Министерство образования и науки РФ
ФГБОУ ВПО «ДАГЕСТАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ 1-4 по дисциплине «Логическое и функциональное и программирование» для студентов, обучающихся по направлению подготовки бакалавров 01.03.02 – «Прикладная математика и информатика»

Махачкала 2021

Методические указания к выполнению лабораторных работ 1-4 по дисциплине «Логическое и функциональное и программирование» для студентов, обучающихся по направлению подготовки бакалавров 01.03.02 – «Прикладная математика и информатика» - Махачкала: ДГТУ, 2021г. -30 с.

Методические указания содержат описания лабораторных работ по следующим темам: «Создание электронных презентаций MS PowerPoint» , «Общие сведения об информатике», «Операционная система Windows'7», «Основы работы с текстовым редактором Microsoft Word»; примеры заданий с описанием технологии выполнения; задания к выполнению лабораторных работ; контрольные вопросы.

Выполнение этих работ позволит студентам приобрести практические навыки работы на персональном компьютере.

Составители:

Л.М.Гаджимахадова - ст. преподаватель кафедры прикладной математики и информатики,

Исабекова Т.И. - зав. кафедрой ПМ и И, к.ф.-м.н., доцент,

Алиосманова О.А.- ст. преподаватель кафедры прикладной математики и информатики.

Рецензенты:

Декан факультета информатики и информационных технологий ДГУ, д.т.н., профессор С.А..Ахмедов

доцент кафедры ПОВТиАС,

д.т.н. А.Г. Мустафаев

Печатается согласно постановлению Ученого совета Дагестанского государственного технического университета

«_____»_____ 2021г. (протокол № __ _____)

Лабораторная работа №1

Общий обзор языка Пролог. Основные механизмы программирования .

Цель работы: ознакомление с режимом Test Goal системы программирования Visual Prolog, приобретение практических навыков составления, отладки и выполнения простейшей программы в системе программирования Visual Prolog (режим Test Goal).

Теоретический материал

Рассмотрим простейшие элементы, из которых состоит программа на языке Prolog. Имена переменных начинаются с прописной латинской буквы или подчеркивания и состоят из латинских букв, цифр и символа подчеркивания. Одиночный символ подчеркивания обозначает анонимную переменную. Примеры переменных:

```
_ X Variable_1 This_is_my_var
```

Атомы (константы) начинаются со строчной латинской буквы и состоят из латинских букв, цифр и символа подчеркивания. Кроме того, существуют т. н. графические атомы, которые представляют собой один или более символов из набора

```
# $ & * + - . / : < = > ? @ ^ ~ \
```

Наконец, есть два специальных атома `[]` и `{}`. Примеры атомов:

```
john name_123 =\=
```

Целочисленные константы могут задаваться в десятичной (`12345`, -1), восьмеричной (`0o177`, `0O15`), шестнадцатеричной (`0x89AB`, `0X89ab`) или двоичной (`0b10010111`) системах счисления так же, как в многих языках семейства C. Кроме того, целое число может быть задано в виде `0'x`, что означает числовой код символа `x`. Константы с плавающей запятой также задаются привычным образом, например: `1.0, 3.1415E0, 1E+2, -123.456e-8`.
Отметим, что знак минус перед числом является частью этого числа.

Текстовые строки заключаются в апострофы: `'Hello World'`. Как и в языке C, можно использовать для задания символов escape-последовательности, начинающиеся с символа \ (обратная косая черта). Сюда относятся следующие способы задания символов:

- Буквенное задание специальных символов: `'\\'` (символ \), `'\a'` (звуковой сигнал), `'\b'` (забой символа), `'\f'` (перевод формата), `'\n'` (перевод строки), `'\r'` (возврат каретки), `'\t'` (горизонтальная табуляция), `'\v'` (вертикальная табуляция).
- Числовое задание символов их кодом в кодировке ASCII: `'\377\'` (восьмеричный код), `'\xff\'` (шестнадцатеричный код).

Однострочные комментарии начинаются с лексемы % и продолжаются до конца строки. Многострочные комментарии) заключаются в лексемы /* и `*/`.
Вложение комментариев одного типа друг в друга запрещено, но однострочный комментарий может быть вложен в многострочный:

```
/* % таким образом */
```

Структура программы

Программа на языке Prolog представляет собой набор директив и предикатов. Каждая директива или предикат должны заканчиваться точкой, за которой следует перевод строки или пробел. Предикат определяется своим именем и количеством аргументов. *n*местный предикат с именем *name* в документации принято кратко обозначать как *name/n*. В программе на языке Prolog предикат задается как набор *утверждений* (clauses), каждое из которых представляет собой либо *факт*, либо *правило*. Рассмотрим эти понятия на примерах.

Высказывание «X счастлив, если он богат и знаменит» может быть записано на Prolog'е следующим правилом:

```
happy(X) :- rich(X), famous(X).
```

Слева от атома `:-` указывается определяемый предикат (в нашем случае *happy/1*), а справа – условие его истинности. Символ запятая обозначает конъюнкцию, т. е. логическое «и».

Высказывание «X счастлив, если он или богат и знаменит, или здоров» записывется на Prolog'е двумя правилами:

```
happy(X) :- rich(X), famous(X).
happy(X) :- wealthy(X).
```

Таким образом, логическое высказывание должно быть представлено как дизъюнкция высказываний, каждое из которых представляет собой конъюнкцию простых высказываний. Каждый из элементов дизъюнкции записывается как отдельное правило, в котором конъюнкты разделены запятыми.

Факт – это утверждение, не содержащее атома `:-`, например:

```
rich(john). famous(john).
wealthy(mary).
```

Отметим, что факты могут содержать переменные. Например, утверждение «все любят жизнь» может быть записано фактом

```
likes(X, life).
```

Рассмотрим, каким образом Prolog-машина исполняет запрос пользователя. В качестве примера будем использовать следующую программу, описывающую библейских персонажей.

```
male(isaac).                                % 1 male(lot).
% 2
 female(yiscah).                             % 3
female(milcah).                             % 4
 father(abraham, isaac).                     % 5
father(haran, lot).                         % 6
father(haran, milcah).                      % 7
father(haran, yiscah).                      % 8
 son(X, Y) :- father(Y, X), male(X).        % 9
daughter(X, Y) :- father(Y, X), female(X). % 10
```

Здесь определены предикаты:

male(X) – «X - мужчина»; **female(X)** – «X - женщина»; **father(X, Y)** – «X – отец Y»; **son(X, Y)** – «X – сын Y»; **daughter(X, Y)** – «X – дочь Y».

Предположим, что мы задали цель **son(lot, haran)**. Для проверки истинности данной цели Prolog-машина просматривает список заданных в программе предикатов, пытаясь найти предикат *son/2*. При этом предикаты просматриваются в том порядке, как они заданы в программе. В нашем случае будет найдет предикат в строке 9 и машина произведет *редукцию* цели **son(lot, haran)** к цели **father(haran, lot), male(lot)**. Редукция – это основной шаг вычислений в логическом программировании, который состоит в замене текущей цели на новую цель, заданную правой частью правила. Теперь для проверки истинности новой цели необходимо установить истинность обоих конъюнктов **father(haran, lot)** и **male(lot)**. Производится просмотр всего списка правил с самого начала с целью проверки первой подцели. В нашем случае первая подцель соответствует факту 6, поэтому цель будет редуцирована к цели **male(lot)**. Еще один просмотр, и факт 2 редуцирует цель к пустой, что означает завершение работы с ответом *yes*.

Унификация переменных

Предположим теперь, что мы задали цель **daugther(X, Y)**. Это означает, что мы хотим найте все пары значений переменных **X** и **Y**, при которых данное высказывание истинно. Первая редукция по правилу 10 даст новую цель **father(Y, X), female(X)**. Каким образом будет проверяться истинность первой подцели? Prolog-машина найдет первый факт **father(abraham, isaac)** и отождествит переменную **X** с атомом **abraham**, а переменную **Y** с атомом **isaac**. Эта процедура называется *унификацией* переменной с конкретным значением (константой). При каждой унификации машина запоминает точку, в которой эта унификация произошла. Теперь цель редуцирована к **female(isaac)**; однако, попытка найти такой факт окончится неудачей. В результате будет произведен откат к последней запомненной точке, и машина попытается произвести новую унификацию переменных. Унификация **X=haran, Y=lot** вновь приводит к неудаче и откату. Следующая унификация **X=haran, Y=milcah** дает редукцию цели к **female(milcah)**, которая успешно редуцируется к пустой цели благодаря факту 4. Таким образом, мы получаем первый ответ: **X=haran, Y=milcah**. Поскольку просмотр фактов для предиката **father** еще не окончен, вновь произойдет откат и повторный поиск решения. На этот раз мы получим успешную унификацию **X=haran, Y=yiscah**, и только теперь работа будет завершена.

Таким образом, работа Prolog-машины состоит в переборе вариантов, при котором производится унификация переменных цели со всеми возможными их значениями, заданными в программе. Каждая унификация создает свою точку возврата, при возвращении в которую переменнная, унифицированная с конкрентным значением, вновь становится неопределенной. Работа программы завершается, когда будут перебраны все возможные варианты унификации переменных.

Управление процессом перебора

Программист может управлять процессом перебора с помощью предиката *отсечения* (cut) **!**. Этот предикат удаляет все точки возврата, созданные с момента активации предиката, в котором он расположен, и успешно завершает текущую ветвь перебора.

Пояснить работу отсечения можно на следующем простом примере.

```
man(a). man(b).  woman(c). woman(d).
pair(X, Y) :- man(X), woman(Y).
```

Если мы зададим цель **pair(X, Y)**, то получим четыре варианта ответов:

X = a Y = c
X = a Y = d
X = b Y = c
X = b Y=d

Изменим теперь определение предиката **pair** на следующее:

```
pair(X, Y) :- man(X), !, woman(Y).
```

Это означает, что после первой унификации переменной **X** ее значение будет зафиксировано. Действительно, та же цель теперь дает всего два ответа:

X = a Y = c
X = a Y = d

Наконец, изменим предикат **pair** еще раз:

```
pair(X, Y) :- man(X), woman(Y), !.
```

Мы запретили откат после унификации переменных **X** и **Y**. Теперь, как и следовало ожидать, работа программы завершается после нахождения первого ответа:

X = a Y = c

В сочетании с отсечением часто используются два нульместных встроенных предиката: **true** (успешное завершение ветви перебора) и **fail** (неудачное завершение, вызывающее откат).

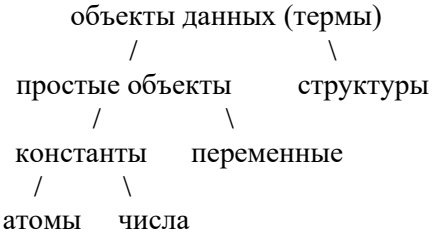
В дополнение к запятой (конъюнкция) есть еще две управляющие конструкции. Конструкция **A; B** представляет собой операционный аналог дизъюнкции. Она обрабатывается следующим образом: сначала создается точка возврата и исполняется **A**. Если при этом произойдет возврат, то исполняется **B**.

Конструкция **A -> B** сначала исполняет **A**. Если исполнение **A** заканчивается неудачей, то выполнение всей конструкции закончено неудачей. В противном случае после исполнения **A** исполняется **B**.

