Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»

М.Ю. Галкина

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Лекции

Оглавление

Введение. Классификация языков программирования

По одной из классификаций языки программирования можно разделить на два типа: процедурные (операторные) и непроцедурные. К непроцедурным языкам относят объектно-ориентированные языки и декларативные. Декларативные языки, в свою очередь, делятся на функциональные и логические. На практике языки программирования часто обычно содержат в себе черты различных типов. На процедурном языке часто можно написать функциональную или объектно-ориентированную программу или ее часть и наоборот. Поэтому, точнее было бы вместо типа языка говорить о стиле или методе программирования. Естественно, что различные языки поддерживают разные стили в разной степени.

Программа на процедурном языке состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. Типичными операторами являются операторы присваивания, циклов, ввода-вывода, передачи управления. Из операторов можно составлять подпрограммы. Основа процедурного программирования: взятие значения какой-либо переменной, выполнение с ним какого-нибудь действия, сохранение нового значения и так до тех пор, пока не будет получено желаемое окончательное значение. К процедурным языкам относят такие языки программирования как Бейсик, Паскаль, Си.

Объектно-ориентированные языки – языки, в которых данные и функции, имеющие доступ к ним, рассматриваются как один модуль. Такое объединение называется инкапсуляцией. Объекты программы образуют иерархическую систему и могут наследовать методы и элементы данных у других объектов. Объектно-ориентированные программы используют событийный механизм управления. Различные воздействия на программные объекты рассматриваются как последовательность событий. Работа программы состоит в том, что объекты, составляющие программу, реагируют на эти события. К объектно-ориентированным языкам можно отнести Object Pascal, C++, Java.

Функциональный стиль программирования является обобщением и развитием одного простого наблюдения – любую программу можно рассматривать как функцию, в качестве аргументов которой выступают входные данные этой программы, а в качестве значений – результаты ее работы. Основным методом программирования при таком подходе является суперпозиция функций. Функции часто прямо или опосредованно вызывают себя. “Чистое” функциональное программирование не признает операторов присваиваний, циклов и передач управления, функции взаимодействуют между собой только через аргументы и значения. На практике предположения чисто функционального подхода часто нарушаются. Некоторые функции обмениваются информацией через побочные эффекты. Другие, так называемые специальные функции, выполняются не по общим правилам (для их работы не требуется определения всех аргументов). Повторные вычисления при функциональном подходе к программированию осуществляются через рекурсию, разветвление вычислений основано на механизме обработки

условного выражения. Как компьютер выполняет функциональную программу? Основное правило выполнения этой программы является прямым следствием главного предположения функционального программирования – программа есть функция в математическом смысле. Из этого утверждения следует, что сначала определяются все аргументы функции, а затем вычисляется ее значение. Выполнение этого правила приводит к тому, что выполнение функциональной программы сводится к последовательности замен примитивных и составных функций, которая постепенно упрощает исходную суперпозицию до одного значения – результата. На каждом шаге выполнения программы заменяется значением одна из функций, аргументы которой известны.

В определенном смысле использование рекурсивных функций «экономит мышление» – позволяет компактно записывать решение задачи. Кроме того, часто весьма трудные для алгоритмических языков задачи с помощью рекурсивных функций естественно и легко формулируются и изящно решаются. В качестве примера рассмотрим известную задачу о Ханойских башнях. Эту задачу придумали буддийские монахи. Они верили, что время решения этой задачи для 64 дисков, соответствует времени наступления конца света. Задача заключается в следующем. Имеется три вертикальных стержня A, B, C и совокупность n круглых дисков различного диаметра с отверстием. В исходном состоянии диски нанизаны по порядку в соответствии со своим размером на диск A (рис.1).

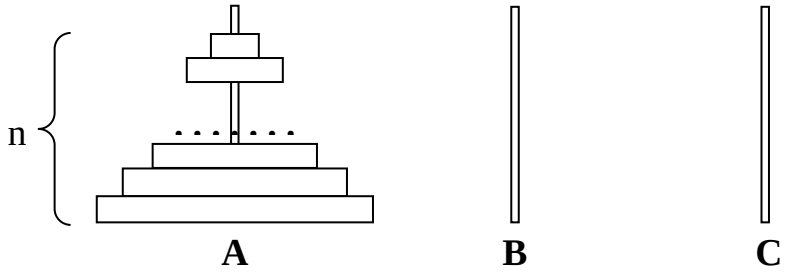


Рис. 1. Задача о Ханойских башнях

Требуется перенести все диски на стержень B, расположив их в том же порядке, соблюдая следующие правила:

- за один раз можно перенести только один диск;
- больший по размеру диск нельзя положить на меньший.

Третий стержень можно использовать как вспомогательный. Если он свободен или там лежит больший диск, то на него можно переложить очередной диск на то время, пока переносится ниже лежащий диск. В этой идее и содержится решение задачи. Ее лишь надо обобщить. Алгоритм решения задачи можно сформулировать следующим образом:

1. перенести со стержня A n-1 дисков на вспомогательный стержень C (задача Ханойские башни для n-1);
2. перенести нижний диск со стержня A на стержень B;
3. перенести со стержня C n-1 дисков на стержень B (задача Ханойские башни для n-1).

Для одного и двух дисков задача решается быстро. Для трех дисков в соответствии с изложенным алгоритмом следует выполнить следующие переносы: A → B, A → C, B → C, A → B, C → A, C → B, A → B. Для большего количества дисков количество переносов резко возрастает. Для решения задачи с 10 дисками нужно выполнить 1023 переноса, в случае n дисков решение задачи требует 2^n-1 переносов. Если считать, что для 1 переноса требуется 1 микросекунда, то время решения задачи с 64 дисками составит 585000 лет.

К функциональным языкам программирования относят такие языки, как Lisp, Haskell.

Основным отличием логических языков от функциональных является то, что выполнение программы может идти помимо обычного выполнения и в обратном направлении: на основе результата вычислять исходные данные. В логических языках описываются объекты предметной области, зависимости между ними и цель задачи, а решение задачи находится автоматически. Представителями логических языков являются Planner, Prolog.

Первое время программирование на Lisp и Prolog будет похоже на игру в волейбол одной рукой и будет вызывать чувство протеста. Но затем это чувство заменится на восхищение мощностью и лаконичностью этих языков.

1. Настройка используемого программного обеспечения

1.1 Программное обеспечение для Лиспа

Используемое свободно распространяемое программное обеспечение состоит из интерпретатора и интегрированной оболочки.

Интерпретатор GNU Clisp 2.49 можно скачать по ссылке http://sourceforge.net/projects/clisp/files/clisp/.

Это самораспаковывающийся архив 4,4 Мб.

GNU Clisp реализован немецкими студентами Бруно Хайбле (Bruno Haible) и Михаэлем Штоллем (Michael Stoll) в 1992 г. Он соответствует ANSI Common Lisp стандарту, работает под Unix, Windows. Для запуска интерпретатора следует запустить файл clisp.exe.

Консольный интерфейс GNU Clisp приведен на рис.2.

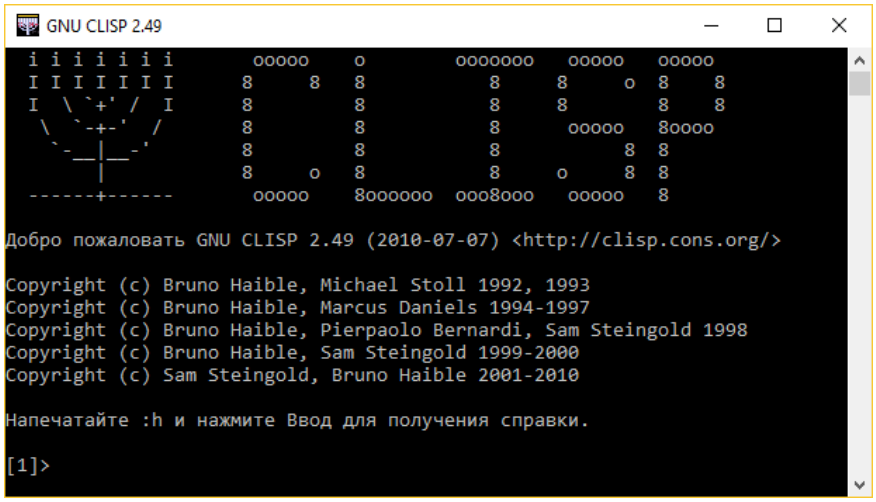


Рис. 2. Интерфейс интерпретатора GNU Clisp

Значок «>» является приглашением для ввода команды. Когда пользователь заканчивает ввод вызова функции или выражения, то интерпретатор сразу же вычисляет значение этого выражения, выдает на экран это значение и очередное приглашение для ввода команды. Если при выполнении команды появилась ошибка, то предлагаются варианты продолжения. В том числе, вернуться на предыдущий уровень (до ошибки) можно с помощью команды abort. Для завершения работы следует ввести команду (exit) или (bye).

GNU Clisp представляет собой полноценную среду для написания, отладки и исполнения программ на языке Lisp, но как и любое консольное приложение, возможности встроенного редактора сильно ограничены, именно поэтому мы будем использовать оболочку LispIDE для GNU Clisp, которая предоставляет дополнительные удобные возможности, в первую очередь для редактирования.

Редактор LispIDE можно скачать по ссылке http://daansystems.com/lispide/ – установщик (архив для скачивания находится внизу страницы). После запуска редактора первый раз необходимо в открывшемся окне прописать путь к файлу clisp.exe (рис.3).

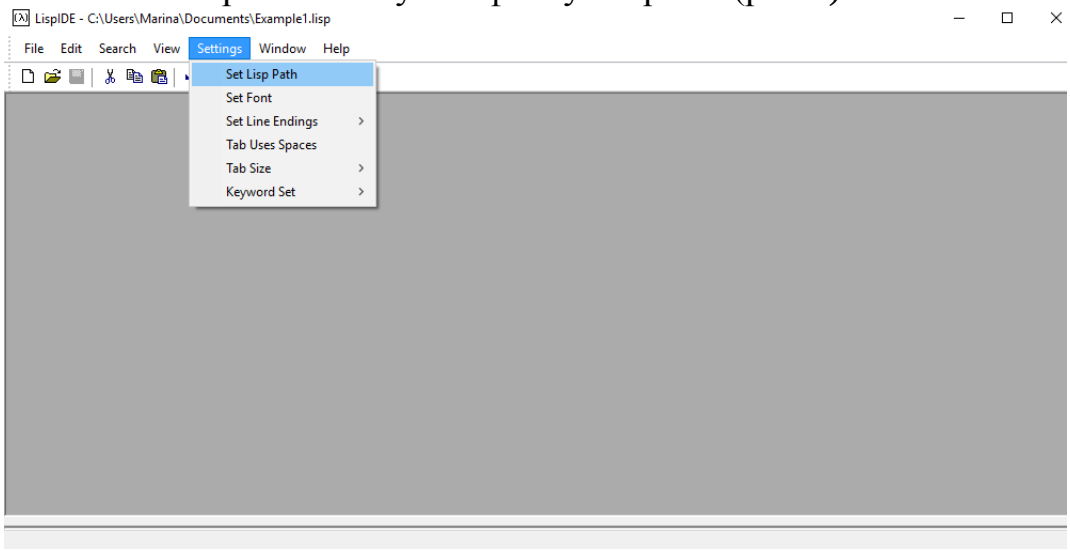


Рис. 3. Установка пути к интерпретатору Clisp (шаг 1)

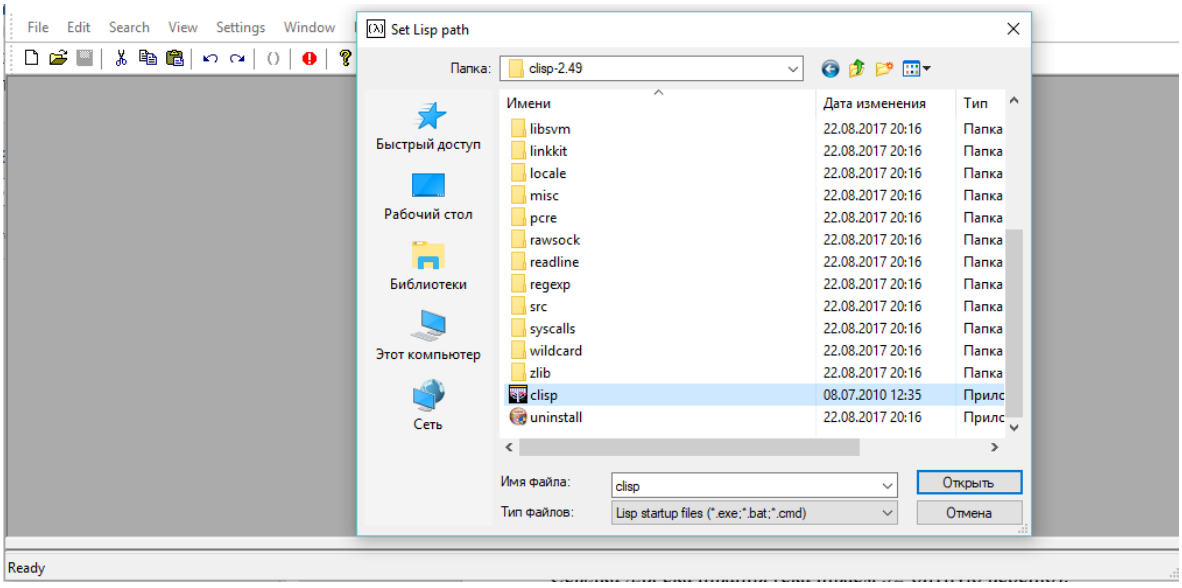
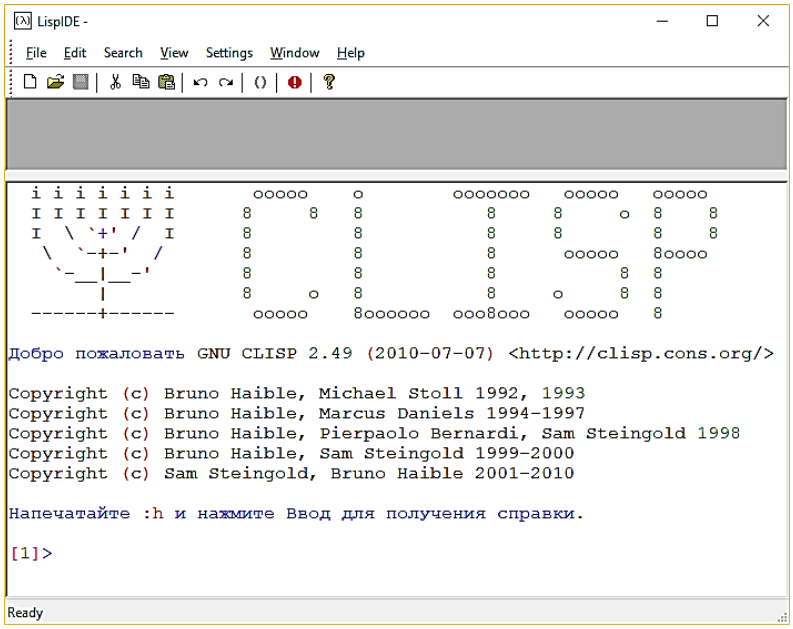


Рис. 4. Установка пути к интерпретатору Clisp (шаг 2)

В результате окно LispIDE должно принять вид, изображенный на рис. 5.



Внимание!

Если нижнее окно (окно интерпретатора Clisp) не отображается, то необходимо при зажатой левой клавише мыши над горизонтальным разделителем рабочих областей перетащить его вверх.

Рис. 5. Окно LispIDE с окном интерпретатора Clisp

Далее открываем файл с программой на Clisp или создаем новый, выбрав соответствующую пиктограмму на панели инструментов. На рис. 6 изображено окно создания нового файла.

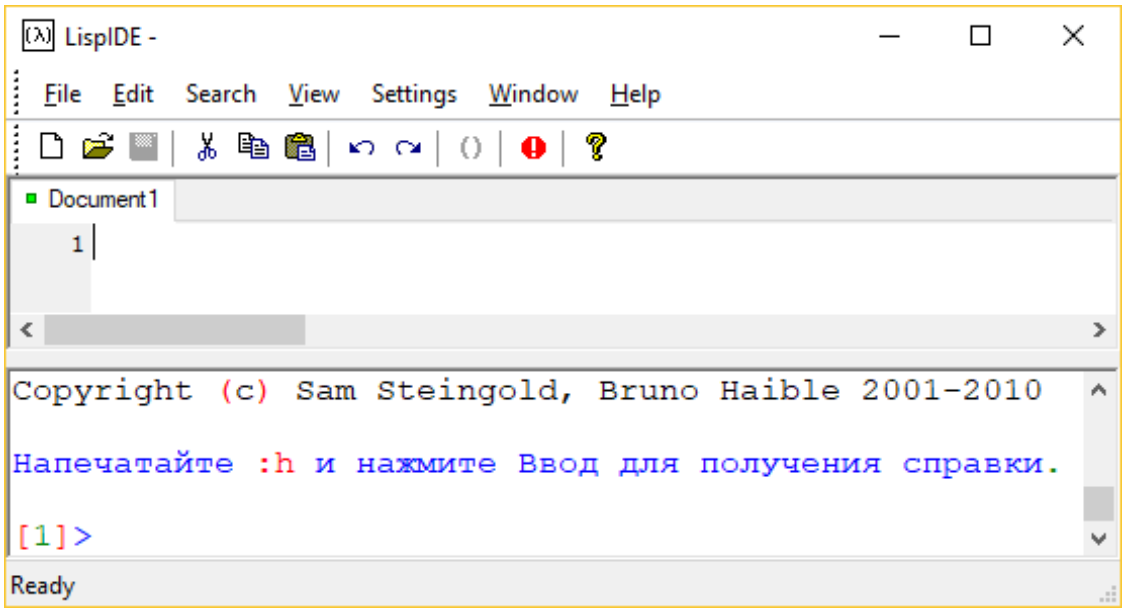


Рис. 6. Окно LispIDE с окнами редактора нового файла и интерпретатора Clisp

Замечание: LispIDE не позволяет сохранять в папки или файлы с названиями на русском языке.

1.2 Программное обеспечение для Пролога

Также, как и в случае с Лиспом, используемое свободно распространяемое программное обеспечение состоит из интерпретатора и интегрированной оболочки.

Интерпретатор SWI-Prolog 7.6.4 (или новее) можно скачать по ссылке (скачиваем 32-битную версию вне зависимости от разрядности системы): http://www.swi-prolog.org/download/stable – установщик.

Консольный интерфейс SWI-Prolog изображен на рис.7.

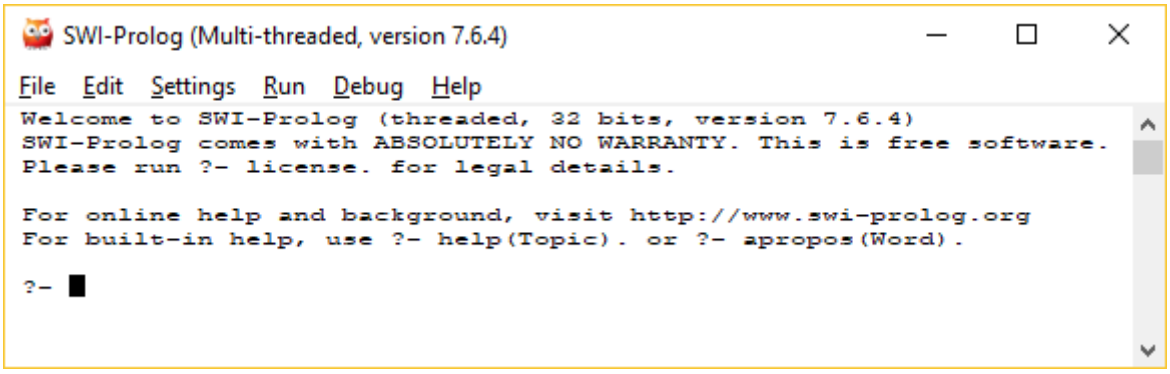


Рис. 7. Окно интерпретатора SWI-Prolog

SWI-Prolog представляет собой полноценную среду для написания, отладки и исполнения программ на языке SWI-Prolog. Встроенный редактор открывается в новом окне при выборе в меню File пункта редактирования файла или создания нового (рис.8).

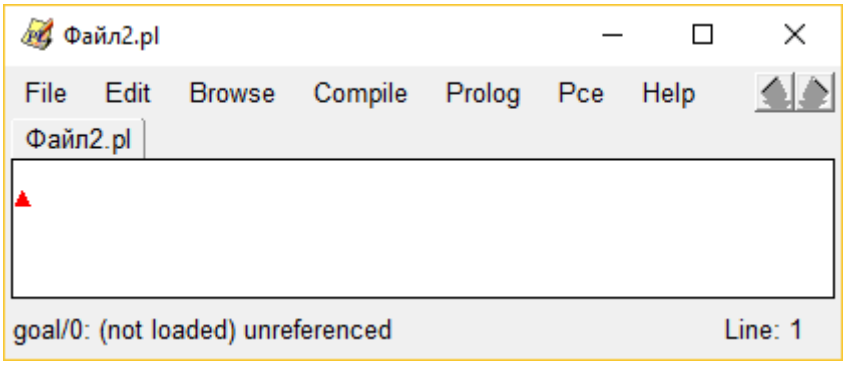


Рис. 8. Окно редактора SWI-Prolog

Мы будем использовать вместо встроенного редактора оболочку SWI-Prolog-Editor 4.26, которая представляет дополнительные удобные возможности.

Запуск программы на выполнение из окна редактора осуществляется выбором Compile-Compile buffer. Далее в окне интерпретатора вводится вызов предиката, и далее работа идет с двумя окнами.

Далее вместо встроенного редактора будет использоваться оболочка SWI-Prolog-Editor 4.26, которая представляет дополнительные удобные возможности. Редактор SWI-Prolog-Editor 4.26 (Может работать только с 32-битной версией SWI-Prolog) можно скачать по ссылке со страницы http://www.frankengel.org/index.php/schule/informatik/informatik-software/145-informatik-software.

После запуска установщика SWI-Prolog-Editor на первом шаге выберите сохранение настроек в INI-файлах и, нажав кнопку Язык, в появившемся окне окне выберите russian (рис. 9 – 10).

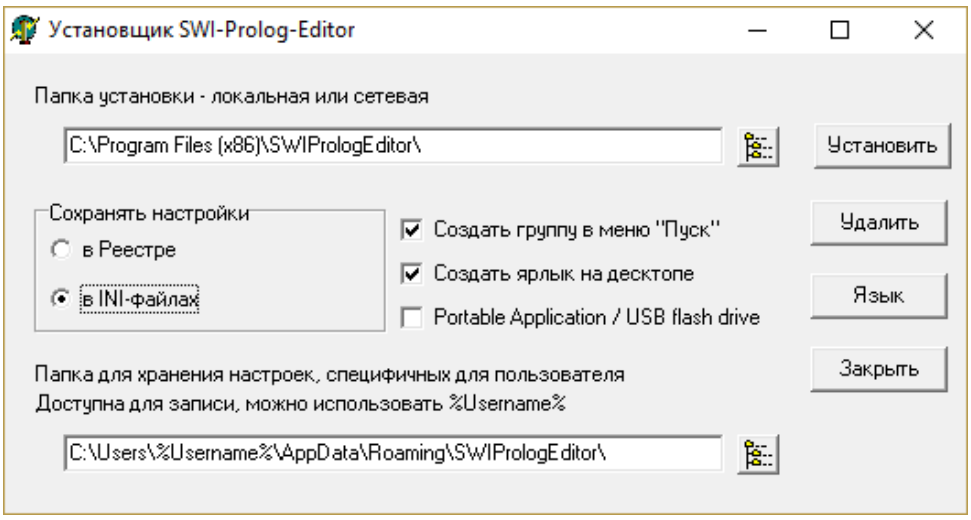


Рис. 9. Окно установки SWI-Prolog-Editor

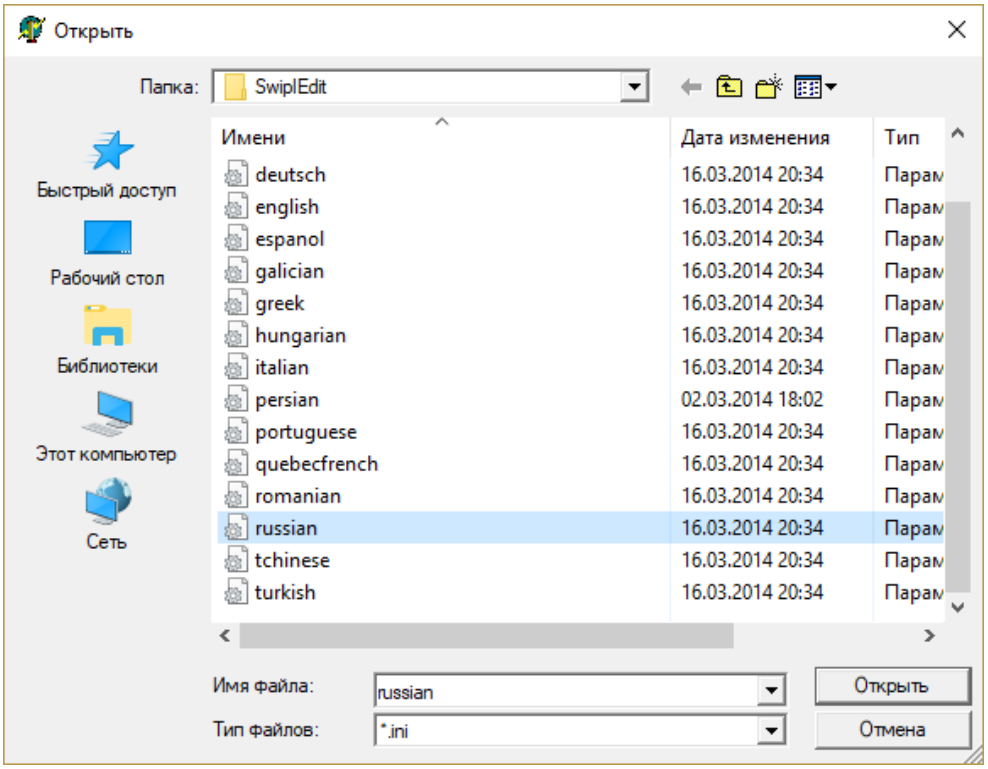


Рис. 10. Выбор языка при установке SWI-Prolog-Editor

На рис. 11 изображена интегрированная среда SWI-Prolog-Editor, запущенная после установки.

