соответствуют именам переменных алгоритмических языков программирования. *Имя символа* состоит из букв латинского алфавита (в языке не различаются прописные и строчные буквы), цифр и специальных знаков (+| -| *| %| &| _| !| @| <| >| =| ?| /| {| }| ^| ~). Порядок следования указанных знаков может быть произвольным. Поэтому имена символов могут начинаться с цифры или спецзнака. Например: **1A, %1A, *col13**. Для приписывания символам значений используют специальные функции. Изначально символы значений не имеют.

Лисп поддерживает работу с целыми, рациональными и вещественными *числами*. Примеры соответствующих чисел: **537, 2/3, 2.0E-3**.

Символы и числа представляют простейшие объекты языка Лисп, поэтому их называют атомами. *Атом* определяется с помощью следующей формулы:

атом::=*символ* | *число* | ()

Пара скобок **( )** в языке Лисп имеет двойное значение. С одной стороны, она обозначает пустой список, а с другой — логическое значение *ложь*. Для обозначения логического значения истина (true) в Лиспе используется символ **T**. Ложь или пустой список также обозначаются символом **NIL**.

Из атомов можно построить основную структуру данных языка Лисп — *s-выражение*, которое определяется с помощью следующей рекурсивной формулы:

s-выражение ::= *атом* | (*s-выражение* . *s-выражение*)

Примеры s-выражений:

gruppa
(gruppa1 . gruppa2)
((gruppa1 . gruppa2) . gruppa3)

Обратите внимание, что точка отделяется от знаков, стоящих слева и справа, пробелом. Конструкция (*s-выражение* . s-*выражение*) называется *точечной парой*.

Удобно точечные пары изображать в виде диаграмм. На рисунке 1.1 показана диаграмма для точечной пары **(A . B)**. Такая диаграмма дает наглядное представление о форме хранения точечной пары в памяти ЭВМ и называется лисповской ячейкой.

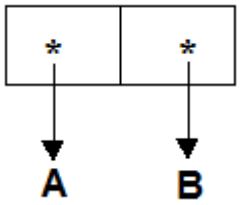

Рисунок 1.1 — Лисповская ячейка

Эта ячейка состоит из двух полей. В первом поле хранится указатель на первый элемент точечной пары, а во втором поле — указатель на второй элемент.

В Лиспе выделяют специальный тип s-выражений, называемый списком. *Список* определяется следующим образом:

список ::= **NIL** | (*s-выражение* . *список*)

Например, s-выражения **(A .(B .(C .(D . NIL))))** и **((A . B) . ((C . D) . NIL))** — списки. Указанные списки также можно также представить в виде последовательности элементов, отделяемых пробелом и заключенных в круглые скобки: **(A B C D), ((A . B) . (C . D))**. Список, заданный в виде последовательности элементов, всегда можно представить в точечной записи. Например, следующие записи эквиваленты:

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

Для перехода от точечной записи выражения к записи его в форме последовательности элементов точку, стоящую перед открывающейся круглой скобкой заменяют пробелом, удалив упомянутую открывающуюся и парную закрывающуюся скобку.

1.2.2. Базовые функции и предикаты языка Лисп

S-выражение, которое может быть вычислено Лисп–интерпретатором, называется *формой. Вызов функции* выполняется по ее имени из формы, записанной в виде списка:

$(f_n \ \ a_1 \ \ a_2 \ \ \ldots \ \ a_n)$,

где f_n — имя вызываемой функции; $a_1, a_2 \ldots a_n$ — аргументы функции, которые задаются вычисляемыми s- выражениями. Например, запись **(* 2 3)** соответствует вызову функции умножения и передает ей в качестве аргументов значения **2** и **3**.

Для доступа к различным элементам списков и конструирования из этих элементов других списков в языке Лисп используются следующие *базовые функции*: **CAR, CDR** и **CONS**.

Функция **CAR** выделяет первый элемент списка или точечной пары. Например,

(car '(a b)) → **A**

(car '(a . b)) → A

Здесь апостроф, обозначает вызов специальной формы **QUOTE** , которая блокирует вычисление значений s-выражений **(a . b)** или **(a b)**. Если этого не сделать, то Лисп-интерпретатор (**EVAL**) предпримет попытку вычислить эти s-выражения и вернет сообщение об ошибке **UNDEFINED-FUNCTION** (неопределенная функция).

Для выделения хвоста списка или второго элемента точечной пары применяется функция **CDR**. Например,

(cdr '(a b)) → (B)
(cdr '(a . b)) → B

Для выделения произвольных элементов списка можно использовать *композиции функций* **CAR** и **CDR**. Например,

(car (car '((a b) (b c)))) → A
(car (cdr (cdr '(1 2 3)))) → 3

Композиции функций **CAR** и **CDR** встречаются довольно часто, поэтому для них введены специальные имена:

(car (car x)) ↔ (caar x),
(car (cdr x)) ↔ (cadr x),
(cdr (car x)) ↔ (cdar x),
(cdr (cdr x)) ↔ (cddr x),
…
(cdr (cdr (cdr (cdr x)))) ↔ (cddddr x).

Для создания точечной пары из s-выражений в Лиспе применяется функция **CONS** (конструктор):

(cons 'a 'b) → (A . B)

Конструирование списков с помощью функции **CONS** осуществляется последовательными вызовами:

(cons ('x (cons 'y (cons 'z NIL)))) → (X Y Z)

Кроме трех базовых функций **CAR, CDR** и **CONS,** при обработке списков часто используют *базовые предикаты* **ATOM** , **EQ, EQUAL**.

Предикат **ATOM** проверяет, является ли аргумент атомом. Например:

(atom 'a) → T
(atom nil) → T

Предикат **EQ** сравнивает значения двух своих аргументов. При этом выясняется, представлены ли аргументы в памяти одной и той же физической структурой данных. Предикат **EQ** возвращает значение **T**, если его аргументы — символы, и ссылаются на одну и ту же физическую структуру. Например,

(eq 'a 'a) → T
(eq '(a b c) '(a b c)) → NIL

Так как функция **EQ** требует, чтобы ее аргументы были символами, то при сравнении чисел могут получаться неожиданные результаты. Например,

(eq 5.0 5.0) → NIL

В Лиспе имеется предикат **EQL**, аналогичный **EQ**, но дополнительно позволяющий сравнивать числовые значения одинаковых типов. Например,

(eql 5.0 5.0) → T

Сравнение объектов Лисп-программы на уровне абстрактных структур выполняется с помощью предиката **EQUAL**. Данный предикат обобщает предикат **EQL** и может сравнивать символы, однотипные числа, строки, списки:

(equal 'a 'a) → T
(equal '(a b c) '(a b c)) → T
(equal "lisp" "lisp") → T
(equal 5.0 5.0) → T

Однако

(equal 5.0 5) → NIL

1.2.3. Дополнительные функции и предикаты работы со списками

Помимо примитивных базовых функций, в Лиспе для работы со списками используется широкий спектр функций более высокого уровня.

Функции для выделения 1-го, 2-го, …, 10-го элемента списка:

FIRST— эквивалента **CAR**, возвращает первый элемент вписка;

SECOND — возвращает второй элемент списка;

THIRD — возвращает третий элемент списка и т.д.;

TENTH — возвращает десятый элемент списка. Имена указанных функций соответствуют порядковым числительным английского языка.

В Лиспе имеется ряд других полезных дополнительных функций для обработки списков:

REST — эквивалентна **CDR**, возвращает хвост вписка;

LAST — выделяет последний элемент списка (результат – список);

NTH — выделяет n-й элемент списка;

SUBST — выполняет замену элементов в списке;

BUTLAST— выделяет список без последних элементов;

MEMBER — проверяет принадлежность элемента списку;

LENGTH — вычисляет длину списка;

REVERSE — выполняет обращение списка;

ADJOIN — добавляет элемент в множество, представленное списком;

REMOVE — удаляет элемент из списка;

APPEND — создаёт список путем конкатенации своих аргументов;

LIST — создаёт список из произвольного количества своих аргументов.

Синтаксис вызова этих функций и любых других функций, а также назначение функций обычно отображается в среде программирования при наборе имени функции. Более детальные сведения приведены в [1].

В Лиспе имеется большая группа предикатов, классифицирующих типы значений: **NULL, NUMBER, INTEGERP, FLOATP, RATIONALP, SYMBOLP, LISTP** и др.

Предикат **NULL** позволяет установить, представляет ли его аргумент пустой список.

Предикат **NUMBERP** устанавливает, является ли его аргумент числом.

Предикаты **INTEGERP, FLOATP, RATIONALP** позволяют выяснить, является ли значение аргумента целым, вещественным или рациональным числом.

Предикаты **SYMBOLP, CONSP, LISTP** позволяют определить принадлежность своего аргумента множеству символов, точечных пар или списков:

```
(symbolp '(a b c)) → NIL
(consp '(a b c)) → T
(listp '(a b c)) → T
(consp '( )) → NIL
(listp '( )) → T
```

Для проверки свойств числовых значений в Лиспе используются предикаты: **ZEROP** (равенство нулю), **ODDP** (нечетность), **EVENP** (четность), **MINUSP** (отрицательность).

Рассмотренные предикаты обычно применяются при построении тест-форм условных выражений.

1.2.4. Арифметические функции и предикаты

Хотя Лисп и предназначен в основном для обработки символов, он содержит широкий спектр арифметических функций. Аргументы этих функций должны иметь числовые значения. Ниже приведены примеры вызовов функций, выполняющих основные арифметические операции:

```
(+ 2 3)          → 5      ; сложение 2-х аргументов
(+ 3 1 2 3)      → 9      ; сложение 2-х и более аргументов
(+ 9)            → 9      ; унарный +
(- 13 1 2 3)     → 7      ;вычитание 2-х и более аргументов
(- - 9)          → 9      ;
(* 2 1.5)        → 3.0    ; умножение 2-х аргументов
(* 2 2 3)        → 12     ;умножение 2-х и более аргументов
(/ 7 2 2)        → 7/4    ;деление целочисленных аргументов
(/ 7 2 2.0)      → 1.75   ;деление с преобразованием
                          ;к вещественному значению
(/ 5.0)          → 0.2    ; обратное значение
(/ 2)            → 1/2    ;
(min 2 3 1)      → 1      ; минимум
(max 2 3 1)      → 3      ; максимум
(abs - 5)        → 5      ; абсолютное значение
(rem 17 4)       → 1      ; остаток от деления.
```

Для сравнения скалярных чисел в Лиспе используются предикаты, имена которых задаются знаками отношений (**=, /=** (не равно), **<, >, <=, >=**). Эти предикаты могут иметь произвольное количество аргументов.

В Лиспе имеется также большой набор элементарных математических функций: **EXP, EXPT, LOG, SQRT, SIN, COS, TAN, ASIN, ACOS, ATAN** и др. Аргументы тригонометрических функций задаются в радианах.

Функции **TRUNCATE, ROUND, FLOAT** выполняют соответственно отбрасывание дробной части числа, округление и приведение значения к вещественному виду.

1.2.5. Присваивание значений

Переменные в языке Лисп обозначаются с помощью символов. Чтобы присвоить символу значение, применяется функция **SET.** Первый аргумент этой функции представляется формой, вычисляющей символ, которому присваивается значение, а второй — формой, определяющей приписываемое значение, например:

(set 'x '(a b c)) $\rightarrow$ **(A B C)**
(set (car x) '(d e)) $\rightarrow$ **(D E)**

Здесь **X** присваивается значение **(A B C)**, а **A** значение — **(D E).**

Символу можно также присвоить значение с помощью вызова специальной формы **SETQ** или макроса **SETF**. Формат вызова специальной формы **SETQ**:

(setq {*символ форма* }***)**

Здесь запись {}* обозначает повторение конструкции, записанной в фигурных скобках ноль или более раз. В отличие от **SET**, форма **SETQ** автоматически блокирует вычисление значений символов и может выполнять одновременно несколько присваиваний. Поэтому ниже в примерах нет необходимости в блокировке вычислений форм, представленных именами символов **X, Y, Z**:

(setq x '(a b c)) $\rightarrow$ **(A B C)**
(setq y 12 z (+ y 10)) $\rightarrow$ **22**

Присваивание значений символам в форме **SETQ** выполняется последовательно слева направо, т.е. в момент присваивания значения символу **Z**, символ **Y** уже получил значение **12**. Вызов **SETQ** возвращает последнее присвоенное значение или **NIL**, если аргументы формы **SETQ** не были заданы.

Макрос **SETF** (set field) является обобщенной формой, используемой для присваивания значений самым различным объектам. Формат вызова **SETF**:

(setf {*место форма* }*)

Здесь *место* — вычисляемое выражение, позволяющее определить лисповскую ячейку памяти, значение которой необходимо изменить.

Примеры:

(setf a 8.8 d 2.0) $\rightarrow$ **2.0**

(setq x '(a b c)) → (A B C)
(setf (car x) 2 (cadr x) 3) → 3
x → (2 3 C)

В первом примере символ **A** получает значение **8,8** , а символ **d** — **2,0**. В качестве результата вызова возвращается последнее присвоенное значение — **2,0**. Второй вызов **SETF** , по сути, меняет в списке **X** = **(A B C)** вхождение **A** на **2**, а **B** на **3**. Т.е. после вызова **SETF** символ **X** будет иметь значение **(2 3 C**). Обработка пар аргументов макроса **SETF** выполняется последовательно слева направо. Макрос **SETF** позволяет изменять не только значение символа, но и другие его атрибуты.

1.2.6. Определение функций

В Лиспе различают описания *именованных и неименованных (анонимных) функций*. Для описания неименованных функций применяется *лямбда-выражение,* определяемое с помощью формулы:

(**lambda** ({*переменная*}*){*форма*}*)

Конструкция ({*переменная*}*) называется *лямбда-списком* и представляет собой список формальных параметров, которые соответствуют аргументам функции. Последовательность форм в лямбда выражении — {*форма*}* — определяет правило формирования значения функции и называется *телом* функции. В качестве результата возвращается значение последней вычисленной формы.

Пусть требуется определить функцию, вычисляющую x^3. Тогда лямбда-выражение будет иметь вид:

(lambda (x) (* x x x))

Чтобы применить неименованную функцию, определенную лямбда-выражением, к некоторым аргументам, необходимо в форме вызова функции подставить лямбда-выражение вместо имени функции, например:

((lambda (x) (* x x x)) 2) → 8

Такие вызовы называются *лямбда-вызовами.*

Для определения именованных функций в Лиспе применяется форма **DEFUN**:

(**defun** *имя лямбда-список* {*форма*}*)

Результатом вызова формы **DEFUN** является имя определяемой функции. Например:

(defun stepen3 (x) (* x x x)) → STEPEN3
(defun spisok (x y) (cons x (cons y nil))) → SPISOK

Теперь функции **STEPEN3** и **SPISOK** можно вызывать по имени:

(stepen3 2) → 8
(spisok 'a 'b) → (A B)

Отметим, что имя функции представляется символом. Поэтому можно говорить о том, что **DEFUN** устанавливает связь между символом и определением функции. Таким образом, с одним и тем же символом можно связать не только значение, но и функцию.

При задании формальных параметров в форме **DEFUN** или в лямбда-списке выделяют :

- обязательные параметры;
- необязательные параметры (**optional**);
- остаточные параметры(**rest**) — список значений тех фактических параметров, которые остались свободными, т.е. им не были назначены соответствующие формальные параметры;
- ключевые параметры (**key**), которые при вызове идентифицируются с помощью своего имени, перед которым ставят двоеточие.

Рассмотренные выше примеры определения функций содержали только обязательные параметры. Если необходимо в списке формальных параметров указать параметры, отличные от обязательных, то перед ними записывают соответствующие ключевые слова: **&OPTIONAL, &REST, &KEY** [1]. Например,

```
(defun func (x &optional (y (+x 5))) (+x y))
```

Здесь **X** — обязательный параметр, а **У** — необязательный. Если параметр **У** не будет задан при вызове функции, то он получит значение по умолчанию, равное **X+5**.

1.2.7. Связывание и область действия переменных

Процесс назначения переменной определенного значения на определенном участке программы называется *связыванием* (binding). Для установления *локальной связи* используется форма **LET**:

(**let** ({(*переменная форма-инициализатор*)}*){ *форма* }*)

Здесь *форма-инициализатор* задает новое значение переменной, которое доступно только в переделах **LET**. По окончании **LET**-вызова восстанавливается старое значение переменной в соответствии со связями, которые имела переменная во внешнем окружении **LET**. Значение, возвращаемое **LET**-вызовом, соответствует последней вычисленной форме, входящей в **LET**.

Связывание переменных внутри формы **LET** выполняется параллельно. Если необходимо, чтобы связи локальных переменных устанавливались последовательно, то применяется форма **LET*.**

Переменные, имеющие локальные связи, в Коммон Лиспе называются *лексическими* или *статическими*. Область действия такой переменной ограничивается той формой, в которой переменная определена, т.е. связи лексической переменной действительны в некотором фиксированном текстуальном (лексическом) окружении.

Альтернативой лексическим переменным являются *динамические* или *специальные* переменные. Область действия такой переменной определяется не тек-

стуальным окружением, в котором переменная используется, а динамическим окружением, возникающим в процессе выполнения программы. В Коммон Лиспе переменные по умолчанию являются лексическими, а динамические (специальные) переменные **должны быть объявлены** одним из способов (последствия использования необъявленных переменных в стандарте ANSI неопределены):

(**defvar** *имя* [*начальное значение*])
(**defparameter** *имя* [*начальное значение*])
(**declare (special** *имя*)**)** ; внутри let или defun

Чтобы отличать такие переменные, им дают особые имена. Обычная практика состоит в том, чтобы такие имена начинались и заканчивались знаком “ * ”. Значение динамической переменной определяется по последней связи. Все глобальные переменные в Коммон Лиспе являются динамическими. Значения динамических переменных определяются не статическими связями на этапе компиляции, а динамическими, возникающими во время выполнения программы. Лексические и динамические переменные ведут себя по-разному в замыканиях.