Эти две конструкции могут сочетаться, образуя аналог условного оператора «если A то B, иначе C»: **A -> B ; C**.

Упомянем здесь еще один предикат, который представляет собой операционный аналог отрицания. **\+ A** завершается успешно, если выполнение A заканчивается неудачей, и наоборот.

Объекты данных в Прологе называются термами. **Терм** может быть: константой, переменной, структурой (составной терм). Классификация объектов данных приведена на схеме:



Атом - это синтаксически неделимый терм. В качестве атома могут быть: **имя** Visual Prolog (имя – это начинающаяся с буквы последовательность букв, цифр и символа подчеркивания), строчное представление числа (например, “1048”, “25.6”), отдельный символ (кроме пробела).

Числа в Visual Prolog записываются точно так же, как и в других языках программирования.

Константы относятся к одному из стандартных доменов (типов), приведенных в таблице 1 (кроме приведенных в таблице Visual Prolog имеет и другие стандартные домены: byte, short, ushort, word, unsigned, long, ulong, dword).

Переменная - имя, начинающееся с большой (прописной) буквы или знака подчеркивания. Когда значение переменной неважно, то в качестве имени переменной используется знак подчеркивания. Такая переменная называется **анонимной**.

Структуры (сложные термы) - это объекты, которые состоят из нескольких компонент. Структура записывается с помощью указания ее **функтора** и **компонент**. Компоненты заключаются в круглые скобки и разделяются запятыми. Число компонент в структуре называется **арностью** структуры.

Пример структуры: data_r(12,mart,1962). Здесь data_r – функтор, 12, mart, 1962 – компоненты. Арность приведенной структуры равна трем.

Программа на Visual Prolog состоит из нескольких разделов, каждому из которых предшествует ключевое слово. Типичная простейшая структура программы представлена ниже:

```
/*     комментарии       */ domains
          <описание доменов (типов данных)> predicates
                    <описание предикатов>
clauses
                    <предложения или утверждения>
goal
                    <целевое утверждение>
```

В программе не обязательно наличие всех разделов. Обычно в программе должны быть по крайней мере разделы predicates и clauses.

В разделе domains объявляются любые нестандартные домены, используемые для аргументов предикатов. Домены на Прологе являются аналогами типов в других языках.

Наиболее простым способом описания доменов в разделе domains является следующий: name=d, где name – нестандартные домены, определенные пользователем; d - один из стандартных доменов (integer, real, char, string, symbol и др.).

Предикат (отношение) в общем случае - это структура вида: predname(komp1,komp2,...), где predname - имя предиката, komp1,... - **типы** компонентов, описанные в разделе domains или стандартные типы.

Например, domains

```
fio=string
den,god=integer
    mes=symbol predicates
anketa(fio,den,mes,god)
```

Если в предикатах используются только стандартные типы данных, то раздел domains может отсутствовать:

```
    predicates
        anketa(string,integer,symbol,integer)
```

Предикат может состоять только из одного имени, напр.,

```
predicates result
```

В разделе clauses размещаются **предложения** (утверждения). Предложение представляет собой факт или правило, соответствующее одному из объявленных предикатов.

Факт - простейший вид утверждения, которое устанавливает отношение между объектами. Пример факта: anketa(“Иванов”,5,august,1950).

Этот факт содержит атом anketa, который является именем предиката, и в скобках после него - список термов, соответствующих компонентам этого предиката. **Факт всегда заканчивается точкой**. Факты содержат утверждения, которые всегда являются безусловно верными. Компонентами факта могут быть только константы.

Все предложения раздела clauses, описывающие один и тот же предикат, должны записываться друг за другом.

Практическое задание

1. Запустить среду визуальной разработки **Visual Prolog** двойным щелчком мыши по пиктограмме.
2. Создать новый проект (выбрать команду **Project – New Project**, активизируется диалоговое окно **Application Expert**).
3. Определить имя проекта и рабочий каталог.

Имя проекта набирается в поле **Project Name**. После щелчка в поле **Name of .VPR File** появляется имя проекта с расширением **.vpr**.

На вкладке **Target** выбрать параметры: **Windows32** в списке **Platform**, **Easywin** – в списке **UI Strategy**, **exe** – в списке **Target Type**, **Prolog** – в списке **Main Program**.

В поле **Base Directory** на вкладке **General** указать директорий для проекта (можно создать новый директорий, используя кнопку **Browse…**).

4. Нажать кнопку **Create** для того, чтобы создать файлы проекта по умолчанию.

Открыть окно редактора (использовать команду меню **File –New**). Появится новое окно редактирования с именем **noname.pro**. Редактор среды визуальной разработки – стандартный текстовый редактор. Можно использовать клавиши управления курсором и мышь так же, как и в других редакторах. Он поддерживает команды **Cut, Copy, Delete, Undo** и **Redo**, которые находятся в меню **Edit**. В меню **Edit** также показаны комбинации «горячих» клавиш для этих действий.

5. Набрать текст программы (см. в п.4).
6. Выполнить программу (команда **Project – Test Goal** или комбинация клавиш **Ctrl-G**).
7. **Обработка ошибок.** При наличии ошибок в программе отобразится окно **Errors (Warnings)**, которое будет содержать список обнаруженных ошибок. Дважды щелкнув на одной из этих ошибок, попадаем на место ошибки в исходном тексте. Можно воспользоваться клавишей **F1** для вывода на экран интерактивной справочной системы Visual Prolog. Когда окно помощи откроется, следует щелкнуть по кнопке **Search**, набрать номер ошибки, и на экране появится соответствующее окно помощи с более полной информацией об ошибке.
8. **Проверка правильности настройки системы.** Для проверки того, что система настроена должным образом, можно набрать в окне редактора, например, следующий текст:

GOAL write (“Добро пожаловать”),nl.

Этого достаточно для программы, чтобы она могла быть выполнена. Результат выполнения программы **Добро пожаловать yes**

будет расположен в отдельном окне, которое необходимо закрыть перед тем, как выполнять другую программу.

9. **Установка шрифта.** Если имеются проблемы с использованием в программе кириллицы, следует щелкнуть правой клавишей мыши в любом месте поля редактора, в появившемся контекстном меню выбрать команду **Font…**, затем в открывшемся диалоговом окне выбрать шрифт **System** или **Tahoma**.

Содержание отчёта

Отчет по лабораторной работе должен содержать: цель работы, постановка задачи, текст Пролог-программы, вопросы к программе и ответы системы, выводы.

Контрольные вопросы:

1. Где был разработан Пролог?
2. К какой группе языков относится Пролог?
3. Что является теоретической основой Пролога?
4. Что такое терм? Какие объекты данных Пролога могут использоваться в качестве термов?
5. Что может быть в качестве атома?
6. Как записываются в Visual Prolog числа?
7. Чем отличаются записи констант и переменных?
8. Что такое анонимная переменная, когда она используется?
9. Форма записи структур (сложных термов).
10. Из каких основных разделов состоит программа Visual Prolog, назначение этих разделов?
11. Может ли быть в программе Visual Prolog несколько одинаковых разделов?
12. С какого раздела начинается выполнение программы Visual Prolog?
13. В каком разделе программы описываются пользовательские (нестандартные) типы данных?
14. Что такое предикат и в каких разделах программы его имя может использоваться?
15. Что является аналогом предиката в процедурных языках?
16. Какие объекты могут быть в качестве компонент факта?
17. Какой тип предложения записывается в разделе goal?
18. Способы записи комментариев в программе Visual Prolog.

Лабораторная работа №2

Организация рекурсивных программ. Рекусивные структуры данных. Использование отсечений в Пролог программах

Цель работы: приобретение практических навыков составления и отладки программ с использованием рекурсии, приобретение практических навыков использования отсечения в программах на Турбо-Прологе.

Теоретический материал

Одним из основных методов программирования на языке Prolog является рекурсия. Для того, чтобы привести примеры рекурсивных предикатов, кратко рассмотрим, как в Prolog’е реализованы арифметические операции.

Арифметическое выражение строится из чисел, переменных и операций. При вычислении значение выражения каждая переменная должна быть унифицирована с константой. Значением выражения является либо целое, либо плавающее число. В следующей таблице собрана информация о всех арифметических операциях языка Prolog.

Выражение	Результат	Комментарий
переменная	значение выражения, с которой она унифицирована	IF → IF
целое число	это число	I → I
плавающее число	это число	F → F

`- E`	унарный минус	IF → IF
`E1 + E2`	сложение	IF, IF → IF
`E1 - E2`	вычитание	IF, IF → IF
`E1 * E2`	умножение	IF, IF → IF
`E1 / E2`	деление	IF, IF → F
`E1 // E2`	частное, округленное до целого	I, I → I
`E1 rem E2`	остаток от деления	I, I → I
`E1 mod E2`	остаток по модулю	I, I → I
`E1 /\ E2`	побитовое «и»	I, I → I
`E1 \/ E2`	побитовое «или»	I, I → I
`\ E`	побитовое «не»	I → I
`E1 << E2`	битовый сдвиг влево	I, I → I
`E1 >> E2`	битовый сдвиг вправо	I, I → I
`abs(E)`	абсолютное значение	IF → IF
`sign(E)`	знак числа (-1, 0 или 1)	IF → IF
`E1 ** E2`	возведение в степень	IF, IF → F
`sqrt(E)`	квадратный корень	IF → F
`atan(E)`	арктангенс	IF → F
`cos(E)`	косинус	IF → F
`sin(E)`	синус	IF → F
`exp(E)`	экспонента	IF → F
`log(E)`	натуральный логарифм	IF → F
`float(E)`	преобразование в плавающее чсло	IF → F
`ceiling(E)`	округление вверх до целого	F → I
`floor(E)`	округление вниз до целого	F → I
`round(E)`	округление до ближайшего целого	F → I
`truncate(E)`	усечение до целого	F → I
`float_fractional_part(E)`	дробная часть числа	F → F
`float_integer_part(E)`	целая часть числа	F → F

Поле «Комментарий» имеет следующий смысл:

- I → I: унарная операция с целым аргументом и целым результатом.
- F → F: унарная операция с плавающим аргументом и плавающим результатом.
- F → I: унарная операция с плавающим аргументом и целым результатом.
- IF → F: унарная операция с целым или плавающим аргументом и плавающим результатом.
- IF → IF: унарная операция с целым или плавающим аргументом и результатом того же типа.
- I, I → I: бинарная операция с целыми аргументами и целым результатом.
- IF, IF → IF: бинарная операция с целыми или плавающими аргументами; если оба аргумента целые, то результат целый, в противном случае – результат плавающий.

Встроенный инфиксный предикат Result is Expression истинен, если Result может быть унифицирован со значением выражения Expression.

Для сравнения числовых значений используются следующие инфиксные предикаты:

- E1 =:= E2 истинно, если E1 равно E2.
- E1 =\= E2 истинно, если E1 не равно E2.
- E1 < E2 истинно, если E1 меньше E2.
- E1 <= E2 истинно, если E1 меньше или равно E2.
- E1 > E2 истинно, если E1 больше E2.
- E1 >= E2 истинно, если E1 больше или равно E2.