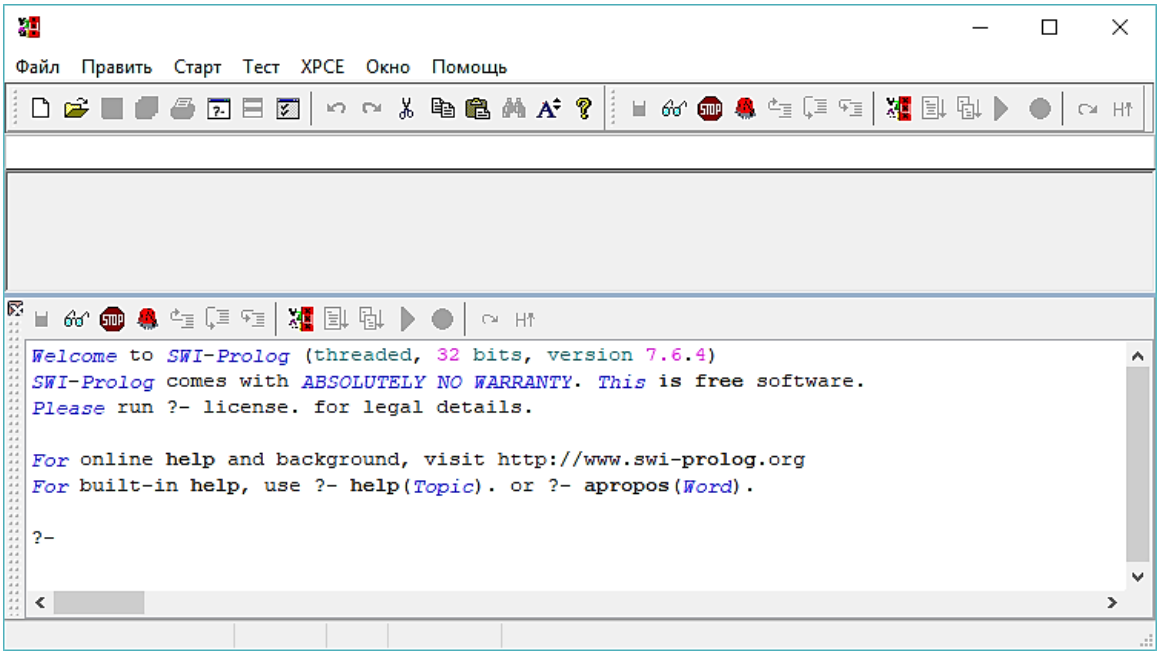


Рис. 11. Интегрированная среда SWI-Prolog-Editor с окном SWI-Prolog

Если связывание SWI-Prolog-Editor с SWI-Prolog не произошло автоматически, то необходимо указать путь к исполняемому файлу SWI-Prolog. Для этого выберите пункт главного меню Окно – Конфигурация и в открывшемся окне пропишите путь к папке bin, в которой находится исполняемый файл swipl.exe (рис. 12 – 13).

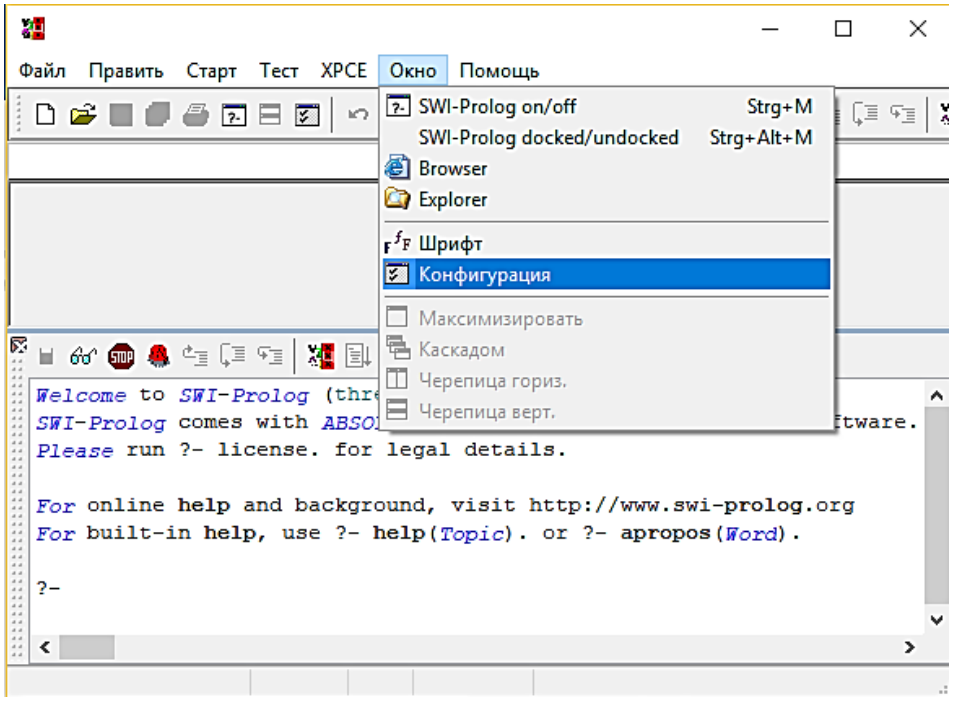


Рис. 12. Настройка связи SWI-Prolog-Editor с интерпретатором SWI-Prolog (шаг 1)

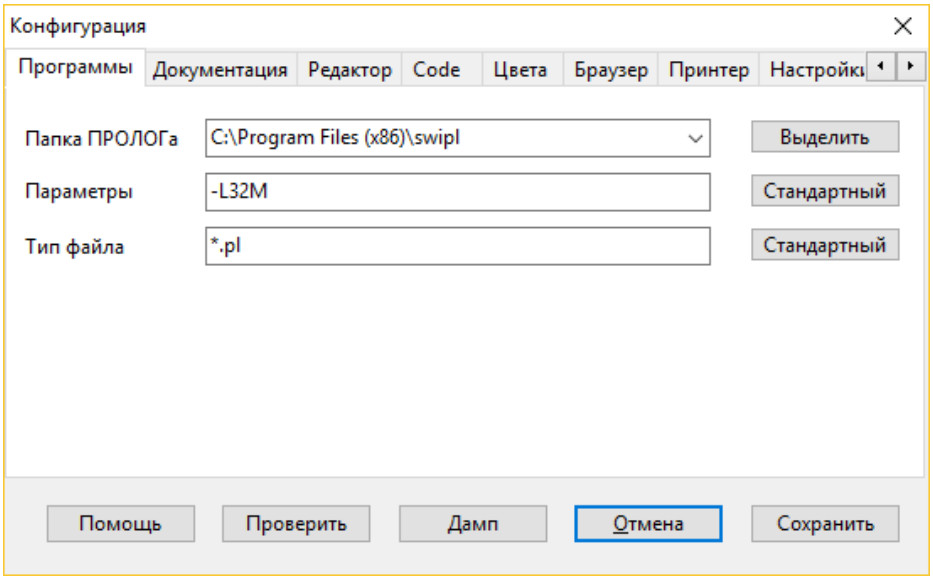


Рис. 13. Настройка связи SWI-Prolog-Editor с интерпретатором SWI-Prolog (шаг 2)

Настройка кодовой страницы необходима для правильного сопоставления строковых констант, набранных русским алфавитом, между текстом программы в среде SWI-Prolog-Editor и языком SWI-Prolog. Для настройки в этом же окне установите номер кодовой страницы 1251 и сохраните выбранные настройки, нажав кнопку Сохранить (рис. 14).

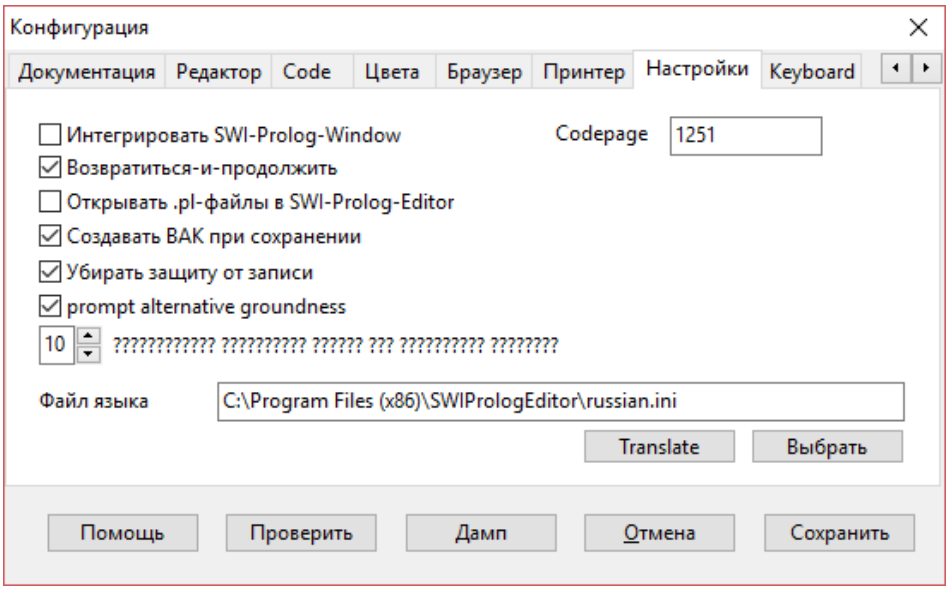


Рис. 14. Настройка кодовой страницы

Далее открываем файл с программой на SWI-Prolog или создаем новый, выбрав соответствующую пиктограмму на панели инструментов (рис.15).

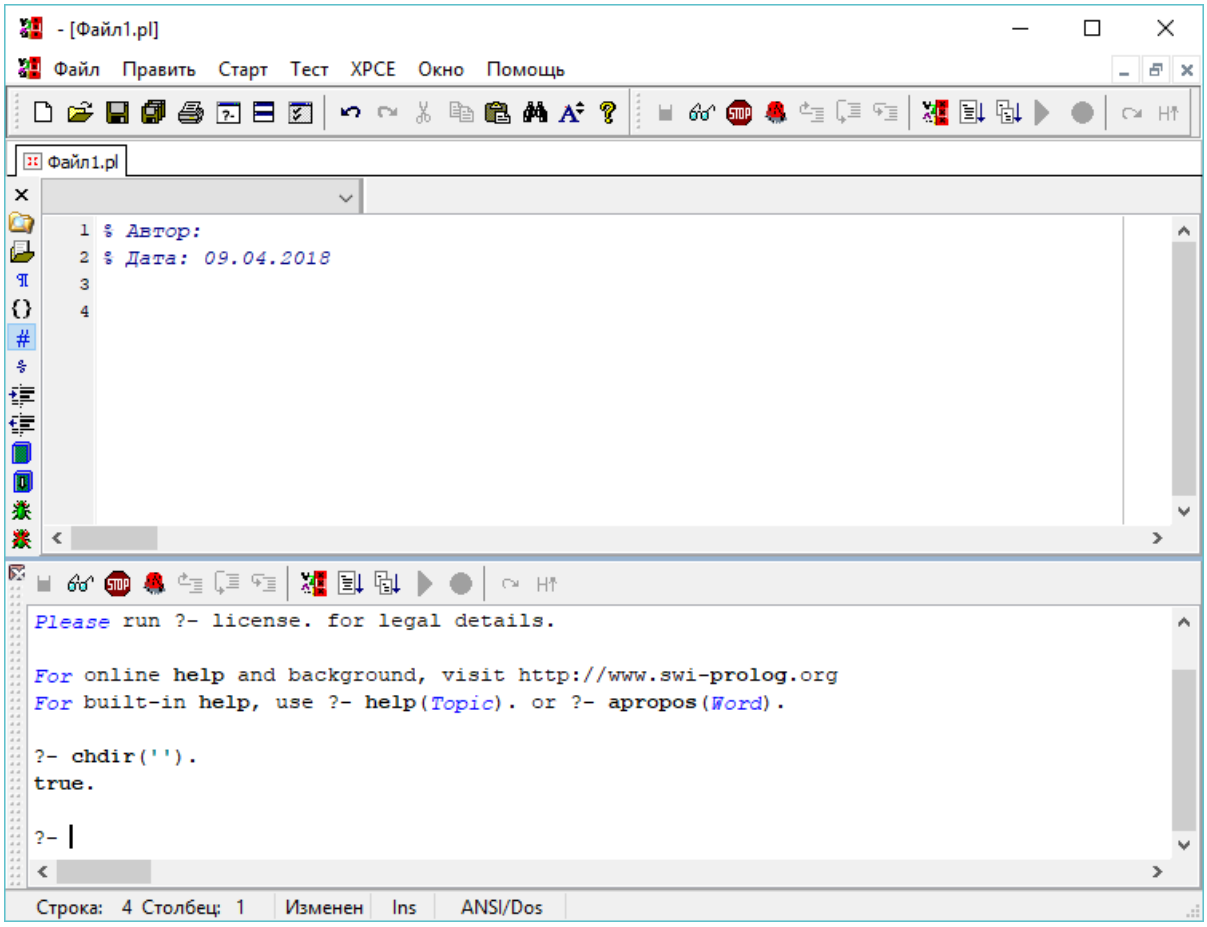


Рис. 15. Интегрированная среда SWI-Prolog-Editor с окном SWI-Prolog и окном редактирования нового файла

2. Функциональное программирование. Основы языка Lisp

Язык Lisp был разработан в Америке Дж.Маккарти в 1961 году и ориентирован прежде всего на символьную обработку, которая не накладывает таких жестких требований к эффективности, как вычислительные задачи. Название языка Lisp – сокращенная запись List processing (“Обработка списков”). Списки – это наиболее гибкая форма представления информации. С помощью списков удобно представлять множества, графы, правила вывода и другие сложные объекты. Большая часть имеющихся на рынке программ символьной обработки, работы с естественным языком написаны на Lisp. Это программы автоматического доказательства теорем, игры в шахматы, экспертные системы и т.д. На Lisp реализован AutoCAD - система автоматизации инженерных расчетов, дизайна и комплектации изделий из имеющихся элементов, и популярный текстовый редактор Emacs для систем UNIX, Linux. В настоящее время Lisp используется как основное средство программирования в системе AutoCAD – системе автоматического проектирования. В нашей стране Lisp не получил широкого распространения, хотя Маккарти еще в 1968 году в Вычислительном центре СО АН заложил основу реализации языка на машине БЭСМ-6. БЭСМ-6 была мощной 48-битовой машиной с быстродействием около 1 миллиона операций в секунду и использовалась для научно-технических расчетов. Но, тем не менее, в Москве, Санкт-Петербурге, Тбилиси появились новые реализации

Lisp для машин других серий. В 1978 году появился первый учебник по Lisp на русском языке.

Основа Lisp – лямбда-исчисление Черча, формализм для представления функций и способов их комбинирования. Например, в соответствии с этим формализмом функция может являться аргументом функции. Черты этого формализма есть в Паскале.

Перечислим ряд удивительных свойств языка. Во-первых, это однообразная форма представления программ и данных. Во-вторых, это использование в качестве основной управляющей конструкции рекурсии. В-третьих, широкое использование данных список и алгоритмов обработки этих данных. Простота синтаксиса Lisp является одновременно его достоинством и недостатком. Начинающий программист может выучить основные правила синтаксиса за несколько минут. Основной проблемой этого синтаксиса являются скобки. Часто в определении функции накапливается до 10-15 вложенных уровней скобок. Такие конструкции чрезвычайно трудно читать и отлаживать, и ошибка в расположении скобок является наиболее типичной синтаксической ошибкой в Lisp. Часто встречаются ошибки, когда выражение для определения функции выглядит «осмысленно», но семантика этого выражения не совпадает с тем, что в него хотели вложить. Существует даже шутливое толкование названия языка: Lots of Idiotic Silly Parentheses. Неудобством языка является то, что существует много диалектов и, к сожалению, ни один из них не принят в качестве стандарта. Все реализации языка являются интерпретаторами, т.е. любая команда сразу обрабатывается.

Файлы, содержащие программы, написанные на языке Clisp, имеют расширение lsp.

2.1 Типы данных в Clisp

Основные типы данных в Clisp – это атомы, списки, точечные пары. Классификация атомов приведена на рис. 16.

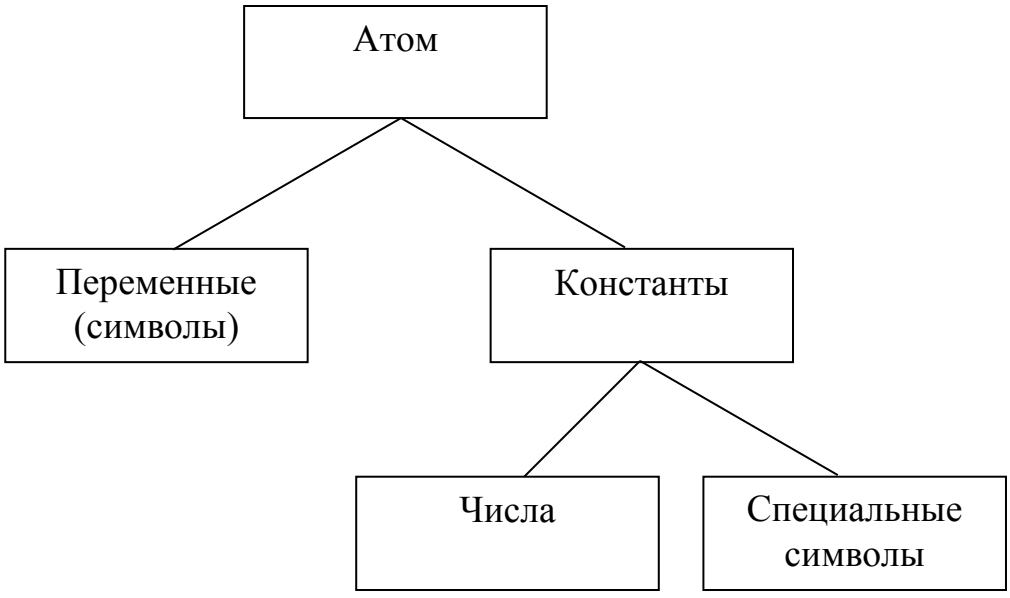


Рис. 16. Классификация атомов в Clisp

Переменная – это последовательность из букв, цифр и специальных знаков. Переменные представляют другие объекты: числа, другие символы, функции. Например, символами являются *, as-2. *Числа* состоят из цифр и точки, которым может предшествовать знак + или –. Число не может представлять других объектов, кроме самого себя. Например, -34.5, 25 – числа. *Специальные символы*: *t* и *nil*. Символ *t* обозначает логическое значение истина, а *nil* – имеет два значения: логическая ложь и пустой список.

Все структуры данных в Clisp строятся из атомов. *Списком* называется упорядоченная последовательность, элементами которой являются атомы или списки. Список заключается в скобки, а элементы разделяются пробелами.

Пример (a) – список из одного элемента;
() или nil – пустой список;
((my house) has (big windows)) – список из трех элементов.

Часто бывает удобно не различать атомы и списки. В этом случае говорят о символьных выражениях (*s–выражениях*).

Про точечные пары будет изложено позже.

2.2 Встроенные функции

В Clisp вызов функции записывается в префиксной нотации. Сначала идет имя функции, пробел, затем аргументы через пробел, и все это заключается в скобки.

Пример 1 Математическая запись f(x) соответствует лисповской записи (f x), а математическая запись x·y–z соответствует (– (* x y) z).

Видно, что по внешнему виду функция и список не различаются. Поэтому, для того, чтобы выражение в скобках воспринималось как список, а не вызов функции, используется специальная функция **QUOTE** (читается как квэут). Эта функция блокирует вычисления и соответствует математической функции f(x)=x. Причем, значение аргумента не вычисляется. Часто вместо (**QUOTE** x) пишут 'x (знак апострофа находится на клавише с буквой Э на английской раскладке). Перед константой знак ' не ставят, т.к. константа и ее значение совпадают. Самая левая **QUOTE** блокирует все вычисления в своем аргументе.

В примерах после символа **–>** идет возвращаемое интерпретатором значение.

Пример 2 Рассмотрим вычисление функции (**QUOTE** (+ 2 3)) двумя способами: в окне интерпретатора и из окна редактора (рис. 17 – 18).

Для вычисления в окне интерпретатора после знака приглашения > вводим (**QUOTE** (+ 2 3)) и нажимаем Enter. В следующей строке появится возвращаемое этой функцией значение.

Можно набрать (**QUOTE** (+ 2 3)) в окне редактора, установить курсор после последней скобки и нажать пиктограмму из двух скобок на панели инструментов. Возвращаемое значение отобразится в окне интерпретатора. Вместо установки курсора после последней скобки функции можно выделить всю функцию и также нажать пиктограмму со скобками. Вместо нажатия пиктограммы можно использовать комбинацию клавиш Shift+Enter. Важно помнить, что LispIDE посылает исполняющей среде ту часть выражения, которая

окружена выделенными скобками, а не все выражение в целом. Неправильное установка курсора или выделение перед выполнением функции может приводить к неправильным результатам.

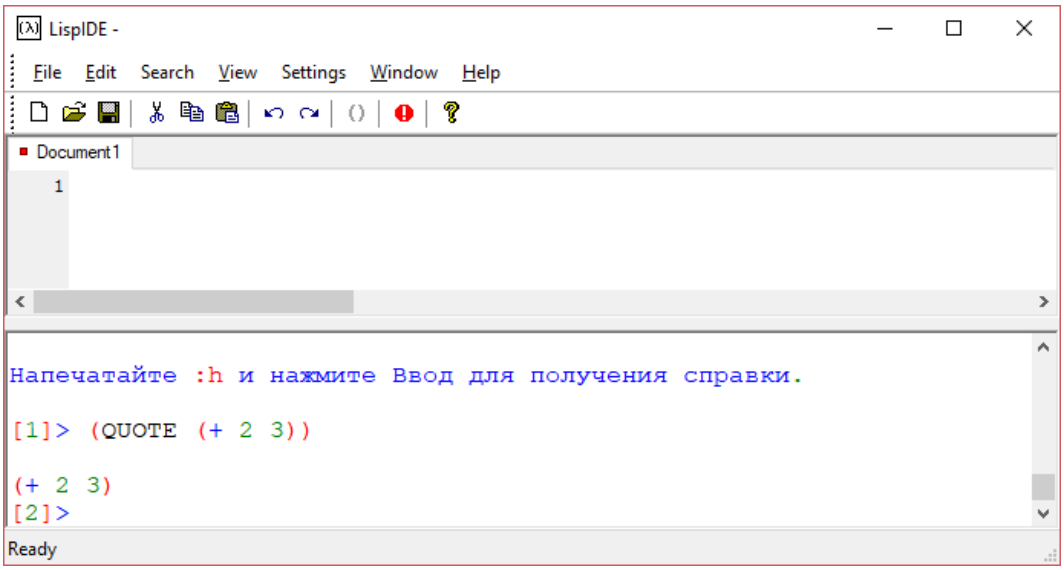


Рис. 17. Вычисление значения функции из окна интерпретатора

Рис. 18. Вычисление значения функции из окна редактора

В дальнейшем после символа «–>» будут приводиться возвращаемые значения функций.

Для перезапуска интерпретатора достаточно нажать пиктограмму с восклицательным знаком на панели инструментов.

Clisp не различает строчные и прописные буквы. В лекциях названия функций написаны прописными буквами для удобства.