1.2.8. Условные вычисления

Для выполнения условных вычислений в Лиспе могут применяться формы **COND, IF, WHEN, UNLESS** и др. [1]. Форма **COND** позволяет выбрать одну из n-ветвей алгоритма:

```
(cond
    (тест-форма1   форма-11  форма-12 … )      ;клоз-1
    (тест-форма2   форма-21  форма-22 … )      ;клоз-2
      …
    (тест-формаi   форма-i1  форма-i2 … )      ;клоз-i
      …
    (тест-формаN  форма-N1  форма-N2…))        ;клоз-N
```

Значение, возвращаемое формой **COND**, определяется следующими правилами. Поочередно для каждого клоза (англ. clause — предложение, пункт) проверяются значения тест-форм. Если тест-форма имеет значение **T**, например *тест-формаi*, то вычисляются формы клоза-i, т.е. *форма-i1*, *форма-i2* и т.д. Значение последней вычисленной формы выбранного клоза и будет представлять результат, возвращаемый формой **COND** в точку вызова. Если ни одна из тест-форм не получила значение T, то результатом вызова формы **COND** будет **NIL**.

Обычно в качестве последней тест-формы используется символ **T**. Тогда формы, входящие в последний клоз, будут вычисляться, когда ни одна из предшествующих тест-проверок не выполнится. Рассмотрим простой пример:

```
(cond ((eq x 'Bern) 'Switzerland)
      ((eq x 'Paris) 'France)
      ((eq x 'Berlin) 'Germany)
      ((eq x 'Kiev) 'Ukraine)
      (t 'unknown))
```

Здесь тест-формы реализованы с помощью предиката **EQ**. Если значением **X** является атом, представляющий название одной из указанных столиц, то в качестве результата возвращается название соответствующего государства. Во всех остальных случаях возвращается значение **UNKNOWN**.

В рассмотренном примере выбиралась одна из пяти ветвей алгоритма. Когда требуется выбрать одну из двух ветвей алгоритма, то используют специальную форму **IF**. Формат вызова формы:

(**if** *тест-форма then-форма* [*else-форма*])

Если результатом вычисления тест-формы является значение **T**, то **IF** возвращает значение, определяемое *then-формой*, иначе – *else-формой*, которая может отсутствовать. Например, вычисление абсолютного значения X можно реализовать в виде:

(if (>= x 0) x (- x))

1.3. Варианты заданий

Описать на языке Лисп функцию f(x y z) от трёх аргументов, которая формирует из своих аргументов список и выполняет его обработку в соответствии с вариантом задания, указанным в таблице 1.1.

Таблица 1.1 — Варианты заданий

Вариант	Задание
1.	Проверить, является ли первый элемент подсписком, и найти его длину. Если первый элемент — подсписок, то вернуть исходный список при условии, что длина подсписка меньше либо равна 2, иначе вернуть подсписок без последнего элемента. Иначе вернуть исходный список без первого элемента.
2.	Проверить, является ли второй элемент списка вещественным или рациональным числом. Если является, то поменять его на квадрат этого числа. Если не является, то вернуть исходный список без последнего элемента.
3.	Проверить, является ли третий элемент списка целым числом или символом. В случае числового значения заменить его на его квадрат. В случае символа вернуть имя символа.
4.	Проверить, является ли первый элемент списка четным числом. Если является, то вернуть исходный список, выполнив замену первого элемента на ближайшее большее нечетное число, иначе удалить первый элемент.
5.	Проверить, является ли второй элемент списка отрицательным числом. Если не является, то вернуть исходный список без последнего элемента, иначе — без первого элемента.
6.	Проверить, является ли третий элемент списка рациональным числом. Если является, то вернуть исходный список без этого элемента, иначе

	поменять местами первый и третий элементы исходного списка.

Продолжение таблицы 1.1

Вариант	Задание
7.	Найти среднее арифметическое первого и второго элементов списка и заменить значения этих элементов средним арифметическим.
8.	Найти среднее арифметическое первого и третьего, первого и второго элементов списка и вернуть список из этих средних значений.
9.	Найти произведение первого и второго, второго и третьего элементов списка и заменить первый и второй элементы исходного списка значениями произведений.
10.	Проверить, является ли хотя бы один из элементов списка числом. Если является, то вернуть исходный список без первого элемента, иначе поменять местами первый и третий элементы исходного списка.
11.	Проверить, является ли хотя бы один из элементов списка списком. Если является, то вернуть этот список без первого элемента, иначе поменять местами первый и второй элементы исходного списка.
12.	Заменить первый элемент списка на Bilbo, если он является четным числом, иначе поменять местами первый и второй элементы исходного списка.
13.	Заменить второй элемент списка на Baggins, если все элементы списка символы, иначе вернуть список без последнего элемента.
14.	Проверить, является ли второй элемент списка отрицательным нечетным числом. Если является, то вернуть исходный список без последнего элемента, иначе поменять местами первый и второй элементы исходного списка.
15.	Проверить, кратны ли все элементы списка 3. Если кратны, то вернуть список результатов делений на 3, иначе поменять местами второй и третий элементы исходного списка.
16.	Проверить, являются ли элементы списка отрицательными числами. Если являются, то вернуть исходный список без второго элемента, иначе поменять местами первый и второй элементы исходного списка.
17.	Проверить, является ли элементы списка четными числами. Если являются, то вернуть исходный список без двух последних элементов, иначе заменить первый и второй элементы исходного списка квадратами их значений.
18.	Проверить, является ли первый элемент списка строкой или символом. Если является, то вернуть исходный список без первого элемента, иначе поменять местами первый и второй элементы исходного списка.
19.	Проверить, является ли второй элемент списком. Если является, то удалить его из исходного списка и добавить все его элементы, кроме первого, в конец исходного списка.
20.	Проверить, является ли третий элемент списка целым или вещественным числом. Если является, то создать список из трех элементов: первый элемент — знак числа; второй — модуль числа; третий — квадрат

	числа.

Продолжение таблицы 1.1

Вариант	Задание
21.	Проверить, делится ли первый элемент списка на 3 и 5 без остатка. Если условие выполняется, то заменить второй и третий элементы на 3 и 5.
22.	Проверить, кратен ли второй элемент списка пяти или семи. Если условие выполняется, то заменить второй элемент на 35, иначе вернуть исходный список без второго элемента
23.	Для списка из числовых элементов проверить, является ли третий элемент списка результатом возведения в степень первого элемента с показателем, равным второму элементу. Если является, то вернуть T, иначе — NIL.
24.	Для списка из числовых элементов проверить, является ли первый элемент списка квадратом третьего элемента. Если является, то вернуть исходный список без второго элемента.
25.	Если первый и последний элементы исходного списка — символы, то вернуть список из первого и последнего элементов, в противном случае вернуть исходный список, из которого удален третий элемент.

1.4. Порядок выполнения лабораторной работы

1.4.1. Изучить среду программирования на языке Лисп по методическим указаниям [2].

1.4.2. Изучить пункт 1.2 настоящих методических указаний и выполнить все приведенные в этом разделе примеры в интерактивном режиме в окне **REPL** среды программирования [2].

1.4.3. Определить на языке Лисп функцию в соответствии с вариантом задания. Определение выполнить различными способами, применяя базовые функции обработки списков (см. п. 1.2.2) и дополнительные функции (см. п.1.2.3).

1.4.4. Создать в среде программирования Лисп-проект в соответствии с методическими указаниями [2], содержащий подготовленные определения функций и вызовы функций при различных значениях аргументов, в том числе и не допустимых.

1.4.5. Выполнить отладку проекта.

1.4.6. Зафиксировать результаты выполнения функций и ошибки, возникающие при недопустимых значениях аргументов (в виде экранных копий).

1.4.7. Объяснить результаты и локализовать вызовы функции, порождающие ошибки.

1.4.8. Исследовать отдельно каждую базовую и дополнительную функцию, входящую в определения функций, выполненных в п.1.4.3. Привести примеры правильных и неправильных вызовов.

1.5. Содержание отчета

Цель работы, вариант задания, описание определений функций в соответствии с вариантом задания, описание тестовых примеров, результаты выполнения функций проекта, результаты выполнения отдельных базовых и дополнительных функций, входящих в определения функций в соответствии с вариантом задания, выводы.

1.6. Контрольные вопросы

1.6.1. Определите следующие основные понятия языка Лисп: символ, атом, s-выражение, точечная пара, список.

1.6.2. Представьте список **((a b) (c d) (e f))** в точечной форме записи.

1.6.3. Изобразите список **((a b) (c d) (e f))** в виде рисунка с использованием понятия лисповской ячейки.

1.6.4. Сформулируйте правила интерпретации, используемые в Лиспе.

1.6.5. Назовите и объясните базовые функции обработки списков языка Лисп.

1.6.6. Назовите и объясните дополнительные функции обработки списков языка Лисп.

1.6.7. Назовите и объясните основные арифметические функции языка Лисп.

1.6.8. Назовите и объясните основные предикаты классифицирующие типы значений.

1.6.9. Назовите и объясните предикаты сравнения **EQ, EQL, EQUAL** и =.

1.6.10. В чем отличие функций **CONS, LIST** и **APPEND**?

1.6.11. Объясните основные формы для присваивания значений символам и приведите примеры.

1.6.12. Как определить неименованную функцию в языке Лисп? Приведите пример.

1.6.13. Как определить именованную функцию в языке Лисп? Приведите пример.

1.6.14. Определите на языке Лисп функцию, возвращающую список, составленный из двух ее аргументов.

1.6.15. Что такое связывание? Как устанавливаются локальные связи?

1.6.16. Что понимают под статическими (лексическими) и динамическими (специальными) переменными?

1.6.17. Как объявить динамическую переменную?

1.6.18. Объясните условную форму **COND**. Приведите пример.

1.6.19. Объясните условную форму **IF**. Приведите пример.

2. ЛАБОРАТОРНАЯ РАБОТА № 2 «СРАВНЕНИЕ ИТЕРАЦИОННОГО И РЕКУРСИВНОГО МЕТОДОВ РЕШЕНИЯ ЗАДАЧ»

2.1. Цель работы

Исследование способов организации циклических вычислений в языке Лисп с помощью итерационного и рекурсивного методов, сравнение указанных методов по вычислительной эффективности и выразительности, получение практических навыков работы со списочными структурами.

2.2. Краткие теоретические сведения

2.2.1. Функциональное программирование

Язык Лисп относится к языкам *функционального программирования*. В основе функционального программирования лежит идея, заключающаяся в том, что в результате каждого действия возникает значение. Полученное значение становится аргументом следующих действий, и т.д., конечный результат возвращается пользователю. Действия оформляются в виде функций, которые могут включать управляющие условные структуры, и вложенные, часто вызывающие самих себя (рекурсивные), вызовы функций. В функциональном программировании нет присваиваний, передач управлений. При таком подходе к программированию рекурсия является единственным способом организации повторяющихся вычислений.

Большинство языков императивного программирования позволяют функциям изменять значения глобальных переменных, т.е. создавать *побочные эффекты*. ***В функциональном программировании оператор присваивания не используется, объекты программы нельзя изменять и уничтожать путем переприсваивания, можно только создавать новые путем декомпозиции и синтеза из существующих объектов.*** При этом программист избавляется от необходимости явного управления памятью. Память для новых объектов выделяется автоматически. Для очистки памяти от ненужных структур используется автоматический *сборщик мусора* (**gc** — garbage collector).

В функциональном программировании функции не должны создавать побочные эффекты. Из этого следует ещё одно преимущество функциональных языков — *параллелизм вычислений*. Действительно, если все функции для вычислений используют только свои собственные параметры, то можно вычислять независимые функции в произвольном порядке или параллельно.

Язык Коммон Лисп, рассматриваемый в методических указаниях, не является «чистым» языком функционального программирования. Для повышения эффективности вычислений в нем имеется большой набор императивных конструкций, допускающих присваивание и разрушение структур в памяти, организацию итерационных циклов.

2.2.2. Итерационные циклические вычисления

Для организации повторяющихся вычислений в Лиспе имеется ряд форм, позволяющих определить циклический вычислительный процесс. К таким формам относят формы **LOOP , DO, DOLIST, DOTIMES**.

Простейший цикл можно организовать вызовом формы **LOOP**:

(**loop** {*форма*}*)

Здесь конструкция {*форма*}* образует тело цикла. Цикл завершается, когда управление, в одной из форм, передается макровызову возврата **RETURN**. При этом **RETURN** вычисляет значение, возвращаемое в точку вызова **LOOP**.

Пример. Вычисление суммы чисел от 1 до n:

```
(defun nsum (n)
      (let ((sum 0))
            (loop (cond ((zerop n) (return sum))
                        (t (setf sum (+ sum n)) (setf n (- n 1)) ) ))))
```

Макроформа **DO** позволяет задавать: локальные переменные цикла, их начальные значения и правила обновления; условие окончания цикла. Общий формат вызова формы **DO**:

```
(do ({({переменная} [начальное_значение  [обновление] ] )}*)
    (условие_окончания {форма}*)
    {форма}*)              ; тело цикла
```

Первый аргумент **DO**-вызова — список, состоящий из подсписков. Элементами подсписков являются: переменные цикла; формы, задающие начальные значения переменных; выражения, определяющие правила обновления переменных. Если начальное значение переменной не задано, то оно равно **NIL**. Если не задана форма обновления, то значение переменной может обновляться в теле цикла.

На каждой итерации вычисляется условие окончания цикла. Когда условие окончания цикла получает значение “истина“, то последовательно вычисляются формы, входящие во второй список **DO**-вызова. Значение последней вычисленной формы и будет результатом. Если условие окончания цикла ложно, то вычисляются формы, образующие тело цикла. Присваивание значений переменным цикла в **DO**-вызове выполняется параллельно. Если необходимо последовательное присваивание, то применяется макровызов **DO***.

Определим функцию **MY-INTERSECTION**, которая формирует список общих элементов двух списков с помощью **DO***:

```
(defun my-intersection (a b)
      (do* ( (aa a (rest aa))          ; управление переменной aa
             (e (first a) (first aa))  ; управление e – элемент списка a
             (inter ( )) )             ; управление  inter – список общих эл-тов
           ( (null aa) inter)          ; выход из  цикла и возврат inter
           (if (member e b)            ; если e входит в список b
                 (setf inter (cons e inter))) )) ; то добавить e в inter
```

В цикле сначала вычисляется значение переменной **AA**, а затем **E**. При этом переменная **E** соответствует первому элементу очередного хвоста списка **AA**, т.е. **E** — это очередной элемент входного списка, представленного формальным параметром **A**. В переменной **INTER** накапливается результат. Её начальное значение равно **( )**. Когда список **AA** будет исчерпан, то функция вернет в качестве результата значение переменной **INTER**.

Часто необходимо выполнять циклическую обработку каждого элемента списка. В этом случае рекомендуется использовать макроформу **DOLIST**:

(**dolist** (*переменная список* [*результат*])
{ *форма* }*)

Здесь действия повторяются для переменной цикла, автоматически принимающей значения, равные очередному элементу списка, заданного в макроформе.

Если требуется выполнить некоторые действия N раз, то используется макроформа **DOTIMES**:

(**dotimes** (*переменная счетчик* [*результат*])
{*форма*}*)

Здесь действия выполняются для целочисленных значений переменной, задаваемых формой *счетчик* и лежащих в диапазоне от нуля до значения счетчика минус один. Когда переменная цикла станет равной *счетчику*, цикл завершится, возвратив значение формы *результат*.

2.2.3. Рекурсивные функции

Функция называется *рекурсивной*, если она вызывает саму себя. Такой вызов называют *хвостовой рекурсией*, если он соответствует последнему действию в определении функции. Рассмотрим пример вычисления факториала:

```
(defun factorial  (n)
      (cond ((zerop n) 1)
            (t (* (factorial  (- n 1)) n))))
```

Данная функция непосредственно реализует рекурсивное определение факториала $N! = (N-1)!\,N$, применяемое в математике. Поскольку в этом определении последней выполняемой операцией является произведение, то это не хвостовая рекурсия. На рисунке 2.1 изображена схема рекурсивных вызовов при вычислении факториала числа 4. Овалы, представленные на рисунке, изображают вызовы функции **FACTORIAL**. Рассмотренная схема представляет собой *простую линейную рекурсию*, для которой рекурсивный вызов выполняется в ветви **COND** только один раз.