Примеры

Пользуясь только что введенными обозначениями, дадим рекурсивное определение факториала. Здесь **fact(N, F)** истинно, если **F = N!**.

```
fact(1, 1).
fact(X, F) :- fact(X1, F1), X is X1 + 1, F is F1 * X.
```

Первое утверждение задает базу рекурсии: 0! = 1, а второе – шаг рекурсии: *n*! = *n* * (*n*-1)!. Обратите внимание, что Prolog позволяет нам запросить и цель **fact(5, F)**, т. е. вычисление значения 5!, и цель **fact(X, 120)**, т. е.
вычисление такого X, что X! = 120.

Аналогичным образом можно определить числа Фибоначчи:

```
fib(0, 1). fib(1, 1). fib(N, F) :- N > 1, N1 is N - 1, N2
is N - 2,              fib(N1, F1), fib(N2, F2), F is F1 +
F2.
```

Еще одним примером может случить вычисление наибольшего общего делителя двух неотрицательных чисел с помощью алгоритма Евклида.

```
gcd(X, 0, X) :- X > 0. gcd(X, Y, G) :- Y > 0, Z is X mod Y, gcd(Y,Z,G).
```

Классическим примером рекурсии является вычисление факториала F=N! = 1*2*3*...*(N-1)*N. Это выражение можно записать так: N!=(N-1)!*N. Последнее выражение - пример рекурсивных вычислений, т.е. каждый последующий результат получается путем использования предыдущего результата. Для того чтобы рекурсивный метод решения был результативным, он должен в конце концов привести к задаче, решаемой непосредственно. Решить ее позволяют утверждения, называемые граничными условиями (для нашего случая граничным условием является 0!=1). На Прологе процедура вычисления факториала может быть описана следующим образом:

```
/* Граничное условие */ factorial(0,1).   % 0!=1
/* Рекурсивное правило вычисления F=N! */ factorial(N,F):-
% F=N!                N>0,N1=N-1,                factorial(N1,F1),
% F1=(N-1)!                F=F1*N.            % N!=(N-1)!*N
```

При использовании стандартных предикатов ввода-вывода, позволяющих запрограммировать удобный интерфейс пользователя с программой, программа вычисления факториала на Visual Prolog может быть записана в следующем виде:

```
/* Вычисление факториала F=N! */ domains n=integer f=real
predicates factorial(n,f) result clauses factorial(0,1). factorial(N,F):-
```

```
N>0,N1=N-1, factorial(N1,F1),F=F1*N.   result:-   write("Введите число
N"),nl,   write("N="),readint(N),
          factorial(N,F),   write(N,"!=",F),nl.
      goal   result.
```

В приведенной процедуре вычисления факториала тело правила начинается с рекурсивного вызова определяемого предиката. Такая рекурсия называется **левосторонней (нисходящей).**

Следующая программа демонстрирует другой вариант определения факториала. Здесь **рекурсивный** вызов заменен на **итеративный (восходящая рекурсия).** Итеративной процедурой называется такая процедура, которая вызывает себя только один раз, причем этот вызов расположен в самом конце процедуры. Это так называемая **правосторонняя рекурсия**.

```
/* Программа итеративного вычисления факториала */ predicates
  factorial (integer, real)
  factorial_aux(integer, real, integer, real) clauses
factorial(N,F):-factorial_aux(N,F,0,1).   factorial_aux(N,F,I,P):-
        I<N, NewI=I+1, NewP=P*NewI,   factorial_aux(N,F,NewI,NewP).
factorial_aux(N,F,I,F):-I>=N.

  goal   write("N="), readinf(N),
    factorial(N,F),   write("F=", F),nl.
```

Для обеспечения итеративного характера процедуры возникла необходимость определить вспомогательный предикат factorial_aux с двумя дополнительными параметрами: первый из них служит для проверки условия окончания цикла, второй – для возврата вычисленного значения через всю порожденную итеративную последовательность вызовов.

Использование отсечения в пролог-программах.

В процессе достижения цели Пролог-система осуществляет автоматический перебор вариантов, делая возврат при неуспехе какого-либо из них. Такой перебор – полезный программный механизм, поскольку он освобождает пользователя от необходимости программировать этот перебор самому. С другой стороны, ничем не ограниченный перебор может быть источником неэффективности программы. Поэтому иногда требуется его ограничить или исключить вовсе. Для этого в Прологе предусмотрено специальное целевое утверждение **!**, называемое **отсечением** (cut).

Отсечение реализуется следующим образом: после согласования целевого утверждения, стоящего перед отсечением, все предложения с тем же предикатом, расположенные после отсечения, не рассматриваются.

Можно выделить три основные случая использования отсечения:

1) **для устранения бесконечного цикла**; Пример: вычисление суммы 1 + 2 + … + N. сумма(1, 1) :- !. сумма(N, R) :- N1=N-1, сумма(N1, R1), R = R1 + N. Если граничное условие будет записано в виде сумма(1, 1).

без отсечения, то сопоставление головы правила с запросом в разделе goal будет происходить успешно и при N <= 0, т.е. делается попытка доказать цель сумма(0, R), что в свою очередь приводит к цели сумма(-1, R) и т.д., т.е. образуется бесконечный цикл.

2) **при программировании взаимоисключающих утверждений**;

Пример: нахождение большего числа среди двух чисел X и Y можно запрограммировать в виде отношения max(X, Y, Max), где Max = X, если X >= Y и Max = Y, если Y > X. Это соответствует двум предложениям программы: max1(X, Y, X) :- X >= Y. max1(X, Y, Y) :- X < Y.

Эти предложения являются взаимоисключающими, т.е. если выполняется первое, второе обязательно терпит неудачу, и наоборот. Поэтому возможна более экономная запись при использовании отсечения: max2(X, Y, X) :- X >= Y, !. max2(_, Y, Y).

3) **при необходимости неудачного завершения доказательства цели**;

Пример: присвоение какому-либо объекту категории в зависимости от величины заданного критерия в соответствии с таблицей 2.

Таблица 2. Присвоение категории

Критерий	Категория
свыше 80 до 100	A
свыше 40 до 80	B
от 0 до 40	C

Возможный вариант программы: категория(Кр, _) :- Кр > 100, !, fail; Кр < 0, !, fail. категория(Кр, ‘A’) :- Кр > 80, !. категория(Кр, ‘B’) :- Кр > 40, !.

категория(_, ‘C’).

При запросе категория(200, X) произойдет сопоставление запроса с головой первого утверждения. Цель Кр > 100 будет достигнута. Затем будет доказана цельотсечение. Но когда встретится предикат fail, который всегда вызывает состояние неудачи, то стоящий перед ним предикат отсечения остановит работу механизма возврата и в результате ответом на запрос будет No Solution (Нет решения). Комбинация !, fail вводит возникновение неудачи.

При объявлении доменов Visual Prolog позволяет использовать рекурсию. Это бывает полезным при формализации знаний о некоторых объектах. В такого рода объявлениях, как правило, используются альтернативные домены, представленные в виде структур.

В качестве примера рассмотрим, каким образом формализуются знания при расчете сопротивления последовательно-параллельной электрической схемы, приведенной на рис. 3.

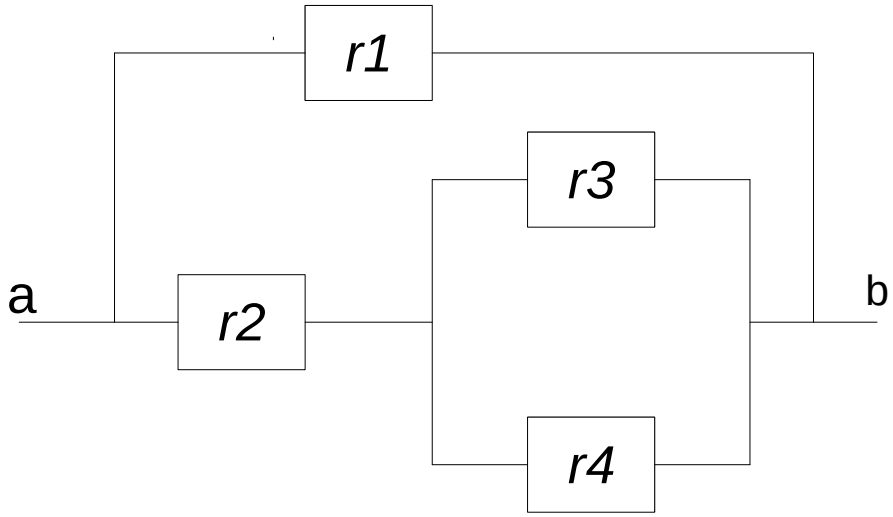


Рис. 3. Расчетная электрическая схема

Введем домен, определяющий представление схемы: отдельным резистором, последовательным соединением резисторов, параллельным соединением резисторов.

typ=res(symbol);
posl(typ,typ); par(typ,typ) В структуре res(symbol) параметр в скобках определяет символическое имя отдельного резистора или некоторой схемы из резисторов.

Назначение вводимых предикатов:

resist(симв_имя, Сопр) – описание резистора, обозначенного на схеме именем **симв_имя** и имеющего сопротивление **Сопр** (или схемы с именем симв_имя и сопротивлением Сопр); **schema(симв_имя, опис_схемы)** – описание схемы соединения резисторов; первый параметр – символическое имя схемы, второй – описание последовательности соединения элементов схемы; **sopr_el(опис_схемы, Сопр)** – определение сопротивления **Сопр** отдельного варианта

схемы **опис_схемы**: резистора (или схемы), последовательной ветви, параллельной ветви;
result(симв_имя, Сопр) – целевое утверждение для определения общего сопротивления **Сопр** рассчитываемой схемы с именем **симв_имя**.

Текст Пролог-программы

```
domains     typ=res(symbol);
posl(typ,typ);
        par(typ,typ) predicates
resist(symbol, real)
schema(symbol, typ)
sopr_el(typ, real)
result(symbol, real) goal
result(ab, R).
clauses resist(r1,3). resist(r2,5). resist(r3,4). resist(r4,8). schema(ab,
par(res(r1),posl(res(r2),par(res(r3),res(r4))))).  sopr_el(res(E),R):-resist(E,R).
sopr_el(posl(C1,C2),R):- sopr_el(C1,R1), sopr_el(C2,R2),R=R1+R2.  sopr_el(par(C1,C2),R):-
sopr_el(C1,R1), sopr_el(C2,R2),R=(R1*R2)/(R1+R2).
   result(N,R):-schema(N,C),sopr_el(C,R).
```

Практическое задание

1. Возвести вещественное число a в целую степень n (степень n может быть положительной, нулевой или отрицательной). Составить два варианта программы:

а) используя рекурсивное выражение a^n=a^(n-1)*a;

б) используя проверку четности степени: для нечетной степени используется рекурсивное выражение варианта а), для четной степени - a^n = a^(n/2)*a^(n/2).

2. Задачи на рекурсию.

Используя рекурсию, запрограммировать решение следующих задач:

3.1. Найти сумму целых последовательных чисел в интервале от M до N (M<=N).

3.2. Просуммировать целые последовательные нечетные числа, т.е. 1+3+5+7+…, от 1 до N (учесть, что при вводе значения N оно может быть четным).