Пример 3 Рассмотрим еще 2 примера вычислений:

1. (+ '2 3) –> 5;
2. (**QUOTE** '(+1 2)) –> '(+ 1 2).

2.2.1 Арифметические функции

В Clisp присутствуют основные арифметические функции:

– Сложение. Обозначается, как + и может иметь несколько аргументов. Например, a+b+c+d будет иметь вид (**+** a b c d).

– Вычитание. Обозначается, как – и может иметь несколько аргументов. Например, a–b–c–d будет иметь вид (**–** a b c d).

– Умножение. Обозначается, как ***** и может иметь несколько аргументов. Например, a·b·c будет иметь вид (***** a b c).

– Деление. Обозначается, как / и может иметь несколько аргументов. Например, 16:2:4 будет иметь вид (***/*** 16 2 4).

– Взятие модуля. Обозначается как **ABS** и имеет один аргумент. Например, |a| будет иметь вид (**ABS** a).

Суперпозиция функций всегда вычисляется «изнутри наружу».

Пример 4 Запишем на Clisp следующие выражения:

1. (1+2)·(3+4)
2. (a+b)/2
3. x^2+2x-5

В первом случае, получаем (* (+ 1 2) (+ 3 4)), во втором – (/ (+ a b)), в третьем – (+ (* x x) (* 2 x) –5).

2.2.2 Функции обработки списков

Разделим список на голову и хвост. Головой назовем первый элемент списка, а хвостом – список без первого элемента. Например, для списка ((1 2) 4 (5)) головой будет список (1 2), а хвостом – список (4 (5)).

Функция **CAR** находит голову списка. Результатом работы функции будет s-выражение. Вызов функции имеет вид:

(**CAR** список).

Функция **CDR** находит хвост списка. Результатом работы функции будет список. Вызов функции имеет вид:

(**CDR** список).

Пример 5 Рассмотрим примеры работы функций **CAR** и **CDR**:

1. (**CAR** '(a b c)) → a;
2. (**CDR** '(a b c)) → (b c);
3. (**CDR** '(a)) → nil.

Последовательно применяя функции **CAR** и **CDR** можно выделить любой элемент списка. Только следует помнить, что функции применяются «изнутри – наружу».

Пример 6 С помощью композиций функций **CAR** и **CDR** выделим в списке ((a b c) (d e) (f)) элемент c.

(**CAR** (**CDR** (**CDR** (**CAR** '((a b c) (d e) (f)))))) → c.

Допускается сокращение (**CADDAR** '((a b c) (d e) (f))). Следует иметь ввиду, что при сокращении не может идти подряд больше четырех букв A и D.

Пример 7 Для выделения в списке (1(2 3((4 *)5)6)) элемента * можно выполнить: (**CADAAR**(**CDDADR** '(1 (2 3 ((4 *) 5) 6)))).

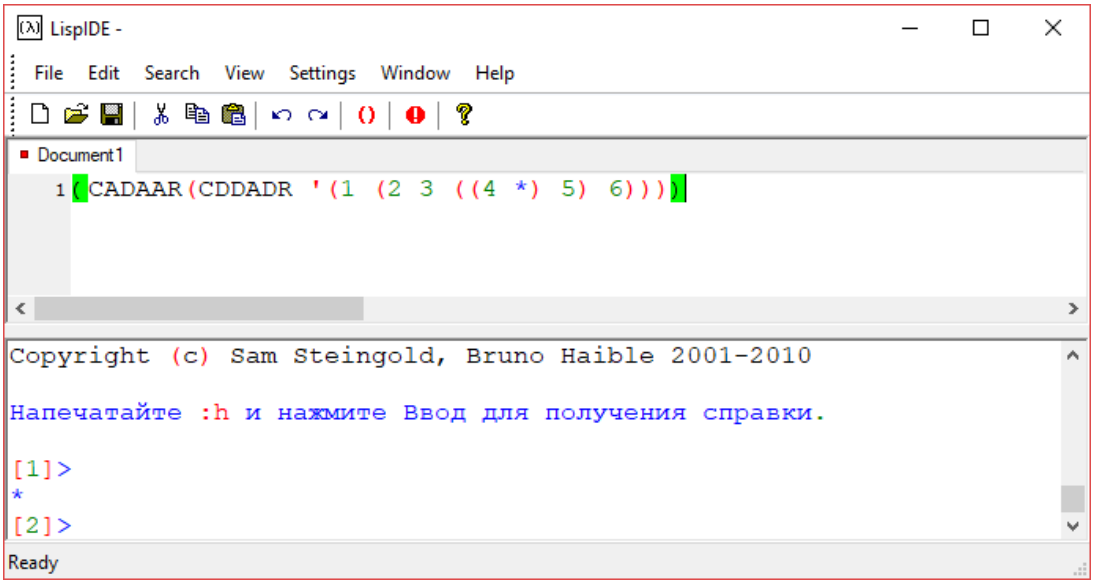


Рис. 19. Выделение элемента списка из Примера 7

Функция **CONS** создает новый список, головой которого является первый аргумент функции, а хвостом – второй аргумент функции. Результатом работы функции будет список. Вызов функции имеет вид:

(**CONS** s-выражение список).

Пример 8 Рассмотрим примеры работы функции **CONS**:

1. (**CONS** 'a '(b c)) → (a b c);
2. (**CONS** 'a nil)) → (a);
3. (**CONS** '(1 2) '(3 4)) → ((1 2) 3 4);
4. (**CONS** (**+** 1 2) '(* 3 4)) → (3 * 3 4);
5. (**CONS** nil nil)) → (nil).

Если второй аргумент функции **CONS** не является списком, то образуется так называемая *точечная пара*. О ней речь пойдет позже.

Пример 9 (**CONS** 1 2) → (1.2).

Можно заметить, что функции **CAR** и **CDR** являются обратными для **CONS**.

Пример 10

1. (**CONS** (**CAR** '(1 2 3)) (**CDR** '(1 2 3))) → (1 2 3);
2. (**CAR** (**CONS** 1 '(2 3))) → 1;
3. (**CDR** (**CONS** 1 '(2 3))) → (2 3).

Функция **LIST** создает новый список, элементами которого являются аргументы функции. Результатом работы функции будет список. Вызов функции имеет вид:

(**LIST** s_1 … s_n), где s_i – s-выражение.

Пример 11 Рассмотрим примеры работы функции **LIST**:

1. (**LIST** 'a 'b '(+ 1)) → (a b (+ 1));
2. (**LIST** '((a)) nil) → (((a)) nil).

Для любой функции **LIST** можно составить эквивалентную композицию функций **CONS**. Для примера 12 (1) эквивалентная композиция функций **CONS** будет иметь вид: (**CONS** 'a (**CONS** 'b (**CONS** '(+ 1) nil))).

Функция **APPEND** создает новый список, элементами которого являются элементы списков - аргументов функции. Результатом работы функции будет список. Вызов функции имеет вид:

(**APPEND** sp_1 … sp_n), где sp_i – список.

Пример 12 Рассмотрим примеры работы функции **APPEND**:

1. (**APPEND** '(1 2) '(3) '(+ *)) → (1 2 3 + *);
2. (**APPEND** '((1)) '()) →((1)).

Пример 13 Из атомов 1, 2, 3, *nil* создадим список (1 (((2)) 3)) двумя способами:

а) с помощью композиций функций **CONS**

(**CONS** 1 (**CONS** (**CONS** (**CONS** (**CONS** 2 nil) nil) (**CONS** 3 nil)) nil));

б) с помощью композиций функций **LIST**

(**LIST** 1 (**LIST** (**LIST** (**LIST** 2)) 3)).

Решение данной задачи приведено на рис. 20 (композиция функций задается в командной строке интерпретатора).

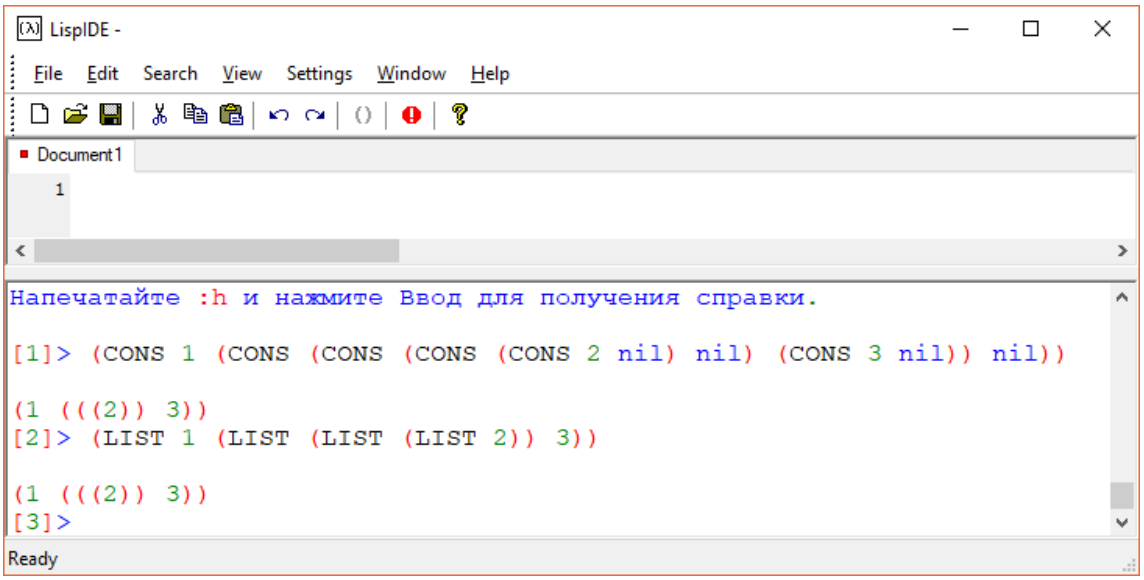


Рис. 20. Создание списка из атомов в примере 13 в командной строке интерпретатора

Функция **LAST** создает список из одного элемента – последнего элемента списка. Результатом работы функции будет список. Вызов функции имеет вид:

(**LAST** список).

Пример 14 Рассмотрим примеры работы функции **LAST**:

1. (**LAST** '(1 2 3)) → (3)**;**
2. (**LAST** '())→ nil**.**

Функция **BUTLAST** создает список из всех элементов, кроме последнего. Результатом работы функции будет список. Вызов функции имеет вид:

(**BUTLAST** список).

Пример 15

1. (**BUTLAST** '(1 2 3))→ (1 2);
2. (**BUTLAST** ' (a))→ nil.

Функция **REVERSE** возвращает список, являющийся «перевернутым» списком-аргументом. Перестановка элементов осуществляется только на верхнем уровне. Результатом работы функции будет список. Вызов функции имеет вид:

(**REVERSE** список).

Пример 16

1. (**REVERSE** '(1 2 3))→ (3 2 1);
2. (**REVERSE** '(1 (2 3) ((4))))→ (((4))(2 3)1);
3. (**REVERSE** '())→ nil;
4. (**REVERSE** (**CDR** (**REVERSE** '(1 2 3 4))→ (1 2 3).

Заметим, что композиция функций в последнем примере эквивалентна (**BUTLAST** '(1 2 3 4)).

2.3 Определение функций пользователем

2.3.1 Лямбда-функции

Определение функций и их вычисление в Лиспе основано на лямбда-исчислении Черча, представляющем простой и точный формализм для описания функций. Для описания функции в Лиспе используется *лямбда-выражение*, которое имеет вид:

(**LAMBDA** (x_1 x_2 … x_n) S_1 S_2 … S_k),

где x_1, x_2, …, x_n – формальные параметры функции, S_1, S_2, …, S_k – последовательность s-выражений, которые образуют *тело функции* и описывают вычисления. Список из формальных параметров называется *лямбда-списком*. Лямбда-выражение соответствует используемому в других языках определению функции.

Например, функцию $f\ (x,\ y) = x^2 + y^2$ можно определить с помощью следующего лямбда-выражения:

(**LAMBDA** (x y) (+ (* x x) (* y y))).

Лямбда-выражение нельзя вычислить, оно не имеет значения. Но можно организовать лямбда-вызов, который соответствует вызову функции и имеет вид:

(лямбда-выражение a_1 a_2 … a_n),

где a_1, a_2, …, a_n – вычислимые s-выражения, задающие вычисления фактических параметров.

Например, для того, чтобы найти вычислить значение функции $f\ (2,\ 3)$ где функция задается формулой $f\ (x,\ y) = x^2 + y^2$, можно организовать следующий лямбда-вызов:

((**LAMBDA** (x y) (+ (* x x) (* y y))) 2 3) → 13.

Вычисление лямбда-вызова производится в два этапа. Сначала вычисляются значения фактических параметров и соответствующие формальные параметры

связываются с полученными значениями. На следующем этапе с учетом новых связей вычисляется тело функции, и последнее вычисленное значение возвращается в качестве значения лямбда-вызова. После завершения лямбда-вызова фактические параметры получают те связи, которые были у них до вычисления лямбда-вызова, т.е. происходит передача параметров по значению.

Лямбда-вызовы можно объединять между собой и другими формами. Лямбда-вызовы можно ставить как на место тела функции, так и на место фактических параметров.

Пример 1

1. Лямбда-вызов стоит на месте фактического параметра:

((**LAMBDA** (x) (**LIST** 2 x)) ((**LAMBDA** (y) (**LIST** y)) 'a)) → (2 (a)).

Здесь (a) – вычисленное значение фактического параметра;

2. Лямбда-вызов стоит на месте тела функции:

((**LAMBDA** (x) ((**LAMBDA** (y) (**LIST** y x)) 'a)) 'b) → (a b).

Здесь сначала формальный параметр x связывается с фактическим параметром b, затем вычисляется тело функции – лямбда-вызов, при котором формальный параметр y связывается с фактическим параметром a и создается список (a b).

Лямбда-выражение является чисто абстрактным механизмом для определения и описания вычислений. Это безымянная функция, которая пропадает сразу после вычисления значения лямбда-вызова. Ее нельзя использовать еще раз, т.к. она не имеет имени. Тем не менее, безымянные функции используются при передаче функции в качестве аргумента другой функции или при формировании функции в результате вычислений.

2.3.2 Определение функции с именем

Дать имя функции можно с помощью функции **DEFUN**, вызов которой имеет следующий вид:

(**DEFUN** имя_функции лямбда-список тело_функции).

Функция **DEFUN** возвращает имя функции, а ее побочным эффектом является связывание имени функции с лямбда-выражением (**LAMBDA** лямбда-список тело_функции).

Пример 2

1. Определим функцию **LIST2** с двумя аргументами, которая создает список из своих аргументов:

   (**DEFUN LIST2** (x y)
   (**CONS** x (**CONS** y nil))
) → list2;

   Обращение к функции:

   (**LIST2** 'a 1) → (a 1);

2. Определим функцию без аргументов pi, которая приближенно вычисляет число π:

   (**DEFUN PI**() 3.141592) → pi;

   Обращение к функции:

   (PI) → 3.141592.

3. Определим функцию, которая меняет местами первый и четвертый элементы списка.

```
(DEFUN ZAMENA (l)
 (APPEND
      (LIST (CADDDR l) (CADR l) (CADDR l) (CAR l))
            (CDDDDR l)
 )
)→ ZAMENA
```

Обращение к функции:

(ZAMENA '(a b c d e f g)) → (d b c a e f g).

2.3.3 Задание параметров в лямбда-списке

При определении функции можно в лямбда-списке использовать ключевые слова, с помощью которых можно по-разному трактовать аргументы функции при ее вызове. Ключевое слово начинается символом *&*, записывается перед параметрами, на которые действует, и его действие распространяется до следующего ключевого слова. Параметры, указанные до первого *&* обязательны при вызове. До сих пор определяли функции без использования ключевых слов, т.е. использовали только обязательные параметры.

Соответствие ключевых слов и типов параметров, обозначаемых ими:

Ключевое слово	Значение
&optional	необязательные параметры
&rest	переменное количество параметров
&key	ключевые параметры

Если параметр необязательный, то при вызове функции он может отсутствовать. Для такого параметра можно указать его значение по умолчанию, либо не указывать, тогда оно будет равно *nil*.

Пример 3

1. **(DEFUN S (&optional** (x 2) (y 5)) (+ x y))

После такого определения можно выполнять вызов функции **S** с одним, двумя параметрами или без параметров. При этом, если при вызове функции значения первого или обоих параметров не заданы, то они считается, что они равны соответственно 2 и 5.

(S 8) → 13

(S 8 3) → 11

(S) → 7

2. (**DEFUN P** (x **&optional** y) (**APPEND** x y))

После такого определения можно выполнять вызов функции **P** с одним или двумя параметрами. При этом, если при вызове функции значение второго параметра не задано, то считается, что он равен *nil*.

(**P** '(a b)) → (a b)

(**P** '(a b) '(c)) → (a b c)

Можно написать функцию, вызываемую с любым переменным количеством аргументов. Для этого следует указать перед последним параметром ключевое слово &rest, тогда Lisp будет собирать все аргументы, не попавшие в обязательные параметры, в список и связывать &rest-параметр с этим списком.

Пример 4

1. (**DEFUN R** (x **&rest** y) (**CONS** x y))

После такого определения можно использовать вызов функции **R** с одним, двумя, тремя и т.д. параметрами.

(**R** 1) → (1)

(**R** 1 2) → (1 2)

(**R** 1 2 3 4 5) → (1 2 3 4 5)

2. (**DEFUN T** (x **&optional** y **&rest** z) (**LIST** x y z))

После такого определения можно использовать вызов функции **T** с одним, двумя, тремя и т.д. параметрами.

(**T** 1) → (1 nil nil)

(**T** 1 2) → (1 2 nil)

(**T** 1 2 3 4 5) → (1 2 (3 4 5))

С помощью ключевого слова &key можно определить ключевые параметры, которые задаются при вызове с помощью ключей. Ключом называется имя формального параметра, перед которым поставлено двоеточие. При обращении к функции после ключа должен следовать фактический параметр. Ключевые параметры при вызове можно перечислять в любом порядке, не зная их порядок в определении функции. Ключевые параметры являются необязательными. В определении функции для ключевых параметров можно указывать значения по умолчанию. Если значение не определено по умолчанию и ключевой параметр отсутствует при обращении к функции, то его значение равно *nil*. С помощью ключевых параметров можно определять функции, которые в зависимости от используемой при вызове комбинации параметров запускаются с их различными значениями.

Пример 5

(**DEFUN U** (**&key (**x 1) y) (**LIST** x y))

После такого определения можно использовать вызов функции **U** с одним, двумя параметрами и без параметров.

(**U** :y 2) → (1 2)

(**U** :y 2 :x 3) → (3 2)

(**U**) → (1 nil)

2.4 Предикаты

Если перед вычислением функции необходимо убедиться, что ее аргументы принадлежат области определения, или возникает задача подсчета элементов списка определенного типа, то используют специальные функции – предикаты. *Предикатом* называется функция, которая используется для распознавания или идентификации и возвращает в качестве результата логическое значение – специальные символы *t* или *nil*. Часто имена предикатов заканчиваются на P (от слова Predicate).

Предикат **ATOM** возвращает значение *t*, если аргумент является атомом. Вызов предиката имеет вид:

(**ATOM** s-выражение).

Пример 1 Рассмотрим примеры работы предиката **ATOM**:

1. (**ATOM** 'x) → t;
2. (**ATOM** '((1) 2)) → nil;
3. (**ATOM** nil) → t.

Предикат **LISTP** возвращает значение t, если аргумент является списком. Вызов предиката имеет вид:

(**LISTP** s-выражение).

Пример 2 Рассмотрим примеры работы предиката **LISTP**.

1. (**LISTP** 'x) → nil;
2. (**LISTP** '((1) 2)) → t;
3. (**LISTP** nil) → t.

Предикат **SYMBOLP** возвращает значение t, если аргумент является атомом – переменной или специальным символом. Вызов предиката имеет вид:

(**SYMBOLP** s-выражение).

Пример 3 Рассмотрим примеры работы предиката **SYMBOLP**.

1. (**SYMBOLP** '+x) → t;
2. (**SYMBOLP** 2) → nil;
3. (**SYMBOLP** '(a b c)) → nil;
4. (**SYMBOLP** t) → t.

Предикат **NUMBERP** возвращает значение t, если аргумент является числом. Вызов предиката имеет вид:

(**NUMBERP** s-выражение).

Пример 4 Рассмотрим примеры работы предиката **NUMBERP**.

1. (**NUMBERP** 56) → t;
2. (**NUMBERP** 't) → nil.

Предикат **NULL** возвращает значение t, если аргумент является пустым списком. Вызов предиката имеет вид:

(**NULL** s-выражение).

Пример 5 Рассмотрим примеры работы предиката **NULL**.

1. (**NULL** '(5 6)) → nil;
2. (**NULL** (**ATOM** '(1 2))) → t;

3. (**NULL** (**LIST** '(1 2))) → nil.

Рассмотрим еще несколько предикатов, аргументами которых являются числа.

Предикат = возвращает значение *t*, если все аргументы – числа равны между собой. Вызов предиката имеет вид:

(= n_1 … n_m), где n_i – число.

Пример 6 Рассмотрим примеры работы предиката =.

1. (**=** 1 1 2) → nil;
2. (**=** 4 4) → t.

Предикат < возвращает значение t, если все аргументы – числа упорядочены в порядке возрастания. Вызов предиката имеет вид:

(**<** n_1 … n_m), где n_i – число.

Этот предикат можно использовать для проверки попадания числового значения в заданный диапазон.

Пример 7 Рассмотрим примеры работы предиката <.

1. (**<** 1 1 2) → nil;
2. (**<** 1 x 2) → $\begin{cases} t \text{ , если значение x попадает в интервал (1;2);} \\ nil\text{, если значение x не попадает в интервал (1;2);} \end{cases}$
3. (**<** 3 6 8 9) → t.

Аналогично определяются предикаты: > (проверка аргументов – чисел на упорядоченность по убыванию); <= (проверка аргументов – чисел на упорядоченность по неубыванию); >= (проверка аргументов – чисел на упорядоченность по невозрастанию); /= (проверка аргументов – чисел на неравенство всех пар рядом стоящих аргументов).

Пример 8 Рассмотрим примеры работы предикатов >, ≤, ≥, /=:

1. (**>** 4 1 -5) → t;
2. (**<=** 3 3 8 9) → t;
3. (**>=** 9 9 5 5) → t;
4. (**/=** 1 4 1) → t.

Для сравнения s-выражений используется предикаты **EQUAL** и **EQ**. Предикат **EQUAL** возвращает значение *t*, если совпадают внешние структуры s-выражений. Вызов предиката имеет вид:

(**EQUAL** s_1 s_2), где s_i (i=1,2) – s-выражение.

Предикат **EQ** возвращает значение *t*, если совпадают внешние структуры s-выражений и их физические адреса. При использовании **EQ** следует помнить, что одноименные переменные всегда хранятся в одном месте, а списки – нет. Кроме того, малые целые числа (от -2^{24} до $2^{24}-1$) определяются уникально, а большие целые числа и дробные числа не определяются уникально. Вызов предиката **EQ** производится аналогично вызову **EQUAL**.

Пример 9 Рассмотрим примеры работы предикатов **EQUAL** и **EQ**:

1. (**EQUAL** 'a 'a) → t;
2. (**EQUAL** '(a b) '(a b)) → t;
3. (**EQUAL** '(a) 'a) → nil;
4. (**EQ** 'a 'a) → t;
5. (**EQ** 100000000 100000000) → nil;
6. (**EQ** '(a) '(a)) → nil. Здесь список (a) создан дважды и хранится по разным адресам;
7. (**EQ** 2.25 2.25) → nil.

Заметим, что функции **EQUAL** и **EQ** иногда называют *примитивными функциями сопоставления с образцом*.

2.5 Псевдофункция SETQ

Мы уже видели, что перед любыми лисповскими константами (числами и символами *t* и *nil*) не надо ставить апостроф, т.к. значением константы является сама константа. Символы могут обозначать некоторые выражения. Связать символ с некоторым значением можно при помощи функции **SETQ**. Эта функция является *псевдофункцией*, поскольку кроме возвращаемого значения (видимый эффект) имеет побочный эффект. Она возвращает в качестве своего значения вычисленное значение второго аргумента, а связывание является побочным эффектом работы этой функции.

Функция **SETQ** имеет четное количество аргументов. Аргументы с нечетными номерами должны быть символами, а с четными – s-выражениями. Функция вычисляет значения аргументов с четными номерами и возвращает значение последнего вычисленного s-выражения. Побочным эффектом работы этой функции является связывание символов-аргументов с нечетными номерами со значениями вычисленных s-выражений. Об автоматическом блокировании вычисления нечетных аргумента напоминает буква Q (от **QUOTE**) в имени функции/ Вызов функции имеет вид:

(**SETQ** p_1 s_1 … p_n s_n), где p_i – символ, s_i – s-выражение.

Пример Рассмотрим пример работы предиката **SETQ**.

(**SETQ** a 1 b 2 c a) → 1, побочный эффект – связывания: a → 1, b → 2, c → 1.