Процесс рекурсивных вызовов можно исследовать, используя возможности трассировки. Включение и выключение механизма трассировки выполняется с помощью входящих в состав интерпретатора директив **TRACE** и **UNTRACE**:

```
(trace factorial)
(factorial 4)
(untrace factorial)
```

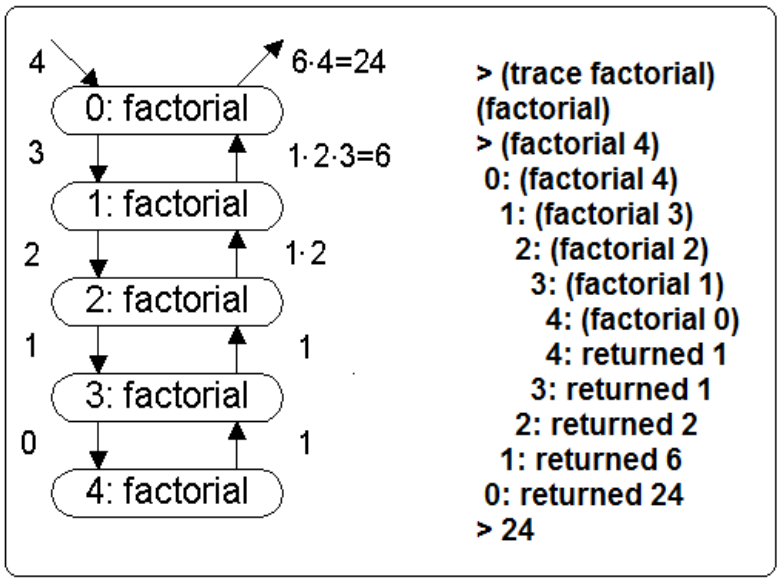

Рисунок 2.1 — Рекурсивное вычисление факториала и его трассировка

Существует еще одна часто встречающаяся схема вычислений, называемая *параллельной рекурсией*. В качестве примера можно указать рекурсивную функцию, вычисляющую числа Фибоначчи [1]. Каждый вызов функции **FIBONACCI** сводится к двум другим параллельным рекурсивным вызовам этой же функции.

Основные правила построения рекурсивных функций:

- определить количество и вид аргументов функции;
- определить вид результата;
- задать минимум одно условие выхода из рекурсии;
- определить формы для вычисления результата;
- определить формы вычисления новых значений аргументов для рекурсивных вызовов.

С целью сопоставления итерационного и рекурсивного подходов решим задачу поиска общих элементов двух списков рекурсивно. Для этого переопределим функцию **MY-INTERSECTION** следующим образом:

```
(defun my-intersection (a b)
(cond
        ((null a) nil)                                  ;правило 1
        ((member (first a) b)                           ;правило 2
        (cons (first a)(my-intersection (rest a) b)))
        (t (my-intersection (rest a) b))))              ;правило 3
```

Функция имеет два аргумента **A** и **B**, которые являются списками. В качестве результата функция возвращает список из элементов, входящих одновременно в списки **A** и **B**. Для этого в функции проверяется вхождение очередного первого элемента из списка **A** в список **B.** При этом после каждой такой проверки

список **A** сокращается на один элемент. Поэтому рекурсия должна будет завершаться, когда список **A** окажется пустым, т.е. когда вызов **(null a)** вернет значение **T (правило 1).**

Если первый элемент списка **A** входит в список **B**, то он добавляется с помощью **CONS** в список общих элементов между хвостом списка **A** (**REST A**) и списком **B**. Для построения указанного списка общих элементов функция **MY-INTERSECTION** рекурсивно применяется к хвосту списка **A** и исходному списку **B (правило 2).**

Если первый элемент списка **A** не входит в список **B**, то нет необходимости добавлять его куда-либо и список общих элементов строится из хвоста списка **A** и списка **B** с помощью соответствующего рекурсивного вызова функции **MY-INTERSECTION (правило3)**.

Таким образом, построение списка общих элементов начинается в тот момент, когда список **A** будет соответствовать пустому списку. При каждом выходе из рекурсии в результирующий список добавляется очередной общий элемент, если он был обнаружен **(**рисунок 2.2).

```
(my-intersection '(a b r c) '(s a d b f e c))

 0: (MY-INTERSECTION (A B R C) (S A D B F E C))
  1: (MY-INTERSECTION (B R C) (S A D B F E C))
   2: (MY-INTERSECTION (R C) (S A D B F E C))
    3: (MY-INTERSECTION (C) (S A D B F E C))
     4: (MY-INTERSECTION NIL (S A D B F E C))
     4: MY-INTERSECTION returned NIL
    3: MY-INTERSECTION returned (C)
   2: MY-INTERSECTION returned (C)
  1: MY-INTERSECTION returned (B C)
 0: MY-INTERSECTION returned (A B C)
(A B C)
```

Рисунок 2.2 — Трассировка рекурсивной функции **MY-INTERSECTION**

2.2.4. Сравнение итерационных и рекурсивных функций

В таблице 2.1. сведены признаки, сопутствующие итерационному и рекурсивному вариантам определения функций [6].

Таблица 2.1 — Отличительные признаки итерационного и рекурсивного вариантов определения функций

Итерационный вариант	Рекурсивный вариант
1. Наличие условия прекращения итераций	1. Наличие условия прекращения порождения рекурсивных вызовов
2. Наличие тела цикла	2. Наличие тела функции
3. Наличие переменных цикла	3. Аналог отсутствует
4. Наличие присваиваний	4. Аналог отсутствует

Различие в 2-х последних признаках говорит о разной природе вычислительных процессов в первом и во втором случае. В частности, сравнение 2-х ва-

риантов определения функции **MY-INTERSECTION** показывает, что в случае итерационного подхода, присваивания используются для изменения значений трех переменных цикла. При этом для формирования результатов вычислений на протяжении всего процесса вычислений необходим объем памяти, достаточный для размещения только этих трех переменных.

Совершенно иная ситуация возникает при выполнении рекурсивной функции. Здесь при каждой активации рекурсивного вызова сохраняется контекст со своими значениями локальных переменных и формальных параметров. В этом варианте потребляемая вычислительным процессом память линейно возрастает с каждым рекурсивным вызовом. Реализация запроса на выделение памяти требует также дополнительного времени процессора.

Таким образом, рекурсивное решение при прочих равных условиях менее эффективно, но часто более выразительно.

Следует отметить, что многие компиляторы при определенных условиях могут преобразовывать хвостовую рекурсию в итерационный вычислительный процесс. В частности, компилятор SBCL выполняет такие преобразования. Поэтому при написании рекурсивных функций хвостовая рекурсия является желаемой. Одним из приемов построения функций с хвостовой рекурсией является введение вспомогательных локальных определений функций с накапливающим (аккумулирующим) параметром. Например, переопределим функцию вычисляющую факториал:

```
(defun factorial1 (n)
  (labels ((factorial-rec (number  product)
             (if (= number 0)
                 product                        ; накапл. параметр
                 (factorial-rec (- number 1) (* product  number)))))
           (factorial-rec n 1)))
```

Здесь **LABELS** — функция, напоминающая **LET**-вызов, но устанавливающая локальное связывание имени **FACTORIAL-REC** с определением функции и оценивающая значение этой функции для вызова **(FACTORIAL-REC N 1)** при значении накапливающего параметра **PRODUCT**, равного 1. В этом определении значения накапливающего параметра вычисляются только при разворачивании рекурсии. При возврате из рекурсии они просто передаются в точку вызова.

2.3. Варианты заданий

Решить приведенную в таблице 2.1 задачу [5,6] двумя способами: с помощью механизмов организации итерационных циклов и рекурсивно. Сравнить оба способа по вычислительной эффективности (затраты памяти и процессорного времени) и выразительности (размер исходного кода). Рекурсивное решение должно удовлетворять требованиям функционального программирования. Выполнить трассировку рекурсивной функции и проследить за выполнением рекурсивного процесса.

Таблица 2.1 — Варианты заданий

Вариант	Задание
1.	Описать функцию, переводящую целое число из десятичной системы в двоичную.
2.	Описать функцию, которая для заданного списка lst формирует результирующий список- путем объединения результата реверсирования lst, результата реверсирования хвоста lst, результата реверсирования хвоста хвоста lst и так далее. Пример : для списка ‘(a b c d) результатом будет — ‘(d c b a d c b d c d).
3.	Описать функцию, которая вставляла бы на заданное место элементы второго списка-аргумента.
4.	Есть список lst и два произвольных лисповских объекта obj1 и obj2. Описать функцию, которая формирует новый список путем замены в списке lst всех вхождений объекта obj1 объектом obj2.
5.	Описать функцию, подсчитывающую произведение всех четных элементов списка
6.	Описать функцию, вычисляющую сумму всех числовых элементов списка с учетом наличия подсписков. Пример : для списка ‘(1 ((2 3) 4 6)) результатом будет — 16.
7.	Описать функцию, возвращающую отсортированный в порядке возрастания список, состоящий из чисел, исключая повторы элементов
8.	Описать функцию, которая на основе списка чисел формирует список-результат следующим образом : первый элемент есть произведение элементов списка, второй — произведение элементов хвоста, третий — произведение элементов хвоста хвоста и так далее. Пример: для списка ‘(1 2 3 4 5 6) результатом будет — ‘(720 720 360 120 30 6).
9.	Описать функцию, которая проверяет, является ли отсортированным в порядке возрастания список, состоящий из чисел
10.	Реализовать функцию включения объекта на заданное место в списке (нумерация элементов — от начала списка).
11.	Описать функцию, которая добавляет число в упорядоченный по возрастанию список без нарушения порядка
12.	Реализовать функцию, которая в исходном списке заменяет все элементы-списки результатами их реверсирования. Реверсирование производить на всех уровнях вложения. Пример: для списка ‘(1 ((2 3) 4) 5 6) результатом будет — ‘(1 (4 (3 2)) 5 6).
13.	Описать функцию, которая объединяет два упорядоченных числовых списка в упорядоченный список
14.	Описать функцию, которая, которая возвращает элемент списка по заданному номеру с конца.
15.	Описать функцию, которая подсчитывает количество разных (отличающихся) элементов списка

Продолжение таблицы 2.1.

Вариант	Задание
16.	Дан список lst и число n. Реализовать функцию, которая удаляет все i+n – е элементы списка.
17.	Описать функцию, которая подсчитывает сумму разных (отличающихся) элементов списка
18.	Даны списки lst1 и lst2. Реализовать функцию, которая удаляет из lst1 все элементы-списки, которые соответствуют тому же множеству, что и lst2. Пример : для списков : lst1=’(1 (2 2 3) 4 (3 2 3) 5), lst2=’(3 2 3 2) результатом будет — ’(1 4 5).
19.	Описать функцию, которая объединяет два списка без повторяющихся элементов
20.	Реализовать функцию, возвращающую T в том случае, если одинаковые атомы расположены в 2-х исходных списках в одном и том же порядке.
21.	Реализовать функцию, меняющую местами первый и последний элементы исходного списка.
22.	Описать функцию, которая возвращает атомарный злемент списка по заданному номеру n, считая от начала. Пример: для списка ‘((2) (3) 4 5 a (e r) g) и n=3 результатом будет — a.
23.	Написать функцию подсчета числа элементов-списков исходного списка на всех уровнях вложения.
24.	Описать функцию, которая возвращает список только из числовых элементов списка-аргумента. Список может содержать подсписки произвольной глубины.
25.	Опишите функцию, которая из исходного списка формирует список, содержащий только символьные атомы.
26.	Описать функцию, аргументами которой являются два списка, а результатом список, содержащий элементы первого списка, не принадлежащие второму списку.
27.	Описать функцию, которая в заданном списке заменяет все элементы-списки значениями сумм входящих в них числовых элементов с учетом вложенных подсписков.
28.	Описать функцию, которая в заданном списке заменяет все элементы-списки значениями количества входящих в них элементов-символов с учетом вложенных подсписков.
29.	Описать функцию, которая для заданного списка проверяет, является ли он отсортированным по возрастанию (убыванию).

2.4. Порядок выполнения лабораторной работы

2.4.1. Ознакомиться по лекционному материалу или учебному пособию [1] с организацией итерационных и рекурсивных вычислений в Лиспе. Выполнить примеры функций, приведенные в настоящей лабораторной работе.

2.4.2. Определить на языке Лисп функцию в соответствии с вариантом задания. Определение выполнить двумя различными способами, применяя формы

для организации итерационных циклических вычислений (см. п. 2.2.2) и правила построения рекурсивных функций (см. п.2.2.3). При этом рекурсивные функции должны быть определены в соответствии с принципами функционального программирования (см. п.2.2.1).

2.4.3. Создать в среде программирования Лисп-проект в соответствии с методическими указаниями [2], содержащий подготовленные определения функций и вызовы функций при различных тестовых значениях аргументов.

2.4.4. Выполнить отладку проекта.

2.4.5. Зафиксировать результаты выполнения функций, определенных двумя способами (в виде экранных копий).

2.4.6. Выполнить и зафиксировать результаты трассировки функций для двух способов определения (в виде экранных копий).

2.4.7. Определить время выполнения каждой из функций в виде количества циклов процессора для каждого из определений. Для этого следует воспользоваться вызовом (**time** *форма*), где *форма* — вызов определенной в лабораторной работе функции с соответствующими аргументами. Чтобы оценить дисперсию отображаемого времени, выполните несколько раз вызов (**time nil**).

2.4.8. Выполнить профилировку функций, определенных двумя способами, воспользовавшись профилировщиком, входящим в SBCL. Для этого следует выполнить вызовы:

(**sb-profile: profile** *имена_профилируемых_функций*)
последовательные вызовы профилируемых функций
(sb-profile: report)
(sb-profile: reset)

Либо воспользоваться соответствующими подпунктами меню в меню **Lisp**. При этом для включения режима профилировки функций следует выделить имя функции, а затем выбрать подпункт меню **Profile Function**. После этого выполнить в окне **REPL** вызовы профилируемых функций и выбрать подпункт меню **Profile Report**. Например, для 2-х вариантов определенных в п.2.2 функций **MY-INTERSECTION** были получены результаты в следующей форме:

```
measuring PROFILE overhead..done
  seconds     |     gc      |  consed  | calls  |  sec/call  |  name
------------------------------------------------------------------------
        0.001 |      0.000  |  499,296 |  1,300 |   0.000001 | MY-INTERSECTION2
        0.000 |      0.000  |        0 |    100 |   0.000000 | MY-INTERSECTION1
------------------------------------------------------------------------
        0.001 |      0.000  |  499,296 |  1,400 |                  | Total

estimated total profiling overhead: 0.00 seconds
overhead estimation parameters:
  3.2000003e-8s/call, 2.994e-6s total profiling, 1.34e-6s internal profiling
```

Чтобы профилировщик выдавал статистически устойчивые результаты, необходимо профилируемые функции вызывать многократно. В приведенном примере они циклически вызывались 100 раз с одни и теми же значениями аргументов. Из результатов профилировки следует, что рекурсивный вариант опреде-

ления функции (**MY-INTERSECTION2)** требует большего количества вызовов (поле calls) из-за рекурсии, резервирует намного больший объем памяти (поле consed) и исполняется дольше (поле seconds).

2.4.9. Сравнить функции по вычислительной эффективности (затраты памяти и процессорного времени) и выразительности (размер исходного кода), объяснить результаты испытаний.

2.5. Содержание отчета

Цель работы, вариант задания, описание определений функций в соответствии с вариантом задания, обоснование выбранных структур функций, включая условие окончания рекурсии в каждом случае и формирование новых значений аргументов при рекурсивном вызове, описание тестовых примеров, результаты выполнения функций проекта, результаты трассировки, оценки времени выполнения функций, таблица с результатами профилировки, сравнение и объяснение результатов испытаний функций, выводы.

2.6. Контрольные вопросы

2.6.1. Сформулируйте основные требования функционального стиля программирования.

2.6.2. Приведите примеры вызовов форм LOOP и DO в общем виде.

2.6.3. Чем отличаются формы DO и DO*?

2.6.4. Приведите примеры вызовов форм DOLIST и DOTIMES.

2.6.5. Что понимают под простой линейной рекурсией?

2.6.6. Что такое параллельная рекурсия?

2.6.7. Определите функцию вычисления факториала с помощью итерационных циклов.

2.6.8. Определите функцию вычисления факториала с помощью рекурсии.

2.6.9. Изобразите трассировку для рекурсивной функции, вычисляющей факториал при n=5.

2.6.10. Сформулируйте основные правила построения рекурсивных функций?

2.6.11. Определите свои рекурсивные варианты функций **member** и **append.**

2.6.12. Назовите основные отличительные признаки функций, реализованных с помощью итерационных циклов и с помощью рекурсии.

2.6.13. Что такое хвостовая рекурсия?

2.6.14. Сравните определение факториала, использующего накапливающий параметр **product** и реализующего хвостовую рекурсию (см. п.2.2.4), с определением, приведенным в п.2.2.3. Найдите значения факториала с помощью двух указанных определений при N=1000 и выполните профилировку. Какое из определений требует меньших объемов памяти и почему?

2.6.15. Что такое профилировка и как она выполняется в SBCL?

3. ЛАБОРАТОРНАЯ РАБОТА 3 «ПРИМЕНЕНИЕ СПИСКОВ И ФУНКЦИЙ ВЫСШИХ ПОРЯДКОВ ДЛЯ ОРГАНИЗАЦИИ БАЗ ДАННЫХ»

3.1. Цель работы

Исследование способов организации простых баз данных с помощью А-списков и списков свойств, получение практических навыков использования и разработки функций высшего порядка, изучение средств файлового ввода-вывода в языке Лисп.

3.2. Краткие теоретические сведения

3.2.1. Ввод-вывод данных в языке Лисп

В Лиспе операции ввода-вывода выполняются с помощью потоков. Потоки представляют собой хранилища данных, из которых можно читать данные или в которые можно записывать данные. Поток характеризуется направлением передачи (параметр :**direction** с возможными значениями — **:input | :output | :io**) и типом элементов (параметр **:element-type** по умолчанию **character**), которые извлекаются из потока или направляются в поток.