3.3. Вычислить значения следующих функций, используя разложение в ряд. Разработать два варианта программы – рекурсивный и итеративный, в итеративном варианте задавать точность вычисления функции. Полученный результат сверить со значением соответствующей стандартной функции для вычисляемого аргумента.

a) e = 1 + 1/1! + 1/2! + …+ 1/k! + … (k=0,1,2,…);
b) ln2 = 1 – 1/2 +1/3 - … + (-1)^(k-1)/k + … (k=1,2,3,…);
c) □/4 = 1 – 1/3 + 1/5 - … + (-1)^(k-1)/(2*k-1) + … (k=1,2,3,…);
d) □^2/6 = 1 + 1/2^2 + 1/3^2 + … + 1/k^2 + … (k=1,2,3,…);
e) □^2/12 = 1 – 1/2^2 + 1/3^2 - … + (-1)^(k-1)/k^2 + … (k=1,2,3,…);
f) □^2/8 = 1 + 1/3^2 + 1/5^2 + … + 1/(2*k-1)^2 + … (k=1,2,3,…);
g) e^x = 1 + x/1! + x^2/2! + … + x^n/n! + … (n=0,1,2,…);
h) sin(x) = x – x^3/3! + x^5/5! - … + (-1)^(m-1)*x^(2*m-1)/(2*m-1)! + … (m=1,2,3,…);
i) cos(x) = 1 – x^2/2! + x^4/4! - … + (-1)^m*x^(2*m)/(2*m)! + … (m=0,1,2,…);
j) arctg(x) = x – x^3/3 + x^5/5 - … + (-1)^(m-1)*x^(2*m-1)/(2*m-1) + … (m=1,2,3,…).

3. Определить возрастной статус человека по известному году рождения в соответствии с таблицей. Разработать два варианта программы: без отсечения и с использованием отсечения.

Возрастной статус человека

Возраст, лет	Статус
< 2	младенец
2 – 12	ребенок

12 – 16	подросток
16 – 25	юноша
25 – 70	мужчина
70 – 100	старик
> 100	долгожитель

Содержание отчёта

Отчет должен содержать: цель работы, постановка задачи, исходный тексты программы, результаты тестирования, выводы.

Контрольные вопросы:

1. Что такое рекурсия?
2. Основная цель использования рекурсии в программах Visual Prolog.
3. Назначение граничного условия в рекурсивном правиле.
4. Принцип построения граничного условия.
5. Может ли быть в рекурсивном правиле более одного граничного условия?
6. Как программируются разветвления в программах Visual Prolog?
7. Левосторонняя (нисходящая) и правосторонняя (восходящая) рекурсии.
8. Принципиальная разница между рекурсивной и итеративной процедурами.
9. Правила сопоставимости двух термов.
10. Поиск с возвратом в программах Visual Prolog.
11. Поточный шаблон предиката.
12. Стандартные предикаты ввода-вывода Visual Prolog.
13. Для чего используется отсечение?
14. Как обозначается отсечение в программе?
15. Механизм работы отсечения.
16. «Красное» и «зеленое» отсечения.
17. Какое отсечение, «красное» или «зеленое» , в процедуре max2 ?
18. Повлияет ли на результат удаление отсечения в процедуре max2 ?
19. Что происходит после выполнения комбинации !, fail?
20. Могут ли располагаться подцели после предиката fail ?
21. Декларативный и процедурный смысл Пролог-программы.
22. Важен ли порядок следования предложений в процедурах max1 и max2 ?
23. Почему домен typ, объявленный в программе раздела 2, является рекурсивным ?
24. Что означают символы «;» в объявлении домена typ ?
25. Объяснить, как производится описание электрической схемы в программе .
26. Можно ли записать в разделе goal целевое утверждение следующим образом: result(AB, R) ?
27. Какие изменения нужно произвести, чтобы в результате выполнения программы определялось общее сопротивление параллельной ветви, составленной из резисторов r3 и r4 ?
28. Как можно описать схему, представленную тремя параллельными ветвями ?
29. При работе с какими другими объектами, кроме электрической схемы, могут использоваться рекурсивные домены ?

Лабораторная работа № 3

Двоичные деревья. Средства ввода-вывода

Цель работы: приобретение практических навыков составления и отладки программ, работающих с B+деревьями.

Теоретический материал

B+дерево является структурой данных, которую можно применять для очень эффективного метода сортировки большого количества данных.

B+деревья находятся во внешней базе данных. Каждый вход в B+дерево – это пара величин: ключ (информационное поле записи, по которому построено B+дерево) и связанный с этим ключом указатель базы данных. При вводе новой записи во внешнюю БД Пролог включает ключ и указатель, соответствующие этой записи, в B+дерево.

B+дерево разбито на отдельные страницы или узлы. Каждый узел B+дерева является либо последним (лист), либо порождает два нижележащих узла. Каждый узел содержит группу ключей из заданного диапазона. По значению ключа можно быстро проверить, не находится ли искомый ключ внутри диапазона конкретного узла. Если да, то быстро находится указатель записи; если значение ключа меньше диапазона ключей узла, то поиск ведется по левой ветви B+дерева, если больше – по правой.

B+дерево **создается** с помощью предиката

bt_create(ИмяБД, ИмяB+дер, ПерB+дер, ДлКл, Пор)(db_selector, string, bt_selector, integer, integer): (i,i,o,i,i)

ИмяБД – имя внешней БД, в которой создается B+дерево;

ИмяB+дер – имя, которое дает B+дереву пользователь;

ПерB+дер – переключатель (ссылка), используемый другими предикатами при выполнении операций с B+деревом;

ДлКл – длина ключа, в пределах которой будет производиться упорядочивание ключей при построении дерева;

Пор – порядок дерева, определяющий, сколько ключей запоминается в каждом узле B+дерева. Для баз данных среднего размера рекомендуется Пор=4.

Для **открытия, закрытия и удаления** B+дерева используются предикаты:

bt_open(ИмяБД, ИмяB+дер, ПерB+дер)(db_selector,string,bt_selector):(i,i,o)
bt_close(ИмяБД, ПерB+дер)(db_selector, bt_selector):(i,i) bt_delete(ИмяБД, ИмяB+дер)(db_selector,bt_selector):(i,i)

Ниже приведен пример правила, проверяющего, существует ли открываемое B+дерево; если его нет, то оно создается:

```
external_open(Bt_sel):-
existfile("parts.dba"),!,    db_open(parts,
"parts.dba",in_file),
bt_open(parts,"parts.tree",Bt_sel).
external_open(Bt_sel):-
        db_create(parts,"parts.dba",in_file),    bt_create(parts,"parts.tree",Bt_sel,10,4).
```

Изменения в B+дереве относительно ключа осуществляются двумя предикатами:

1) key_insert(ИмяБД, ПерB+дер, Ключ, Ссылка)(db_selector, bt_selector, string, ref):(i,i,i,i)

- **вставляет** новый ключ (информационное поле) в B+дерево. Параметр Ключ является новым ключом, а Ссылка является номером ссылки БД, принадлежащим этому ключу. **Пример:** domains db_selector=mydba dbdom=person(family,name,sex)

```
        family, name=string   sex=char goal
db_open(mydba,"dd.bin",in_file),
bt_open(mydba,"personnames",Bt_sel),
        chain_inserta(mydba,namechain,,dbdom,person("Hoffman","Artur",'m',),Ref),
key_insert(mydba,Bt_sel,"Artur",Ref),   db_close(mydba).
```

2) key_delete(ИмяБД, ПерB+дер, Ключ, Ссылка)(db_selector, bt_selector, string, ref):(i,i,i,i)

- **удаляет** ключ из B+дерева.

Для осуществления **поиска** в B+дереве используются следующие предикаты:

key_search(ИмяБД, ПерB+дер, Ключ, Ссылка)(db_selector,bt_selector,string,ref):(i,i,i,o) - находит ключ в B+дереве. Если ключ найден, номер ссылки БД, принадлежащий этому ключу, будет возвращен через параметр Ссылка. Если ключ не найден, предикат будет несогласован, однако внутренний указатель B+дерева будет указывать на ключ, следующий непосредственно за тем, на который указатель указывал до поиска ключа.

key_current(ИмяБД, ПерB+дер, Ключ, Ссылка)(db_selector,bt_selector,string,ref):(i,i,o,o) - возвращает текущий ключ и номер ссылки БД для текущего внутреннего указателя B+дерева.

key_first(ИмяБД, ПерB+дер, Ссылка) (db_selector, bt_selector, ref):(i,i,o) key_last(ИмяБД, ПерB+дер, Ссылка) (db_selector, bt_selector, ref):(i,i,o) key_next(ИмяБД, ПерB+дер, Ссылка) (db_selector, bt_selector, ref):(i,i,o)

key_prev(ИмяБД, ПерB+дер, Ссылка) (db_selector, bt_selector, ref):(i,i,o)

Эти предикаты, имеющие одинаковый формат, перемещают внутренний указатель B+дерева соответственно на первый, последний, следующий и предыдущий ключи в B+дереве и возвращают номер ссылки БД, заданный для соответствующего ключа. **Пример**:

```
/* Создание БД типа группа(Фамилия, Вес). Создание B+дерева по полю «Вес». Вывод фамилий членов группы в порядке убывания веса */ domains
        db_selector = mydba              % объявление имени внешней БД
dbdom = gruppa(string,string)    % объявление домена терма predicates
        insert_term(bt_selector)         % занесение термов в БД
    show_up(bt_selector)             % просмотр B+дерева в обратном порядке clauses
        insert_term(Bt_sel1):-
            write("Фамилия:"),readln(Fam),Fam<>"aaa",
write("Вес   :"),readln(Ves),
            chain_inserta(mydba,chain,dbdom,gruppa(Fam,Ves),Ref),
key_insert(mydba,Bt_sel1,Ves,Ref),       insert_term(Bt_sel1).
        insert_term(_):-!.
show_up(Bt_sel):-
            key_current(mydba,Bt_sel,_,Ref),
ref_term(mydba,dbdom,Ref,gruppa(Fam,Ves)),
write(Ves," ",Fam),nl,fail.      show_up(Bt_sel):-
key_prev(mydba,Bt_sel,_),        show_up(Bt_sel).
show_up(_). goal     db_create(mydba,"gruppa.bin",in_file),
bt_create(mydba,"ves",Bt_sel1,3,3),     insert_term(Bt_sel1),
        key_last(mydba,Bt_sel1,_),
show_up(Bt_sel1),     db_close(mydba).
```

Замечания к программе:

а) Если упорядочивание производится по нескольким полям, то для каждого из этих полей создается свое двоичное дерево.

б) При вставке нового ключа в базу данных для одного из полей тип данных этого поля при ссылке на него в предикате key_insert должен быть string или приведен к string с помощью предикатов преобразования типов.

в) Если вывод по ключу производится в прямом порядке, то строится соответствующее правило (в goal используется предикат key_first вместо key_last, а в правиле show_up вместо key_prev используется key-next).

Практическое задание

Создать внешнюю базу данных на диске следующего формата: student(Номер_зачетки, Фамилия, Средний_балл).

В этой БД по каждому из полей сформировать B+дерево.

Требуется:

а) вывести фамилии студентов в порядке возрастания номеров зачетной книжки;

б) вывести фамилии студентов в алфавитном порядке;

в) вывести фамилии студентов в порядке убывания среднего балла.

В меню предусмотреть операции создания внешней БД, извлечения из БД нужной информации в соответствии с заданием, дополнения БД новыми термами, удаления термов из БД, просмотра содержимого отдельных B+деревьев.