Все образовавшиеся связи действительны в течение всего сеанса работы с интерпретатором Лиспа.

2.6 Последовательные вычисления

В Лиспе имеются структуры, управляющие последовательностью вычислений. Такие структуры будем называть *предложениями*. Предложения выглядят внешне как вызовы функции. Предложение записывается в виде скобочного выражений, первый элемент которого является именем управляющей структуры, а остальные элементы «аргументами». Результатом вычисления так же, как и у функций, является значение. Но следует помнить, что

предложения не являются вызовами функций и разные предложения по-разному используют свои аргументы.

Предложения **PROG1**, **PROGN** позволяют работать с несколькими вычисляемыми формами. Они имеют следующую структуру:

(**PROG1** V_1 V_2 … V_n) и (**PROGN** V_1 V_2 … V_n),

где V_1, V_2, …, V_n – вычислимые выражения.

У этих специальных форм переменное число аргументов, которые они последовательно вычисляют и возвращают в качестве значения предложения значение первого (для **PROG1)** или последнего (для **PROGN**) аргумента.

Пример (**PROGN (SETQ** x 2) **(SETQ** y (* 3 x))) –> 6.

2.7 Разветвление вычислений

Предложение **COND** является основным средством разветвления вычислений. Это синтаксическая форма, позволяющая управлять вычислениями на основе определяемых предикатами условий. Предложение имеет следующую структуру:

(**COND**
(P_1 V_1)
(P_2 V_2)
……
(P_n V_n)
), где P_i – предикат, V_i – вычислимое выражение (i = 1, …, n).

Значение предложения **COND** определяется следующим образом: слева направо (сверху вниз) вычисляются предикаты P_1, P_2, … до тех пор, пока не встретится предикат, возвращающий значение отличное от *nil*. Пусть это будет предикат P_k. Вычисляется выражение V_k и полученное значение возвращается в качестве значения предложения **COND**. Если все предикаты предложения **COND** возвращают *nil*, то само предложение **COND** возвращает *nil*. Рекомендуется в качестве последнего предиката использовать специальный символ *t*, тогда соответствующее ему выражение будет вычисляться во всех случаях, когда ни одно другое условие не выполняется.

При составлении предикатов можно использовать встроенные логические функции **AND** (и), **OR** (или) и **NOT** (отрицание). Число аргументов функций **AND** и **OR** может быть произвольным. В случае истинности предикат **AND** возвращает значение своего последнего аргумента, а предикат **OR** – значение своего первого аргумента, отличного от *nil*.

Пример Примем, что возможны три типа выражений: пустой список, атом, список. Определим функцию **TYPE**, определяющую тип выражения.

```
(DEFUN TYPE (l)
    (COND
        ((NULL l) 'pustoi_spisok)
        ((LISTP l) 'spisok)
        (t 'atom)))
```

Рассмотрим примеры обращения к функции **TYPE.**

(**TYPE** '(1 2)) –> spisok
(**TYPE** (**ATOM** '(a t o m))) –> pustoi_spisok
(**TYPE** 'a) –> atom

В условном предложении может отсутствовать вычислимое выражение V_i (соответствующая строка имеет вид (P_i)) или на его месте может стоять несколько вычислимых выражений (тогда соответствующая строка имеет вид (P_i V_{i1} V_{i2} … V_{ik})). Если предикат P_i возвращает значение отличное от *nil*, в первом случае результатом предложения **COND** будет являться значение P_i, а во втором – после последовательного вычисления выражений V_{i1}, V_{i2}, …, V_{ik} слева направо, результатом предложения **COND** будет значение V_{ik} (неявный **PROGN**).

2.8 Комментарии

Комментариям до конца текущей строки предшествует точка с запятой. Блочный комментарий заключается между #| и |#.

Пример Определим функцию **TWO_EQ**, возвращающую *t*, если в двух списках совпадают два первых элемента.

```
(DEFUN TWO_EQ (l1 l2)
   (COND
      ((OR (NULL l1) (NULL l2)) nil) ;Нет элементов в одном из списков
      ((OR (NULL (CDR l1)) (NULL (CDR l2))) nil) ;В одном из списков один элемент
      ((EQUAL (CAR l1) (CAR l2)) (EQUAL (CADR l1) #| Это второй элемент первого списка |#
                                        (CADR l2) ) #| Это второй элемент второго списка |#))
      (t nil)
  )
)
```

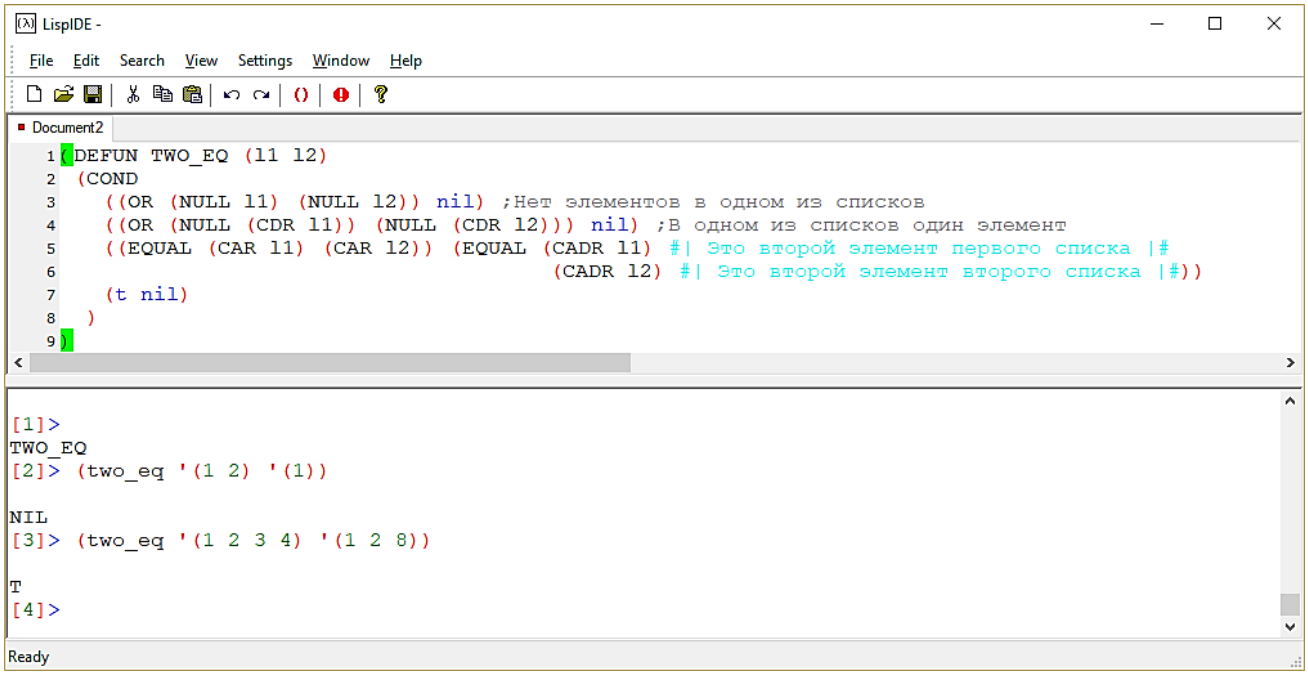


Рис. 21. Комментарии в теле функции **TWO_EQ**

2.9 Интерпретатор языка Лисп EVAL

Интерпретатор Лиспа называется **EVAL** и его можно так же, как и другие функции, вызывать из программы. «Лишний» вызов интерпретатора может, например, снять эффект блокировки вычисления от функции **QUOTE** или найти значение значения выражения, т.е. осуществить двойное вычисление.

Функция **EVAL** вычисляет значение значения аргумента и возвращает его. Вызов функции имеет вид:

(**EVAL** s-выражение).

Пример Рассмотрим примеры работы предиката **EVAL**:

1. (**EVAL** '(+ 4 8)) → 12,
2. (**SETQ** x '(a b c)) → (a b c), побочный эффект – связывание: x → (a b c);
   (**EVAL** 'x) → (a b c);
   (**EVAL** x) → ошибка, т.к. сначала вычисляется значение x, а потом делается попытка вычислить функцию с именем a;
3. (**SETQ** a 'b) → b, побочный эффект – связывание: a → b;
   (**SETQ** b 'c) → c, побочный эффект – связывание: b → c;
   (**EVAL** a) → c, вычисляется значение значения a, т.е. значение b;
   (**EVAL** 'a) → b.
4. (**SETQ** a 3) → 3, побочный эффект – связывание: a → 3;
   (**SETQ** b 'a) → a, побочный эффект – связывание: b → a;
   (**EVAL** b) → 3, вычисляется значение значения a;
   (**SETQ** a 4) → 4, побочный эффект – связывание: a → 4;
   (**EVAL** b) → 4, вычисляется значение значения b, т.е. значение a;
   (**EVAL** 'b) → a.

Для функции **EVAL** нет аналогий в процедурных языках программирования. Используя **EVAL**, мы можем выполнить «оператор», который создан Лисп-

программой и который может меняться в процессе выполнения программы. Лисп позволяет с помощью одних функций формировать определения других функций, программно анализировать и редактировать эти определения как s-выражения, а затем, используя функцию **EVAL**, исполнять их.

2.10 Функции ввода-вывода

Для ввода с клавиатуры (по умолчанию) используется функция **READ**. Функция возвращает введенное с клавиатуры s-выражение. Вызов функции имеет вид:

(**READ**).

Для ввода значений в переменные используют композицию функций **SETQ** и **READ**:

(**SETQ** переменная_1 (**READ**) … переменная_*n* (**READ**)).

Мы уже видели, что все значения, возвращаемые функциями, и значения переменных, набранных в командной строке, автоматически выводятся на экран. Если необходимо ввести данные в процессе работы программы, то для вывода пояснений для ввода используют функции **PRINT**, **PRINC, TERPRI**.

Функция **PRINT** возвращает значение аргумента и, в качестве побочного эффекта, осуществляет переход на следующую строку и выводит на экран значение аргумента. Вызов функции имеет вид:

(**PRINT** *S*),

где *S* – s-выражение или строка, заключенная в кавычки.

Функция **PRINC** возвращает значение аргумента и, в качестве побочного эффекта, выводит на экран это значение. Вызов функции имеет вид:

(**PRINC** *S*),

где *S* – s-выражение или строка, заключенная в кавычки.

В Лиспе можно использовать строки, при этом строка заключается в кавычки.

Функция **TERPRI** всегда возвращает nil и, в качестве побочного эффекта, осуществляет переход на следующую строку. Вызов функции имеет вид:

(**TERPRI**).

Пример Пусть с клавиатуры вводятся два числа и знак операции. Следует вывести на экран сформированное для вычисления выражение и результат вычисления.

Определим функцию *f* без параметров:

```
(DEFUN f()
  (PRINC "Vvedite chislo znak chislo: ") ;Приглашение ввести число, знак, число
  (SETQ a (READ) zn (READ) b (READ)) ;Ввод данных с клавиатуры
  (SETQ rez (EVAL (LIST zn a b))) #|Формируется выражение для вычисления в прединфиксной форме, вычисляется и результат вычислений связывается с переменной rez|#
  (TERPRI) ;Перевод строки
  (PRINC "Rezultat: ")
  (PRINC a) ;Вывод первого числа
  (PRINC zn) ;Вывод знака операции
  (PRINC b) ;Вывод второго числа
```

```
  (PRINC "=") ;Вывод знака равно
  (PRINC rez) ;Вывод результата операции
)
```

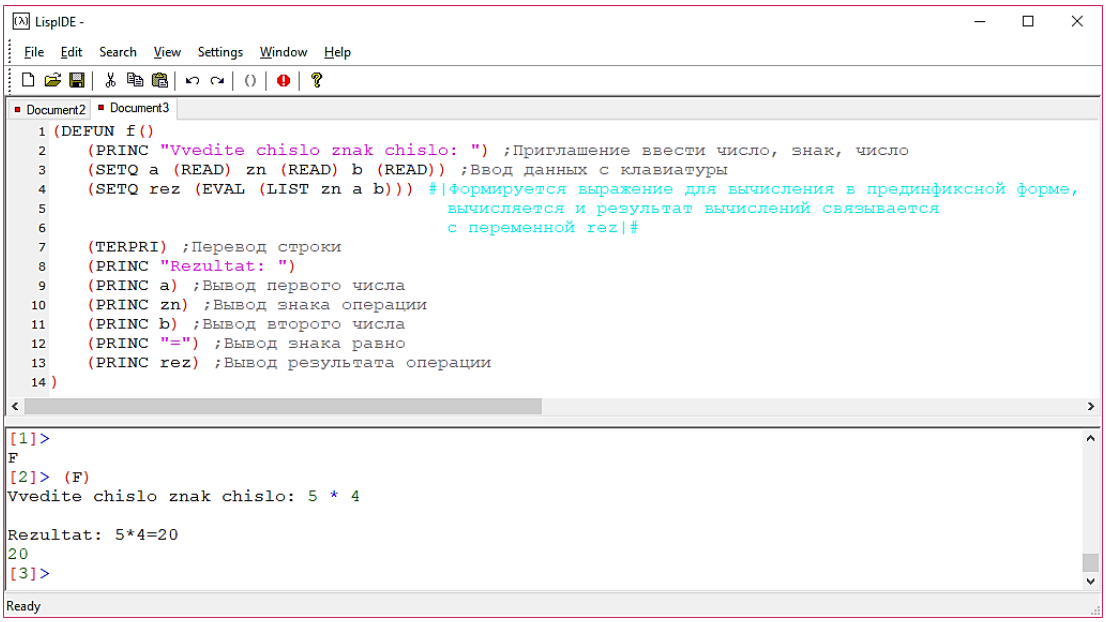


Рис. 22. Функция, осуществляющая ввод с клавиатуры

Замечание: Clisp не поддерживает кириллицу. Комментарии на русском языке, которые имеются в программе, вставлены из Word.

2.11 Рекурсия

Функция называется рекурсивной, если в определяющем ее выражении содержится хотя бы одно обращение к ней самой (явное или через другие функции). Рекурсия в Лиспе основана на математической теории рекурсивных функций. В этой области математики изучаются вопросы, связанные с вычислимостью. Под вычислимыми понимаются такие задачи, которые можно запрограммировать и решить с помощью компьютера. Основная идея рекурсивного определения заключается в том, что функцию с помощью рекуррентных формул можно 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. При определении рекурсивных функций в Лиспе применяется такой же подход. Во-первых, определение рекурсивной функции должно содержать хотя бы одну рекурсивную ветвь. Во-вторых, в рекурсивном описании должно быть условие окончания рекурсии. При написании рекурсивных функций следует обратить внимание на следующее. Когда выполнение функции доходит до рекурсивной ветви, функционирующий вычислительный процесс приостанавливается, а запускается с начала новый такой же процесс, но уже на новом уровне. Прерванный процесс запоминается, он начнет исполняться лишь при окончании запущенного им нового процесса. В свою очередь, новый процесс так же может приостановиться и т.д. Таким образом, образуется стек прерванных процессов, из

которых выполняется лишь последний запущенный процесс. Функция будет выполнена, когда стек прерванных процессов опустеет.

Перечислим наиболее часто встречающиеся ошибки при написании рекурсивных функций:

1. ошибочное условие (например, список не укорачивается), которое приводит к бесконечной рекурсии;
2. неверный порядок условий;
3. отсутствие проверки какого-нибудь случая.

При написании рекурсивных функций для предотвращения бесконечной рекурсии условия остановки рекурсии следует ставить в начале тела функции. В этих условиях следует проверить все особые случаи.

Рекурсия хорошо подходит для работы со списками, так как списки могут содержать в качестве элементов подсписки, т.е. иметь рекурсивное строение. Для обработки рекурсивных структур естественно использовать рекурсивные функции. Так же в Лиспе рекурсия используется для организации повторяющихся вычислений. На рекурсии основано разбиение проблемы и разделение ее на подзадачи, решение которых пытаются свести к уже решенным задачам, или в соответствии с основной идеей рекурсии к решаемой в настоящий момент задаче. При поиске решения задачи в Лиспе часто используется механизм возврата (backtracking), основанный так же на рекурсии. При помощи этого механизма можно вернуться из тупиковой ветви к месту разветвления, аннулировав проделанные вычисления.

Далее рассмотрим различные виды рекурсии, используемые в Лиспе.

2.11.1 Обработка ошибок и трассировка функций

Среда выполнения GNU Clisp организована в виде REPL (read-eval-print loop) – бесконечного цикла чтения, вычисления и вывода введенного выражения. Применительно к среде выполнения GNU Clsip различают так называемый цикл верхнего уровня (Top level loop) и циклы прерывания (break loops). В том случае, когда в цикле возникает ошибка, исполняющая среда переходит в цикл прерывания. Нахождение в таком цикле обозначается меткой Break *n*, где *n* – номер цикла прерывания. В принципе можно не учитывать в каком именно цикле идут вычисления, но желательно все же всю работу выполнять на верхнем уровне, поскольку каждый цикл прерывания инициирует свои кадр стека и лексическое окружение. Для возврата из цикла прерывания необходимо выполнить команду abort или ее сокращение: «:a». Обратите внимание, что оба представления команды записываются без круглых скобок.

Трассировка используется для тестирования функции при ее неверной работе. С помощью трассировки можно отследить вычисление тестируемой функции (или функций) с целью локализации и исправления ошибок. Если некоторая функция помечается как трассируемая, то система информирует об имени функции и значениях аргументов каждый раз при входе в функцию и выводит значение функции, как только оно вычислено.

Трассировка представляет собой специальный макрос, позволяющий трассировать вычисление значения функции или обработку списка. Будем использовать самую простую его форму:

(**TRACE** <имя_функции>).

Если выполнение **TRACE** завершено удачно, то возвращается имя трассируемой функции (в случае одного аргумента). В противном случае возвращается *nil*. Если необходимо трассировать несколько функций, то их имена перечисляются через пробел. При успешном выполнении **TRACE** возвращается список из имен трассируемых функций.

Если была включена трассировка, то при обращении к функции будут отображаться имена вызываемых функций, значения их аргументов в момент вызова и возвращаемые значения после вычислений. Цифрами обозначаются уровни рекурсивных вызовов. После знака «==>» указываются возвращаемые значения соответствующего рекурсивного вызова.

Отключение трассировки всех функций производится с помощью вызова (**UNTRACE**). Если необходимо отключить трассировку только некоторых функций, то их имена перечисляются в качестве аргументов **UNTRACE**.

Примеры трассирования функций будут рассмотрены ниже.

2.11.2 Простая рекурсия

Рекурсия называется *простой*, если вызов функции встречается в некоторой ветви лишь один раз. В процедурном программировании простой рекурсии соответствует обыкновенный цикл. Простую рекурсию можно разделить на два класса:

- *рекурсия по значению*, когда рекурсивный вызов функции определяет результат функции;
- *рекурсия по аргументу*, когда результатом функции является значение другой функции, у которой одним из аргументов является рекурсивный вызов определяемой функции.

Пример 1 Определим функцию **ST**, возводящую число в целую неотрицательную степень. Воспользуемся рекурсивным определением степени:

$$x^n = \begin{cases} 1, n = 0 \\ x \cdot x^{n-1}, n > 0 \end{cases}$$

```
(DEFUN ST (x n)
   (COND
      ((= n 0) 1) ;Условие остановки  рекурсии
      (t (* x (ST x (- n 1))))
   )
 )
```

В данном случае имеем рекурсию по аргументу (рекурсивный вызов **ST** является аргументом функции *****).

На рис. 23 приведен вызов функции **ST**, в определении которой допущена ошибка (в рекурсивном вызове пропущен первый аргумент). Определение функции не дало ошибок, а при вызове с фактическими параметрами возникла

ошибка. Сообщение об ошибке в данной ситуации не слишком информативно, потому что самый верхний вызов функции произведен верно. Обратите внимание на подчеркнутый маркер «Break 1» в командной строке. Как описывалось выше, это означает что мы зашли в цикл прерывания.

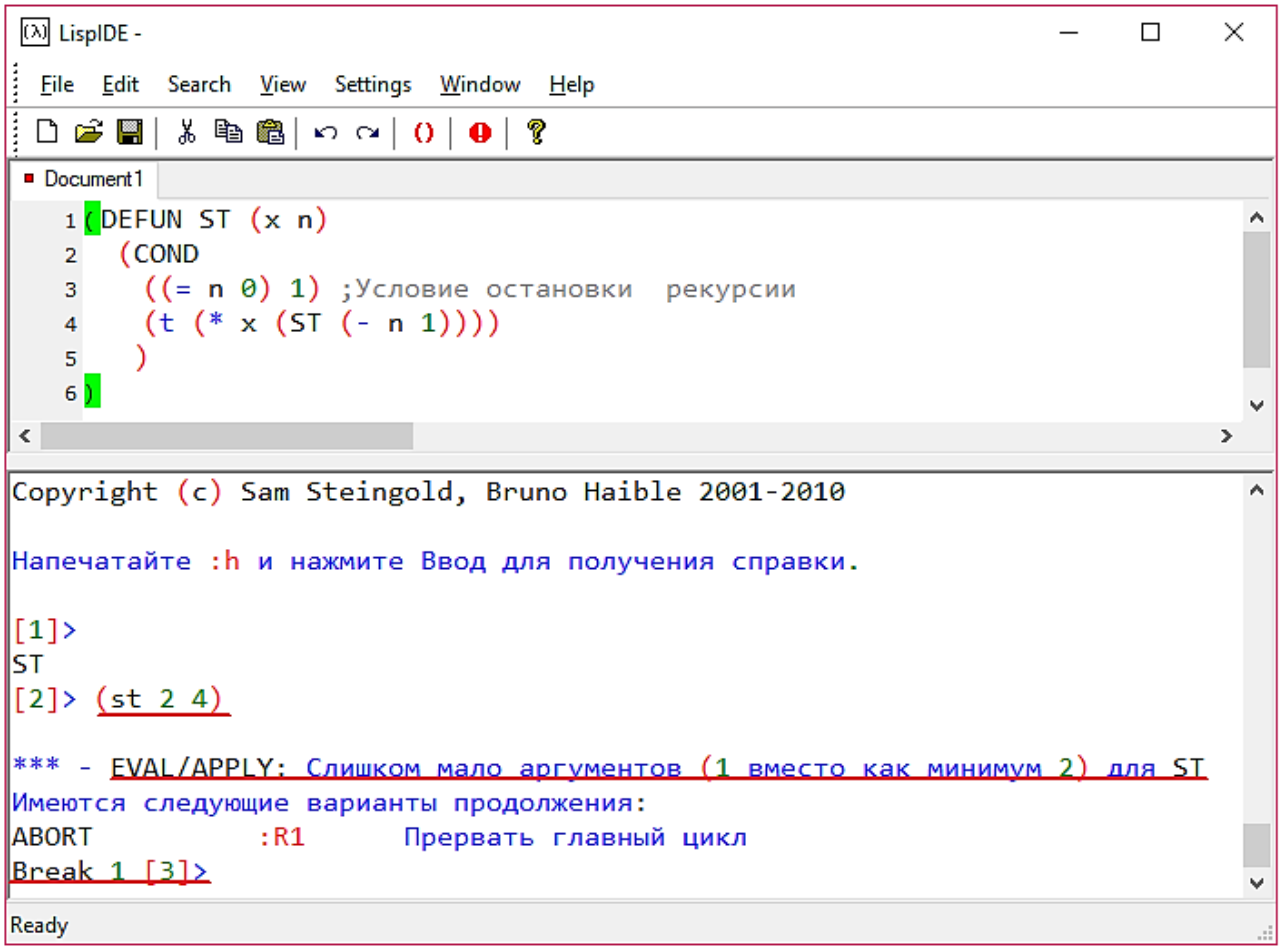


Рис. 23. Вызов функции с ошибкой в определении

Попробуем трассировать нашу функцию, чтобы получить больше информации. Для этого перейдем в главный цикл с помощью команды abort и выполним макрос trace.