Поток в Лиспе связывается с динамической переменной, значения которой и представляют поток. Имеется несколько стандартных потоков: ***STANDARD-INPUT*, *STANDARD-OUTPUT*, *QUERY-IO*, *ERROR-OUTPUT*, *TRACE-OUTPUT*** и др. Эти динамические переменные определяют для функций ввода (**READ**) и вывода (**PRINT**) файлы ввода-вывода. В начале сеанса работы с Лисп системой эти файлы по умолчанию связаны с терминалом. Функции **READ** и **PRINT** соответственно вводят из потоков или выводят в потоки s-выражения.

Если требуется выполнять ввод-вывод из других файлов, то их необходимо открыть. Для этого применяется функция OPEN, упрощенный формат которой можно представить в форме

(**open** *имя-файла* **&KEY**
: direction *значение*
: element-type *значение*
: if-exists *значение*
: if-does-not-exist *значение*)

Например:

```
(setf in (open ”extern.dat” :direction  :input))
```

Здесь первый аргумент функции **OPEN** — строка, указывающая путь доступа к файлу. Для чтения выражений из открытого потока **in** необходимо сообщить его имя функции **READ**, т.е.

```
(read in)
```

В этом случае из файла **”extern.dat”** будет считано одно s-выражение.

Для вывода значений в выходные потоки применяются функции **PRINT, PRIN1, PRINC**. По умолчанию вывод выполняется на терминал. Функция **PRINT** перед выводом значения осуществляет переход на новую строку, а после печати значения выводит пробел. Функция **PRIN1** аналогична **PRINT**, но не выполняет переход на новую строку и не выводит пробел. Функция **PRINC** дает возможность напечатать строку без ограничивающих кавычек (или двоеточия в имени символа-ключа):

(princ ”lisp”) → lisp

Вывод в выходной поток, открытый с помощью функции **OPEN,** выполняют следующим образом:

```
(setf out  (open ”extern.dat”  :direction :output  :if-exists :append))
(print “lisp” out)
```

Здесь первый аргумент функции **PRINT** представляет выводимое значение, а второй — выходной поток **OUT**, связанный с файлом **EXTERN.DAT**.

При работе с файлами удобно использовать макроформу **WITH-OPEN-FILE.** Она позволяет открыть поток, задать режим работы с ним, выполнить необходимую его обработку и закрыть его. Примеры использования этой макроформы приведены в приложении Г методических указаний [2].

При записи данных в файл совместно с **PRINT** рекомендуется использовать макрос **WITH-STANDARD-IO-SYNTAX**, который гарантирует, что переменным, влияющим на поведение функции **PRINT**, присвоены стандартные значения. Используйте этот макрос и при чтении данных из файла для гарантии совместимости операций чтения и записи.

Для гибкого управления формой вывода в Коммон Лиспе применяется функция **FORMAT**:

(format ***поток шаблон*** **&REST** ***аргументы*** **)**

Если поток задан символом **T**, то функция осуществляет вывод на экран. Шаблон представляет собой строку, которая содержит спецификации (коды) управления печатью (таблица 3.1).

Спецификация начинается со знака “~” (аналогично % в функции printf языка Си). Если шаблон не содержит управляющих кодов, то **FORMAT** просто выводит литеры шаблона с помощью **PRINC**. Приведем пример:

```
(format  t ” Смотри: ~S ! ~% А сейчас по-английски: ~R dog~:P ” ’(A B)   5 )
→ Смотри:  (A B ) !
     А сейчас по-английски: five dogs
```

В Коммон Лиспе имеются дополнительные функции для ввода отдельных типов данных. Например, для чтения одиночного символа можно использовать функцию **(READ-CHAR** *поток***)**, а для чтения строки — функцию **(READ-LINE** *поток***)**. Если при чтении строки из потока вводятся числа, то необходимо выполнять преобразование числовой строки в число. При вводе целых чисел преобразование выполняют с помощью функции **(PARSE-INTEGER** *числовая_строка* **:JUNK-**

ALLOWED T), где ключевой параметр позволяет игнорировать ошибки преобразования. Для ввода значений типа «да/нет» можно воспользоваться функцией **(Y-OR-N-P** *строка***).** Эта функция обеспечивает отображение строки-приглашения ввода на экране и последующий ввод логического значения.

Таблица 3.1 — Коды управления печатью

Код	Назначение
~ %	Новая строка
~*	Пропуск аргумента
~A	Вывод очередного аргумента с помощью PRINC
~S	Вывод очередного аргумента с помощью PRIN1
~nT	Вывод со столбца n (табуляция)
~F	Вывод в форме с плавающей запятой
~R	Вывод числа словами
~P	Вывод слова во множественном числе
:	Используется совместно с управляющими кодами и уточняет их действие
~:P	Вывод во множественном числе, если предыдущий аргумент это не 1
~{ …~}	Коды форматирования, указанные между парой этих знаков, циклически применяются к каждому элементу последующего аргумента-списка

Функции чтения — **READ-CHAR, READ-LINE, READ** — могут иметь 2-ой и 3-ий аргументы, которые определяют их поведение при достижении конца файла. Если 2-ой аргумент равен NIL, то функции возвращают в качестве результата при достижении конца файла значение 3-его аргумента (см. приложение Г в [2]).

3.2.2. Ассоциативные списки и списки свойств

Ассоциативным списком или *а-списком* называется список, состоящий из точечных пар

$$((x_1 . y_1)(x_2 . y_2) \ldots (x_n . y_n)),$$

где $\mathbf{x_i}$ и $\mathbf{y_i}$ — произвольные s-выражения, i=1, … , n.

Первый элемент каждой точечной пары представляет собой некоторый ключ, а второй — данные, ассоциированные с ключом. Ассоциативный список ставит в соответствие ключу $\mathbf{x_i}$ выражение $\mathbf{y_i}$. Так как любой список можно представить в виде точечной пары, то список, состоящий из подсписков — а-список. Например,

((ivanov 1978 1)(petrov 1979 2) (sidorov 1980 3))

Для работы с ассоциативными списками в Лиспе имеется ряд встроенных функций. Функция **PAIRLIS** строит а-список из списков ключей и данных. Формат функции:

(pairlis ***ключи данные*** **[** ***а-список*** **])**

Функция добавляет новые точечные пары, образованные из списков *ключи* и *данные*, в начало *а-списка*. Например,

(setf x (pairlis '(a b d) '(1 2 3) '((c . 4))) → ((D . 3) (B . 2) (A . 1) (C . 4))

Если а-список при вызове функции **PAIRLIS** не задан, то образуемые точечные пары добавляются в пустой список.

Функция **ASSOC** выполняет поиск в ассоциативном списке. Она возвращает в качестве значения **первую по порядку** точечную пару, у которой ключ совпадает с заданным аргументом поиска:

(assoc 'd x) → (D . 3)

В Коммон Лиспе имеется также обратная функция **RASSOC**, которая ищет по данным ключ

(rassoc 4 x) → (C . 4)

Функция **ACONS** строит точечную пару из двух своих аргументов и добавляет ее в начало а-списка:

(acons 'a 1 x) → ((A . 1)(D . 3)(B . 2)(A . 1)(C . 4))

Дальнейшим развитием понятия а-список является *список свойств*. *Свойство символа* представляется в виде двух элементов: имени свойства и значения свойства. Свойства символа, если они назначаются, записываются в хранимый вместе с символом список свойств. Для работы со списком свойств символа применяются функции: **GET**— возвращает значение свойства; **REMPROP** — выполняет удаление свойства и его значения из списка свойств; комбинация **GET** и **SETF** — для присвоения или изменения значения свойства; **SYMBOL-PLIST** — для просмотра списка свойств символа [1].

Интересную возможность по хранению списка свойств предоставляет функция **LIST** [7]. Например, вызов **(LIST :A 1 :B 2 :C 3)** позволяет сохранить некоторые свойства, обозначаемые символами-ключами **a,b** и **c,** и их значения в виде списка **(:a 1 :b 2 :c 3)**. Доступ к значениям свойств в таком списке можно выполнять с помощью функции **GETF**, указав имя свойства в виде ключевого параметра:

(setf *database* (list :a 1 :b 2 :c 3)) → (:a 1 :b 2 :c 3)
(getf *database* :a) → 1

Если значение свойства не найдено, то **GETF** возвращает значение необязательного третьего аргумента (по умолчанию **NIL**).

Ассоциативные списки и списки свойств удобно использовать для создания простых баз данных.

3.2.3. Функции высших порядков

Иногда необходимо передать в функцию через ее формальный параметр имя другой функции. Такой параметр называют *функциональным*, а функцию, принимающую этот параметр, — *функционалом*. Функция также может возвращать в

виде результата другую функцию. Такие функции называют функциями *с функциональным значением*. Вызов функции с функциональным значением может быть аргументом функционала, а также использоваться вместо имени функции в вызове. Переданный в качестве параметра функциональный объект внутри принявшей его функции можно использовать только через явный вызов специальных *применяющих функционалов* **FUNCALL** или **APPLY**. Функционалы и функции с функциональным значением называют функциями *высших порядков*. Они являются весьма мощным инструментом, позволяющим Лиспу генерировать Лисп программы.

В качестве примера определим функционал, вычисляющий сумму функций $f(i)$ одного аргумента i = 1…n:

```
(defun funsum ( func n )
      (do    (( i 1 ( + i 1 ))
              ( result 0 ( + result (funcall func i ))) )
              (( =  i  ( + n 1 ))  result )))
```

Здесь **FUNC** — передаваемое имя функции, которая вызывается (применяется) с помощью применяющего функционала **FUNCALL.** Вызов **FUNSUM** можно выполнять двумя способами, например: **(funsum 'sqrt 5)** или **(funsum #'sqrt 5)**. Второй вызов называется *функциональной блокировкой* (см. ниже).

Функционал **APPLY** аналогичен **FUNCALL**, но применяется он не к отдельным аргументам, а к списку аргументов:

(APPLY *имя-функции список***)**

Например, **(apply '+ '(1 2 3 4))** $\rightarrow$ **10** или **(apply #'+ '(1 2 3 4))** $\rightarrow$ **10.**

Кроме применяющих функционалов, в Лиспе имеется группа **MAP**-функционалов, которые называются *отображающими функционалами*. Эти функционалы обеспечивают повторное применение своего функционального аргумента к списку и, тем самым, преобразуют (отображают) одну последовательность элементов в другую. Общий формат вызова отображающих функционалов можно представить в виде

(**MAP**x *имя-функции* **&REST** *список*)

Здесь **MAP**x представляет имя одного из **MAP**-функционалов: **MAPCAR, MAPLIST, MAPCAN, MAPCON, MAPC, MAPL** [1]. Рассмотрим примеры вызовов некоторых из этих функционалов.

MAPCAR. Этот функционал применяет функцию, заданную именем, к каждому элементу списка и в качестве результата возвращает список:

```
(mapcar 'sqrt '( 1 4 9 2 3 )) → (1.0  2.0  3.0 1.414214  1.732051)
(mapcar '+ '( 1 4 9 2 3 ) '( 9 6 1 8 7 ) → (10 10 10 10 10 )
```

MAPLIST. Применяется в тех случаях, когда необходимо повторить вычисления для хвостовых частей списка:

```
(maplist 'cons '(a b) '( 1 2 )) → ( ((a b) 1 2 ) ((b) 2) )
```

Здесь функция **CONS** сначала применяется к спискам **(A B)** и **(1 2)**, а затем к их **CDR**-частям , т.е. к спискам **(B)** и **(2).** Результат вычислений представляется в форме списка.

Для работы с последовательностями в Коммон Лиспе существует ряд мощных функционалов: **MAP, REMOVE, DELETE, SUBSTITUTE, FIND, POSITION, COUNT**. Функционал **MAP** является обобщением функционала **MAPCAR**. Он применяет заданную функцию к каждому элементу последовательности, но дополнительно позволяет указать тип результата:

(**MAP** *тип-результата функция* **&REST** *последовательности*)

Например, **(map 'string #'(lambda (x) (if (evenp x) #\1 #\0)) '(1 2 3 4))** → **"0101".** В примере функциональный параметр задан с помощью лямбда-выражения.

Для удаления из списка элементов, обладающих заданными свойствами, удобно использовать встроенный функционал **REMOVE-IF(–NOT) (англ. удалить-если),** первым аргументом которого является предикат, проверяющий выполнение (для **–NOT** невыполнение) условий удаления для очередного элемента:

(remove-if #'oddp '(1 2 3 4 3 2 1)) → **(2 4 2)**
(remove-if-not #'evenp '(1 2 3 4 3 2 1)) → **(2 4 2)**

При работе с базой данных, представленной в виде списка из подсписков, функционал **REMOVE-IF–NOT** может использоваться для поиска и возврата необходимых записей по заданному ключу поиска. Например, для извлечения из базы данных записей с ключевым полем **:b**, равным **22**, можно применить вызовы:

(setf *database* '((:a 11 :b 12 :c 13) (:a 21 :b 22 :c 23) (:a 21 :b 32 :c 33)…))
(remove-if-not #'(lambda (x) (equal (getf x :b) 22)) *database*) → **((:a 21 :b 22 :c 23))**

Здесь передаваемая в функционал **REMOVE-IF–NOT** функция представлена лямбда-выражением. Запись #'(*форма*) эквивалентна записи (function *форма*) и называется *функциональной блокировкой*. В общем случае, если в функциональной блокировке *форма* представлена лямбда-выражением, то генерируется так называемое *лексическое замыкание* (lexical closure). Суть его состоит в том, что к определению функции, которое задано лямбда-выражением, добавляются связи свободных переменных, входящих в лямбда-выражение, т.е. возникает замыкание функции и некоторого контекста ее определения. Замыкания можно использовать в качестве функционального аргумента. При вызове замыкания значения свободных переменных лямбда-выражения берутся из контекста создания замыкания [1].

Это позволяет определить следующую функцию выборки из упомянутой выше базы данных для произвольного значения поля **:b (B-VALUE)**, определяемого из контекста вызова:

```
(defun select-by-b (b-value)
      (remove-if-not #'(lambda (x) (equal (getf x :b) b-value)) *database*))

(select-by-b 12) → ((:a 11 :b 12 :c 13) )
```

В приведенном определении **B-VALUE** является свободной переменной лямбда-выражения и её значение берется из контекста вызова. Введенное определение легко обобщается на случай выборки записей по значениям других полей, отличных от **:b**. Для этого можно определить функционал **SELECT**, в который передается необходимая функция-селектор **SELECTOR-FUNC**, обеспечивающая проверку наличия в базе данных записи с соответствующим полем и значением:

```
(defun select (selector-func)
      (remove-if-not  selector-func *database*))
```

Где функцию-селектор для выборки, например по ключу **:a**, можно определить так:

```
(defun a-selector (a-value)
      #'(lambda (x) (equal  (getf x :a) a-value)))
```

Тогда выборки из базы данных можно будет выполнять следующим образом:

(select (a-selector 21)) → ((:a 21 :b 22 :c 23) (:a 21 :b 32 :c 33))

Аналогично можно определить функции-селекторы для других полей базы данных.

В приведенных примерах перебор всех записей базы данных выполнялся функционалом **REMOVE-IF–NOT.** При модификации существующих записей базы данных необходимо будет выполнять поиск в базе соответствующей записи и её изменение. Перебор записей в этом случае можно организовать с помощью отображающего функционала **MAPCAR.** Определим функционал, модифицирующий записи базы данных:

```
(defun update (selector-func &key a b c)
   (setf *database*
         (mapcar
               #'(lambda (record)
                     (when (funcall selector-func record)
                           (if a (setf (getf record :a) a))
                           (if b (setf (getf record :b) b))
                           (if c (setf (getf record :c) c)))
                     record) *database*)))
```

Здесь функциональный параметр **SELECTOR-FUNC** представляет функцию-селектор, которая будет применяться к очередной записи **RECORD** базы данных с помощью **MAPCAR.** Если функция-селектор выполняется для очередной записи, то полям записи присваиваются с помощью **SETF** новые значения, передаваемые в функцию **UPDATE** через ключевые параметры **A, B, C.** Пример вызова:

```
(update (a-selector 21) :b 44 :c 55) →
            ((:a 11 :b 12 :c 13) (:a 21 :b 44 :c 55) (:a 21 :b 44 :c 55))
```

Здесь для записей с полем **:A=21** значения полей **:B** и **:C** меняются соответственно на **44** и **55**. Если необходимо обеспечивать более сложный поиск модифицируемой записи, то необходимо усовершенствовать функцию-селектор. Обратите внима-

ние, что функция-селектор по своему назначению соответствует условию **WHERE** языка **SQL**.

3.3. Варианты заданий

Написать программу, обеспечивающую создание на диске базы данных. Структура базы данных определяется одной из таблиц в соответствии с вариантом задания. В функции программы должно входить :

− создание базы данных;
− добавление записи в базу данных;
− сохранение базы данных на диске;
− загрузка базы данных в оперативную память;
− просмотр информации.

Кроме этого, программа должна выполнять дополнительные функции, указанные в варианте задания (таблица 3.2).

Таблица 3.2 — Варианты заданий

Вариант	Номер таблицы и дополнительные функции
1.	Таблица 3.3. Корректировка данных в базе по номеру записи; вывод на дисплей фамилий и номеров групп для всех студентов, если средний балл студента больше 4.0; если таких студентов нет, вывести соответствующее сообщение.
2.	Таблица 3.3. Корректировка данных в базе по фамилии; вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5; если таких студентов нет, вывести соответствующее сообщение.
3.	Таблица 3.3. Корректировка данных в базе по номеру группы; вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2; если таких студентов нет, вывести соответствующее сообщение
4.	Таблица 3.4. Корректировка данных в базе по номеру рейса; вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры; если таких рейсов нет, выдать на дисплей соответствующее сообщение
5.	Таблица 3.4. Корректировка данных в базе по типу самолета; вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры; если таких рейсов нет, выдать на дисплей соответствующее сообщение
6.	Таблица 3.5. Корректировка данных в базе по фамилии; вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры; если таких работников нет, вывести на дисплей соответствующее сообщение.