Дополнительные задачи к работе:

1. Выбрать из базы данных фамилии (вывод на экран по одной в строке), в которых есть хотя бы одна повторяющаяся буква.
2. Выбрать из базы данных фамилии (вывод на экран по одной в строке), в которых первая и последняя буквы одинаковые.
3. Выбрать из базы данных фамилии (вывод каждой в отдельной строке), длина которых меньше длины номера зачетки.
4. Выбрать из базы данных фамилии (вывод на экран по одной в строке), длина которых меньше заданной.
5. Выбрать из базы данных фамилии (вывод на экран по одной в строке), заканчивающиеся на заданную букву.
6. Выбрать из базы данных фамилии (вывод на экран по одной в строке), начинающиеся с заданной буквы.
7. Выбрать из базы данных номера зачеток (вывод на экран по одному в строке), сумма цифровых символов которых меньше заданного числа.
8. Выбрать из базы данных фамилии (вывод на экран по одной в строке), в которых есть заданная буква.

Содержание отчёта

Отчет должен содержать: название работы, постановку задачи и сведения о последовательности её выполнения; результаты тестирования программы, выводы

Контрольные вопросы.

1. Что такое B+деревья, где и для чего они используются ?
2. Отличие B+дерева от двоичного (бинарного) дерева.
3. Механизм упорядочивания данных в двоичном дереве.
4. Что представляет собой ключ B+дерева ?
5. Страница, длина и порядок B+дерева.
6. Как в программе создается B+дерево ?
7. Какие стандартные предикаты используются для открытия, закрытия и удаления B+дерева ?
8. Как можно вставить новый ключ в B+дерево ?
9. Как можно удалить ключ из B+дерева ?
10. Какие стандартные предикаты используются для установки указателя ключа B+дерева на первый, последний, текущий ключ ?
11. Какие стандартные предикаты используются для перемещения указателя на ключ B+дерева вперед и назад ?
12. Какой тип данных должны иметь ключи B+дерева ?
13. Как можно определить имена B+деревьев внешней базы данных (см. лабораторную работу № 12) ?.
14. Как можно получить статистическую информацию о B+дереве ?
15. В каком порядке разместятся в B+дереве следующие значения ключа: 1, 2, 5, 12,43?

Лабораторная работа № 4

Основные понятия языка Лисп. Система программирования mulisp.

Цель работы: приобретение практических навыков работы в системе программирования muLisp.

Теоретический материал

1.1. **Запуск интерпретатора muLisp**.

Запуск интерпретатора осуществляется путем инициализации файла **mulisp.com** (в нашем случае через bat-файл **lisp.bat**).

После выдачи начальных сообщений система высвечивает приглашение (знак доллара $), которое означает, что muLisp готов к вводу с клавиатуры.

Далее пользователь набирает некоторое выражение и нажимает клавишу Enter. Интерпретатор оценивает это выражение и печатает результирующее значение в начале новой строки, затем снова высвечивается приглашение $ и т.д. Этот цикл взаимодействия интерпретатора с пользователем повторяется до тех пор, пока не будет введена системная команда (SYSTEM), которая завершает работу и возвращает управление операционной системе.

1.2. **Механизм работы Лисп-системы**.

Так как Лисп является функциональным языком, то любая программа на Лиспе – тоже функция. Программа на Лиспе записывается в виде s-выражения (символьного выражения) как обращение к функции.

В общем случае обращение к функции имеет следующий вид:

(F Arg1 Arg2 … ArgN).

Здесь F – первый элемент списка, интерпретируется как имя функции, которую необходимо выполнить, взяв в качестве аргументов оставшиеся элементы списка Arg1, Arg2, …, ArgN.

Механизм работы Лисп-системы состоит из трех последовательных шагов (read-evalprint):

- считывание s-выражений (READ); -

интерпретация s-выражений (EVAL); -

печать s-выражения (PRINT).

Интерпретация s-выражений – это единственная и главная задача Лиспинтерпретатора. Ее выполняет функция Лисп-системы EVAL, которая берет одно sвыражение, интерпретирует его, если это возможно, и возвращает другое s-выражение как результат интерпретации первого s-выражения.

2.3. **Прерывания при работе интерпретатора muLisp**.

При работе интерпретатора могут возникнуть внутренние и внешние прерывания (остановы). В этом случае на экране дисплея высвечивается подсказка в виде опций: Continue, Break, Abort, Top-level, Restart, System?

Затем система ждет, пока пользователь выберет одну из опций по ее имени (C, B, A, T, R или S соответственно). Действия опций:

Continue (продолжить): возвращает управление программе, которая вызвала прерывание.

Break (останов): временно приостанавливает выполнение программы и выходит на следующий нижний уровень цикла read-eval-print (чтение – вычисление – печать). Выход из останова – команда (RETURN).

Abort (прерывание): прерывает выполнение программы, присваивает формальным параметрам, размещенным в стеке переменных, первоначальные значения и возвращает управление на текущий уровень цикла read-eval-print.

Top-level (верхний уровень): прерывает выполнение программы, присваивает первоначальные значения формальным параметрам, которые размещены в стеке переменных, выводит текущие входные и выходные данные и возвращает управление верхнему уровню цикла read-eval-print.

Restart (повторный старт): перезагрузка системы.

System (система): выход из системы Лисп.

В любое время в ходе выполнения программы пользователь может прервать ее работу, нажав клавишу Esc на клавиатуре.

2.4. **Редактор muLisp**.

2.4.1. **Загрузка редактора и опции главного меню**.

Загрузка редактора в существующую среду muLisp выполняется командой:
(RDS Edit), а затем (RETURN).

Как только редактор начинает работать, он чистит экран, рисует рамку и выдает головное меню: Edit, Print, Screen, Lisp, Quit:.

Затем система ждет, пока пользователь не наберет одну из опций путем ввода первой буквы ее имени (E, P, S, L или Q).

- Edit: выдает на экран подсказку Edit file: и ждет, пока не будет введено имя файла для редактирования. После ввода имени файла (напр., D:\107215\aaa) нажимается Enter. По умолчанию используется расширение .lsp и текущий каталог.
- Print: выдает на экран подсказку Print file: и ждет, пока будет введено имя файла для печати.
- Screen: выдает на экран подсказку Full, Vertical, Horizontal: и ждет выбора одной из опций (F, V или H). Опция Full screen использует для редактирования полный экран. Опция Vertical split screen использует правую половину экрана для редактирования, а левую – для отладки. Опция Horizontal split screen использует верхнюю половину экрана для редактирования, а нижнюю – для отладки.
- Lisp: завершает работу редактора; на экране появляется знак доллара. Из среды muLisp можно вернуться в редактор по команде (RETURN).
- Quit: завершает работу редактора и системы muLisp.

2.4.2. **Команды управления курсором**.

Основные команды перемещения курсора:
Ctrl-Q, D – в правый конец текущей строки,
Ctrl-Q, S – в левый конец текущей строки,
Ctrl-Q, X – в нижний конец экрана,
Ctrl-Q, E – в верхний конец экрана,
Ctrl-Q, I – влево в предыдущую позицию табуляции.

2.4.3. **Команды скроллинга экрана**.

Ctrl-Z – сдвигает текст вверх на одну строку,
Ctrl-W – сдвигает текст вниз на одну строку,
Ctrl-R – сдвигает текст на высоту экрана вниз,
Ctrl-C - сдвигает текст на высоту экрана вверх, Ctrl-Q, C – перемещает курсор и окно в конец файла, Ctrl-Q, R - перемещает курсор и окно в начало файла.

2.4.4. **Изменение режима ввода текста**.

Для переключения из режима вставки (Insert) в режим замещения (Replace) и наоборот используется команда Ctrl-V.

2.4.5. **Команды удаления и восстановления текста**.

Ctrl-Y – удаляет строку, на которой распологается курсор,
Ctrl-Q, Y – удаляет правый конец строки, начиная с курсора,
Ctrl-U – восстанавливает текст, уничтоженный последней командой удаления.

2.4.6. **Команды блочных операций**.

Ctrl-O, B – отмечает начало блока,

Ctrl-O, E – отмечает конец блока,
Ctrl-O, C – копирует отмеченный блок в место, указанное курсором,
Ctrl-O, M – перемещает отмеченный блок в положение курсора,
Ctrl-O, R – читает текст из указанного файла и вставляет в место, указанное курсором,
Ctrl-O, W – запись отмеченного блока в указанный файл.

2.4.7. **Команды списковых структур.**

Команды списковых структур позволяют перемещать, копировать, уничтожать, считывать из файла и записывать в файл s-выражения. Двухсимвольные команды, начинающиеся с Esc, инициируют команды списковых структур.

Esc-D – перемещает курсор вперед на одно s-выражение,
Esc-S – перемещает курсор назад на одно s-выражение,
Esc-F – перемещает курсор вперед, устанавливая его после конца списка,
Esc-A - перемещает курсор назад, устанавливая его перед началом списка,
Esc-T – уничтожает s-выражение под курсором и справа от курсора,
Esc-Y – уничтожает определение, где располагается курсор,
Esc-L – закрывает редактор и передает управление в окно вычисления Lisp. Возврат в редактор – команда (RETURN).

Esc-! – выдает в окно вычисления Lisp результат вычисления s-выражения под курсором и справа от него. Данная команда дает возможность выборочно переопределять определения редактируемых функций; это плезно при их тестировании.

2.4.8. **Команды сохранения файлов** (начинаются с Ctrl-K).

Через несколько секунд после ввода Ctrl-K выдается подсказка, если пользователь не успел ввести второй символ команды.

Ctrl-K, D – сохраняет текст в виде файла и возвращает управление в головное меню редактора,
Ctrl-K, S – сохраняет текст в виде файла и заново открывает файл для редактирования.

Если в текст вносились изменения, Ctrl-K, A и Ctrl-K, Q высвечивают подсказку:

Abandon <filename>? (Y/N),

где <filename> - имя отредактированного файла. При вводе Y отредактированный текст удаляется, а управление возвращается в головное меню редактора. В противном случае никаких действий не происходит. Если изменения не вносились, текст просто удаляется.

2.4.9. **Команды окна.**

Ctrl-Q, W – позволяет изменить размер окна и его форму на время редактирования файла. Команда выдает подсказку

Full, Vertical, Horizontal и ждет выбора одной из опций (F, V или H) (см. пункт 2.4.1).

2.5. **Трассировка программы**.

Чтобы выполнить трассировку программы, нужно загрузить отладочный пакет muLisp'a с помощью команды

(RDS 'DEBUG)

Трассировка функций осуществляется с помощью функции TRACE. Она вызывается с одной или более функциями для трассировки в качестве своих аргументов. Например, команда

(TRACE 'APP 'REV) трассирует функции APP и REV.

Если после этого вызвать функцию APP (объединение двух списков) следующим образом

(APP '(A B) '(C D E)), то будет выведена трасса для этой функции.

Если вывод трассы осуществляется слишком быстро, то для временной остановки вывода можно набрать на клавиатуре Ctrl-S.

Трасса также может быть выведена командой (HISTORY).

Для запрещения трассировки используется команда CLEAR. Например, команда (CLEAR ‘APP ‘REV) запрещает трассировку функций APP и REV.