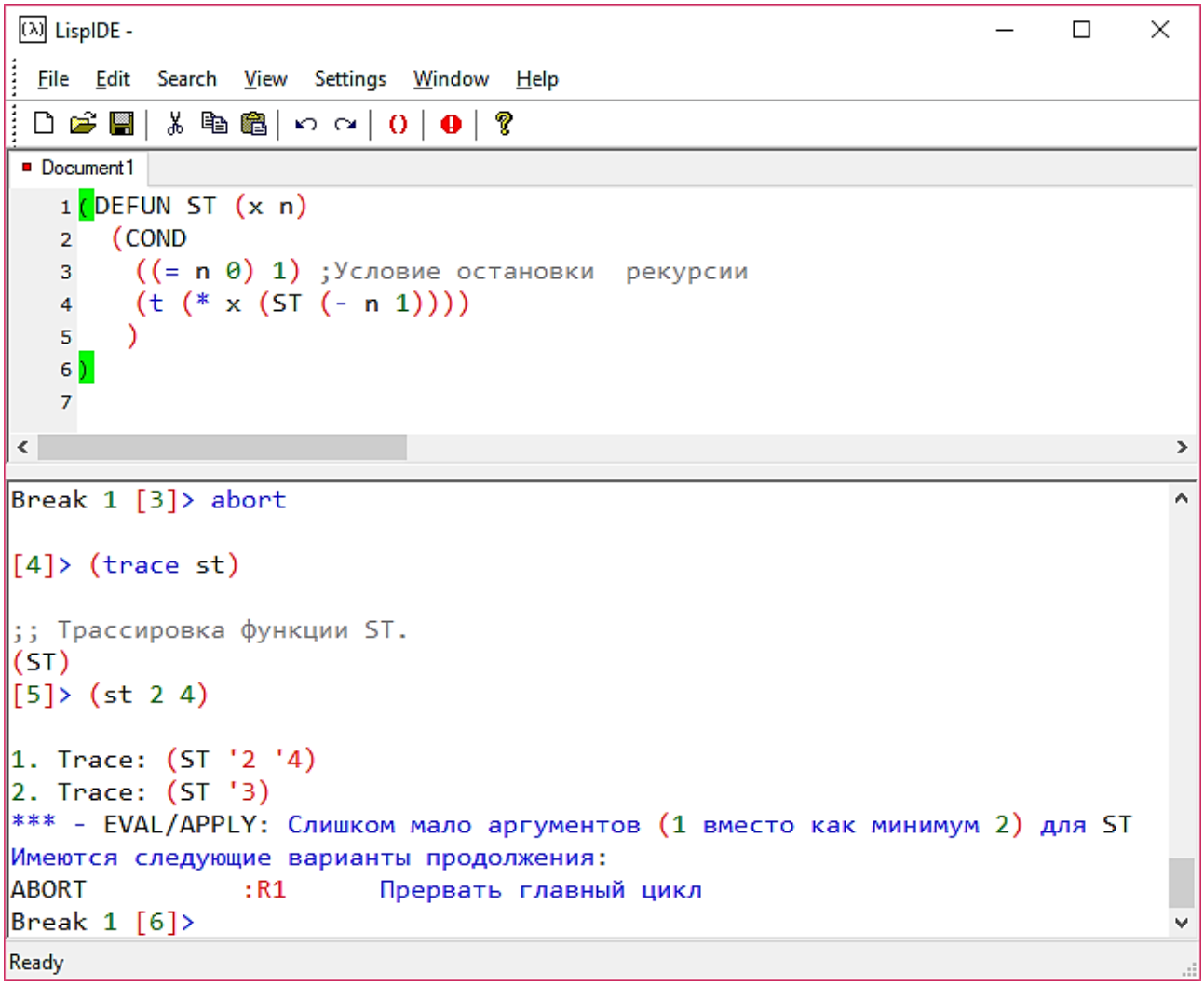


Рис. 24. Результат трассировки функции с ошибкой

Трассировка дает больше информации об источнике ошибки. Видно, что первый вызов прошел успешно, а второй вызов, соответствующий рекурсивному, производится с одним параметром.

Вернемся в главный цикл, исправим ошибки в определении функции, пошлем исправленное определение интерпретатору и снова сделаем вызов (рис. 25). Как видно из рисунка, нет необходимости перезапускать Clisp для переопределения функции, достаточно внести исправления и вновь послать выражение исполняющей среде. Теперь наша функция работает верно.

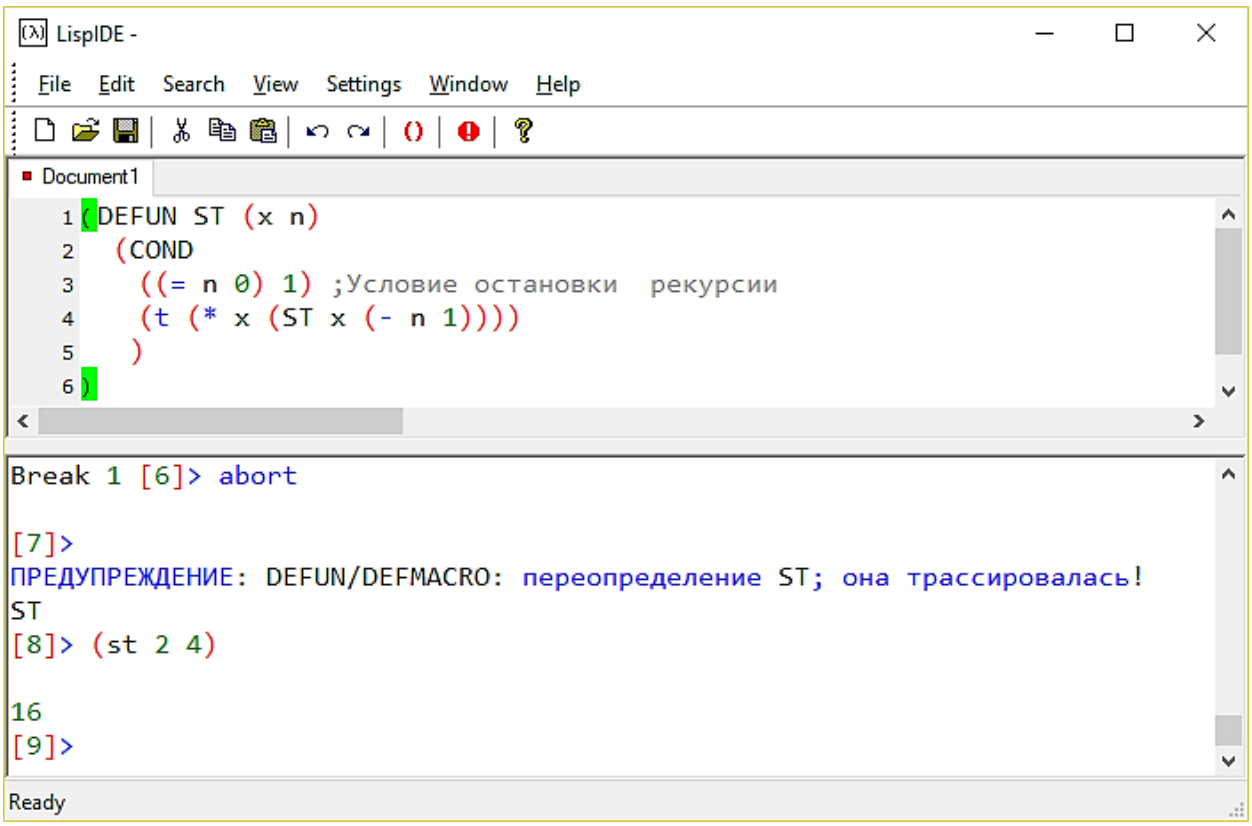


Рис. 25. Результат выполнения функции после исправления ошибки

Посмотрим, как работает вызов функции ST с помощью трассировки (рис. 26).

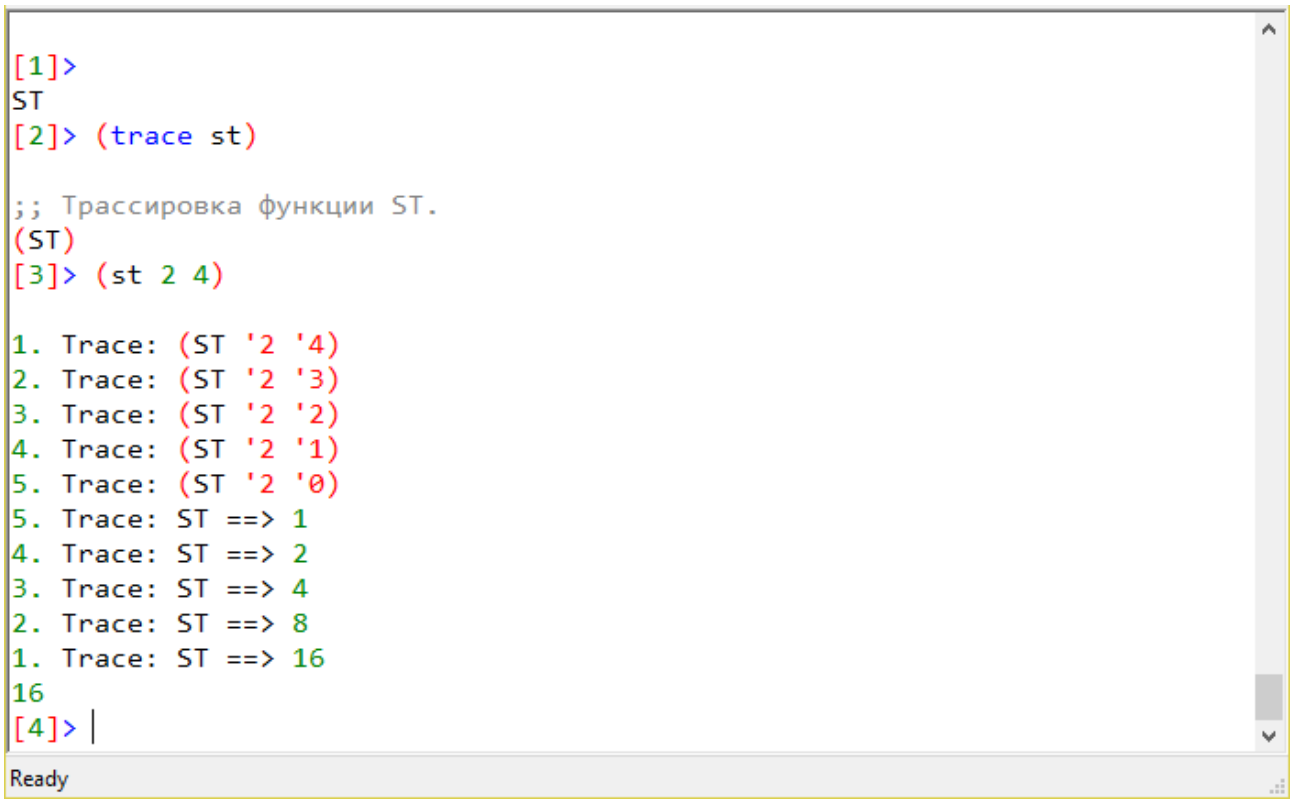


Рис. 26. Трассировка функции

Сначала идет откладывание рекурсивных вызовов в стек прерванных процессов (1 – 4), процесс 5 (останов рекурсии) завершается (возвращает 1), а потом процессы из стека возобновляются от 4 до 1 в обратном порядке попадания в стек.

Пример 2 Определим функцию **COPY**, копирующую список на верхнем уровне (без учета вложенностей). Запишем определение функции словесно:

- скопировать список – это значит соединить голову списка со скопированным хвостом;
- копией пустого списка является пустой список (условие остановки рекурсии).

```
(DEFUN COPY (l)
   (COND
      ((NULL l) l)
      (t  (CONS (CAR l) (COPY (CDR l)) ) )
   )
 )
```

В данном случае имеем рекурсию по аргументу (рекурсивный вызов **COPY** является аргументом функции **CONS**). Внешне функция ничего не делает, но выполнив небольшие изменения, эту функцию можно использовать для различных преобразований списка на верхнем уровне.

Обращение к функции **COPY**: (**COPY** '((1 2) 3 4)) –> ((1 2) 3 4).

При включении трассировки приведенного вызова, на экране отобразится:

```
1. Trace: (COPY '((1 2) 3 4))
2. Trace: (COPY '(3 4))
3. Trace: (COPY '(4))
4. Trace: (COPY 'NIL)
4. Trace: COPY ==> NIL
3. Trace: COPY ==> (4)
2. Trace: COPY ==> (3 4)
1. Trace: COPY ==> ((1 2) 3 4)
```

Пример 3 Определим предикат **ATOM_IN_LIST**, определяющий есть ли среди элементов списка атомы. Запишем определение функции словесно:

- если голова списка является атомом, то предикат возвращает *t*, и дальше список не проверяем (условие остановки рекурсии);
- в противном случае проверяем наличие атомов в хвосте;
- если дошли до конца списка, то атомов нет (условие остановки рекурсии).

```
(DEFUN ATOM_IN_LIST (l)
   (COND
      ((NULL l) nil)
      ((ATOM (CAR l)) t)
      (t (ATOM_IN_LIST (CDR l)))
   )
 )
```

В данном случае имеем рекурсию по значению (рекурсивный вызов **ATOM_IN_LIST** определяет результат функции).

Обращение к функции **ATOM_IN_LIST**:

(**ATOM_IN_LIST** '((1 2) (3))) –> nil

При включении трассировки приведенного вызова, на экране отобразится:

```
1. Trace: (ATOM_IN_LIST '((1 2) (3)))
2. Trace: (ATOM_IN_LIST '((3)))
3. Trace: (ATOM_IN_LIST 'NIL)
3. Trace: ATOM_IN_LIST ==> NIL
2. Trace: ATOM_IN_LIST ==> NIL
1. Trace: ATOM_IN_LIST ==> NIL
```

Пример 4 Определим функцию **MEMBER_S**, проверяющую принадлежность s-выражения списку на верхнем уровне. В случае, если s-выражение принадлежит списку, функция возвращает часть списка, начинающуюся с первого вхождения s-выражения в список. В Clisp имеется аналогичная встроенная функция **MEMBER** (но она использует в своем теле функцию **EQ**, поэтому не работает для вложенных списков)**.** Запишем определение функции **MEMBER_S** словесно:

- если s-выражение совпадает с головой списка, то возвращаем этот список (условие остановки рекурсии);
- в противном случае ищем первое вхождение s-выражения в хвосте списка;
- если дошли до конца списка, то s-выражение не входит в список (условие остановки рекурсии).

```
(DEFUN MEMBER_S (x l)
   (COND
      ((NULL l) l)
      ((EQUAL x (CAR l)) l)
      (t  (MEMBER_S x (CDR l)) )
   )
 )
```

В данном случае имеем рекурсию по значению. Обращения к функции **MEMBER_S**:

```
(MEMBER_S 1 '(2 (3) 1 5)) –> (1 5)
(MEMBER_S 1 '(2 (1) 3 5)) –> nil
(MEMBER_S '(1 2 3) '(6 (8) (1 2 3) 1 5)) –> ((1 2 3) 1 5))
```

Заметим, что встроенная функция **MEMBER** для последнего примера вернула бы *nil*, т.к. в теле встроенной функции для сравнения используется функция **EQ**.

```
(MEMBER '(1 2 3) '(6 (8) (1 2 3) 1 5)) –> nil
```

Пример 5 Определим функцию **REMOVE_S**, удаляющую все вхождения заданного s-выражения в список на верхнем уровне. Запишем определение функции словесно:

- если s-выражение совпало с головой списка, то результат функции – хвост списка с удаленными всеми вхождениями s-выражения;

♦ если s-выражение не совпало с головой списка, то результат функции – соединение головы списка и его хвоста с удаленными всеми вхождениями s-выражения;

♦ если список пуст, то все удалено (условие остановки рекурсии).

```
(DEFUN REMOVE_S  (x l)
    (COND
        ((NULL l)  nil)
        ((EQUAL x (CAR l)) (REMOVE_S x (CDR l)))
        (t  (CONS (CAR l) (REMOVE_S x (CDR l))) )
    )
 )
```

Для этой функции для разных условий предложения **COND** имеем рекурсию как по аргументу (последнее условие), так и по значению (второе условие).

Обращения к функции **REMOVE_S**:

(**REMOVE_S** 1 '(2 1 1 3)) –> (2 3).

Еще одно обращение к функции **REMOVE_S**:

(**REMOVE_S** '(a b) '(1 (a b) ((a b)) 3 (a b))) –> (1 ((a b)) 3).

Заметим, что встроенная функция **REMOVE** вернула бы исходный список, т.к. в теле функции для сравнения используется функция **EQ**.

(**REMOVE** '(a b) '(1 (a b) ((a b)) 3 (a b))) –> (1 (a b) ((a b)) 3 (a b)).

Пример 6 Определим функцию **REVERSE1**, обращающую список на верхнем уровне (в Лиспе имеется аналогичная встроенная функция **REVERSE**). Запишем определение функции словесно:

♦ обратить список – это значит соединить обращенный хвост списка и голову;

♦ если список пуст, то он обращен (условие остановки рекурсии).

```
(DEFUN REVERSE1  (l)
    (COND
        ((NULL l)  nil)
        (t  (APPEND (REVERSE1  (CDR l)) (LIST (CAR l))))
    )
 )
```

Обращения к функции **REVERSE1**:

(**REVERSE1** '(2 1 (5 3))) –> ((5 3) 1 2).

2.11.3 Использование накапливающих параметров

При работе со списками их просматривают слева направо. Но иногда более естественен просмотр справа налево. Например, обращение списка было бы легче осуществить, если бы была возможность просмотра в обратном направлении. Для сохранения промежуточных результатов используют вспомогательные параметры.

Пример 7 При определении функции обращения списка удобно голову списка перекладывать во вспомогательный список, который сначала пуст. Определим **REVERSE2**, обращающую список на верхнем уровне, с

дополнительным параметром для накапливания результата обращения списка. Запишем определение функции **REVERSE2** словесно:

♦ обратить список – это значит обратить хвост, добавив голову исходного списка во вспомогательный список;

♦ если список пуст, то результат функции - вспомогательный список (условие остановки рекурсии).

```
(DEFUN REVERSE2  (l1 &optional l2)
   (COND
      ((NULL l1)  l2)
      (t  (REVERSE2  (CDR l1) (CONS (CAR l1) l2)))
   )
 )
```

Обращение к функции (**REVERSE2** '(1 (2 3) 4 (5 (6)))) с включенной трассировкой:

```
1. Trace: (REVERSE2 '(1 (2 3) 4 (5 (6))))
2. Trace: (REVERSE2 '((2 3) 4 (5 (6))) '(1))
3. Trace: (REVERSE2 '(4 (5 (6))) '((2 3) 1))
4. Trace: (REVERSE2 '((5 (6))) '(4 (2 3) 1))
5. Trace: (REVERSE2 'NIL '((5 (6)) 4 (2 3) 1))
5. Trace: REVERSE2 ==> ((5 (6)) 4 (2 3) 1)
4. Trace: REVERSE2 ==> ((5 (6)) 4 (2 3) 1)
3. Trace: REVERSE2 ==> ((5 (6)) 4 (2 3) 1)
2. Trace: REVERSE2 ==> ((5 (6)) 4 (2 3) 1)
1. Trace: REVERSE2 ==> ((5 (6)) 4 (2 3) 1)
((5 (6)) 4 (2 3) 1)
```

Заметим, что на обратном ходе рекурсии ничего не выполняется.

Пример 8 Определим функцию **COMPARE**, возвращающую *t*, если в числовом списке положительных элементов больше, чем отрицательных. Определим у функции два вспомогательных параметра-счетчика. Запишем определение функции **COMPARE** словесно:

♦ если головой списка является положительный число, то сравниваем количество положительных и отрицательных элементов в хвосте, при этом увеличив на 1 первый счетчик;

♦ если головой списка является отрицательное число, то сравниваем количество положительных и отрицательных элементов в хвосте, увеличив на 1 второй счетчик;

♦ если головой списка является 0, то сравниваем количество положительных и отрицательных элементов в хвосте списка, не изменяя счетчиков;

♦ если список пуст, то сравниваем накопленные значения счетчиков и возвращаем t в случае, если счетчик положительных чисел больше счетчика отрицательных чисел (условие остановки рекурсии).

```
(DEFUN COMPARE  (l &optional (np 0)(no 0))
   (COND
      ((NULL l)  (> np no) )
      ((PLUSP (CAR l)) (COMPARE (CDR l) (+ np 1) no))
      ((MINUSP (CAR l)) (COMPARE  (CDR l) np (+ no 1)))
      (t  (COMPARE  (CDR l) np no))
   )
 )
```

В данном примере имеем рекурсию по значению.

Обращение к функции **COMPARE**: (**COMPARE** ‘(1 -3 0 4 -5 -7)) –> nil.

Пример 9 Определим функцию **POS**, определяющую позицию первого вхождения s-выражения в список (на верхнем уровне) с дополнительным параметром для позиции элемента. Запишем определение функции **POS** словесно:

- ♦ если голова списка совпадает с s-выражением, то результат работы функции – накопленный к этому моменту номер позиции головы (условие остановки рекурсии);
- ♦ если голова списка не совпадает с s-выражением, то определяется позиция первого вхождения s-выражения в хвост списка, при этом значение счетчика позиций увеличивается на 1;
- ♦ если список пуст, то данное s-выражение в списке отсутствует (условие остановки рекурсии).

```
(DEFUN POS (s l  &optional (n 1))
   (COND
      ((NULL l)  l)
      ((EQUAL (CAR l) s) n)
      (t  (POS  s (CDR l) (+ n 1)))
   )
 )
```

В данном примере имеем рекурсию по значению.

Обращение к функции **POS**: (**POS** ‘a ‘(1 -3 a 4 -5 a)) –> 3.

2.11.4 Параллельная рекурсия

Рекурсия называется *параллельной*, если рекурсивный вызов встречается одновременно в нескольких аргументах функции. Такая рекурсия встречается обычно при обработке вложенных списков. В операторном программировании параллельная рекурсия соответствует следующим друг за другом (текстуально) циклам.

Пример 10 Определим функцию **COPY_ALL**, копирующую список на всех уровнях. Запишем определение функции словесно:

- ♦ скопировать список – это значит соединить скопированную голову и скопированный хвост;
- ♦ копия пустого списка – пустой список (условие остановки рекурсии);
- ♦ копия атома – сам атом (условие остановки рекурсии).

```
(DEFUN COPY_ALL  (l)
   (COND
      ((NULL l)  nil )
      ((ATOM l) l)
      (t  (CONS (COPY_ALL (CAR l)) (COPY_ALL (CDR l))))
   )
 )
```

Функция возвращает тот же список, который был ее аргументом. Она разбирает список на атомы, а потом на обратном ходе рекурсии собирает его заново. Функцию можно модифицировать для внесения изменений в исходный список. Параллельность рекурсии не временная, а текстуальная. При выполнении тела функции в глубину идет сначала левый вызов (рекурсия «в глубину»), а потом правый (рекурсия «в ширину»).

Пример 11 Определим функцию **MEMBER_A**, проверяющую принадлежность атома списку на всех уровнях. Функция возвращает *t*, если атом встречается в списке на любом уровне, и *nil* в противном случае. Запишем определение функции словесно:

- ♦ атом может принадлежать либо голове списка, либо хвосту;
- ♦ если аргументом функции является атом, то сравниваются два атома (условие остановки рекурсии);

```
(DEFUN MEMBER_A (x l)
   (COND
      ((ATOM l) (EQUAL x l))
      (t  (OR (MEMBER_A x (CAR l)) (MEMBER_A x (CDR l))))
   )
 )
```

Обращения к функции **MEMBER_A**:

(**MEMBER_A** 1 '((4) ((((1) 2) 3) 4) 5)) –> t;
(**MEMBER_A** nil '(nil)) –> t.

Пример 12 Определим функцию **IN_ONE**, преобразующую список в одноуровневый (удаление вложенных скобок). Запишем определение функции словесно:

- ♦ удалить вложенные скобки в списке – значит соединить два списка: голову исходного списка с удаленными вложенными скобками и хвост с удаленными вложенными скобками;
- ♦ если список пуст, то все вложенные скобки удалены (условие остановки рекурсии);
- ♦ если аргумент функции – атом, то возвращается список из атома.

```
(DEFUN IN_ONE  (l)
   (COND
      ((NULL l)  nil )
      ((ATOM l) (LIST l))
      (t  (APPEND (IN_ONE (CAR l)) (IN_ONE (CDR l))))
```

```
    )
  )
```

Обращение к функции: (**IN_ONE** '((4) ((1) 2) 5)) –> (4 1 2 5).

Пример 13 Определим функцию **MAX_IN_LIST**, находящую максимальный элемент в числовом списке, содержащем подсписки. Запишем определение функции словесно:

- ♦ если список из одного элемента, то ищется максимальный элемент в голове списка;
- ♦ если в списке больше одного элемента, то максимальным элементом в списке является максимальный из двух чисел: максимального элемента в голове списка и максимального элемента в хвосте списка;
- ♦ если аргумент функции – атом, то он и есть максимальный элемент (условие остановки рекурсии).

```
(DEFUN MAX_IN_LIST (l)
    (COND
        ((ATOM l)  l)
        ((NULL (CDR l)) (MAX_IN_LIST (CAR l)))
        (t  (MAX (MAX_IN_LIST (CAR l)) (MAX_IN_LIST (CDR l))))
    )
  )
```

Обращение к функции **MAX_IN_LIST**:
(**MAX_IN_LIST** '((2) (((1 5) 0 4) 8) 3)) –> 8.