Продолжение таблицы 3.2

Вариант	Номер таблицы и дополнительные функции
7.	Таблица 3.6. Корректировка данных в базе по номеру поезда; вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени; если таких поездов нет, выдать на дисплей соответствующее сообщение.
8.	Таблица 3.6. Корректировка данных в базе по пункту назначения; вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры; если таких поездов нет, выдать на дисплей соответствующее сообщение.
9.	Таблица 3.6. Корректировка данных в базе по времени отправления; вывод на экран информации о поезде, номер которого введен с клавиатуры; если таких поездов нет, выдать на дисплей соответствующее сообщение.
10.	Таблица 3.7. Корректировка данных в базе по начальному маршруту; вывод на экран информации о маршруте, номер которого введен с клавиатуры; если таких маршрутов нет, выдать на дисплей соответствующее сообщение.
11.	Таблица 3.7. Корректировка данных в базе по номеру маршрута; вывод на экран информации о маршрутах, которые начинаются или оканчиваются в пункте, название которого введено с клавиатуры; если таких маршрутов нет, выдать на дисплей соответствующее сообщение.
12.	Таблица 3.8. Корректировка данных в базе по фамилии; вывод на экран информации о человеке, номер телефона которого введен с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение.
13.	Таблица 3.8. Корректировка данных в базе по номеру телефона; вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры; если таких нет, выдать на дисплей соответствующее сообщение.
14.	Таблица 3.8. Корректировка данных в базе по году рождения; вывод на экран информации о человеке, чья фамилия введена с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение.
15.	Таблица 3.9. Корректировка данных в базе по фамилии; вывод на экран информации о человеке, чья фамилия введена с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение.
16.	Таблица 3.9. Корректировка данных в базе по знаку зодиака ; вывод на экран информации о людях, родившихся под знаком, название которого введено с клавиатуры; если таких нет, выдать на дисплей соответствующее сообщение.
17.	Таблица 3.9. Корректировка данных в базе по месяцу рождения ; вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры; если таких нет, выдать на дисплей соответствующее сообщение.

Продолжение таблицы 3.2

Вариант	Номер таблицы и дополнительные функции
18.	Таблица 3.10. Корректировка данных в базе по названию товара; вывод на экран информации о товаре, название которого введено с клавиатуры; если таких товаров нет, выдать на дисплей соответствующее сообщение.
19.	Таблица 3.10. Корректировка данных в базе по названию магазина; вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры; если такого магазина нет, выдать на дисплей соответствующее сообщение.
20.	Таблица 3.11. Корректировка данных в базе по расчетному счету плательщика ; вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры; если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.
21.	Таблица 3.12 Корректировка данных в базе по фамилии; вывод на дисплей анкетных данных студентов отличников; если таких студентов нет, вывести соответствующее сообщение.
22.	Таблица 3.12. Корректировка данных в базе по году рождения; вывод на дисплей анкетных данных студентов, получивших одну оценку 3; если таких студентов нет, вывести соответствующее сообщение.
23.	Таблица 3.12. Корректировка данных в базе по году поступления; вывод на дисплей анкетных данных студентов, получивших все двойки; если таких студентов нет, вывести соответствующее сообщение.
24.	Таблица 3.12. Корректировка данных в базе по оценке «физика»; вывод на дисплей анкетных данных студентов, получивших все пятерки; если таких студентов нет, вывести соответствующее сообщение.
25.	Таблица 3.12. Корректировка данных в базе по номеру ; вывод на дисплей анкетных данных студентов, получивших одну оценку 4, а все остальные – 5; если таких студентов нет, вывести соответствующее сообщение.
26.	Таблица 3.12. Корректировка данных в базе по фамилии, которая начинается с литеры ‘А’ ; вывод на дисплей фамилий студентов, которые начинаются с литеры ‘А’, и их оценки; если таких студентов нет, вывести соответствующее сообщение.
27.	Таблица 3.12. Корректировка данных в базе по фамилии, которая начинается с литеры ‘Б’; вывод на дисплей фамилий студентов, которые начинаются с литеры ‘Б’, и год их рождения; если таких студентов нет, вывести соответствующее сообщение.
28.	Таблица 3.12. Корректировка данных в базе по фамилии, которая начинается с литеры ‘Б’или ‘Г’ ; вывод на дисплей фамилий студентов, которые начинаются с литеры ‘Б’или ‘Г’, и год их поступления; если таких студентов нет, вывести соответствующее сообщение.

Таблица 3.3. — Студент группы

Фамилия И.О.	Номер группы	Успеваемость				
		P1	P2	P3	P4	P5

Таблица 3.4. — Рейс самолета

Пункт назначения	Номер рейса	Тип самолета

Таблица 3.5. — Сотрудник

Фамилия И.О.	Должность	Год приема на работу

Таблица 3.6. — Поезд

Пункт назначения	Номер поезда	Время отправления

Таблица 3.7. — Маршрут

Начальный пункт	Конечный пункт	Номер маршрута

Таблица 3.8. — Записная книжка

Фамилия Имя	Номер телефона	Дата рождения		
		день	месяц	год

Таблица 3.9. — Знак зодиака

Фамилия Имя	Знак зодиака	Дата рождения		
		день	месяц	год

Таблица 3.10. — Стоимость

Название товара	Название магазина	Стоимость товара, грн

Таблица 3.11. — Счет

Расчетный счет плательщика	Расчетный счет получателя	Перечисляемая сумма, грн.

Таблица 3.12.— Студент

Номер	Фамилия Имя	Год рождения	Год поступления	Оценки		
				Ф	ВМ	Пр.

3.4. Порядок выполнения лабораторной работы

3.4.1. Ознакомиться по лекционному материалу или учебному пособию [1] с функциями ввода-вывода языка Лисп, функциями обработки А-списков и списков свойств, функционалами и замыканиями. Выполнить примеры функций, приведенные в разделе 3.2 настоящей лабораторной работы.

3.4.2. Ознакомиться с вариантом задания и выбрать одну из списковых структур (А-список, список свойств символа, список символов-ключей и их значений) для хранения записей таблицы. Привести обоснование выбора.

3.4.3. Определить на языке Лисп функции добавления записи в базу, функции сохранения базы на диске и загрузки базы в оперативную память, функцию просмотра базы на экране.

3.4.4. Создать в среде программирования Лисп-проект в соответствии с методическими указаниями [2], содержащий подготовленные определения функций, указанных в п. 3.4.3.

3.4.5. Выполнить частичную отладку проекта.

3.4.6. Подготовить определения дополнительных функций в соответствии с вариантом. При этом выборку записей в базе выполнять с помощью функционалов **REMOVE-IF–NOT или FIND**, а поиск записи для корректировки с помощью отражающих функционалов **MAPCAR или MAP,** следуя общим рекомендациям, указным в п. 3.2.3.

3.4.7. Выполнить полную отладку проекта и зафиксировать результаты работы программы в виде экранных копий.

3.4.8. Придумать 3-4 дополнительных запроса к базе данных и оценить объем возможных изменений (дополнений) в программе

3.5. Содержание отчета

Цель работы, вариант задания, обоснование выбранных списковых структур для представления записей таблицы, описание определений функций для общей работы с базой данных, описание определений дополнительных функций в соответствии с вариантом задания, описание тестовых запросов и результаты их выполнения, оценка объема возможных изменений программы в случае добавления новых запросов, выводы.

3.6. Контрольные вопросы

3.6.1. Что понимают под потоком ввода-вывода? Как открыть поток?

3.6.2. Какие функции используют для записи s-выражений в поток?

3.6.3. Какие функции используют для чтения s-выражений в из потока?

3.6.4. Объясните макроформу **WITH-OPEN-FILE**.

3.6.5. Объясните функцию **FORMAT** и приведите примеры.

3.6.6. Какие функции применяются для ввода символа и строки?

3.6.7. Какая функция применяется для ввода логических значений?

3.6.8. Чем определяется поведение функций чтения при достижении метки конец файла?

3.6.9. Что такое А-список?

3.6.10. Какие функции применяют для работы с А-списком?

3.6.11. Что такое список свойств символа и как он хранится?

3.6.12. Какие функции применяются для работы со списком свойств?

3.6.13. Какие с помощью функции LIST создать список из символов-ключей и их значений?

3.6.14. Что такое функционал?

3.6.15. Что такое функция с функциональным значением?

3.6.16. Что такое функции высших порядков?

3.6.17. Назовите и объясните использование применяющих функционалов?

3.6.18. Объясните синтаксис передачи функционального аргумента?

3.6.19. Объясните отображающий функционал **MAPCAR?**

3.6.20. Объясните отображающий функционал **MAPLIST?**

3.6.21. Объясните отображающий функционал **MAP?**

3.6.22. Объясните функционал **REMOVE-IF(–NOT)?**

3.6.23. Что называют функциональной блокировкой?

3.6.24. Что такое лексическое замыкание? Приведите пример.

4. ЛАБОРАТОРНАЯ РАБОТА № 4 «ЕЯ ДОСТУП К БАЗЕ ДАННЫХ НА ОСНОВЕ АЛГОРИТМА СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ»

4.1. Цель работы

Исследование алгоритма сопоставления с образцом и особенностей его применения для формирования запросов к базам данных, а также для организации доступа к базам данных на ограниченном подмножестве естественного языка.

4.2. Краткие теоретические сведения

4.2.1. Общие правила сопоставления с образцом

Сопоставление с образцом является важнейшей операцией, выполняемой во многих системах искусственного интеллекта. Под сопоставлением с образцом (pattern matching) понимается процедура, при которой с известной символьной структурой (образцом) сопоставляется некоторая другая структура (факт, данные) с целью выявления единообразия или подобия структур.

Процедура сопоставление с образцом — неотъемлемая часть многих СИИ. Например, она широко используется в экспертных системах, базирующихся на продукционной модели представления знаний. В этом случае предпосылки продукционных правил выступают в качестве образцов, а факты, хранящиеся в рабочей памяти (базе данных), выступают в роли распознаваемых образов. Если предпосылка правила сопоставима с одним из фактов в рабочей памяти, то активизируется соответствующее правило и в базу данных добавляется новый факт. Таким образом, осуществляется прямой логический вывод [1].

Рассмотрим сопоставление с образцом, представляя образец и сопоставляемые с ним факты (данные) в виде списков языка Лисп. Простейшие возможности сопоставления реализуются встроенными предикатами Лиспа, например **EQ, EQL, EQUAL.**

Для большей гибкости при сопоставлении в образце применяются условные обозначения в виде символов **?** и **$**. Символу **?** из образца можно поставить в соответствие любой элемент из списка данных, находящийся на том же месте, но только один; символу **$** можно поставить в соответствие один или несколько элементов из списка данных, в том числе и пустой элемент. Ниже приведен пример, когда образец и данные сопоставимы:

ОБРАЗЕЦ — (? позволяет распознавать $)
ДАННЫЕ — (сопоставление позволяет распознавать ситуации или причины)

Обозначим сопоставимость символьных структур знаком **:=:**, и сформулируем основные правила сопоставления:

ОБРАЗЕЦ **:=:** ДАННЫЕ
если ; правило 1 , пример: ( ):=: ( )
ДАННЫЕ = NIL и ОБРАЗЕЦ = NIL

или ; правило 2 , пример: ($):=: ( )
ДАННЫЕ = NIL и (CAR ОБРАЗЕЦ) = $
и (CDR ОБРАЗЕЦ) = NIL
или ; правило 3 , пример: (a b c d):=: (a b c d)
(ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ) ≠ NIL
и (CAR ОБРАЗЕЦ) = (CAR ДАННЫЕ)
и (CDR ОБРАЗЕЦ) **:=:** (CDR ДАННЫЕ)
или ; правило 4 , пример: (? b c d):=: (a b c d)
(ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
и (CAR ОБРАЗЕЦ) = ?
и (CDR ОБРАЗЕЦ) **:=:** (CDR ДАННЫЕ)
или ; правило 5, пример: ($ b c d):=: (b c d)
(ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
и (CAR ОБРАЗЕЦ) = $
и (CDR ОБРАЗЕЦ) **:=:** ДАННЫЕ
или ; правило 6, пример: ($ d):=: (a b c d)
(ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
и (CAR ОБРАЗЕЦ) = $
и ОБРАЗЕЦ **:=:** (CDR ДАННЫЕ)

Обратите внимание, что введенные правила определяют понятие “сопоставимость” рекурсивно. Назначение каждого из правил видно из примеров, записанных рядом с номером правила.

4.2.2. Базовая функция сопоставления с образцом

Определим функцию **(MATCH P D)**, выполняющую сопоставление образца **P** и данных **D** с помощью введенных выше правил (полный листинг функции приведен в приложении А):

```
; функция сопоставления  образца p и данных d
    (defun match (p d)
      (cond
              ;правило 1
               ((and (null p)(null d)) t)
               ;правило 2
               ((and (null d)
                     (eq (car p) '$)
                     (null (cdr p))) t)

               ; один из списков исчерпан
               ((or (null p)(null d)) nil)

              ;правило 3 и правило 4
               ((or (equal (car p) '?)
                    (equal (car p) (car d)))
                (match (cdr p)(cdr d)))

               ;правило 5 и 6
               ((eq (car p) '$)
                (cond ((match (cdr p) d) t)
                      ((match p (cdr d)) t))) ))
```

Функцию **MATCH** можно легко расширить, чтобы она могла сопоставлять списки с вложенными подсписками. Для этого достаточно добавить в текст функции следующее правило:

```
;правило 7 - сопоставление списков, включающих подсписки
                ((and (not (atom (car p)))
                      (not (atom (car d)))
                      (match (car p)(car d)))
                 (match (cdr p) (cdr d)) )
```

В этом случае функция **MATCH** будет обеспечивать сопоставление более сложных символьных структур. Например,

```
(match '( (a    b) ? (?   d)  $)
         '( (a    b) c (c   d)  1  2  3  4  5))
→T
```

4.2.3. Применение функции сопоставления с образцом для формирования запросов к базе данных

Несмотря на то, что введенная функция **MATCH** реализует простейшую схему сопоставления, её уже можно использовать для решения практических задач, например, доступа к базе данных. Определим для этого функцию **GET-MATCHES** с двумя аргументами, представленными s-выражениями. Первый аргумент будет представлять собой запрос к базе данных, записываемый в виде образца, а второй – список, соответствующий базе данных, в которой выполняется поиск. Функция **GET-MATCHES** возвращает в качестве результата записи базы данных, сопоставимые с запросом (образцом).

Пусть рассматривается база данных, содержащая сведения о служащих:

```
(setq *database*
        '(((Иванов Сергей)      14000 1235)
          ((Абрамов Николай)  17500 3325)
          ((Гавриленко Ольга)  14000 3143)
          ((Денисенко Сергей)  22000 2528)
          ((Коль Татьяна) 18250 3133)
           … ))
```

В базе данных хранятся данные о годовых доходах служащих и их табельные номера. Определим функцию **GET-MATCHES**, используя рекурсию:

```
(defun get-matches (p db)
  ;db - база данных
  ;p  - запрос (образец)
  (cond ((null db) nil)
        ((match p (car db))
         (cons (car db) (get-matches p (cdr db))))
        (t (get-matches p (cdr db)))))
```

С помощью функции **GET-MATCHES**, можно получать ответы на следующие запросы:

```
;проверка наличия конкретной записи
(get-matches '((Денисенко Сергей) 22000 2528) *database*)
→ (((Денисенко Сергей) 22000 2528))

;выдать сведения о всех сотрудниках с годовым доходом 14000
(get-matches '(? 14000 ?) *database*)
→ ((( Иванов Сергей) 14000 1235) ((Гавриленко Ольга) 14000 3143))

;выдать сведения о всех сотрудниках, которых зовут Сергей
(get-matches '((? Сергей) ? ?) *database*)
→ (((Иванов Сергей) 14000 1235) ((Денисенко Сергей) 22000 2528))
```

Кстати, функция будет успешно работать и с базой данной, представленной в виде списков с символами-ключами без переписывания её кода:

```
(setq *database2* (list
      (list :FIO '(Иванов Сергей)      :Z 14000 :N 1235)
      (list :FIO '(Абрамов Николай)  :Z 17500 :N 3325)
      (list :FIO '(Гавриленко Ольга) :Z 14000 :N 3143)
      (list :FIO '(Денисенко Сергей) :Z 22000 :N 2528)
      (list :FIO '(Коль Татьяна)       :Z 18250 :N 3133)
            ))
;выдать сведения о всех сотрудниках с годовым доходом 14000
(get-matches '(:FIO ? :Z 14000 :N  ?)*database2*) →
((:FIO (Иванов Сергей) :Z 14000 :N 1235)
 (:FIO (Гавриленко Ольга) :Z 14000 :N 3143))
```

Единственное отличие заключается в форме записи образца при вызове функции **GET-MATCHES.**

4.2.4. Усовершенствование базовой функции сопоставления с образцом

Можно достигнуть еще большей универсальности приведенного выше алгоритма сопоставления, включив в него дополнительные параметры-переменные. Имена переменных в образце начинаются символом **'?'** или **'$'**. Например, **?V** или **$V**. В первом случае в переменную **V** будет подставляться произвольный символ данных, во втором — сегмент списка данных. Подстановка значений в переменные позволит получать ответы на запросы вида:

```
(match '(AVTO ?MODEL ?YEAR) '(AVTO ford 1998)) → T
model → ford
year → 1998
```

Реализация соответствующих правил приведена в [1]. Функцию **MATCH** можно совершенствовать и далее, используя ограничивающий параметр в образце

(**restrict**), который дает возможность накладывать определенные ограничения на сопоставляемое значение. Например,

```
(match '((restrict  ?V  integerp  evenp) b c)
       '(36  b  c)) → T
(match '((restrict ?V integerp evenp) b c)
       '(36.0  b  c)) → NIL
```

В примере параметр **?V** из подсписка ограничений сопоставляется с первым элементом списка данных. Если рассматриваемый элемент данных является целым четным числом, то сопоставление успешно. Использование переменных в образце позволяет запоминать сделанные подстановки и таким образом обрабатывать более сложные запросы к базе данных.