2.6. **Примитивы Лиспа**:

- функция (**car L**) – возвращает голову списка L, напр., (car ‘((a b) c d)) возвращает (A B);
- функция (**cdr L**) – возвращает хвост списка L, напр., (cdr ‘((a b) c d)) возвращает (C D);
- функция (**cons S1 S2**) – включает элемент S1 в начало списка S2, напр., (cons ‘a ‘(b c)) возвращает (A B C);
- предикат (**atom S**) – проверяет, является ли аргумент S атомом, напр., (atom ‘()) возвратит T, а (atom ‘(nil)) возвратит NIL;
- предикат (**eq S1 S2**) – проверяет тождественность двух символов S1 и S2, напр., (eq ‘a (car ‘(a b c))) возвратит T, а (eq ‘a ‘(a)) возвратит NIL;
- функция (**quote S**) – блокирует вычисление выражения S, значением функции является символьное выражение S, представленное в вызове, напр., (quote (+ 2 3)) возвращает (+ 2 3). Синонимом для (quote (+ 2 3)) является запись ‘(+ 2 3).

Практическое задание

Загрузить muLisp.

3.2.Ознакомиться с возможностями редактора muLisp.

3.3.Проделать следующие вычисления с помощью интерпретатора Лиспа:

a) 3.234*(4.56+21.45)
b) 5.67+6.342+12.97
c) (454-214-657)/(456+678*3)

3.4.Написать выражение, вычисляющее среднее арифметическое чисел 23, 5, 43 и 17.

3.5.Определить значение следующих выражений:

a) ‘(+ 2 (* 3 4))
b) (+ 2 ‘(* 3 4))
c) (+ 2 (‘* 3 4))
d) (+ 2 (* 3 ‘4))
e) (quote ‘quote)
f) (quote 2)
g) ‘(quote nil)

3.6.Записать последовательность вызовов CAR и CDR, выделяющие из приведенных ниже списков символ «цель». Упростить эти вызовы с помощью функций C…R.

a) (1 2 цель 3 4)
b) ((1) (2 цель) (3 (4)))
c) ((1 (2 (3 4 цель))))

3.7.Вычислить значения следующих выражений. Проверить результат на компьютере.

a) (cons nil ‘(суть пустой список))
b) (cons nil nil)
c) (cons ‘(nil) ‘(nil))
d) (cons (car ‘(a b)) (cdr ‘(a b)))
e) (car ‘(car (a b c)))
f) (cdr (car (cdr ‘(a b c))))

3.8.Проверить, какие из следующих вызовов возвращают значение T ?

a) (atom ‘(cdr nil))

b) (atom (* 2 (+ 2 3)))
c) (atom (atom (+ 2 3)))
d) (eq nil ( ))
e) (eq t (atom (car '(мышь))))

3.9.Выполнить трассировку функции APP, объединяющей два списка. Определение функции:

```
( defun app ( lst1 lst2 )
  (( null lst1 ) lst2 )
  ( cons ( car lst1 )
  ( app ( cdr lst1 ) lst2 )))
```

4.**Содержание отчета**: цель работы, постановка задачи, диалог пользователя с интерпретатором muLisp, выводы. (Для формирования отчета использовать редактор muLisp).

Содержание отчёта

Отчет должен содержать:
- название работы, постановку задачи и сведения о последовательности её выполнения;
- ответы на контрольные вопросы ;
- основные этапы работы и результаты их выполнения.

Контрольные вопросы

1. Почему Lisp называют языком функционального программирования ?
2. Чем отличается запись списка на языке Lisp от аналогичной записи на языке Пролог ?
3. Инфиксная и префиксная формы записи выражений; какая из этих форм принята в Lisp ?
4. Пять базовых функций (примитивов) Lisp.
5. Для чего используется функция QUOTE, чем ее можно заменить ?
6. Что означает символ $ в начале строки при работе с интерпретатором muLisp ?
7. Как производится выход из системы muLisp ?
8. Как будет интерпретироваться системой muLisp s-выражение (a b c d e) ?
9. Как загружается редактор в среду muLisp ?
10. Как можно запустить на выполнение функцию, набранную в редакторе muLisp?
11. По какой команде можно перейти из редактора в окно вычислений muLisp ?
12. По какой команде можно вернуться из окна вычислений в редактор ?
13. Как можно сохранить отредактированный в редакторе текст в виде файла ?
14. Как производится трассировка программы в среде muLisp ?

Лабораторная работа № 5

О п р е д е л е н и е п о л ь з о в а т е л ь с к о й ф у н к ц и и

Цель работы: приобретение практических навыков определения функций на языке Lisp.

Теоретический материал

Данные в Лисп представляются **атомами, списками, консами, символьными выражениями**. Взаимосвязь понятий иллюстрируется рис. 4.

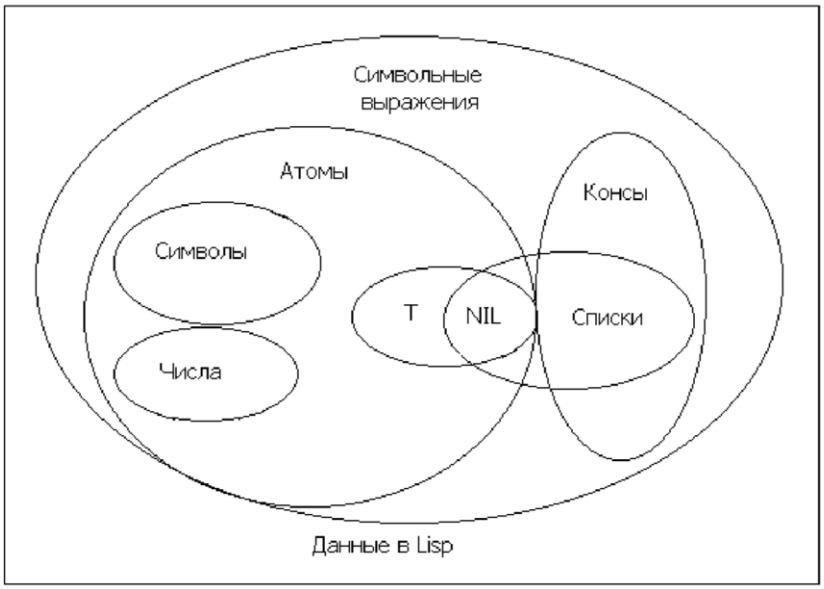


Рис. 4. Данные в Lisp

Атомы, простейшие объекты в Lisp, делятся на символьные и числовые. **Символьный атом** – это последовательность букв, цифр и специальных символов, например, X7, Month8, A-1. **Числовой атом** – это последовательность цифр, включающая возможно символы ‘+’, ‘-‘, ‘.’, ‘/’, например, 12, -8, 3.14, 2/7. Числовые атомы интерпретируются как константы, символьные – как константы и как переменные. Атом T означает логическое значение “истина”, атом NIL – логическое значение “ложь” или пустой список.

Списки – это последовательность элементов, разделенных пробелами и заключенных в круглые скобки. Элементами списка могут быть любые объекты Лиспа: атомы, списки, консы, например, (1 a 2 b 3 c), ((x1 0) (x2 1)), ((y.blue) (z.yellow)). Пустой список не содержит элементов и обозначается ( ) или NIL. **Консы** – это пара элементов, разделенных точкой и заключенных в круглые скобки.

Список (e1 e2 … eN) представляется суперпозицией консов (e1. (e2. (… (eN.NIL) …))). **Символьным выражением** в Лисп является один из следующих объектов:

1) атом,
2) список (s1 … sn), 3) конс (s1.s2), где s1, s2, …, sn – символьные выражения.

Например, (2 (T.NIL).(y z)) является символьным выражением.

2.2. **Префиксная форма записи выражений**.

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

Например, в математике вызов функции записывается следующим образом: f(x), g(x,y), h(x, g(y,z)) и т.д.

На Лиспе эти же записи выглядят следующим образом:

(f x), (g x y), (h x (g y z)) и т.д.

Соответственно арифметические выражения x+y, x*(y+z), x*x-y*y на Лиспе будут записаны в виде: (+ x y), (* x (+ y z)), (- (* x x) (* y y)).

2.3. *Примитивы Лиспа, используемые при определении функций:*

- ***функции*** **car, cdr, cons, quote**, ***предикаты*** **atom, eq** (см. лаб. работу 16);

- ***математические функции***:

(+ x1 x2 … xN) – выполняет сложение всех аргументов;

(- x1 x2 … xN) – возвращает разность между первым аргументом и всеми остальными;

(* x1 x2 … xN) – выполняет произведение всех аргументов;

(/ x1 x2 … xN) – возвращает результат деления первого аргумента на все остальные;

(**min x1 x2 … xN**) – возвращает наименьшее xi;

(**max x1 x2 … xN**) – возвращает наибольшее xi;

- ***функции сравнения чисел***:

(= x1 x2 … xN) – числа равны;

(/= x1 x2 … xN) – числа не равны;

(< x1 x2 … xN) – числа возрастают:

(> x1 x2 … xN) – числа убывают;

(<= x1 x2 … xN) – числа возрастают или равны;

(>= x1 x2 … xN) – числа убывают или равны; -

логические функции:

(**and S1,S2,…,Sn**) – логическое И;

(**or S1,S2,…,Sn**) – логическое ИЛИ;

(**not S**) – логическое НЕ;

- ***функции проверки знака числа***:

(plusp число) – число положительное;

(minusp число) – число отрицательное;

(zerop число) – нуль;

- ***функции назначения***:

(setq x Sv) – переменной x приписывается значение выражения Sv;

(set Sv1 Sv2) – присваивает значение выражения Sv2 значению выражения Sv2;

- предикат (**null Sv**) – проверяет, является ли значением аргумента Sv пустой список, напр., (null (cdr ‘(a))) возвращает T;

- предикат **(equal S1 S2)** – проверяет идентичность записей S1 и S2, позволяет сравнивать однотипные числа, а также проверяет одинаковость двух списков, напр., (equal 5 (+ 2 3)) возвращает T; (equal ‘(a b c) (cons ‘a ‘(b c))) возвращает T;

- функция **(list x1 x2 … xN)** – возвращает в качестве своего значения список из значений аргументов, напр., (list ‘a ‘b (+ 2 3)) возвращает (A B 5).

2.4. **Управляющие структуры**.

В Лиспе имеются управляющие структуры для организации последовательных вычислений, разветвлений и циклов. Управляющие структуры (будем называть их предложениями) выглядят внешне как вызовы функций. Предложение записывается в виде скобочного выражения, первый

элемент которого действует как имя управляющей структуры, а остальные элементы – как аргументы. В качестве аргументов используются формы. Под **формой** понимается такое символьное выражение, значение которого может быть найдено интерпретатором.

Для **последовательных** вычислений используются предложения **prog1** и **progn**.

(prog1 форма1 форма2 … формаN)

(progn форма1 форма2 … формаN)

У этих предложений переменное число аргументов, которые они последовательно вычисляют и возвращают в качестве значения значение первого (prog1) или последнего (progn) аргумента.