Пример 14 Определим функцию **REVERSE_ALL**, обращающую список на всех уровнях. Запишем определение функции словесно:

- ♦ обратить список – значит соединить два списка: обращенный на всех уровнях хвост и обращенную на всех уровнях голову исходного списка;
- ♦ если аргумент функции – атом, то возвращается атом (атом обращать не надо, условие остановки рекурсии);
- ♦ если список из одного элемента, то возвращается список из обращенной головы.

```
(DEFUN REVERSE_ALL (l)
    (COND
        ((ATOM l)  l)
        ((NULL (CDR l)) (LIST (REVERSE_ALL (CAR l))))
        (t (APPEND (REVERSE_ALL (CDR l)) (REVERSE_ALL (LIST (CAR l)
))))
    )
)
```

Обращения к функции **REVERSE_ALL**:
(**REVERSE_ALL** '((a) (((b c) d e) f) r)) –> (r (f (e d (c b))) (a)).

Пример 15 Определим функцию **FIB_1**, вычисляющую n-ый член последовательности Фибоначчи: $f_1=1$, $f_2=1$, $f_n=f_{n-1}+f_{n-2}$.

```
(DEFUN FIB_1 (n)
 (COND
    ((OR (= n 1) (= n 2)) 1)
    (t (+ (FIB_1 (- n 1)) (FIB_1 (- n 2))))
  )
)
```

Недостатком такой рекурсии является то, что происходит дублирование вычислений. При такой организации рекурсии число вызовов растет как $a \cdot b^{n-2}$, где $a \approx 1.8$, $b \approx 1.6$, хотя необходим только n-1 вызов. Напишем другой вариант рекурсивной функции **FIB_2**. Новая функция будет с двумя накапливающими параметрами – соседними членами последовательности, с помощью которых вычисляется следующий член последовательности.

```
(DEFUN FIB_2 (n &optional (a 1)(b 1))
  (COND
    ((OR (= n 1) (= n 2)) b)
    (t (FIB_2 (- n 1) b (+ a b)))
  )
)
```

В этом случае члены последовательности вычисляются от 3-го к n-му. При вычислении n-го члена последовательности Фибоначчи функция вызывается n-1 раз.

Трассировка вызова (**FIB_2** 6):

```
1. Trace: (FIB_2 '6)
2. Trace: (FIB_2 '5 '1 '2)
3. Trace: (FIB_2 '4 '2 '3)
4. Trace: (FIB_2 '3 '3 '5)
5. Trace: (FIB_2 '2 '5 '8)
5. Trace: FIB_2 ==> 8
4. Trace: FIB_2 ==> 8
3. Trace: FIB_2 ==> 8
2. Trace: FIB_2 ==> 8
1. Trace: FIB_2 ==> 8
8
```

2.11.5 Взаимная рекурсия

Рекурсия называется *взаимной между двумя или более функциями*, если они вызывают друг друга.

Пример 16 Определим функцию обращения списка на всех уровнях с использованием взаимной рекурсии. Функция **REVRSE_V** для обращения каждого подсписка использует функцию **REVERSE_P**, которая в свою очередь использует функцию **REVERSE_V**.

```
(DEFUN  REVERSE_V (l)
  (COND
     ((ATOM l) l)
     (t (REVERSE_P l))
```

```
  )
)
(DEFUN REVERSE_P (l1 &optional l2)
  (COND
    ((NULL l1) l2)
    (t (REVERSE_P  (CDR l1) (CONS (REVERSE_V (CAR l1)) l2)))
  )
)
```

Трассировка вызова (**REVERSE_V** '(1 (2 3))) при включенной трассировке функций **REVERSE_V, REVERSE_P**:

```
1. Trace: (REVERSE_V '(1 (2 3)))
2. Trace: (REVERSE_P '(1 (2 3)))
3. Trace: (REVERSE_V '1)
3. Trace: REVERSE_V ==> 1
3. Trace: (REVERSE_P '((2 3)) '(1))
4. Trace: (REVERSE_V '(2 3))
5. Trace: (REVERSE_P '(2 3))
6. Trace: (REVERSE_V '2)
6. Trace: REVERSE_V ==> 2
6. Trace: (REVERSE_P '(3) '(2))
7. Trace: (REVERSE_V '3)
7. Trace: REVERSE_V ==> 3
7. Trace: (REVERSE_P 'NIL '(3 2))
7. Trace: REVERSE_P ==> (3 2)
6. Trace: REVERSE_P ==> (3 2)
5. Trace: REVERSE_P ==> (3 2)
4. Trace: REVERSE_V ==> (3 2)
4. Trace: (REVERSE_P 'NIL '((3 2) 1))
4. Trace: REVERSE_P ==> ((3 2) 1)
3. Trace: REVERSE_P ==> ((3 2) 1)
2. Trace: REVERSE_P ==> ((3 2) 1)
1. Trace: REVERSE_V ==> ((3 2) 1)
((3 2) 1)
```

2.11.6 Вложенные циклы

Многократные повторения, соответствующие вложенным циклам в процедурных языках программирования, в функциональном программировании осуществляются с помощью нескольких функций, каждая из которых соответствует простому циклу. При этом вызов одной функции используется в качестве аргумента рекурсивного вызова другой функции.

Пример 17 Определим функцию **SORT**, сортирующую числовой список по неубыванию методом вставки. Опишем ее работу:

- ♦ отсортировать список – это значит добавить голову списка в нужное место отсортированного хвоста;
- ♦ пустой список упорядочен (условие окончания рекурсии).

Для добавления элемента в отсортированный по неубыванию список определим функцию **ADD**, работающую следующим образом:

- ♦ если добавляемый элемент больше головы списка, то добавляем его в подходящее место хвоста;
- ♦ если добавляемый элемент не больше головы списка, то добавляем элемент перед головой (условие окончания рекурсии);
- ♦ если список пустой, то результат добавления – список из добавляемого элемента (условие окончания рекурсии).

```
(DEFUN  SORT_V (l)
  (COND
     ((NULL l) l)
     (t (ADD (CAR l) (SORT_V (CDR l))))
  )
)
(DEFUN ADD (x l)
  (COND
     ((NULL l) (LIST x))
     ((<= x (CAR l)) (CONS x l))
     (t (CONS (CAR l) (ADD x (CDR l))))
  )
)
```

2.12 Внутреннее представление s-выражений

Каждому атому в программе ставится в соответствие ячейка памяти, называемая информационной ячейкой атома. Сам атом заменяется во внутреннем представлении выражений адресом его информационной ячейки. Через информационную ячейку можно получить доступ к списку свойств атома. Среди прочих свойств в этом списке содержится как внешнее представление этого атома (последовательность символов, представляющая его в программе), так и указатель на его значение. Например, побочным эффектом функции работы **SETQ** является замещение указателя в поле значений символа.

Оперативная память, с которой работает интерпретатор Лиспа, логически разбивается на области – *списочные ячейки*. Списочная ячейка состоит из двух полей, каждое из которых содержит указатель. Указатель может ссылаться на другую списочную ячейку или на объект Лиспа. Графически списочную ячейку можно представить в виде, изображенном на рис. 27.

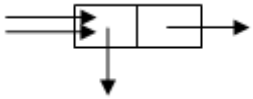


Рис. 27. Списочная ячейка

Список представляется последовательностью списочных ячеек, связанных через указатели в правой части. Правое поле последней списочной ячейки ссылается на пустой список, т.е. атом nil. Графически ссылку на пустой список

изображают в виде перечеркнутого поля. Указатели из левых ячеек ссылаются на структуры, являющиеся элементами списка.

Пример 1 Рассмотрим побочный эффект работы функции (**SETQ** x '(a b c)). Этот эффект – создание штриховой стрелки на рис. 28.

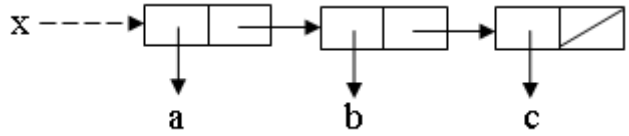


Рис. 28. Побочный эффект функции **SETQ**

В графическом представлении становится понятна работа функций **CAR**, **CDR** и **CONS**. Функция **CAR** возвращает значение левой списочной ячейки (в примере это указатель на атом a), а функция **CDR** – значение правой списочной ячейки (в примере это указатель на список (b c)). Функция **CONS** создает новую списочную ячейку, содержимое левого поля которого – это указатель на первый аргумент функции, а содержимое правого поля – это указатель на второй аргумент функции. Таким образом, новая списочная ячейка связывает две существующие структуры в одну новую структуру. Это довольно эффективное решение с точки зрения создания новых структур и их представления в памяти. Заметим, что применение функции **CONS** не изменяет структур аргументов функции.

Пример 2 Рассмотрим последовательность вызовов:
(**SETQ** x '(b c)) → (b c), побочный эффект – связывание: x → (b c);
(**SETQ** y '(a b c)) → (a b c), побочный эффект – связывание: y → (a b c);
(**SETQ** z (**CONS** x y)) → ((b c) a b c), побочный эффект – связывание: z → ((b c) a b c).

Структура, связанная с переменной z, представлена графически на рис. 30.

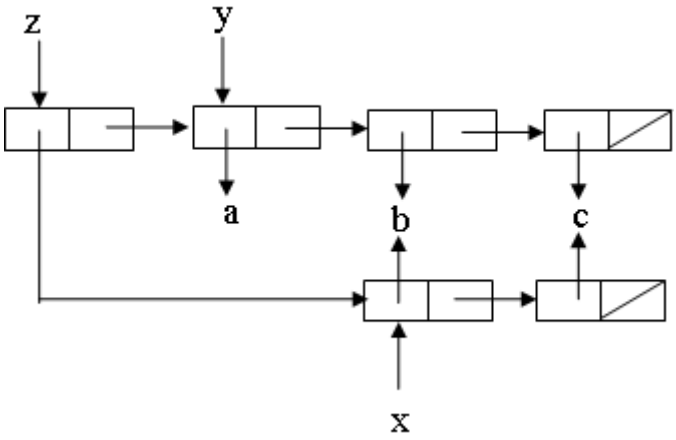


Рис. 30. Созданная структура из примера 2

Из рисунка видно, что логически идентичные атомы содержатся в системе один раз. Однако, логически идентичные списки могут быть представлены различными списочными ячейками. Например, для созданной выше структуры имеем:
(**CAR** z) → (b c);

(**CDDR** z) → (b c).

На рис. 31 графически представлена работа этих функций.

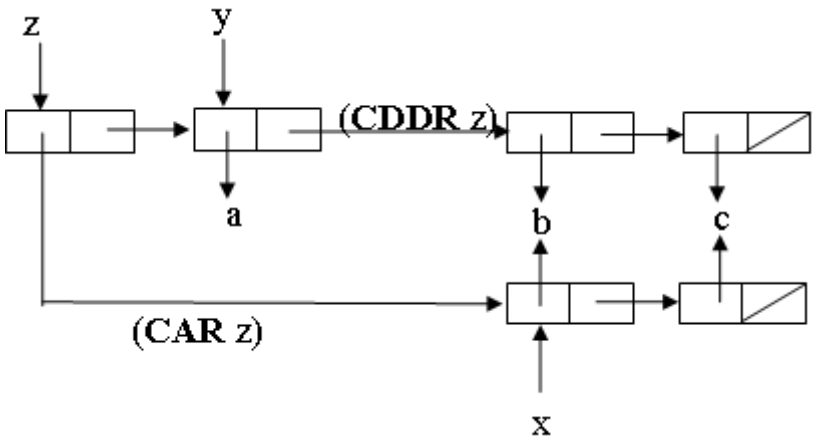


Рис. 31. Работа функций (**CAR** z) и (**CDDR** z)

Полученные списки логически идентичны, а физические адреса для них будут разные, поэтому

(**EQUAL** (**CAR** z) (**CDDR** z)) → t;

(**EQ** (**CAR** z) (**CDDR** z)) → nil.

Список ((b c) a b c) можно было создать другим способом:

(**SETQ** x '(b c)) → (b c), побочный эффект – связывание: x → (b c);

(**SETQ** y (**CONS** 'a x)) → (a b c), побочный эффект – связывание: y → (a b c);

(**SETQ** z1 (**CONS** x y)) → ((b c) a b c), побочный эффект – связывание: z1 → ((b c) a b c).

Структуру, связанную с переменной z1, представлена графически на рис. 32.

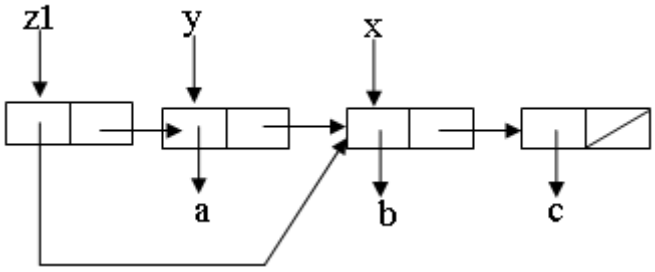


Рис. 32. Созданная структура, связанная с z1

Тогда,

(**EQUAL** (**CAR** z1) (**CDDR** z1)) → t;

(**EQ** (**CAR** z1) (**CDDR** z1)) → t.

В результате вычислений в памяти могут возникнуть структуры, на которые нельзя сослаться. Такие структуры называются *мусором*. Образование мусора происходит в тех случаях, когда вычисленная струтура не сохраняется с помощью **SETQ** или когда теряется ссылка на старое значение в результате побочного эффекта нового вызова **SETQ** или другой функции.

Пример 3 Образование мусора.

1. Представим графически структуру, созданную следующей функцией:
(**SETQ** spis '((a b) c d)) → ((a b) c d), побочный эффект – связывание: spis → ((a b) c d).

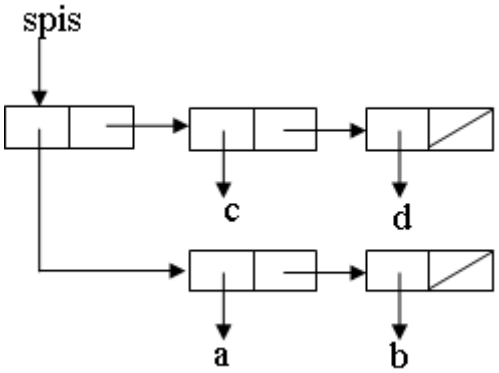


Рис. 33. Созданная структура spis

После присваивания нового значения переменной spis побочным эффектом функции **SETQ**
(**SETQ** spis (**CDR** spis)) → (c d), побочный эффект – связывание: spis → (c d), уже нельзя будет «добраться» до двух списочных ячеек. Они стали мусором.

2. Значение вызова (**CONS** 'a (**LIST** 'b)) лишь выводится на экран дисплея, после чего созданная им в памяти структура станет мусором.

Для повторного использования ставшей мусором памяти в Лиспе предусмотрен специальный сборщик мусора, который автоматически запускается, когда в памяти остается мало свободного места. Сборщик мусора перебирает все ячейки и собирает являющиеся мусором ячейки в список свободной памяти для того, чтобы их можно было использовать заново.

Все рассмотренные до сих пор функции манипулировали выражениями, не вызывая каких-либо изменений в уже существующих структурах. Значения переменных можно было изменить лишь целиком. Единственное, что могло произойти со старым значением – это лишь то, что оно могло пропасть. В Лиспе имеются специальные функции, которые изменяют внутреннюю структуру списков. Такие функции называются *структуроразрушающими*. Например, рассмотрим, как с помощью структуроразрушающей функции можно повысить эффективность функции **APPEND**.

Пример 4 Рассмотрим последовательность вызовов:
(**SETQ** x '(a b)) → (a b), побочный эффект – связывание: x → (a b);
(**SETQ** y '(c d)) → (c d), побочный эффект – связывание: y → (c d);
(**SETQ** z (**APPEND** x y)) → (a b c d), побочный эффект – связывание: z → (a b c d).

Может показаться, что приведенный вызов функции **APPEND** изменяет указатели так, как это указано штриховыми стрелками на рис.34.

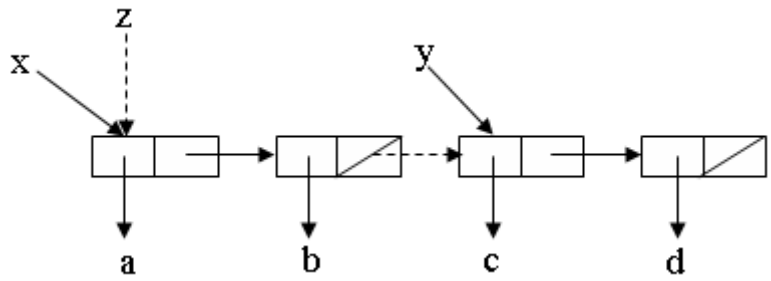


Рис. 34. Неверное представление структуры z

Но это не верно, так как значение переменной x не может измениться, поскольку **APPEND** не является структуроразрушающей функцией. Вычисляется новое значение, которое присваивается переменной z. На самом деле **APPEND** создает копию списка – первого аргумента (рис. 35).

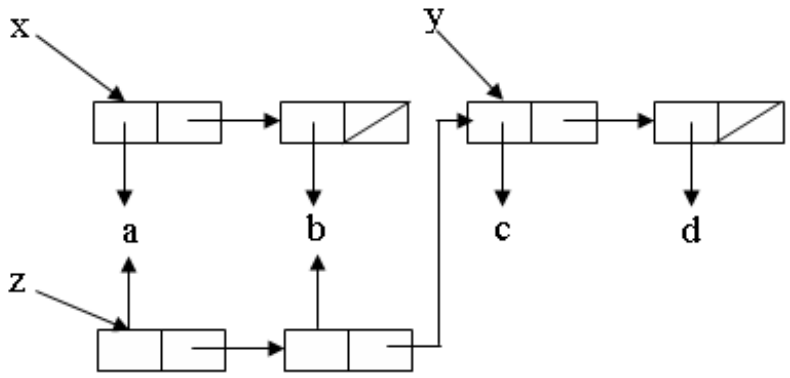
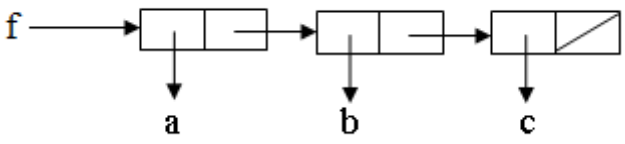


Рис. 35. Верное представление структуры z

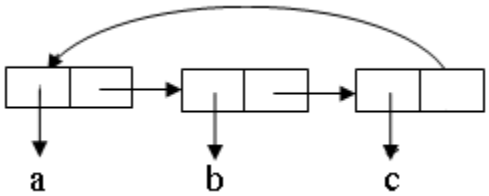
Очевидно. что если первый аргумент функции **APPEND** является списком из 1000 элементов, а второй – списком из одного элемента, то будет создано 1000 новых ячеек, хотя нужно добавить всего лишь один элемент к 1000 имеющимся. Если нам не важно, что значение переменной x может измениться, то можно использовать соединение списков как на рисунке 7 с помощью структуроразрушающей функции **NCONC**. Эта функция просто изменяет указатель в поле **CDR** последней ячейки списка – первого аргумента на начало списка – второго аргумента. **NCONC** может создавать циклические списки.

Пример 5

(**SETQ** f '(a b c))–> (a b c)



(**NCONC** f f) Здесь будет переполнение стека, т.к. будет строиться список вида (a b c a b c…..…).



Рассмотрим две функции, которые изменяют структуру своих аргументов: **RPLACA** (replace the car) и **RPLACD** (replace the cdr).

RPLACA (список s-выражение)

Функция **RPLACA** заменяет указатель на голову списка на значение s-выражения. **RPLACA** возвращает измененный список.

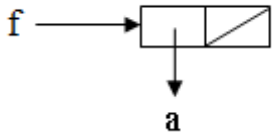
Пример 6

(**SETQ** f '((a) b c))–> ((a) b c)
(**RPLACA** f 'z)–>(z b c)

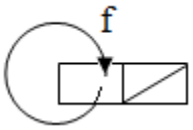
Произошла замена головы списка. Кажется, что такого эффекта можно было бы достичь с помощью функции **CONS**.
Например, (**SETQ f (CONS** 'z (**CDR** f))). Но вызов с использованием **CONS** строит новую списочную структуру, а **RPLACA** модифицирует существующую. Функция **RPLACA** обладает побочным эффектом - после ее вызова могут измениться значения сразу многих выражений, которые ссылаются на модифицированную ячейку. Но главное коварство функции **RPLACA** заключается в том, что с ее помощью можно создавать циклические списочные структуры.

Пример 7

(**SETQ** f '(a))–> (a)



(**RPLACA** f f) Здесь будет переполнение стека, т.к. будет строиться список вида (((((((.....))))))) .



RPLACD (список s-выражение)

Функция **RPLACD** заменяет указатель на хвост списка на значение s-выражения. **RPLACD** возвращает измененный список или точечную пару (если s-выражение не являлось списком).

Пример 8

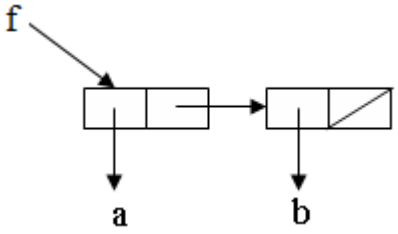
(**SETQ** f '((a) b c))–> ((a) b c)
(**RPLACD** f '(z))–>((a) z)

Произошла замена хвоста списка. Кажется, что такого эффекта можно было бы достичь с помощью функции **CONS**.

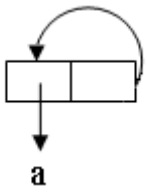
Например, (**SETQ f (CONS** (**CAR** f) '(z))). Но вызов с использованием **CONS** опять строит новую списочную структуру, а **RPLACD** модифицирует существующую. С использованием **RPLACD** так же можно создавать циклические списки.

Пример 9

(**SETQ** f '(a b))–> (a b)



(**RPLACD** f f) Здесь будет переполнение стека, т.к. будет строиться список вида `(a a …… a)`.



В чисто функциональном программировании такие функции не используются. В практическом программировании функции, изменяющие структуры, иногда используют. Например, если нужно сделать небольшие изменения в большой структуре данных. С помощью структуроразрушающих функций можно эффективно менять за один раз несколько структур, если у этих структур есть совмещенные в памяти подструктуры. Но такие изменения могут привести к сюрпризам в тех частях программы, которые не знают об изменениях, что осложняет написание программ и их документирование. Обычно использование структуроразрушающих функций стараются избегать.

2.13 Точечная пара

При определении функции **CONS**, предполагалось, что ее второй аргумент должен быть списком. На самом деле, если второй аргумент атом, результатом работы функции будет так называемая *точечная пара*, левым элементом которой является первый аргумент функции, а правым – второй аргумент.

Пример 1 (**CONS** 'a 'b) → (a .b). Графически созданная структура представлена на рис. 36.

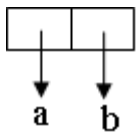


Рис. 36. Точечная пара

Тогда функция **CAR** от точечной пары будет возвращать выражение, стоящее слева от точки, а поле **CDR** – выражение справа от точки. Точечная нотация позволяет расширить класс объектов Лиспа. Ситуацию можно сравнить с добавлением класса комплексных чисел к имеющемуся классу действительных чисел в математическом анализе.

Любой список можно представить в точечной нотации. Преобразования можно осуществить следующим образом: $(a_1\ a_2\ \ldots a_n) \Leftrightarrow (a_1\ .\ (a_2\ .\ \ldots\ (a_n\ .\ nil)\ \ldots\ ))$. Таким образом, каждый пробел заменяется точкой, за которой ставится открывающаяся скобка. Соответствующая закрывающаяся скобка ставится непосредственно перед ближайшей справа от этого пробела закрывающейся скобкой, не имеющей парной открывающей скобки также справа от пробела. После каждого последнего элемента списка добавляется *nil*.

Пример 2 (a b (c d) e) ⇔ (a . (b . ((c .(d . nil)) . (e . nil)))).

Интерпретатор может привести записанное в точечной нотации выражение частично или полностью к списочной нотации. Полностью преобразование

можно осуществить только тогда, когда правый элемент каждой точечной пары является списком или точечной парой. Переход к списочной записи осуществляется по следующему правилу: если точка стоит перед открывающейся скобкой, то она заменяется пробелом и одновременно убирается соответствующая закрывающаяся скобка. Это же правило позволяет избавиться и от лишних *nil*, если помнить, что *nil* эквивалентен ().

Пример 3

1. (a . ((b . nil) . (c . nil))) ⇔ (a (b) c);
2. (a . (b . c)) ⇔ (a b . c);
3. ((a . b) . (c . d)) ⇔ ((a .b) c . d);
4. (nil . (a . b)) ⇔ (nil a . b).

Точечная запись списков обладает некоторыми преимуществами по сравнению со списочной записью. Она является более общей, т.к. любой список можно переписать в точечных обозначениях, но уже простейшее точечное выражение (a . b) не может быть представлено списочной записью. Использование точечных пар позволяет сократить объем необходимой памяти. Например, структура данных (a b c), записанная в виде списка требует трех ячеек, тогда как представление тех же данных в виде (a b . c) требует только двух ячеек.