4.2.5. Обработка запросов к БД на естественном языке (ЕЯ)

Используя рассмотренную функцию сопоставления с образцом, можно написать простейшую программу, поддерживающую беседу на естественном языке. Такая программа основана на распознавании с помощью функции **MATCH** ответа пользователя и формировании подходящего по смыслу вопроса для следующего шага диалога с пользователем [1]. Эту же идею можно использовать для формирования запросов к базе данных на естественном языке.

Существующие ЕЯ-интерфейсы баз данных используют различные подходы [1,3]: метод сопоставления с образцом, синтаксические грамматики, семантические грамматики, некоторый внутренний язык. Интерфейсы, основанные на методе сопоставления с образцом, являются наиболее простыми.

Приведем фрагмент программы возможной обработки ЕЯ-запросов к базе данных, представленной динамической переменной ***database*** (см. п.4.2.3) **.** Запросы вводятся в виде списка слов на естественном языке (ЕЯ). В программе примеры обрабатываемых ЕЯ-запросов представлены в виде комментариев:

```
(setq fact (read))    ; ввод ЕЯ запроса

; Запрос 1: (есть ли в базе запись о сотруднике Денисенко Сергей)

(cond
  ((match '( $ запись $ Денисенко Сергей $ ) fact)     ; распознавание  ЕЯ запроса
  (princ (get-matches '((Денисенко Сергей) ? ?) *database*))) ; запрос к БД
       ;→ (((Денисенко Сергей) 22000 2528))

; Запрос 2: (выдать сведения о всех сотрудниках у которых  годовой доход 14000)

   ((match '( $ всех $  годовой доход $ 14000 ) fact)
   (princ (get-matches '(? 14000 ?) *database*)))
       ;→ ((( Иванов Сергей) 14000 1235) ((Гавриленко Ольга) 14000 3143))

; Запрос 3: (выдать сведения о всех сотрудниках  которых зовут Сергей)

  ((match '( $ всех $  зовут Сергей ) fact)
```

```
(princ (get-matches '((? Сергей) ? ?) *database*)))
    ;→ (((Иванов Сергей) 14000 1235) ((Денисенко Сергей) 22000 2528))

; Запрос 4: (как зовут сотрудника по фамилии Денисенко)

((match '( $   зовут $ фамилии Денисенко ) fact)
(get-matches '((Денисенко ?V) ? ?) *database*) (princ V))…)
    ;→ Сергей
```

Поясним работу фрагмента программы на примере первого запроса. ЕЯ-запрос, введенный с помощью функции **READ**, запоминается в переменой **FACT**. В форме **COND** введенный запрос сопоставляется с образцом, который подбирается так, чтобы описать распознаваемый запрос в общем виде. В примере вводится запрос **«(есть ли в базе запись о сотруднике Денисенко Сергей)»**, который сопоставляется с образцом **«($ запись $ Денисенко Сергей $)»**. Так как ЕЯ-запрос сопоставим с образцом, то выполняется действие — вызывается функция **GET-MATCHES**, которая выполняет поиск в базе данных нужной записи опять таки по образцу **«((Денисенко Сергей) ? ?)» .** Найденное значение выводится на экран с помощью функции **PRINC**.

Таким образом, каждый ЕЯ-запрос обрабатывается с помощью правила «образец — действие». Каждое из правил представляет клоз **COND** формы, которая легко дополняется новыми правилами.

ЕЯ-интерфейсы БД, построенные на основе метода сопоставления с образцом, могут приводить к ошибочным ответам, так как не учитывают семантических отношений между элементами запросов. Более гибкими являются ЕЯ-интерфейсы, реализующие синтаксический разбор запросов к БД. При этом синтаксический анализ можно также выполнять на основе грамматических правил с использованием схемы сопоставления с образцом [1].

4.3. Варианты заданий

Для базы данных, созданной в лабораторной работе 3, необходимо написать на языке Лисп интерфейс, который позволяет выполнять ЕЯ-запросы с помощью алгоритма сопоставления с образцом. Ответ на запрос должен также представляться на естественном языке в виде списка слов предложения. Кроме запроса, заданного по варианту задания, предусмотреть 5-6 различных дополнительных запросов.

4.4. Порядок выполнения лабораторной работы

4.4.1. Изучить по лекционному материалу или учебному пособию [1] алгоритм сопоставления с образцом.

4.4.2. Написать функцию сопоставления с образцом в соответствии с примером, приведенным в [1]. Выполнить отладку функции по тестовым примерам.

4.4.3. Написать функцию обеспечивающую поиск записей в базе данных, по запросу, представленному в виде образца. В качестве аналога можно использовать

функцию **GET-MATCHES**. Рекомендуется переопределить её с использованием отображающего функционала **MAPCAR**.

4.4.4. Разработать запросы и соответствующие образцы к ним для тестирования разработанных функций. По результатам тестирования внести необходимые изменения (расширения) в разработанные функции.

4.4.5. Дополнить разработанную программу правилами обработки ЕЯ-запросов. Для этого разработать соответствующие образцы, обеспечивающие распознавание ЕЯ-запросов и фрагменты кода, осуществляющего необходимые действия в случае успешного распознавания. Выполнить тестирование программы для всех разработанных запросов.

4.4.6. Написать функции, необходимые для преобразования результатов поиска в ЕЯ-форму, и дополнить ими правила из п.3.4.5. Выполнить окончательную отладку программы.

4.4.7. Выполнить анализ работы программы на разных запросах, определить ограничения программы и зафиксировать результаты её работы в виде экранных копий.

4.4.8. Факультативно предлагается расширить программу таким образом, чтобы она могла вести с пользователем диалог на естественном языке в рамках сведений, содержащихся в базе данных.

4.5. Содержание отчета

Цель работы, вариант задания, общие правила построения функции сопоставления с образцом, описание выполненных дополнений для функции сопоставления с образцом, описание разработанной функции поиска записей в БД по образцу, описание набора правил распознавания ЕЯ-запросов, описание тестовых ЕЯ-запросов и результатов их выполнения, оценка ограничений программы, выводы.

4.6. Контрольные вопросы

4.6.1. Объясните суть задачи сопоставления с образцом? Где встречается эта задача?

4.6.2. Какие функции языка Лисп выполняют простейшее сопоставление с образцом?

4.6.3. Какие специальные символы используют в образце и что они обозначают?

4.6.4. Объясните общие правила сопоставления с образцом.

4.6.5. Напишите базовую функцию сопоставления с образцом.

4.6.6. Как расширить базовую функцию сопоставления с образцом, чтобы она могла сопоставлять списки, включающие подсписки?

4.6.7. Как выполнить доступ к базе данных по образцу? Определите необходимую функцию.

4.6.8. Как обозначают переменные в образце, если необходимо запоминать подстановки в ходе выполнения сопоставления?

4.6.9. Определите дополнительные правила для базовой функции сопоставления с образцом, если требуется запоминать подстановки.

4.6.10. Для чего вводится пакет ограничений образца?

4.6.11. Определите правила обработки пакета ограничений образца.

4.6.13. Объясните общий принцип построения программы, распознающей ЕЯ-запросы по образцу.

4.6.14. Приведите пример правил для распознавания ЕЯ-запроса по образцу.

4.6.15. Какие недостатки свойственны ЕЯ-интерфейсу, построенному на принципе сопоставления с образцом?

4.6.17. Что необходимо учитывать, чтобы улучшить обработку ЕЯ-запросов?

5. ЛАБОРАТОРНАЯ РАБОТА № 5 «МЕТОДЫ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

5.1. Цель работы

Исследование методов поиска решений задач в пространстве состояний и овладение методологией решения логических задач с применением этих методов.

5.2. Краткие теоретические сведения

5.2.1. Формулировка задач в пространстве состояний

При решении любой задачи можно выделить два этапа: выбор способа представления задачи и собственно решение. Задача поиска в пространстве состояний, определяется совокупностью четырех составляющих

$$(S_0, S, F, G), \tag{5.1}$$

где S — множество возможных состояний задачи; S_0 — множество начальных состояний, $S_0 \in S$; F — множество операторов, преобразующих одни состояния в другие; G — множество целевых состояний, $G \in S$ [1].

Каждый оператор $f \in F$ является функцией, отображающей одно состояние в другое — $s_j = f(s_i)$, где $s_i, s_j \in S$. Решением задачи является последовательность операторов $f_i \in F$, преобразующих начальные состояния в конечные, т.е. $f_n(f_{n-1}(\ldots(f_2(f_1(S_0)))\ldots)) \in G$. Если такая последовательность не одна и задан критерий оптимальности, то возможен поиск оптимальной последовательности.

Поиск решения в пространстве состояний представляют в виде ***графа состояний.*** Множество вершин графа соответствует состояниям задачи, а множество дуг (ребер) — операторам. Тогда решение задачи может интерпретироваться как поиск пути на графе.

Уточним сказанное на примере **задачи игры в восемь**. В игре используются фишки, пронумерованные от 1 до 8 (рисунок 5.1). Фишки располагаются в ячейках матрицы размером 3×3. Любая фишка, смежная с пустой ячейкой, может быть передвинута в позицию, соответствующую пустой ячейке. В игре ищут последовательность ходов, переводящих игру из начального состояния в конечное состояние, изображенное на рисунке 5.1 справа.

Формулировка этой задачи в пространстве состояний требует определения всех составляющих выражения (5.1).

Состояния. Описание состояния должно определять местонахождение каждой из восьми фишек и пустой ячейки.

Начальное состояние. В качестве начального состояния может быть выбрано любое состояние.

Целевое состояние. Показано на рисунке 5.1 справа.

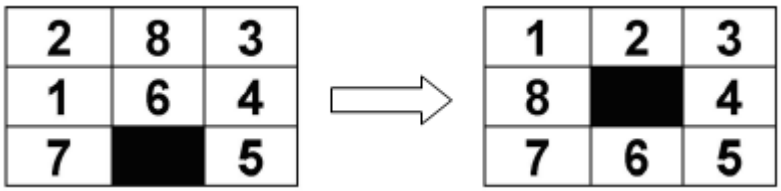

Рисунок 5.1 – Состояния игры в восемь

Функция (оператор) преобразования состояний. Эта функция формирует допустимые состояния, которые являются результатом осуществления одного из четырех действий (теоретически возможных ходов: *влево, вправо, вверх, вниз*).

Проверка цели. Позволяет проверить, соответствует ли данное текущее состояние целевому состоянию.

Часть графа поиска решения игры в восемь изображена на рисунке 5.2

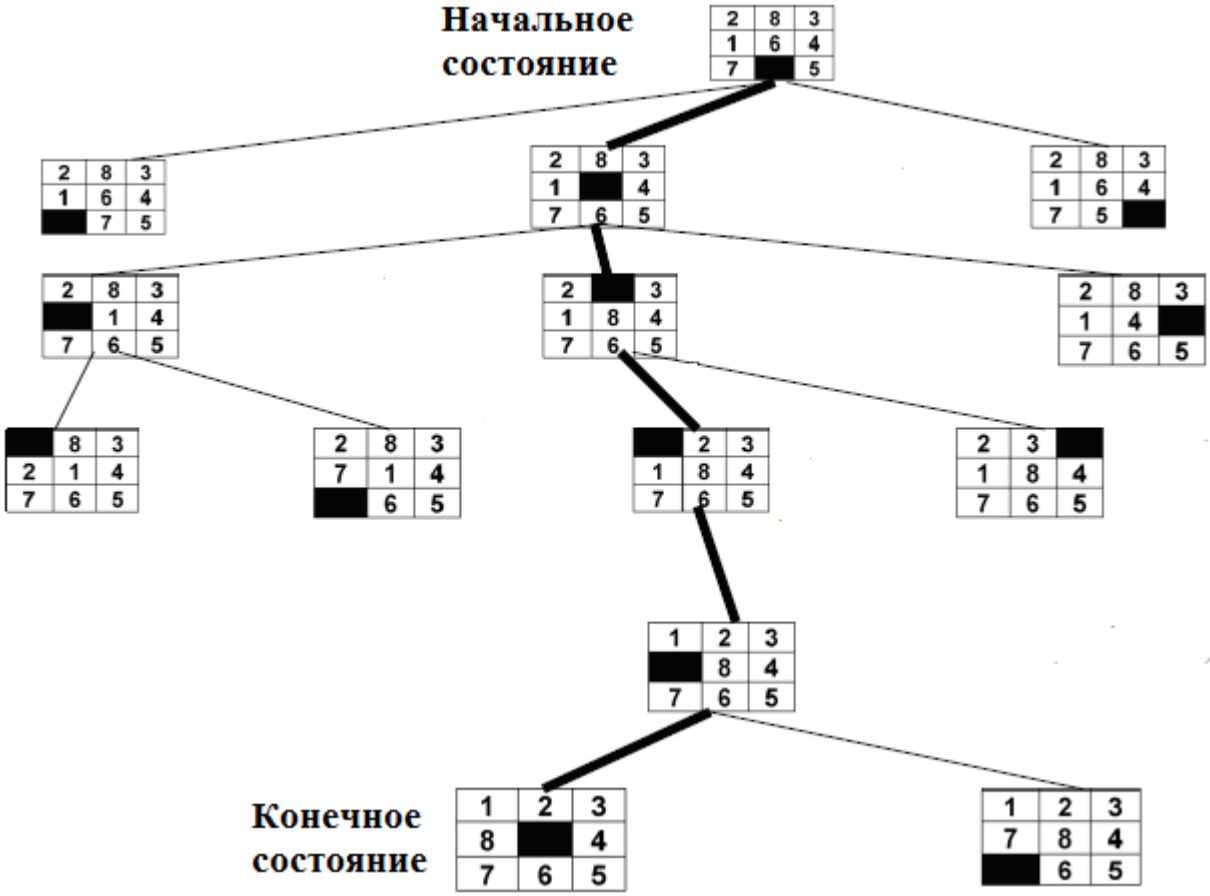

Рисунок 5.2 — Подграф поиска решения игры в восемь

Процесс применения операторов к некоторой вершине с целью получения всех ее дочерних вершин называется ***раскрытием вершины.*** При этом могут задаваться ***условия применимости*** оператора к вершине (состоянию).

5.2.2. Методы поиска решений задач в пространстве состояний

Методы (стратегии) поиска в пространстве состояний подразделяются на две группы: методы “слепого” и упорядоченного (эвристического) поиска. В методах “слепого” поиска выполняется полный просмотр всего пространства состояний, что может приводить к проблеме *комбинаторного взрыва*. Для сокращения

количества просматриваемых вариантов применяют методы упорядоченного (направленного) поиска.

Стратегии поиска в пространстве состояний обычно сравнивают с помощью следующих критериев [4].

Полнота. Гарантируется ли обнаружение решения, если оно существует?

Оптимальность. Стратегию называют оптимальной, если она обеспечивает нахождение решения, которое не обязательно будет наилучшим, но известно, что оно принадлежит подмножеству решений, обладающих некоторыми заданными свойствами, согласно которым мы относим их к оптимальным.

Минимальность. Стратегию называют минимальной, если она гарантирует нахождение наилучшего решения, т.е. минимальность является более сильным случаем оптимальности.

Временная сложность. Время, необходимое для нахождения решения.

Пространственная сложность. Объем памяти, необходимый для решения задачи.

Рассматриваемые ниже стратегии поиска используют два списка: список открытых вершин и список закрытых вершин. В *списке открытых вершин* (**OPEN**) находятся идентификаторы (имена) вершин, подлежащих раскрытию; *в списке закрытых вершин* (**CLOSED**) находятся имена уже раскрытых вершин и имя вершины, раскрываемой в данный момент. Список **CLOSED** позволяет запоминать уже рассмотренные состояния с целью исключения их повторного раскрытия.

5.2.3 Методы слепого поиска

5.2.3.1. Поиск в ширину. В начале поиска список **CLOSED** пустой, а **OPEN** содержит только начальную вершину. Каждый раз из списка OPEN выбирается для раскрытия первая вершина. Раскрытая вершина перемещается в список **CLOSED**, а ее дочерние вершины помещаются в ***конец*** списка **OPEN**, т.е. принцип формирования списка OPEN соответствует очереди. Согласно этой стратегии, вершины глубиной ***k***, раскрываются после того как будут раскрыты все вершины глубиной ***k-1***. В этом случае фронт поиска растет в ширину. Для построения обратного пути (из целевой вершины в начальную вершину) все дочерние вершины снабжаются указателями на соответствующие родительские вершины. В приведенном ниже алгоритме поиска в ширину функция **first(OPEN)** выбирает из списка **OPEN** первую вершину.

```
Procedure Breadth_First;
Begin
        Поместить начальную вершину в список OPEN;
        CLOSED:='пустой список';
        While OPEN<>'пустой список' Do
          Begin
            n:=first(OPEN);
            If n='целевая вершина' Then Выход(УСПЕХ);
            Переместить n из списка OPEN в CLOSED;
```

Раскрыть вершину n и поместить все ее дочерние вершины, отсутствующие в списках OPEN и CLOSED, в конец списка OPEN, связав с каждой дочерней вершиной указатель на n;
End;
Выход(НЕУДАЧА);
End.