Prog1 вычисляет форму1, затем оставшиеся формы и возвращает результат вычисления формы1. Prog1 часто используется для того, чтобы не вводить временные переменные для хранения результатов в процессе вычисления других выражений.

```
(setq FOO '(a b c d))                          -------→ (A B C D)
(prog1 (car FOO) (setq FOO (cdr FOO)))         --------→  A
FOO        ------→ (B C D)
```

Progn последовательно вычисляет формы, начиная с формы1 и выдает значение последней формы.

```
(progn (setq num1 (+ 2 5))
       (setq num2 (* 3 4))
       ((< num1 num2)
          ((minusp num1)
             (* 3 num2))
          (+ num1 num2))
       (- num1 num2))              ----------→ 19
```

Основным средством **разветвления** вычислений является предложение **cond.**

(cond (p1 a1) (p2 a2) … (pN aN))

Здесь p1, p2, …, pN – предикаты, a1, a2, …, aN – произвольные выражения.

Cond последовательно вычисляет предикаты pi до появления значения, отличного от nil, или до исчерпания всех выражений в cond-аргументах. В первом случае значением cond будет значение выражения ai, соответствующее первому, отличному от nil, предикату pi. Если значения всех предикатов есть nil, то значением cond будет nil.

Пример. Сколько элементов в списке ? (Два варианта – 1 или больше1).

```
(cond ((cdr L) 'больше 1)
      ( t   1))
```

В простом случае для разветвлений может использоваться предложение **if**:

(if условие то-форма иначе-форма)

Пример. (if (atom t) 'атом 'список) --------→ атом

Для программирования циклических вычислений обычно используется рекурсия. 2.5.

Определение функций.

Определение функций в среде Лисп-системы осуществляется с помощью примитива defun.

(defun name (arg1 arg2 … argN)

(Ev1) (Ev2) … (EvN)),

где name – имя определяемой функции,

arg1, arg2, …, argN – список формальных параметров в виде переменных,

Ev1, Ev2, …, EvN – s-выражения, связывающие переменные определяемой функции и вычисляющие ее значение.

Пример 1. Определим функцию, выделяющую второй элемент списка:

```
(defun Второй (Lst)
(car (cdr Lst)))
```

Пример 2. Определим логическую функцию И:

```
(defun И (x y)
  (cond (x y)
        (t nil)))
```

Пример 3. Определим рекурсивную функцию mem, проверяющую, является ли атом x элементом некоторого списка y.

```
(defun mem (x y)
  (cond ((null y) nil)
        ((eq x (car y)) t)
        ( t (mem x (cdr y)))))
```

Пример 4. Определим функцию ! вычисления факториала:

```
(defun ! (N)
  (cond ((zerop N) 1)
        (t (* N (! (- N 1))))))
```

Практическое задание

Определить функцию, вычисляющую x + y – x*y.

3.2. Определить функции (null x), (caddr x) и (list x1 x2 x3) с помощью базовых функций (использовать имена null1, caddr1, list1, чтобы не переопределять одноименные встроенные функции Лисп).

3.3. Определить логическую функцию ИЛИ (OR) (имя OR не использовать).

3.4. Определить логическую функцию НЕ (NOT) (имя NOT не использовать).

3.5. Определить N-ое число Фибоначчи.

3.6. Подсчитать количество элементов списка.

3.7. Все элементы списка увеличить на заданное число.

3.8. Выделить последний элемент списка.

3. 9. Добавить заданный элемент в конец списка.

3.10. Удалить из списка последний элемент.

3.11. Удалить из списка первое вхождение заданного элемента на верхнем уровне. 3.12. Удалить из списка все вхождения заданного элемента на верхнем уровне

3.13. Удалить из списка каждый второй элемент.

3.14. Определить среднее арифметическое элементов списка.

3.15. Подсчитать количество отрицательных элементов списка.

3.16. Определить минимальный элемент целочисленного списка (не используя встроенную функцию min).

3.17. Заменить в списке все элементы со значением X на значение Y.

3.18. Удалить начало списка до заданного элемента X (включительно).

3.19. Получить новый отсортированный список путем вставки заданного элемента в исходный отсортированный в порядке возрастания элементов список.

3.20. Определить функцию, разбивающую список (a b c d …) на пары ((a b) (c d) …).

3.21. Определить функцию, которая, чередуя элементы списков (a b …) и (1 2 …), образует новый список (a 1 b 2 …).

3.22. Определить функцию (первый-совпадающий x y), которая возвращает первый элемент, входящий в оба списка x и y, в противном случае nil.

3.23. Проверить, является ли список множеством, т.е. входит ли каждый элемент в список лишь один раз.

3.24. Преобразовать список в множество, т.е. удалить из списка повторяющиеся элементы.

3.25. Определить предикат Равенство-множеств, проверяющий совпадение двух множеств (независимо от порядка следования элементов).

3.26. Определить предикат Подмножество, проверяющий, является ли одно множество подмножеством другого.

3.27. Определить функцию Пересечение, определяющую общие для дух множеств элементы.

3.28. Определить предикат Непересекающиеся, проверяющий, что два множества не имеют общих элементов.

3.29. Определить функцию Объединение, формирующую объединение двух множеств.

3.30. Определить функцию Разность, формирующую разность двух множеств, т.е. удаляющую из первого множества все общие со вторым множеством элементы.

4. **Содержание отчета**: цель работы, постановка задачи, тексты определяемых функций, результаты тестирования функций, выводы.

Замечания:

- при определении функций использовать только следующие примитивы Lisp: car, cdr, cons, atom, eq, quote, plusp, minusp, zerop, set, setq, equal, null, list, математические функции, функции сравнения чисел и логические функции and , or, not;

- в заданиях 3.22 – 3.30 использовать вспомогательную функцию, проверящую принадлежность заданного элемента списку;

- для формирования отчета использовать редактор muLisp.

5.**Содержание отчета**: цель работы, постановка задачи, диалог пользователя с интерпретатором muLisp, выводы. (Для формирования отчета использовать редактор muLisp).

Содержание отчёта

Отчет должен содержать:

- название работы, постановку задачи и сведения о последовательности её выполнения;
- ответы на контрольные вопросы ;
- основные этапы работы и результаты их выполнения.

Контрольные вопросы

1. Какие объекты относятся к символьным выражениям Lisp ?
2. Какие объекты относятся к атомам Lisp ?
3. Что означает атом NIL ?
4. Что такое конс ?
5. Префиксная форма записи выражений в Lisp.
6. Какие функции в Lisp называются предикатами ?

7. Управляющие структуры Lisp для последовательных вычислений.
8. Управляющие структуры Lisp для разветвления вычислений.
9. Основной механизм программирования циклических вычислений.
10. Определение функций в Lisp.
11. Можно ли в качестве имени определяемой функции взять имя стандартной функции Lisp.
12. Принципиальное отличие восходящей рекурсии от нисходящей рекурсии.
13. Вспомогательные (промежуточные) функции.
14. Суть двумерной рекурсии.

Лабораторная работа №4.

Использование управляющих структур. Использование рекурсивного подхода Функционалы и макросы.

Цель: Изучить основы программирования с применением рекурсии. Научиться работать с функционалами и макросами.

Теоретический материал

Управляющие структуры

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

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

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

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

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

Предложение COND является основным средством организации разветвления вычислений.

Структура условного предложения :

(COND (< проверка-1 > < действие-1 >) (< проверка-2 > < действие-2 >)

.....

(< проверка-n > < действие-n >))

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

Значение COND определяется следующим образом:

1. Выражения < проверка-i >, выполняющие роль предикатов вычисляются последовательно, слева направо, до тех пор, пока не встретится выражение, значением которого не является NIL.
2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения COND.
3. Если истинного значения нет, то значением COND будет NIL.

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

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

Пример: (функция проверяет тип аргумента)

```
( defun classify ( arg )
( cond
( ( null arg ) nil )
( ( list arg ) 'list )
( ( numberp arg ) 'number )
```

(t 'atom))

(classify 'a)
atom
(classify 5)
number
Еще один пример:
(defun double1 (num)
(cond
((numberp num) (* num 2) (
t ' не-число))
Эта функция гарантировано удваивает число, отбрасывая не числовые аргументы.
В COND могут отсутствовать результирующие выражения для предикатов, а так же присутствовать несколько действий.
(COND
(< проверка-1 >)
(< проверка-2 > < действие-2 >)
(< проверка-3 > < дейст.-31 > < дейст.-32 > < дейст.-33 >))
Если нет действия - результат значение предиката. Если не одно действие - результат значение последнего аргумента.
COND наиболее общее условное предложение. Часто пользуются более простыми условными предложениями.
(IF < условие > < то форма > < иначе форма >)
Пример:
(if (atom x) 'atom 'not - atom)
Условные предложения WHEN и UNLESS являются частными случаями условного предложения IF:
Если условие соблюдается, то выполняются формы.
(WHEN < условие >
< форма-1 > < форма-2 > < форма-3 > ...)
Если условие не соблюдается, то выполняются формы.
(UNLESS < условие >
< форма-1 > < форма-2 > < форма-3 > ...)
Любую логическую функцию можно заменить COND-выражением и наоборот.
Пример:
car-функция с проверкой:
(defun gcar (l)
(cond
((listp l) (car l))
(t nil)))
то же через логические функции:
(defun gcar1 (l)
(and
(listp l) (car l)))

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

4.4 Ввод и вывод информации

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

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

Теперь мы рассмотрим специальные функции ввода и вывода Лиспа.

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

(READ)

функция без аргументов.

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

(READ)

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

(setq x ' (read))

(+ 1 2) - вводимое выражение

(+ 1 2) - значение

x

(+ 1 2)

(eval x)

3

Еще один пример:

(defun tr (arg)

(list (+ arg (read)) (read)))

(tr 8)

1

4 cat

(22 cat)

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

Для вывода выражений можно использовать функцию print.

(PRINT < arg >)

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

* (print (+ 2 3))

5 - вывод 5 - значение print и read - псевдофункции, у которых кроме значения есть побочный эффект.

Значение функции - значение ее аргумента.

Побочный эффект - печать этого значения.

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

Пример:

(setq row ' (x x x))

(x x x)

(print (cdr row))

(x x) - печать

(x x) - значение

(cons 'o (print (cdr row)))

(x x) - печать

(o x x) - значение

PROGN, PROG1, PROG2

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

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

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

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

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

В Лиспе часто используется так называемый неявный PROGN, т.е вычисляется последовательность форм, а в качестве значения берется значение последней формы. При определении функций может использоваться неявный PROGN .