Более компактное представление может сократить и объем вычислений за счет меньшего количества обращений к памяти. Когда выражение строится из небольшого заранее известного количества элементов, точечные конструкции предпочтительнее списков. Если же число элементов велико и может изменяться, то целесообразнее использовать списочные конструкции. Точечные пары применяются в теории, книгах и справочниках по Лиспу, системном программировании. Большинство программистов не используют точечные пары, поскольку по сравнению с требуемой в таком случае внимательностью получаемый выигрыш в объеме памяти и скорости вычислений не заметен.

2.14 Функционалы

До сих пор во всех примерах аргументами функций были данные, представленные s-выражениями. Однако в Лиспе функции могут выступать в качестве аргументов, точнее аргументом функции может быть определяющее функцию лямбда-выражение или имя другой функции. Такой аргумент называется *функциональным*, а функция, имеющая функциональный аргумент, называется *функционалом*.

2.14.1 Аппликативные (применяющие) функционалы

Применяющим функционалом называется функционал, который применяет функциональный аргумент к остальным параметрам. Применяющие функционалы похожи на функцию **EVAL**, которая вычисляет значение произвольной формы. Применяющий функционал вычисляет значение вызова функционального аргумента.

В Clisp имеются два встроенных применяющих функционала: **APPLY** и **FUNCALL**.

Функционал **APPLY** является функцией двух аргументов: функционального аргумента и списка данных. Этот функционал вычисляет значение функционального аргумента (функции от n переменных) для фактических параметров, которые являются элементами списка. Вызов функционала имеет вид:

(**APPLY** *fn sp*),

где *fn* – имя функции или лямбда-выражение, *sp* – список.

Пример 1

1. (**APPLY** '**+** '(1 2 3)) →(**+** 1 2 3) →6;
2. (**APPLY** (**LAMBDA** (x y z) (+ x y z)) '(1 2 3)) →(**+** 1 2 3) →6;
3. (**APPLY** '**CONS** '(a (b c))) →(**CONS** 'a '(b c)) →(a b c).

При использовании функционала наблюдается большая гибкость, чем при вызове обычной функции. В зависимости от значения функционального аргумента можно осуществлять различные вычисления над одними и теми же данными.

Пример 2 Напишем функциональный предикат **ALL**, который возвращает *t* в том и только в том случае, если функциональный аргумент истинен для каждого элемента списка.

```
(DEFUN ALL (p l)
  (COND
     ((NULL l) t)
     ((NULL (CDR l)) (APPLY p (LIST (CAR l))))
     ((APPLY p (LIST (CAR l))) (ALL p (CDR l)))
     (t nil)
  )
)
```

Рассмотрим примеры работы функционала **ALL**.

(**ALL** 'atom '((1 2) 3 4)) → nil;
(**ALL** 'atom '(1 2 3 4)) → t;
(**ALL** 'plusp '(1 2 3 4)) → t;
(**ALL** (**LAMBDA** (x) (**<=** x 0)) '(-1 -3 -4 0)) → t.

Функционал **FUNCALL** является функцией не менее двух аргументов: функционального аргумента и данных. Этот функционал работает аналогично **APPLY**, но аргументы функционального аргумента (функции от n переменных) задаются не списком, а как аргументы **FUNCALL**, начиная со второго. Вызов функционала имеет вид:

(**FUNCALL** *fn* a_1 a_2 … a_n),

где *fn* – имя функции или лямбда-выражение, a_i (i = 1, … ,n)– значение i-го аргумента для *fn*.

Пример 3

1. (**FUNCALL** '**+** 1 2 3) →(**+** 1 2 3) →6;
2. (**FUNCALL** '(**LAMBDA** (x y) (+ x y)) 1 2) →(**+** 1 2) →6;

3. (**FUNCALL** '**LIST** '(a b)) →(**LIST** '(a b)) →((a b)).

Заметим, что если в последнем примере заменить **FUNCALL** на **APPLY**, то получим: (**APPLY** '**LIST** '(a b)) →(**LIST** a b) →(a b).

Пример 4 Напишем функцию сортировки числового списка в виде функционала, у которого функциональный аргумент будет задавать порядок сортировки.

```
(DEFUN  SORT (l p)
  (COND
     ((NULL l) l)
     (t (ADD_ORD (CAR l) (SORT (CDR l) p) p))
  )
)
(DEFUN ADD_ORD (x l p)
  (COND
     ((NULL l) (LIST x))
     ((FUNCALL p x (CAR l)) (CONS x l))
     (t (CONS (CAR l) (ADD_ORD x (CDR l) p)))
  )
)
```

Рассмотрим примеры работы функционала **SORT**:

(**SORT** '(5 1 3 2 4) '<) → (1 2 3 4 5);
(**SORT** '(5 1 3 1 2 4 2) (**LAMBDA** (x y) (>= x y))) → (5 4 3 2 2 1 1);
(**SORT** '(ab sc aefg srt) 'string>) → (srt sc aefd ab).

2.14.2 Отображающие функционалы или MAP-функции

Отображающие функционалы с помощью функционального аргумента преобразуют список в новый список или порождают побочный эффект, связанный с этим списком. Такие функционалы начинаются на MAP.

В Clisp имеются встроенные отображающие функционалы **MAPCAR** и **MAPLIST**.

Функционал **MAPCAR** является функцией не менее двух аргументов: функционального аргумента и списков данных. Этот функционал возвращает список, состоящий из результатов последовательного применения функционального аргумента (функции *n* переменных) к соответствующим элементам n списков. Число аргументов-списков должно быть равно числу аргументов функционального аргумента. Вызов функционала имеет вид:

(**MAPCAR** *fn* sp_1 sp_2 … sp_n),

где *fn* – имя функции или лямбда-выражение, sp_i – список (i = 1, …, n).

Пример 1

1. (**MAPCAR** 'atom '(a b c)) → ((**ATOM** a) (**ATOM** b) (**ATOM** c)) → (t t t);
2. (**MAPCAR** '(**LAMBDA** (y) (**LIST** y)) '(a b c)) →
   → ((**LIST** a) (**LIST** b) (**LIST** c)) → ((a) (b) (c));
3. (**DEFUN PARA**(x) (**LIST** x '*))→ para;
   (**MAPCAR** 'para '(a b c)) → ((a *) (b *) (c *));

4. (**MAPCAR** '+ '(1 2) '(2 3) '(3 4)) → (6 9);
5. (**MAPCAR** 'cons '(a b c) '((1) (2) (3))) → ((a 1) (b 2) (c 3)).

Как правило, MAP-функция применяется к одному аргументу-списку, т.е. функциональный аргумент является функцией одной переменной. Использование MAP-функций используется при программировании специальных циклов, поскольку с их помощью можно сократить запись повторяющихся вычислений.

Пример 2 Напишем функцию **SUM3**, вычисляющую сумму кубов элементов списка.

```
(DEFUN SUM3 (l)
 (EVAL (CONS '+ (MAPCAR '* l l))))
```

Рассмотрим пример работы функции **SUM3**: (**SUM3** '(1 2 3)) → 36.

Пример 3 Напишем функцию **DECART**, вычисляющую декартово произведение двух множеств, через композицию двух вложенных вызовов функционала **MAPCAR**.

```
(DEFUN DECART (l1 l2)
 (MAPCAR (LAMBDA (x) (MAPCAR (LAMBDA (y) (LIST x y)) l2)) l1))
```

Рассмотрим пример работы функции **DECART**:

(**DECART** '(1 2 3) '(a b)) → (((1 a) (1 b)) ((2 a) (2 b)) ((3 a) (3 b))).

Отображающий функционал **MAPLIST** действует подобно **MAPCAR**, но действия осуществляются не над элементами списков, а над последовательными хвостами этих списков, начиная с самих списков. Вызов функционала имеет вид:

(**MAPLIST** *fn* sp_1 sp_2 … sp_n),

где *fn* – имя функции или лямбда-выражение, sp_i – список (i = 1 ,…, n).

Пример 4

1. (**MAPLIST** 'reverse '(a b c)) → ((c b a) (c b) (c));
2. (**MAPLIST** '(**LAMBDA** (x) x) '(a b c)) → ((a b c) (b c) (c));
3. (**MAPLIST** 'append '(a b) '(1 2)) → ((a b 1 2) (b 2)).

Среди отображающих функционалов выделяют объединяющие функционалы **MAPCAN** и **MAPCON**. Работа их аналогична соответственно **MAPCAR** и **MAPLIST**. Различие заключается в способе построения результирующего списка. Если функционалы **MAPCAR** и **MAPLIST** строят новый список из результатов применения функционального аргумента с помощью функции **LIST**, то функционалы **MAPCAN** и **MAPCON** для построения нового списка используют структуроразрушающую псевдофункцию **NCONC**, которая делает на внешнем уровне то же самое, что и функция **APPEND.** Функционалы **MAPCAN** и **MAPCON** удобно использовать в качестве фильтров для удаления нежелательных элементов из списка.

Вызов функционала **MAPCAN** имеет вид:

(**MAPCAN** *fn* sp_1 sp_2 … sp_n),

где *fn* – имя функции или лямбда-выражение, sp_i – список (i = 1 ,…, n).

Пример 5 Удаление из числового списка всех элементов, кроме отрицательных.

```
(MAPCAN (LAMBDA (x)
              (COND
                 ((MINUSP x) (LIST x))
                  (t nil)
               )) '(-3 1 4 -5 0)) → (-3 -5)
```

Заметим, что использование **MAPCAR** не привело бы к такому результату, т.к. (**APPEND** '(1) nil) → (1), а (**LIST** 1 nil) → (1 nil).

Вызов функционала **MAPCON** имеет вид:

(**MAPCON** *fn* sp_1 sp_2 … sp_n),

где *fn* – имя функции или лямбда-выражение, sp_i – список (i = 1 ,…, n).

Пример 6

1. (**MAPCON** 'reverse '(1 2 3)) → (3 2 1 3 2 3).

2. Преобразование одноуровнего списка в множество:

```
(MAPCON (LAMBDA (l)
              (COND
                 ((MEMBER (CAR l) (CDR l)) nil)
                  (t (LIST (CAR l)))
               )) '(1 2 3 1 1 4 2 3)) → (1 4 2 3)
```

Отметим, что функционал **MAPCON** может легко «зацикливать» списки, поэтому использовать его нужно осторожно.

3. Логическое программирование. Основы языка Пролог

Логическое программирование базируется на убеждении, что не человека следует обучать мышлению в терминах операций компьютера, а компьютер должен выполнять инструкции, свойственные человеку. В чистом виде логическое программирование предполагает, что инструкции даже не задаются, а сведения о задаче формулируются в виде логических аксиом. Такое множество аксиом является альтернативой обычной программе. Подобная программа может выполняться при постановке задачи, формализованной в виде логического утверждения, подлежащего доказательству (*целевого утверждения*).

Идея использования логики исчисления предикатов I порядка в качестве основы языка программирования возникла в 60-е годы, когда создавались многочисленные системы автоматического доказательства теорем и вопросно-ответные системы. В 1965 г. Робинсон предложил принцип резолюции, который в настоящее время лежит в основе большинства систем поиска логического вывода. Метод резолюций был использован в системе GPS (general problem solver). В нашей стране была разработана система ПРИЗ, которая может доказать любую теорему из школьного учебника геометрии.

Язык программирования PROLOG (programming in logic) был разработан и впервые реализован в 1972 г. группой сотрудников Марсельского университета во главе с Колмероэ. Группа занималась проблемой автоматического перевода с

одного языка на другой. Основа этого языка - исчисления предикатов I порядка и метод резолюций.

При программировании на Прологе усилия программиста должны быть направлены на описание логической модели фрагмента предметной области решаемой задачи в терминах объектов предметной области, их свойств и отношений между собой, а не деталей программной реализации. Фактически Пролог представляет собой не столько язык для программирования, сколько язык для описания данных и логики их обработки. Программа на Прологе не является таковой в классическом понимании, поскольку не содержит явных управляющих конструкций типа условных операторов, операторов цикла и т. д. Она представляет собой модель фрагмента предметной области, о котором идет речь в задаче. И решение задачи записывается не в терминах компьютера, а в терминах предметной области решаемой задачи, в духе модного сейчас объектно-ориентированного программирования.

Суть Пролога – программирование в терминах целей. Программист описывает условие задачи, пользуясь понятиями объектов различных типов и отношений между ними, и формулирует вопрос. Пролог-система обеспечивает ответ на вопрос, находя автоматически последовательность вычисления решения, используя встроенную процедуру поиска. До 1981 г. число исследователей, занимавшихся логическим программированием, составляло около сотни во всем мире. В 1981 году Prolog был выбран в качестве базового языка компьютеров пятого поколения, и количество исследователей логического программирования резко возросло. Одной из наиболее интересных тем исследований является связь логического программирования с параллелизмом.

Где же используется Пролог в настоящее время? Это область автоматического доказательства теорем, построение экспертных систем, машинные игры с эвристиками (например, шахматы), автоматический перевод с одного языка на другой.

В настоящее время создано достаточно много реализаций языка Пролог: Wisdom Prolog, SWI Prolog, Turbo Prolog, Visual Prolog, Arity Prolog и т.д.

В нашем курсе будем использовать SWI Prolog. SWI-Prolog развивается с 1987 года. Его создателем и основным разработчиком является Ян Вьелемакер (Jan Wielemaker). Название SWI происходит от Sociaal-Wetenschappelijke Informatica (гол. социально-научная информатика), первоначального названия группы в Амстердамском университете, где работает Вьелемакер.

SWI-Prolog позволяет разрабатывать приложения любой направленности, включая Web-приложения и параллельные вычисления, но основным направлением использования является разработка экспертных систем, программ обработки естественного языка, обучающих программ, интеллектуальных игр и т.п. Это интерпретатор. Файлы, содержащие программы, написанные на языке SWI Prolog, имеют расширение pl.

3.1 Факты и правила

Как уже отмечалось Пролог использует исчисление предикатов первого порядка. Предикаты определяют отношения между объектами.

Пример 1 Рассмотрим дерево родственных отношений, изображенное на рис. 37.

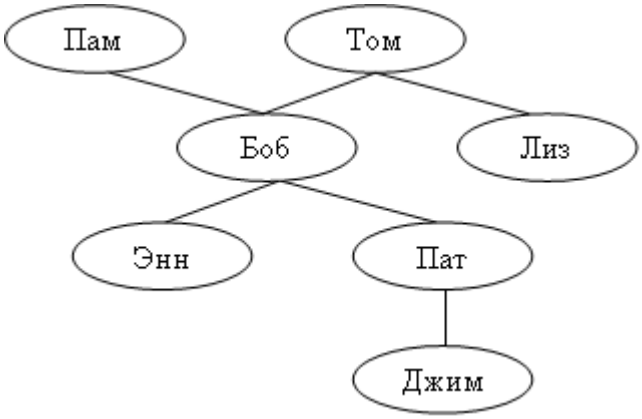


Рис. 37. Дерево родственных отношений

Это дерево можно описать следующей Пролог-программой (рис. 38).

```
родитель(пам, боб).
родитель(том, боб).
родитель(том, лиз).
родитель(боб, энн).
родитель(боб, пат).
родитель(пат, джим).
```

У нас имеется отношение родитель между двумя объектами. Описаны 6 фактов наличия отношений между конкретными объектами. Имена объектов начинаются с маленьких букв (они являются константами). В Прологе принято соглашение, что константы начинаются с маленькой буквы, а переменные – с большой.

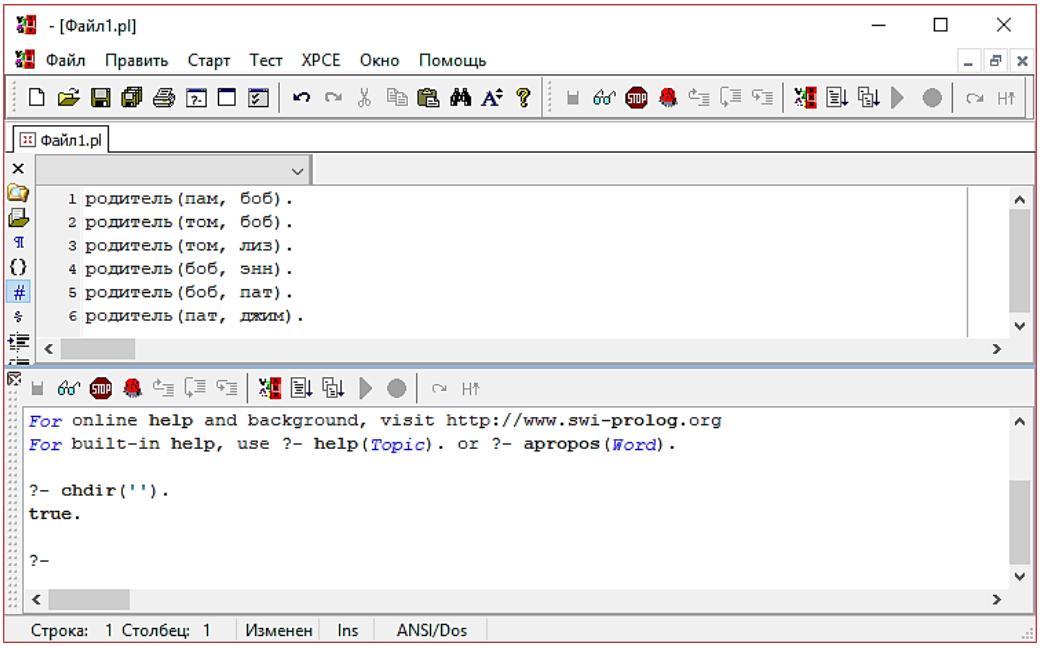


Рис. 38. Программа на SWI-Prolog

После набора такой Пролог-программы в SWI-Prolog-Edit можно загрузить программу в интерпретатор Пролога (нажав клавишу F9 или пиктограмму на панели инструментов). Если программа загружается первый раз, то будет

предложено сохранить ее. После успешной загрузки программы в интерпретатор SWI-Prolog в командной строке появится сообщение (может быть небольшая задержка перед появлением сообщения).
?- consult(‘<имя файла>’).
true.
Слово true свидетельствует об успешной загрузке (рис. 39).

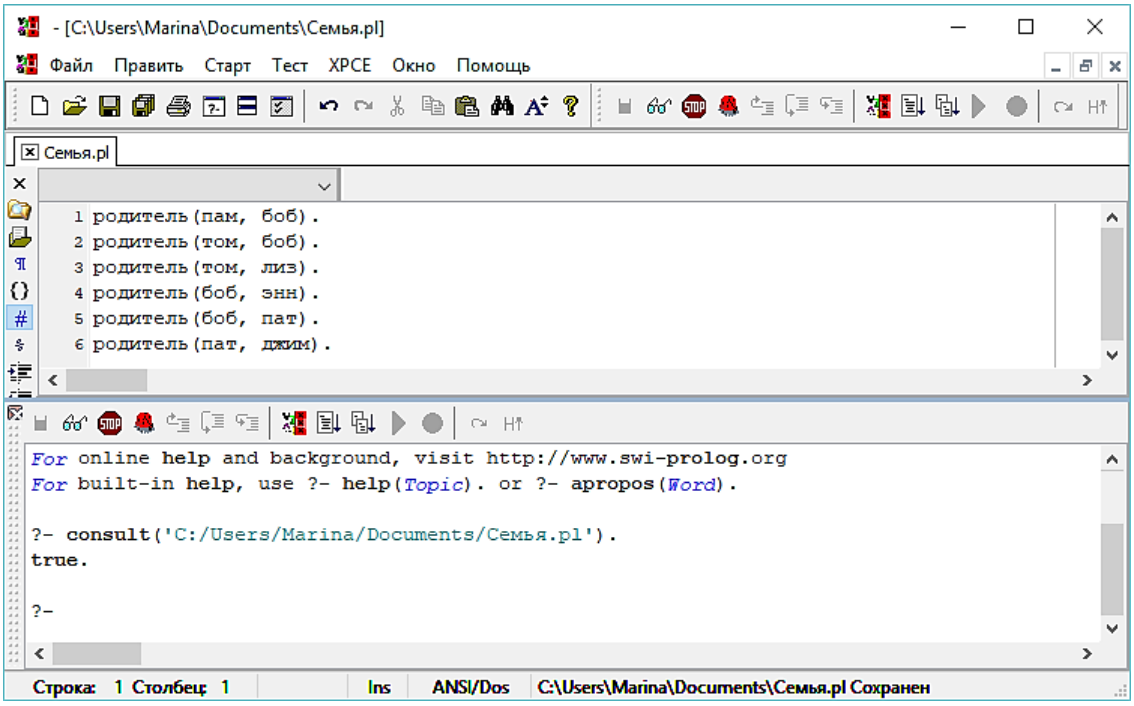


Рис. 39. Загрузка программы в интерпретатор SWI-Prolog

Замечание: Если в полном пути к сохраненному файлу есть названия на кириллице, то загрузка будет неуспешной. В этом случае следует переключиться на русскую раскладку и повторить загрузку еще раз.

Теперь можно задавать вопросы, касающиеся отношения родитель. Запрос к программе набирается после приглашения «?-» и должен заканчиваться точкой. Для выполнения набранного запроса необходимо нажать клавишу Enter. Ответ будет выдан под запросом. Запрос может быть набран в несколько строк - для перехода на новую строку используется нажатие клавиши Enter. В том случае, если строка будет заканчиваться точкой и будет нажата клавиша Enter, SWI-Prolog начнет выполнение запроса. Если возможны несколько вариантов ответа на запрос, то для получения каждого следующего варианта используется ввод знака «;» или нажатие клавиши Enter. Прекратить выполнение программы (выдачу альтернативных ответов) можно нажав клавишу «a».

Замечание: Клавиша Enter во встроенном редакторе SWI-Prolog используется для прекращения поиска альтернативных ответов.

Вопросы могут быть простые и сложные (в качестве связки «и» при составлении сложного вопроса используется запятая). Ответы Пролог-системы выводятся сразу после вопроса. Могут быть следующие варианты ответов:

- true (да);
- false (соответствует нет или не найдены значения переменных в вопросе);

- Перечисляются возможные значения переменных в вопросе. Если решение единственно, то после решения выводится точка и выводится приглашение интерпретатора «?-». Если решение не единственно, то Пролог ожидает дальнейших указаний по поиску решений. При нажатии клавиши «а» поиск решений прекращается, при нажатии Enter или вводе ; – продолжается.

На рис. 40 – 44 приведены различные варианты ответов SWI-Prolog на заданные вопросы.

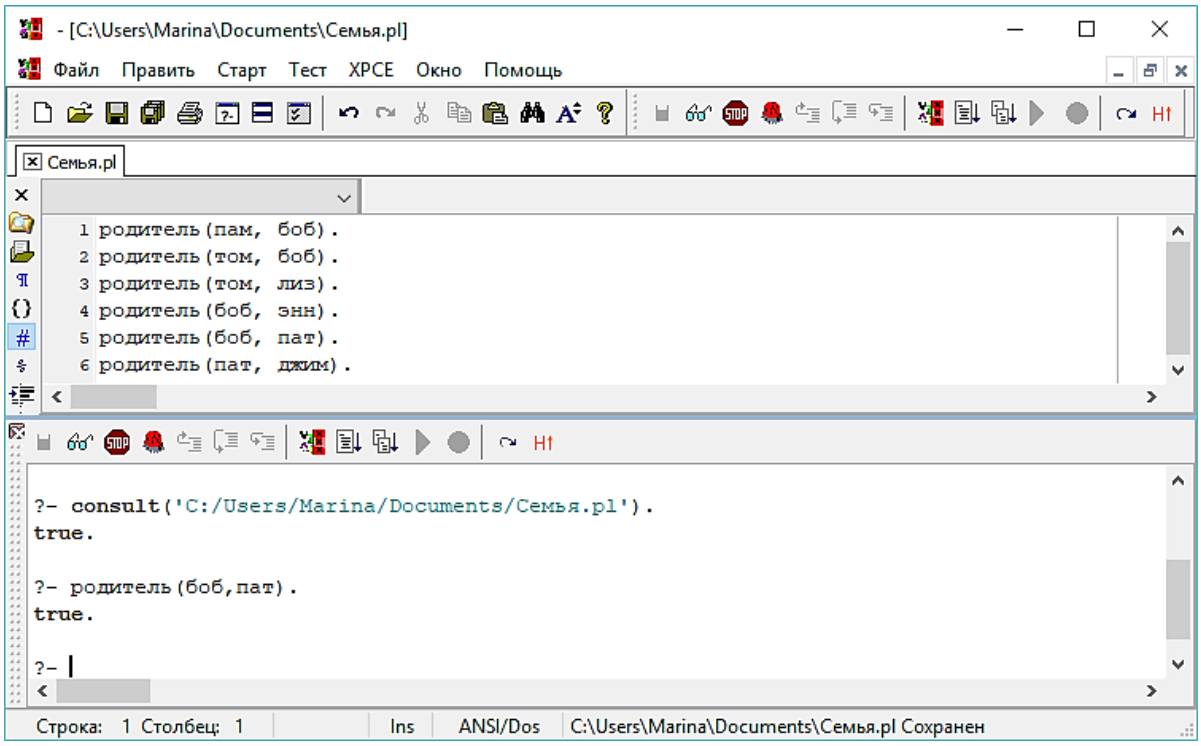


Рис. 40. Ответ интерпретатора на вопрос «Боб является родителем Пат?»