Если повторяющиеся вершины не исключаются из рассмотрения (т.е. список **CLOSED** не создаётся и строится не граф поиска, а **дерево поиска**!) и каждая вершина имеет B дочерних вершин, то при остановке поиска на глубине, равной d, максимальное число раскрытых вершин будет равно $B+B^2+B^3+\ldots+B^d+(B^{d+1}-B)$ [4]. Обычно вместо этой формулы употребляют её обозначение $O(B^{d+1})$, которое называют ***экспоненциальной оценкой сложности***. Если полагать, что раскрытие каждой дочерней вершины требует единицы времени, то $O(B^{d+1})$ является ***оценкой временной сложности***. При этом каждая вершина должна сохраняться в памяти до получения решения. Поэтому ***оценка пространственной сложности*** будет также равна $O(B^{d+1})$. Это приводит к тому, что стратегия поиска в ширину может использоваться только задач, которые характеризуются пространством поиска небольшой размерности. Но при этом она удовлетворяет ***критерию полноты и минимальности*** (обеспечивает нахождение целевого состояния, которое находится на минимальной глубине дерева поиска, т.е. самого поверхностного решения).

5.2.3.2. Поиск по критерию стоимости (алгоритм равных цен). Поиск в ширину является оптимальным, если стоимости раскрытия всех вершин равны. Если с каждой вершиной связать стоимость её раскрытия и из списка **OPEN** выбирать вершину с наименьшей стоимостью, то рассмотренная выше процедура будет обеспечивать нахождение **оптимального** решения по критерию стоимости (при условии модификации стоимости вершины в списке **OPEN**, при обнаружении пути с меньшей стоимостью). Стоимость раскрытия $g(V_i)$ некоторой вершины V_i равна

$$g(V_i)=g(V)+c(V,V_i), \qquad (5.2)$$

где V — родительская вершина для вершины V_i; $c(V,V_i)$ — стоимость пути из вершины V в вершину V_i; $g(V)$ — стоимость раскрытия родительской вершины (для начальной вершины $g(V)=0$).

Рассмотренная стратегия гарантирует **полноту** поиска, если стоимость каждого участка пути положительная величина. Так как поиск в этом случае направляется стоимостью путей, то ***временная и пространственная сложности*** (при построении дерева поиска) в наихудшем случае будут равны $O(B^{1+C/c})$, где C — стоимость оптимального решения, c — минимальная стоимость каждого действия. Эта оценка может быть больше $O(B^d)$. Это связано с тем, что процедура поиска по критерию стоимости часто обследует поддеревья поиска, состоящие из мелких этапов небольшой стоимости, прежде чем перейти к исследованию путей, в которые входят крупные, но возможно более полезные этапы [4].

5.2.3.3. Поиск в глубину. При поиске в глубину всегда раскрывается самая глубокая вершина в текущем фронте поиска. Процедура поиска в глубину отличается от рассмотренной процедуры поиска в ширину тем, что дочерние вершины,

получаемые при раскрытии вершины **n**, помещаются в **начало** списка **OPEN** т.е. принцип формирования списка **OPEN** соответствует стеку.

Поиск в глубину требует хранения только единственного пути от корня до листового узла. Для дерева поиска с коэффициентом ветвления ***B*** и максимальной глубиной ***m*** поиск в глубину требует хранения ***Bm+1*** узлов, т.е ***пространственная сложность*** равна ***O(Bm)***, что намного меньше по сравнению в рассмотренными выше стратегиями [4]. При **поиске в глубину с возвратами** требуется еще меньше памяти. В этом случае каждый раз формируется только одна из дочерних вершин и запоминается информация о том, какая вершина должна быть сформирована следующей. Таким образом, требуется только ***O(m***) ячеек памяти.

Однако поиск в глубину ***не является полным*** (в случае не ограниченной глубины) и ***оптимальным*** (не обеспечивает гарантированное нахождение наиболее поверхностного целевого узла). В наихудшем случае ***временная сложность*** равна $O(B^m)$, где ***m*** — максимальная глубина дерева поиска (может быть гораздо больше по сравнению с ***d*** — глубиной самого поверхностного решения).

Проблему деревьев поиска неограниченной глубины можно решить, предусматривая применение во время поиска заранее определенного ***предела глубины L.*** Это означает, что вершины на глубине ***L*** рассматриваются таким образом, как если бы они не имели дочерних вершин. Такая стратегия поиска называется **поиском с ограничением глубины**. Однако при этом вводится дополнительный источник **неполноты**, если будет выбрано ***L<d***, т.е. самая поверхностная цель находится за пределами глубины. А при выборе ***L>d*** поиск с ограничением глубины будет **неоптимальным** (не гарантируется получение самого поверхностного решения).

5.2.3.4. Поиск в глубину с итеративным углублением. Эта стратегия поиска позволяет найти наилучший предел глубины. Для этого применяется процедура поиска с ограничением по глубине. При этом предел глубины постепенно увеличивается (в начале он равен 0, затем 1, затем 2 и т.д.) до тех пор пока не будет найдена цель. Это происходит, когда предел глубины достигает значения ***d***— глубины самого поверхностного решения. Конечно, здесь допускается повторное формирование одних и тех же состояний. Однако такие повторные операции не являются слишком дорогостоящими, и временная сложность оценивается значением $O(B^d)$ [4]**.**

В поиске с итеративным углублением (по дереву поиска) сочетаются преимущества поиска в ширину (является **полным**) и поиска в глубину (малое значение **пространственной сложности**, равное ***O(Bd)***).

Так как при поиске в ширину некоторые вершины формируются на глубине ***d+1***, то поиск с итеративным углублением фактически осуществляется быстрее, чем поиск в ширину, несмотря на повторное формирование состояний.

5.2.3.5. Исключение повторяющихся состояний. Так как в рассмотренной выше обобщенной процедуре поиска в ширину (см. п.5.2.3.1) для исключения повторяющихся состояний используется список **CLOSED**, в котором запоминаются уже раскрытые вершины, то **временная сложность** рассмотренных алгоритмов при **поиске на графе состояний** будет меньше, чем при **поиске на дереве состо-**

яний. Но из-за того, что запоминаются все рассмотренные состояния, то поиск в глубину и поиск с итеративным углублением уже не будут характеризоваться линейными оценками **пространственной сложности**. И рассмотренные методы поиска в этом случае могут оказаться неосуществимыми из-за недостаточного объема памяти. ***В этом заключается фундаментальный компромисс между пространством и временем.***

5.2.4 Эвристический поиск

5.2.4.1. Общая характеристика методов эвристического поиска. Основная идея таких методов состоит в использовании дополнительной информации для ускорения процесса поиска. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. **эвристик.** Использование эвристик позволяет сократить количество просматриваемых вариантов при поиске решения задачи, что ведет к более быстрому достижению цели.

В алгоритмах эвристического поиска список открытых вершин упорядочивается по возрастанию некоторой ***оценочной функцией***, формируемой на основе эвристических правил. Оценочная функция может включать две составляющие, одна из которых называется эвристической и характеризует близость текущей и целевой вершин. Чем меньше значение эвристической составляющей оценочной функции, тем ближе рассматриваемая вершина к целевой вершине. В зависимости от способа формирования оценочной функции выделяют следующие алгоритмы эвристического поиска: алгоритм “подъема на гору”, алгоритм глобального учета соответствия цели, А-алгоритм [1]. Наиболее общим является А-алгоритм.

5.2.4.2. А- алгоритм. А-алгоритм похож на алгоритм равных цен, но в отличие от него учитывает как уже сделанные затраты, так предстоящие затраты. Такого учета можно добиться, если воспользоваться оценочной функцией:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

где $\hat{g}(n)$ – оценка стоимости пути из начальной вершины в вершину *n,* которая вычисляется в соответствии с (5.2); $\hat{h}(n)$– **эвристическая** оценка стоимости кратчайшего пути из вершины *n* в целевую вершину (предстоящие затраты). Чем меньше значение $\hat{h}(n)$, тем более перспективен путь, на котором находится вершина *n* . В ходе поиска раскрываются вершины с минимальным значением оценочной функции $\hat{f}(n)$.

Готовых рецептов в отношении построения эвристической составляющей оценочной функции не существует. При решении каждой конкретной задачи используется ранее накопленный опыт решения подобных задач. Например, для **игры в восемь** это может быть количество фишек, которые находятся не на своем месте или сумма расстояний каждой фишки от своего до целевого места.

В общем случае оценки $\hat{f}(n)$ меняют свои значения в процессе поиска. Это приводит к тому, что вершины из списка **CLOSED** могут перемещаться обратно в список **OPEN**:

Procedure A_Search;
Begin

Поместить начальную вершину *s* в список OPEN: $\hat{f}(s) = \hat{h}(s)$**;**
CLOSED:='пустой список';
While OPEN<>'пустой список' Do Begin
n:=first(OPEN);
If n='целевая вершина' Then Выход(УСПЕХ);
Переместить n из OPEN в CLOSED;
Раскрыть n и для всех ее дочерних вершин n_i **вычислить оценку** $\hat{f}(n, n_i) = \hat{g}(n, n_i) + \hat{h}(n_i)$**;**
Поместить дочерние вершины, которых нет в списках OPEN и CLOSED, в список OPEN, связав с каждой вершиной указатель на вершину n;
Для дочерних вершин n_i**, которые уже содержатся в OPEN, сравнить оценки** $\hat{f}(n,n_i)$ **и** $\hat{f}(n_i)$**, если** $\hat{f}(n,n_i) < \hat{f}(n_i)$**, то связать с вершиной** n_i **новую оценку** $\hat{f}(n,n_i)$ **и указатель на вершину n;**
Если вершина n_i **содержится в списке CLOSED и** $\hat{f}(n,n_i) < \hat{f}(n_i)$**, то связать с вершиной** n_i **новую оценку** $\hat{f}(n,n_i)$**, переместить её в список OPEN и установить указатель на n;**
Упорядочить список OPEN по возрастанию $\hat{f}(n_i)$**;**
End;
Выход(НЕУДАЧА);
End.

5.2.4.3. Свойства А-алгоритма. Свойства А-алгоритма существенно зависят от условий, которым удовлетворяет или не удовлетворяет эвристическая часть оценочной функции $\hat{f}(n)$ [1]:

1) А-алгоритм соответствует алгоритму равных цен, если $h(n)=0$;
2) А-алгоритм **гарантирует оптимальное решение**, если $\hat{h}(n) \le h(n)$; в этом случае он называется **А* – алгоритмом**. А*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает;
3) А-алгоритм обеспечивает однократное раскрытие вершин, если выполняется ***условие монотонности*** $\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j)$,где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j;;
4) алгоритм A_1* эвристически более сильный, чем алгоритм A_2* при условии $\hat{h}_1(n) > \hat{h}_2(n)$. Эвристически более сильный алгоритм A_1* в большей степени сокращает пространство поиска;
5) А*-алгоритм полностью информирован, если $\hat{h}(n)=h(n)$. В этом случае никакого поиска не происходит и приближение к цели идет по оптимальному пути;

6) при $\hat{h}(n)>h(n)$ А-алгоритм не гарантирует получение оптимального решения, однако часто решение получается быстро.

Таким образом, при определенных условиях А-алгоритм является **полным, оптимальным** и эффективным в отношении **временной сложности**. Однако из-за того, что алгоритм хранит все сформированные узлы, оценка **пространственной сложности** по-прежнему является экспоненциальной.

Преодолеть проблему пространственной сложности за счет небольшого увеличения времени выполнения позволяют алгоритмы **RBFS** и **MA*** [4].

5.2.5. Критерии качества методов поиска

Кроме приведенных критериев временной и пространственной сложности, часто используют критерий целенаправленности поиска и фактор эффективного ветвления.

Целенаправленность поиска — позволяет узнать, в какой мере поиск идет в направлении к цели. Целенаправленность поиска характеризуется отношением ***P=L/N***, где ***L*** — длина найденного пути до цели, а ***N*** — общее число построенных в процессе поиска вершин. Информированный поиск характеризуется большими значениями ***P*** по сравнению со слепым поиском. При этом в пределе ***P*** стремится к единице.

Фактор эффективного ветвления (*B*) — основан на представлении о дереве с глубиной, равной длине пути ***L***, и общим числом вершин ***N***, причем у каждой вершины ровно ***B*** дочерних вершин. Связь этих параметров выражается формулой [1]:

$$N=B(B^L-1)/(B-1).$$

Зависимость ***B*** от ***N*** при различных ***L*** может быть представлена практически линейной диаграммой. Величина ***B***, близкая к 1, соответствует точно направленному к цели поиску. Целенаправленность можно связать с ***B*** и длиной пути соотношением

$$P=[L\ (B-1)]\ /\ [B\ (B^L-1)].$$

Фактор эффективного ветвления ***B*** может быть использован для предсказания того, сколько вершин было бы построено при поиске пути той или иной длины.

5.2.6. Базовые программы поиска решений в пространстве состояний

Листинг базовой программы поиска в глубину на языке Лисп приведен в приложении В методических указаний [2]. Листинг базовой программы, реализующей поиск в соответствии с А-алгоритмом, приведен в приложении Б настоящих методических указаний. Программы выполняют поиск на заранее заданных графах состояний. Чтобы адаптировать приведенные базовые программы поиска решений к варианту задания, необходимо, прежде всего, переопределить функцию **LIST-CHILDREN**, которая для заданной вершины находит список дочерних вершин,

т.е. раскрывает вершину. Это функция должна быть написана таким образом, чтобы дочерние вершины генерировались в соответствии с условиями задачи. Поскольку для многих задач такие вершины генерируются по правилам из условия задачи, то при написании **LIST-CHILDREN** рекомендуется для реализации правил использовать функцию **MATCH** из предыдущей лабораторной работы.

5.3. Варианты заданий

Требуется представить задачу, заданную вариантом задания, в пространстве состояний. Написать на языке Лисп программу, которая находит решение задачи в соответствии с заданным методом. Оценить качество работы метода перебора по критериям временной и пространственной сложности, целенаправленности поиска. Варианты заданий приведены в таблице 5.1. Задание по варианту состоит из двух цифр, например: вариант 14 — (4, 3). Здесь первая цифра означает номер задачи (таблица 5.2), а вторая — номер метода поиска решения (таблица 5.3).

Таблица 5.1 — Варианты заданий

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

Таблица 5.2 — Условия задач

№	Условие задачи
1	Игра в восемь. Формулировка задачи приведена в п.5.2.1. Начальное состояние должно генерироваться случайным образом.
2	Задача о коммивояжере. Коммивояжер должен посетить каждый из N заданных городов, изображенных на карте. Между каждой парой городов имеется путь, длина которого указана на этой карте. Нужно, отправляясь из стартового города, найти самый короткий путь, по которому коммивояжер по одному разу проходит через каждый из городов и затем возвращается в стартовый город. N вводится с клавиатуры (N>5). Программа должна генерировать расстояния между городами автоматически и отображать их.

Продолжение таблицы 5.2

№	Условие задачи
3	Задача о двух кувшинах. Дан кувшин с водой емкостью N литров и пустой кувшин емкостью M литров. Требуется получить заданную емкость L литров (L<N и L<M). Воду можно либо выливать, либо переливать из одного кувшина в другой. (Кувшины можно полностью наполнять водой из неограниченного резервуара). Значения N, M, L вводятся с клавиатуры, например, соответственно: 5, 2, 1.
4	Задача о миссионерах и каннибалах. Три миссионера и три каннибала находятся на левом берегу реки. Все хотят перебраться на другой берег. Имеется лодка, вмещающая не более двух человек. Если на каком-то берегу каннибалов окажется больше, чем миссионеров, то они съедят миссионеров. Требуется найти последовательность перемещений лодки с одного берега на другой, гарантирующую безопасность миссионерам.
5	Задача об N ферзях. Разместить N ферзей на шахматной доске N×N так, чтобы на каждой горизонтали, вертикали и диагонали стояло не более одного ферзя.
6	Задача о шахматном коне (задача Эйлера). Требуется обойти все клетки шахматной доски ходом коня.
7	Задача “Ханойская башня”. Имеется три стержня, на первом из которых помещены N дисков разного диаметра, причем меньший диск обязательно лежит на большем. Требуется переместить все диски на третий стержень, двигая их по очереди. Один ход состоит в снятии верхнего диска с одного из стержней и перемещении его поверх дисков (если они есть) другого стержня. Ограничение: больший диск нельзя класть на меньший.
8	Задача о трех монетах. Из трех монет, лежащих на столе, первая и третья лежат кверху гербом, вторая – кверху цифрой. Требуется в три хода перевернуть монеты так, чтобы они все лежали одинаково – либо гербом, либо цифрой кверху. Каждый ход состоит в переворачивании двух из трех монет.
9	Путь коня. На шахматной доске N×N, из которой вырезано несколько клеток, заданы две клетки. Построить минимальный путь коня из одной клетки в другую.
10	Магараджи пери. К берегу Ганга подъехали сразу трое магараджи со своими пери. Всем они хотели переправиться на другой берег. У берега стояла лодка для 2-х человек. Обычаи не позволяли ни одной пери оставаться в лодке или на берегу одной с чужим мужем, если рядом не будет своего. Обычаи не запрещают пери самой управлять лодкой. Требуется найти последовательность перемещений лодки.