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

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

Пример:

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

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

( print-sum )
        ( type two number )
```

Основная идея рекурсивного определения заключается в том, что функцию можно с помощью рекуррентных формул свести к некоторым начальным значениям, к ранее определенным функциям или к самой определяемой функции, но с более «простыми» аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям.

Рекурсивная процедура, во-первых содержит всегда по крайней мере одну терминальную ветвь и условие окончания. Во-вторых, когда процедура доходит до рекурсивной ветви, то функционирующий процесс приостанавливается, и новый такой же процесс запускается сначала, но уже на новом уровне. Прерванный процесс каким-нибудь образом запоминается. Он будет ждать и начнет исполняться лишь после окончания нового процесса. В свою очередь, новый процесс может приостановиться, ожидать и т. д.

Будем говорить о рекурсии по значению и рекурсии по аргументам. В первом случае вызов является выражением, определяющим результат функции. Во втором - в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции. Аргументом рекурсивного вызова может быть вновь рекурсивный вызов и таких вызовов может быть много.

Рассмотрим следующие формы рекурсии:
простая рекурсия;
параллельная рекурсия;
взаимная рекурсия.

Рекурсия называется простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.
Для примера напишем функцию вычисления чисел Фибоначчи (F(1)=1; F(2)=1; F(n)=F(n-1)+F(n-2) при n>2):

```
(DEFUN FIB (N)
(IF (> N 0)
(IF (OR N=1 N=2) 1
(+ (FIB (- N 1)) (FIB (- N 2))))
‘ОШИБКА_ВВОДА))
```

Рекурсию называют параллельной, если она встречается одновременно в нескольких аргументах функции:

```
(DEFUN f ...
...(g ... (f ...) (f ...) ...)
...)
```

Рассмотрим использование параллельной рекурсии на примере преобразования списочной структуры в одноуровневый список:

```
(DEFUN PREOBR (L)
(COND
((NULL L) NIL)
((ATOM L) (CONS (CAR L) NIL))
(T (APPEND
(PREOBR (CAR L))
(PREOBR (CDR L))))))
```

Рекурсия является взаимной между двумя и более функциями, если они вызывают друг друга:

```
(DEFUN f ...
...(g ...) ...)
(DEFUN g ...
...(f ...) ...)
```

Для примера напишем функцию обращения или зеркального отражения в виде двух взаимно рекурсивных функций следующим образом:

```
(DEFUN obr (l)
(COND ((ATOM l) l)
(T (per l nil))))
(DEFUN per (l res)
(COND ((NULL l) res)
(T (per (CDR l)
(CONS (obr (CAR l)) res)))))
```

2. Применяющие функционалы.
Функции, которые позволяют вызывать другие функции, т. е. применять функциональный аргумент к его параметрам называют применяющими функционалами. Они дают возможность интерпретировать и преобразовывать данные в программу и применять ее в вычислениях.
APPLY
APPLY является функцией двух аргументов, из которых первый аргумент представляет собой функцию, которая применяется к элементам списка, составляющим второй аргумент функции APPLY:

(APPLY fn список)

_(SETQ a ‘+) p +
_(APPLY a ‘(1 2 3)) p 6
_(APPLY ‘+ ‘(4 5 6)) p 15

FUNCALL.
Функционал FUNCALL по своему действию аналогичен APPLY, но аргументы для вызываемой он принимает не списком, а по отдельности:
(FUNCALL fn x1 x2 ... xn)

_(FUNCALL ‘+ 4 5 6) p 15

FUNCALL и APPLY позволяют задавать вычисления (функцию) произвольной формой, например, как в вызове функции, или символом, значением которого является функциональный объект. Таким образом появляется возможность использовать синонимы имени функции. С другой стороны, имя функции можно использовать как обыкновенную переменную, например для хранения другой функции (имени или лямбда-выражения), и эти два смысла (значение и определение) не будут мешать друг другу:

_(SETQ list ‘+) p +
_(FUNCALL list 1 2) p 3
_(LIST 1 2) p (1 2)

3. Отображающие функционалы.
Отображающие или MAP-функционалы являются функциями, которые являются функциями, которые некоторым образом отображают список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью. Каждая из них имеет более двух аргументов, значением первого должно быть имя определенной ранее или базовой функции, или лямбда-выражение, вызываемое MAP-функцией итерационно, а остальные аргументы служат для задания аргументов на каждой итерации. Естественно, что количество аргументов в обращении к MAP-функции должно быть согласовано с предусмотренным количеством аргументов у аргумента-функции. Различие между всеми MAP-функциями состоит в правилах формирования возвращаемого значения и механизме выбора аргументов итерирующей функции на каждом шаге.
Рассмотрим основные типы MAP-функций.
MAPCAR.
Значение этой функции вычисляется путем применения функции fn к последовательным элементам xi списка, являющегося вторым аргументом функции. Например в случае одного списка получается следующее выражение:
(MAPCAR fn ‘(x1 x2 ... xn))
В качестве значения функционала возвращается список, построенный из результатов вызовов функционального аргумента MAPCAR.

_(MAPCAR ‘LISTP ‘((f) h k (i u)) p (T NIL NIL T)
_(SETQ x ‘(a b c)) p (a b c)
_(MAPCAR ‘CONS x ‘(1 2 3)) p ((a . 1) (b . 2) (c . 3))

MAPLIST.

MAPLIST действует подобно MAPCAR, но действия осуществляет не над элементами списка, а над последовательными CDR этого списка.

_(MAPLIST ‘LIST ‘((f) h k (i u)) p (T T T T)
_(MAPLIST ‘CONS ‘(a b c) ‘(1 2 3)) p (((a b c) 1 2 3) ((b c) 2 3) ((c) 3))

Функционалы MAPCAR и MAPLIST используются для программирования циклов специального вида и в определении других функций, поскольку с их помощью можно сократить запись повторяющихся вычислений.
Функции MAPCAN и MAPCON являются аналогами функций MAPCAR и MAPLIST. Отличие состоит в том, что MAPCAN и MAPCON не строят, используя LIST, новый список из результатов, а объединяют списки, являющиеся результатами, в один список.

4. Макросы.
Программное формирование выражений наиболее естественно осуществляется с помощью макросов. Макросы дают возможность писать компактные, ориентированные на задачу программы, которые автоматически преобразуются в более сложный, но более близкий машине эффективный лисповский код. При наличии макросредств некоторые функции в языке могут быть определены в виде макрофункций. Такое определение фактически задает закон предварительного построения тела функции непосредственно перед фазой интерпретации.
Синтаксис определения макроса выглядит так же, как синтаксис используемой при определении функций формы DEFUN:
(DEFMACRO имя лямбда-список тело)
Вызов макроса совпадает по форме с вызовом функции, но его вычисление отличается от вычисления вызова функции. Первое отличие состоит в том, что в макросе не вычисляются аргументы. Тело макроса вычисляется с аргументами в том виде, как они записаны.
Второе отличие состоит в том, что интерпретация функций, определенных как макро, производится в два этапа. На первом, называемом макрорасширением, происходит формирование лямбда-определения функции в зависимости от текущего контекста, на втором осуществляется интерпретация созданного лямбда-выражения.

_(DEFMACRO setqq (x y)
(LIST ‘SETQ x (LIST ‘QUOTE y))) p setqq
_(setqq a (b c)) p (b c)
_a p (b c)

Макросы отличаются от функций и в отношении контекста вычислений. Во время расширения макроса доступны синтаксические связи из контекста определения. Вычисление же полученной в результате расширения формы производится вне контекста макровызова, и поэтому статические связи из макроса не действуют. Использование макрофункций облегчает построение языка с лиспоподобной структурой, имеющего свой синтаксис, более удобный для пользователя. Чрезмерное использование макросредств затрудняет чтение и понимание программ.

Практическое задание

1. Напишите рекурсивную функцию, определяющую сколько раз функция FIB вызывает саму себя. Очевидно, что FIB(1) и FIB(2) не вызывают функцию FIB.
2. Напишите функцию для вычисления полиномов Лежандра (P0(x)=1, P1(x)=x, Pn+1(x)= ((2*n+1)*x*Pn(x)-n*Pn-1(x))/(n+1) при n>1).
3. Напишите функцию:
вычисляющую число атомов на верхнем уровне списка (Для списка (a в ((a) c) e) оно равно трем.);
определяющую число подсписков на верхнем уровне списка;

вычисляющую полное число подсписков, входящих в данный список на любом уровне.
4. Напишите функцию:
от двух аргументов X и N, которая создает список из N раз повторенных элементов X;
удаляющую повторные вхождения элементов в список;
которая из данного списка строит список списков его элементов, например, (a b) p ((a) (b));
вычисляющую максимальный уровень вложения подсписков в списке;
единственным аргументом которой являлся бы список списков, объединяющую все эти списки в один;
зависящую от трех аргументов X, N и V, добавляющую X на N-е место в список V.
5. Напишите функцию:
аналогичную функции SUBST, но в которой третий аргумент W обязательно должен быть списком;
которая должна производить замены X на Y только на верхнем уровне W;
заменяющую Y на число, равное глубине вложения Y в W, например Y=A, W=((A B) A (C (A (A D)))) p ((2 B) 1 (C (3 (4 D))));
аналогичную функции SUBST, но производящую взаимную замену X на Y, т. е. X p Y, Y p X.
6. Вычислите значения следующих вызовов:
(APPLY ‘LIST ‘(a b));
(FUNCALL ‘LIST ‘(a b));
(FUNCALL ‘APPLY ‘LIST ‘(a b));
(FUNCALL ‘LIST ‘APPLY ‘(a b);
7. Определите функционал (A-APPLY f x), который применяет каждую функцию fi списка
f = (f1 f2 ... fn)
к соответствующему элементу xi списка
x = (x1 x2 ... xn)
и возвращает список, сформированный из результатов.
8. Определите функциональный предикат (КАЖДЫЙ пред список), который истинен в том и только в том случае, когда, являющийся функциональным аргументом предикат пред истинен для всех элементов списка список.
9. Определите функциональный предикат (НЕКОТОРЫЙ пред список), который истинен, когда предикат истинен хотя бы для одного элемента списка.
10. Определите FUNCALL через функционал APPLY.
11. Определите функционал (MAPLIST fn список) для одного списочного аргумента.
12. Определите макрос, который возвращает свой вызов.
13. Определите лисповскую форму (IF условие p q) в виде макроса.

Примеры написания функций.

```
;Subst - заменяет все вхождения Y в W на X.
(DEFUN subst (x y w)
(COND ((NULL w) NIL) ;проверка на окончание списка
((EQUAL ‘y ‘w) x)
((ATOM ‘w) w) ;
(t (CONS (subst x y (car w)) ;поиск в глубину
(subst x y (cdr w)))))) ;поиск в ширину


;COMPARE1 - сравнение с образцом
(defun compare1 (p d)
(cond ((and (null p) (null d)) t) ;исчерпались списки?
((or (null p) (null d)) nil) ;одинакова длина списков?
((or (equal1 (car p) '&) ;присутствует в образце атом &
(equal1 (car p) (car d))) ;или головы списков равны
(compare1 (cdr p) (cdr d))) ;& сопоставим с любым атомом
((equal1 (car p) '*) ;присутствует в образце атом *
(cond ((compare1 (cdr p) d)) ;* ни с чем не сопоставима
((compare1 (cdr p) (cdr d))) ;* сопоставима с одним атомом
```

((compare1 p (cdr d))))))) ;* сопоставима с несколь ;кими атомами

Вопросы.

1. Что такое рекурсия?
2. Назовите достоинства ее использования?
3. Что такое функционал?
4. Назовите особенности применяющих и отображающих функционалов?
5. Для чего они используются?
6. Что такое макрос?
7. Когда их используют?

Список литературы

СОДЕРЖАНИЕ

Лабораторная работа №1 . Системы счисления. Приемы работы с двоичной, восьмиричной, шестнадцатиричной системами……...3
Лабораторная работа №2 Операционная система Windows.……..8
Лабораторная работа № Текстовый редактор Microsoft Word...……13
Лабораторная работа № 4 Освоение технологии работы с электронными таблицами MS Excel…….18
Список литературы …….…….…29

Методические указания

Редактор
Подписано в печать
Формат 60х84 1/16 Бумага газетная.
Печать офсетная 2.0 п.л., уч.-изд.л.
Тираж 100 экз.
Заказ №
Редакционно-издательский отдел ДГТУ

ИПЦ ДГТУ
367015 Махачкала, пр.Шамиля, 70