Продолжение таблицы 5.2

№	Условие задачи
11	Раскраска карт. Дана карта, содержащая N областей, некоторые из которых являются соседними. Требуется раскрасить карту, используя только 4 цвета, таким образом, чтобы не было 2-х смежных областей одного цвета.
12	Задача о волке, козе и капусте. На берегу реки находятся фермер, волк, коза и капуста. Фермеру необходимо перевезти животных и капусту на другой берег. При этом в отсутствии фермера волк и коза, равно как и коза с капустой, не могут находиться на одном берегу. В лодке можно перевозить только одно животное или капусту. Требуется определить последовательность перевозки.

Таблица 5.3 — Методы поиска решений

№	Название метода
1	Поиск в ширину
2	Поиск по критерию стоимости (алгоритм равных цен0
3	Поиск в глубину
4	Поиск в глубину с возвратом
5	Поиск в глубину с ограничением глубины
6	Поиск с итеративным углублением
7	A*-алгоритм

5.4. Порядок выполнения лабораторной работы

5.4.1. Изучить по лекционному материалу или учебному пособию [1] методы поиска решений задач в пространстве состояний и базовые программы поиска решений на языке Лисп.

5.4.2. Разработать описание заданной задачи в пространстве состояний: указать форму описания состояний, определить операторы преобразования состояний, определить начальные и конечные состояния, критерии достижения цели.

5.4.3. Представить пространство состояний в виде графа (подграфа).

5.4.4. Разработать программу, выполняющую поиск решения задачи в соответствии с заданным методом, используя в качестве основы базовые программы, указанные в п. 5.2.6.

5.4.5. Разработать тестовые наборы данных, выполнить отладку программы по тестовым примерам.

5.4.6. Оценить эффективность используемого метода поиска по критериям временной и пространственной сложности, а также по критерию целенаправленности, выполнив необходимые эксперименты с программой (при необходимости в программу следует включить соответствующие счетчики).

5.4.7. Сравнить полученные экспериментальные значения критериев с теоретическими оценками.

5.4.8. По результатам анализа сделать заключение о применимости метода, заданного вариантом к решению поставленной задачи. Сформулировать предложения по усовершенствованию выбранных представлений или выбору иного метода.

5.5. Содержание отчета

Цель работы, вариант задания, краткий обзор методов поиска решений задач в пространстве состояний (подходящих для решения задачи в соответствии с вариантом задания), описание представления задачи (описание состояний, операторов, начального и конечного состояний, критериев достижения цели), представление пространства состояний в виде графа, тексты программ с комментариями и обоснованиями, полученные результаты и их анализ (с обязательным указанием наборов тестовых данных), выводы по проведенным экспериментам.

5.6. Контрольные вопросы

5.6.1. Назовите основные способы представления задач в ИИ?

5.6.2. Опишите состояния, операторы преобразования состояний, функции стоимости для следующих задач (см. табл. 5.2):

а) задача о коммивояжере;

б) задача о миссионерах и каннибалах;

в) задача о раскраске карт.

5.6.3. Для задачи о двух кувшинах (см. табл. 5.2) емкостью 5 литров и 2 литра, построить дерево поиска, если требуется налить во второй кувшин ровно 1 литр воды.

5.6.4. Напишите на псевдоязыке процедуры поиска в ширину и глубину, объясните их различие с алгоритмической точки зрения.

5.6.5. Напишите на псевдоязыке процедуру поиска в соответствии с алгоритмом равных цен.

5.6.6. Сформулируйте принципы поиска, используемые в алгоритмах поиска в глубину: с возвратом, с ограничением глубины и с итеративным углублением.

5.6.7. Объясните основной принцип построения процедур эвристического поиска. Запишите вид оценочной функции и объясните её составляющие.

5.6.8. Напишите на псевдоязыке процедуру поиска в соответствии с А-алгоритм.

5.6.9. Сформулируйте и объясните свойства А-алгоритма.

5.6.10. Определите критерии полноты, оптимальности, минимальности, пространственной и временной сложности.

5.6.11. Сравните основные алгоритмы поиска в пространстве состояний по критериям полноты, оптимальности, пространственной и временной сложности.

5.6.12. Определите критерий целенаправленности поиска и фактор эффективного ветвления.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Бондарев В.Н. Искусственный интеллект: учеб. пособие / В.Н. Бондарев, Ф.Г. Аде.— Севастополь : Изд-во СевНТУ, 2002.—615с.
2. Введение в среду разработки Лисп-приложений Eclipse Cusp: методические указания к лабораторным работам по дисциплине “Методы и системы искусственного интеллекта” для студ. дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии” / СевГУ; сост. В. Н. Бондарев. — Севастополь : Изд-во СевГУ, 2015. — 44 с.
3. Люгер Дж. Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е изд. : Пер. с англ.— М.: Издательский дом «Вильямс» .— 2003.—864с.
4. Рассел С. Искусственный интеллект: современный подход, 2-е изд. / С. Рассел, П. Норвиг: Пер. с англ.— М.: Издательский дом «Вильямс» .— 2006.— 1408с.
5. Функциональное и логическое программирование. Лабораторный практикум. Часть1. Функциональное программирование [Электронный ресурс]/ Авт.-сост. Д.В. Михайлов, Г.М. Емельянов — Великий Новгород: НовГУ им. Ярослава Мудрого, 2007. — 80с. — Режим доступа: http://www.machinelearning.ru/wiki/images/0/0b/Method_fp.pdf . — Последний доступ: 03.01.2012. — Название с экрана.
6. Шалимов П.Ю. Функциональное программирование : учеб. пособие / П.Ю. Шалимов .— Брянск : БГТУ, 2003.— 160с.
7. Seibel P. Practical Common Lisp [Электронный ресурс] / P.Seibel .— New York: Apress, 2005. — Режим доступа: http://www.gigamonkeys.com/book/ — Последний доступ: 03.01.2012. — Название с экрана.

Приложение А
(справочное)
Базовая программа, реализующая алгоритм сопоставления с образцом

```
;;;;<1. Функция сопоставления с образцом

;;;образец - p , данные - d
(defun match (p d)
  (cond
    ;;правило 1
    ((and (null p)(null d)) t)
    ;;правило 2
    ((and (null d)
          (eq (car p) '$)
          (null (cdr p))) t)

    ;;один из списков исчерпан
    ((or (null p)(null d)) nil)

    ;;правило 3 и правило 4
    ((or (equal (car p) '?)
         (equal (car p) (car d)))
     (match (cdr p)(cdr d)))

    ;;правило 5 и 6
    ((eq (car p) '$)
     (cond ((match (cdr p) d) t)
       ((match p (cdr d)) t)))

    ;;правило 7 - сопоставление списков,включающих подсписки
    ((and (not (atom (car p)))
          (not (atom (car d)))
          (match (car p)(car d)))
     (match (cdr p) (cdr d)) )

    ;;правило 8 - подстановка значения в переменную
    ;;пример вызова:(match '(AVTO ?MODEL ?YEAR) '(AVTO ford 1998))
    ;;MODEL=ford, YEAR=1998
    ((and (atom (car p))
          (eq (car-letter (car p)) #\?)
          (match (cdr p)(cdr d)))
     (set  (cdr-name (car p)) (car d)) t)


    ;;правило 9 - подстановка сегмента значений в переменную
    ;;пример вызова: (match '(?F $V) '(Petrov 1975 5 October))
    ;;F=Petrov, V=(1975 5 October)
    ((and (atom (car p))
          (eq (car-letter (car p)) #\$))
     (cond ((match (cdr p)(cdr d))
            (set (cdr-name (car p)) (list (car d)))
            t)
       ((match p (cdr d))
        (set (cdr-name (car p))
             (cons (car d)(eval (cdr-name (car p)))))
        t)))

    ;; правило 10 - обработка пакета ограничений, если в пакете есть «?»
    ((and (not(atom (car p)))
          (eq (caar p)  'restrict)
          (eq (cadar p) '?)
          (and-to-list
```

```
          (mapcar  #'(lambda (pred)
                       (funcall pred (car d))) (cddar p))))
      (match (cdr p)(cdr d)))

    ;; правило 11 - обработка пакета ограничений, если в пакете есть «?V»
    ;; например: (match '((restrict  ?V  integerp  evenp) b c) '(36  b  c))
    ((and (not (atom (car p)))
          (not (atom  d))
          (eq (caar p)  'restrict)
          (eq (car-letter (cadar p)) #\?)
          (and-to-list
           (mapcar  #'(lambda (pred)
                       (funcall pred (car d))) (cddar p)))
          (match (cdr p)(cdr d)))
     (set (cdr-name (cadar p)) (car d))
     t)


    ))

;;;;<2. Вспомогательные функции

;;; выделение первой литеры из имени
(defun car-letter (x)  (if (not (numberp x)) (car (coerce (string x) 'list))))

;;; возвращает имя без первой
(defun cdr-name (x)
  (intern  (coerce (cdr (coerce (string x) 'list)) 'string)))

;;; проверяет,  все ли элементы списка lis имеют значение T
(defun and-to-list ( lis )
  ;lis - список логических значений
  (let ((res t))
  (dolist (temp lis res)
    (setq res (and res temp)))))
```

Приложение Б
(справочное)
Базовая программа, реализующая А-алгоритм

Ниже приведен листинг Лисп-программы , реализующий поиск пути на графе состояний в соответствии с А-алгоритмом. Полное описание программы приведено в [1].

Б.1 Файл main.lisp

```
;;;;<1. Описание графа состояний

;;; граф представляется в виде a- списка с подсписками (p d c h) ,где
;;; p - родительская вершина
;;; d - соответствующая  дочерняя  вершина
;;; c - стоимость участка графа между вершинами p и d
;;; h- оценка стоимости кратчайшего пути из вершины d целевую вершину.

;;; явное описание графа состояний
(setq graph '((s d 1 10)(s c 3 7)(s b 5 4)
              (s a 7 1)(a g 10 0) (b a 1 1)
              (c b 1 4)(c a 2 1)(d c 1 7)))

;;; элементы списков OPEN и CLOSED будут представляться в виде подсписков
;;; ( d p g f), где d - некоторая  вершина на графе состояний
;;; p - ссылка  на родительскую вершину
;;; g - оценка стоимости пути из начальной вершины в вершину d
;;; f - значение  оценочной функции

;;;;<2 Основная Функция поиска в соответствии с А-алгоритмом

(defun a-search(start goal)
  ;; поместить в open стартовую вершину, описываемую
  ;; подсписком со структурой ( s s 0 h)
  (setq open (cons start nil))
  ;; инициализация closed
  (setq closed nil)
  (loop                ; цикл
                       (cond ((null open)(return 'неудача)) ; на графе нет пути
                         (t
                          ;; запомнить первый подсписок списка open
                          (setq n (first open))
                          ;; удалить n из open
                          (setq open (cdr open))
                          ;; внести n в closed
                          (setq closed (cons n closed))
                          ;; проверить, является ли (first n) целевой вершиной
                          (if (eq (first n) goal) (return 'удача))
                          ;; внести в open дочерние вершины и удалить
                          ;; их при необходимости из closed
                          (put-in-list (list-children n))
                          ;; печать промежуточных результатов
                          (terpri)
                          (princ "closed=")(prin1 closed)
                          (terpri)
                          (princ "open=") (prin1 open))))))

;;;;<3 Функция, возвращающая список дочерних вершин для вершины (first n)
(
```

```
defun list-children (n)
  (setq L nil)
  (dolist (temp graph L) ; выполнить действия для каждого подсписка из graph
    ;; переставить первый и второй элементы temp
    (setq rtemp (rev temp))
    ;; выяснить, содержит ли temp дочернюю вершину для (first n)
    (when (eq (first n) (first temp))
      ; вычислить значения оценки g(n) для дочерней вершины
      (setq rtemp (put-third rtemp (+ (third n)(third rtemp))))
      ; вычислить значения оценки f(n) для дочерней вершины
      (setq rtemp (put-fourth rtemp (+ (third rtemp) (fourth rtemp))))
      ;добавляем описание дочерней вершины в виде подсписка rtemp
      ;в результирующий список L
      (setq L (cons rtemp L)))))

;;;;<4 Вспомогательные функции

;;; функция, переставляющая местами первый и второй элементы в temp
(defun rev(temp)
  (setq a (first temp)) (setq temp (rest temp))
  (setq b (first temp)) (setq temp (rest temp))
  (setq temp (cons b (cons a temp))))

;;; функция, меняющая значение третьего элемента в temp (оценка g(n)) на x
(defun put-third (str x)
   (setq temp1 (butlast str 2))
   (setq temp2 (last str))
   (append temp1 (cons x temp2)))

;;; функция, меняющая значение четвертого элемента в temp (оценка f(n)) на x
(defun put-fourth (str x)
  (setq temp1 (butlast str 1))
  (append temp1 (cons x nil)))

;;;Список дочерних вершин, возвращаемый функцией LIST-CHILDREN,
;;;используется в функции PUT-IN-LIST для соответствующей модификации списков OPEN
;;;и CLOSED
(defun put-in-list (dv) ;    dv - a-список дочерних вершин
  (cond
    ;; условие завершения рекурсии
    ((null dv ) nil)
    ;; если очередная дочерняя вершина не входит в
    ;; списки OPEN и CLOSED, то добавить её  в список OPEN
    ((and (not (member1 (caar dv) open))
          (not (member1 (caar dv) closed)))
     (setf open (add (first dv) open))
     (put-in-list (rest dv)))
    ;; если очередная дочерняя вершина  входит в
    ;; один из списков OPEN или CLOSED, то удалить её из
    ;; этих списков и заново внести в список OPEN с учетом значения
    ;; оценочной функции

    (t  (setf open (del (first dv) open))
        (setf closed (del (first dv) closed))
        (setf open (add (first dv) open))
        (put-in-list (rest dv)))))

;;; Функция DEL(V L) выполняет удаление подсписка V, представляющего вершину графа
;;;состояний, из списка L. Удаление выполняется, если оценочная функция для
;;;удаляемой вершины V меньше, либо равна значению оценочной функции вершины из
;;; списка L

(defun del(v l)
  (cond ((null l) nil)  ; завершение рекурсии
```

```
    ;; проверка вхождения v в l
    ((and (eq (first v)(first(first l)))
          ;; сравнение оценочных функций
          (<= (fourth v)(fourth (first l))))
     (setf l (cdr l)))
    (t (append (list(car l)) (del v (cdr l))))))

;;;Вставка дочерних вершин в упорядоченный список OPEN
(defun add(v l)
  (cond ((null l) (cons v l))  ; выход из рекурсии
    ;;сравнить значения оценочных функций
    ((<= (fourth v)(fourth (first l)))
     ;;если меньше, то вставить вершину v в l
     (setf l (cons v l)))
    (t (append (list(car l)) (add v (cdr l))))))

;;; Функция MEMBER1(V L)
(defun member1 (v l)
  (cond ((null l) nil)
    ((eq v (first(first l))) t)
    (t (member1 v (rest l)))))

;;;функция, восстанавливающая по списку closed искомый путь
(defun back-way (goal start)
  (setq g goal)
  (setq L nil)
  (dolist (temp closed L)
    (if (eq (first temp)g)
        (prog1 (setq L (cons (rev temp) L))
               (setq g (second temp))))))
```

Б.2. Результаты выполнения тестового примера

```
a-search>
(a-search '(s s 0 14) 'g)

closed=((S S 0 14))
open=((A S 7 8) (B S 5 9) (C S 3 10) (D S 1 11))
closed=((A S 7 8) (S S 0 14))
open=((B S 5 9) (C S 3 10) (D S 1 11) (G A 17 17))
closed=((B S 5 9) (S S 0 14))
open=((A B 6 7) (C S 3 10) (D S 1 11) (G A 17 17))
closed=((A B 6 7) (B S 5 9) (S S 0 14))
open=((C S 3 10) (D S 1 11) (G A 16 16))
closed=((C S 3 10) (S S 0 14))
open=((A C 5 6) (B C 4 8) (D S 1 11) (G A 16 16))
closed=((A C 5 6) (C S 3 10) (S S 0 14))
open=((B C 4 8) (D S 1 11) (G A 15 15))
closed=((B C 4 8) (C S 3 10) (S S 0 14))
open=((A B 5 6) (D S 1 11) (G A 15 15))
closed=((A B 5 6) (B C 4 8) (C S 3 10) (S S 0 14))
open=((D S 1 11) (G A 15 15))
closed=((D S 1 11) (A B 5 6) (B C 4 8) (S S 0 14))
open=((C D 2 9) (G A 15 15))
closed=((C D 2 9) (D S 1 11) (S S 0 14))
open=((A C 4 5) (B C 3 7) (G A 15 15))
closed=((A C 4 5) (C D 2 9) (D S 1 11) (S S 0 14))
open=((B C 3 7) (G A 14 14))
closed=((B C 3 7) (C D 2 9) (D S 1 11) (S S 0 14))
open=((A B 4 5) (G A 14 14))
closed=((A B 4 5) (B C 3 7) (C D 2 9) (D S 1 11) (S S 0 14))
open=((G A 14 14))
УДАЧА

a-search>
(back-way 'g 's)
((S S 0 14) (S D 1 11) (D C 2 9) (C B 3 7) (B A 4 5) (A G 14 14))
```

Заказ № _____ от « _____ » _____2015г. Тираж _____экз.
Изд-во СевГУ