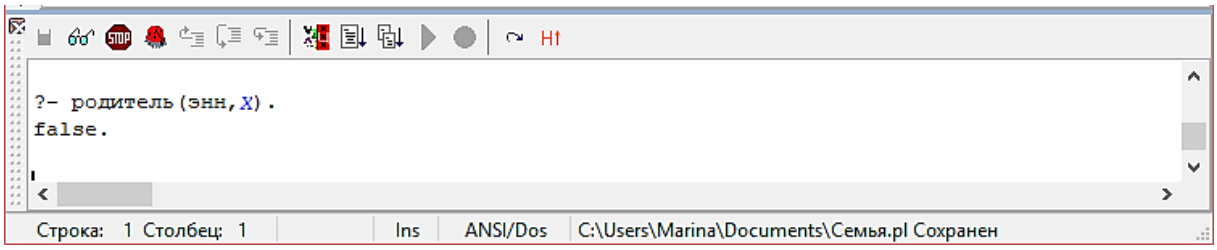


Рис. 41. Ответ интерпретатора на вопрос «Кто дети Энн?»

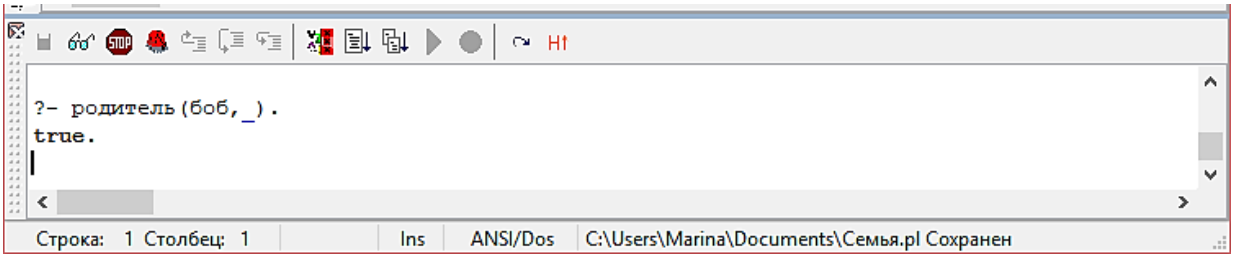


Рис. 42. Ответ интерпретатора на вопрос «У Боба есть дети?»

В приведенном на рис. 42 вопросе используется _ – анонимная переменная (важно только найдет ли Пролог ее значение, а само значение не важно). Если бы вместо _ использовали обычную переменную, то получили бы ответ с альтернативами, изображенный на рис. 43.

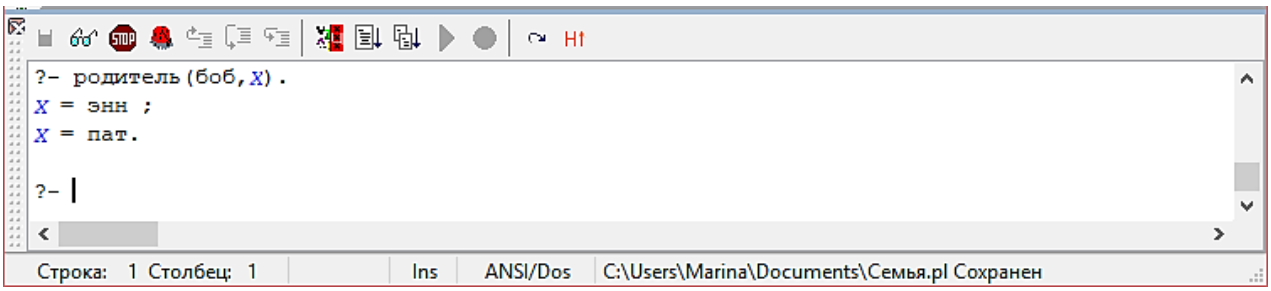


Рис. 43. Ответ интерпретатора на вопрос «Кто дети Боба?»

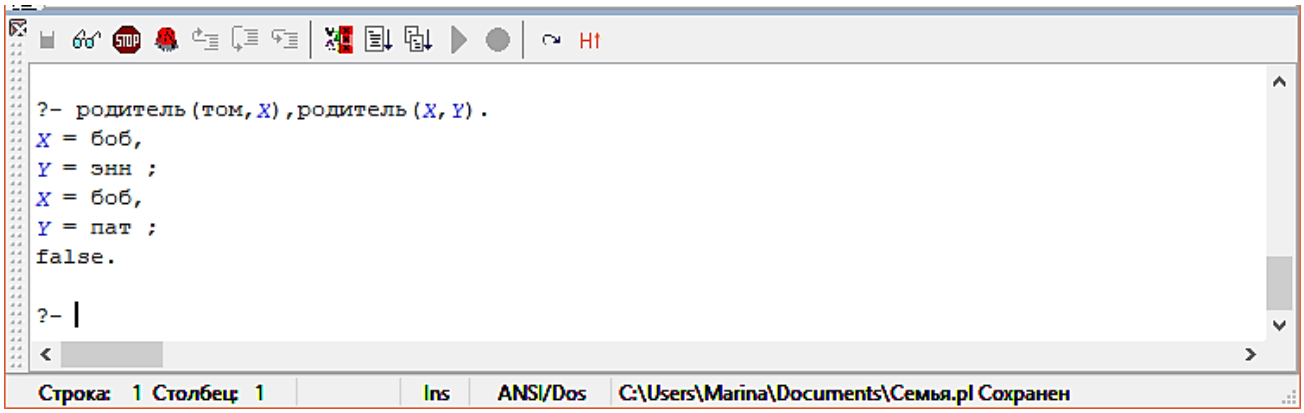


Рис. 44. Ответ интерпретатора на сложный вопрос «Кто внуки Тома?»

Выведенный на рис.44 false свидетельствует о том, что после того, как Пролог нашел внуков Энн и Пат, у него оставался еще вариант поиска детей у Лиз. Но поиск не нашел детей Лиз, которые могли быть внуками Тома.

Итак, в простейшем случае Пролог-программа описывает факты и правила.

Факт – это безусловное утверждение (всегда истинное), характеризующее объект с некоторой стороны или устанавливающее отношение между несколькими объектами. Факт не требует доказательств. Факт имеет следующий вид:

<имя предиката>(O_1,O_2,…,O_n).

Обратим внимание на то, что в конце факта ставится точка. <имя предиката> должно начинаться со строчной буквы и может содержать буквы, цифры, знаки подчеркивания. O_i (i = 1, .., n) - аргументы предиката могут быть конкретными объектами (константами) или абстрактными объектами (переменными). Если конкретные объекты начинаются с буквы, то эта буква должна быть строчной.

Переменные начинаются с прописной буквы или символа подчеркивания. Переменная в Прологе, в отличие от алгоритмических языков программирования, обозначает объект, а не некоторую область памяти. Пролог не поддерживает механизм деструктивного присваивания, позволяющий изменять значение инициализированной переменной, как императивные языки. Переменные могут быть *свободными* или *связанными*. *Свободная переменная* – переменная, которая еще не получила значения. Она не равняется ни нулю, ни пробелу; у нее вообще нет никакого значения. Такие переменные еще называют *неконкретизированными*. Переменная, которая получила какое-то значение и оказалась связанной с определенным объектом, называется *связанной*. Если переменная была конкретизирована каким-то значением и ей сопоставлен

некоторый объект, то эта переменная уже не может быть изменена внутри одного предложения (до точки).

Областью действия переменной в Прологе является одно предложение. В разных предложениях может использоваться одно и то же имя переменной для обозначения разных объектов. Исключением из правила определения области действия является анонимная переменная, которая обозначается символом подчеркивания. Анонимная переменная предписывает интерпретатору проигнорировать значение переменной. Если в правиле несколько анонимных переменных, то все они отличаются друг от друга, несмотря на то, что записаны с использованием одного и того же символа.

Правило – утверждение, которое истинно при выполнении некоторых условий. Правило состоит из условной части (тела) и части вывода (головы). *Головой правила* является предикат, истинность которого следует установить. *Тело правила* состоит из одного или нескольких предикатов, связанных логическими связками: конъюнкция (обозначается запятой), дизъюнкция (обозначается точкой с запятой) и отрицание (означается not или \+). Так же как в логических выражениях, порядок выполнения логических операций можно менять расстановкой скобок. Правило имеет следующий вид:

<голова правила> :– <тело правила>.

Знак «:-» в правиле означает «если». В конце правила также ставится точка. Можно считать, что факт – это правило, имеющее пустое тело. Также в теле правила можно использовать разветвление, которое имеет вид:

(<условие> -> <действие1>;<действие2>)

С помощью правил можно описывать новые отношения.

Пример 2 Пусть имеется двуместное отношение родитель и одноместное отношение мужчина. Опишем эти отношения в виде фактов.

Опишем новое двуместное отношение дед, используя правило:
X является дедом Y, если существует цепочка X– родитель Z, Z – родитель Y, при этом X должен быть мужчиной. На Прологе это запишется следующим образом:

```
дед(X,Y):–родитель(X,Z),родитель(Z,Y),мужчина(X).
```

Пример 3 Пусть имеется двуместное отношение родитель, описанное в виде фактов.

Опишем новое двуместное отношение предок, используя правило:
X является предком Y, если X – родитель Y или существует цепочка людей между X и Y, связанных отношением родитель.

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

Эти правила можно записать по-другому:

```
предок(X,Y):–родитель(X,Y); родитель(X,Z),предок(Z,Y).
```

В данном примере получили рекурсивное определение отношения предок. Добавим это отношение в программу из примера 1 (рис.45).

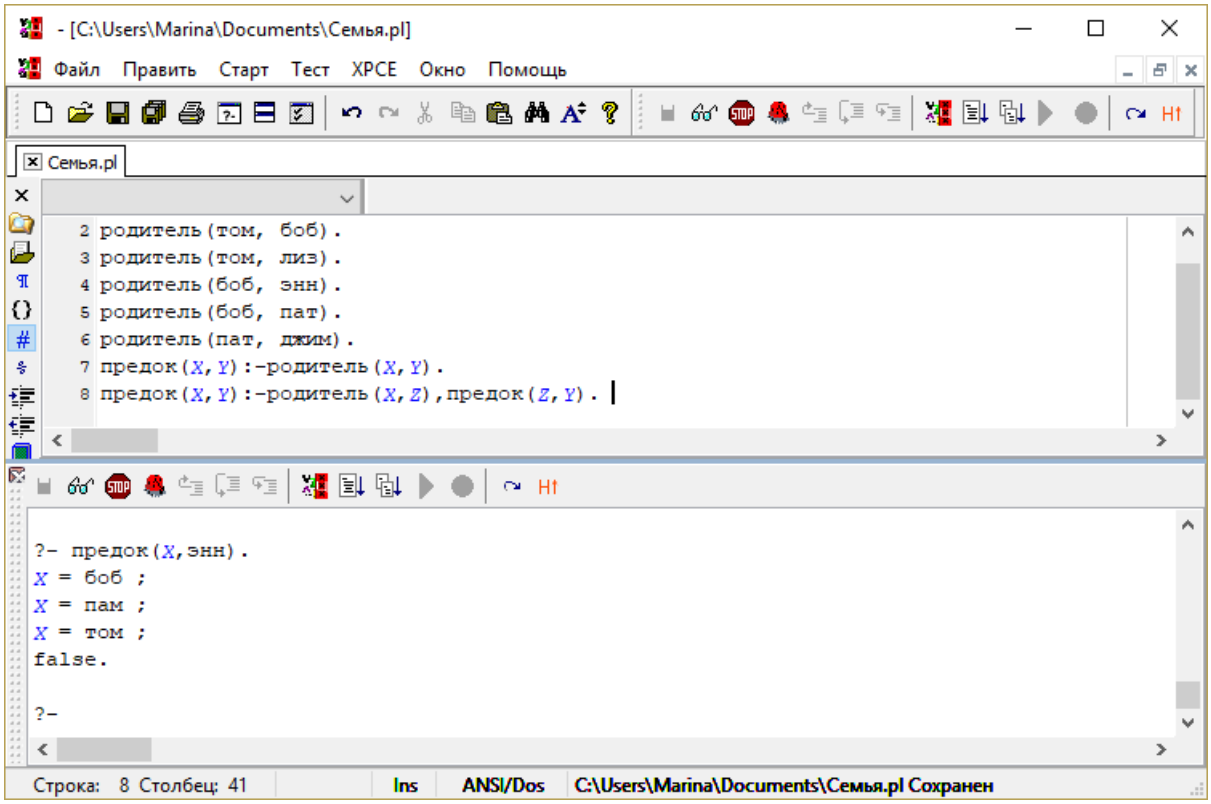


Рис. 45. Ответ интерпретатора на вопрос «Кто предки Энн?»

Замечание: После любой модификации программу требуется заново загрузить интерпретатору.

Перезапуск интерпретатора Пролога осуществляется нажатием Ctrl+F9 или пиктограммы на панели инструментов. Для получения доступа к справочнику наберите команду help.

3.2 Операции в SWI-Prolog

Операция	Назначение операции
=	Унификация (присваивание значения несвязанной переменной)
<, =<, >=, >	Арифметические (только для чисел) операции сравнения
=:=	Арифметическое равенство
=\=	Арифметическое неравенство
is	Вычисление арифметического выражения
@<, @=<, @>=, @>	Операции сравнения для констант и переменных любого типа (чисел, строк, списков и т.д.)
==	Равенство констант и переменных любого типа
\==	Неравенство констант и переменных любого типа
A mod B	Остаток от деления A на B
A//B	Целочисленное частное при делении A на B

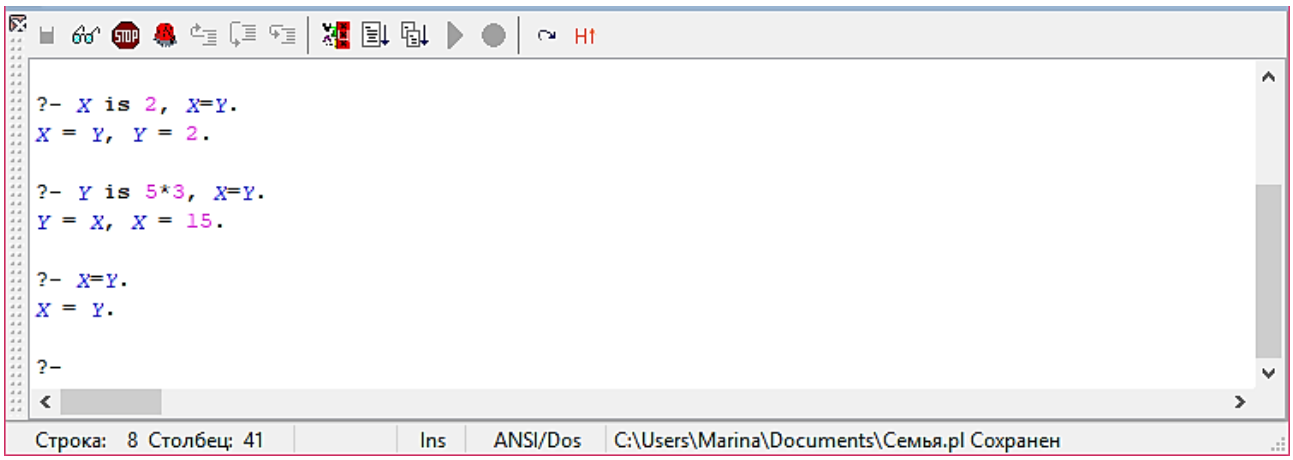


Рис. 46. Примеры применения операций is и =

Из приведенного рис. 46 видно, что несвязанная переменная может находиться как справа, так и слева от знака «=». В последнем вопросе переменные X и Y оказались сцепленными (не связанными, но в случае получения значения одной переменной в этом предложении, вторая получила бы такое же значение).

3.3 Предикаты ввода-вывода, комментарии

Предикат	Назначение предиката
read(A)	Предикат с одним аргументом, чтение значения с клавиатуры в переменную A
write(A)	Предикат с одним аргументом, печать A на экран без перевода строки
writeln(A)	Предикат с одним аргументом, печать A на экран с переводом курсора в начало следующей строки
nl	Перевод курсора в начало следующей строки
format(‘<строка~w>’,X)	<строка> <значение X>
format(‘<строка~w~w>’,[X,Y])	<строка> <значение X><значениеY>
format(‘<строка1~w\n строка2~w>’,[X,Y])	<строка1> <значение X> <строка2> <значениеY>

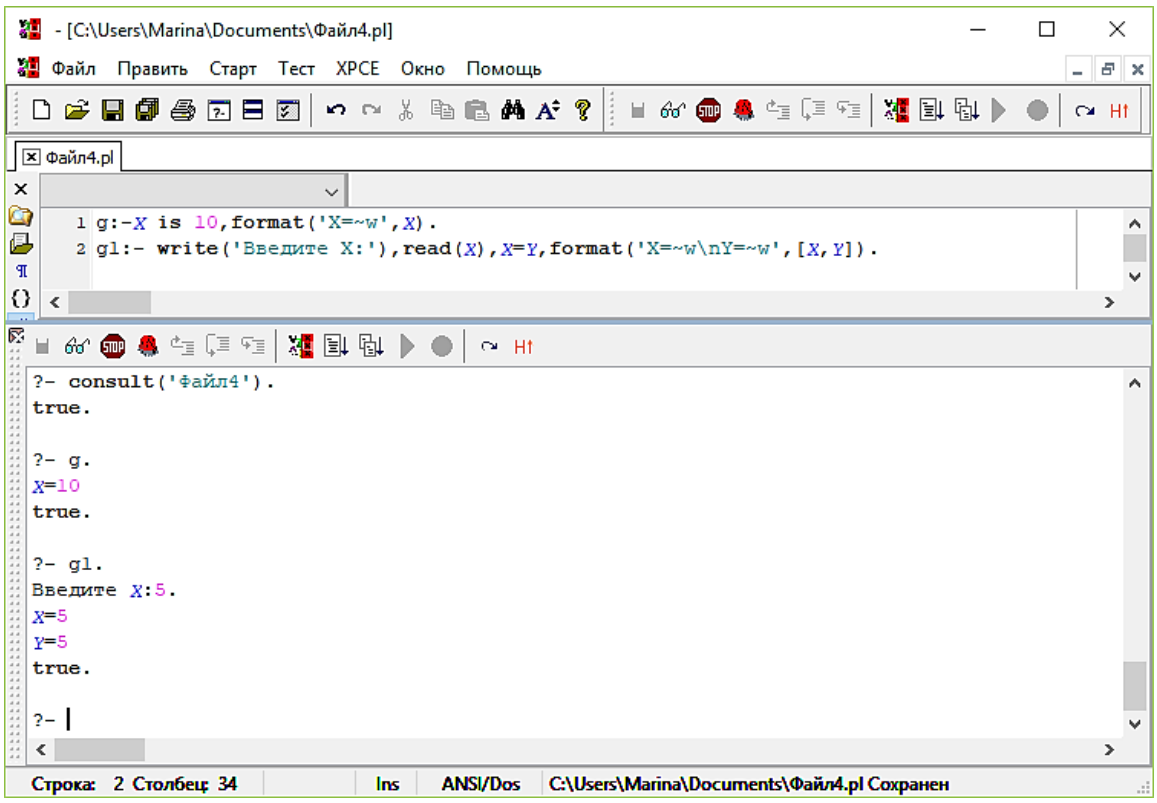


Рис. 47. Примеры использования предикатов read, write и format

При ожидании ввода выводится приглашение «|:». Ввод завершается точкой и нажатием клавиши Enter. Если Enter нажали до ввода точки, то можно ввести точку на следующей строке и еще раз нажать Enter.

Комментариям до конца текущей строки предшествует знак «%», можно воспользоваться пиктограммой на панели инструментов слева. Блочный комментарий заключается между /* и */.

Узнать текущую рабочую папку можно предикатом pwd, содержимое текущей папки – предикатом ls, сменить текущую рабочую папку можно с помощью предиката cd('<путь к папке, слэши дублируются>').

3.4 Поиск решений Пролог-системой

Вопрос к системе – это всегда последовательность, состоящая из одной или нескольких целей. Для ответа на поставленный вопрос Пролог-система должна *достичь всех целей*, т.е. показать, что утверждения вопроса истинны в предположении, что все отношения программы истинны. Если в вопросе имеются переменные, то система должна найти конкретные объекты, которые, будучи подставлены вместо переменных, обеспечат достижение цели. Если система не в состоянии вывести цель из имеющихся фактов и правил, то ответ должен быть отрицательный. Таким образом, факты и правила в программе соответствуют аксиомам, а вопрос – теореме.

При поиске ответа на поставленный вопрос Пролог-система находит факт или правило для содержащегося в вопросе предиката и выполняет операцию сопоставления (унификации) объектов предиката. При этом возможны следующие случаи успешного сопоставления:

- сопоставляются две одинаковые константы;

- сопоставляется свободная переменная с константой (при этом переменная получает значение, равное константе);
- сопоставляется связанная переменная с константой, равной значению переменной;
- сопоставляется свободная переменная с другой свободной переменной.

Никаких значений переменные при этом не получают, но между ними устанавливается связь, таким образом, что в дальнейшем они будут выступать в качестве синонимов. Такие переменные называют сцепленными.

После успешного сопоставления все переменные получают значения и становятся связанными, а предикат считается успешно выполненным (если сопоставление выполнялось с фактом) или заменяется на тело правила (если сопоставление выполнялось с головой правила). Связанные переменные освобождаются, если цель достигнута или сопоставление неуспешно.

Процесс сопоставления похож на использование операции =. Интерпретация этой операции Прологом зависит от того, известны ли оба значения, связанные этой операцией или только одно из них. Если оба значения известны, то операция интерпретируется как оператор сравнения. Если известно только одно значение, то оператор интерпретируется как присваивание, причем присваивание может выполняться как слева направо, так и справа налево в зависимости от того, слева или справа от оператора находится известное значение.

Для достижения целей Пролог использует механизм отката. Выполняется сопоставление подцелей с фактами и головами правил. Сопоставления выполняются слева направо. Возможно, некоторые подцели будут неуспешны при сопоставлении с некоторыми фактами или правилами, поэтому Пролог должен запоминать «точки», в которых он может поискать альтернативные пути решения. Если цель была неуспешной, то выполняется откат влево к ближайшему указателю отката и выполняется новая попытка достичь цели. Этот откат будет повторяться, пока цель не будет достигнута или исчерпаются все указатели отката. Цель была достигнута, но использовались не все указатели отката, то будет продолжен поиск других решений.

Пример Рассмотрим работу Пролог-системы при ответе на вопрос предок(том,энн). Представим шаги вычислений графически (рис. 48).

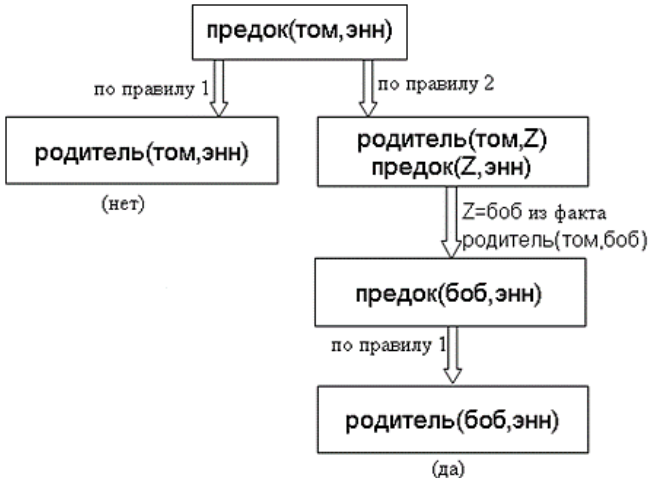


Рис. 48. Поиск решений

Сначала выполняется сопоставление цели и головы первого правила, сопоставление успешно и переменные X и Y становятся связанными (X получает значение том, а Y – значение энн). При этом Пролог-система запоминает, что имеется второе правило, по которому можно будет продолжить поиск решения. Далее начинает выполняться тело первого правила (новая цель родитель(том,энн)). Опять выполняются все возможные сопоставления с фактами предиката родитель. Сопоставление неуспешно, все связи разорваны.

Далее происходит возврат для сопоставления цели и головы второго правила. Сопоставление успешно и переменные X и Y становятся связанными (X получает значение том, а Y – значение энн). Далее начинает выполняться тело второго правила (новая цель – родитель(том,Z)). В нем имеется две подцели, их достижение проверяется по порядку. Первая подцель успешна, т.к. сопоставляется с фактом родитель(том,боб), при этом Z получает значение боб. Новая подцель - предок(боб,энн). Опять выполняется составление подцели и головы первого правила. Оно успешно, при этом X получает значение боб, а Y – значение энн. Новая подцель – правая часть первого правила родитель(боб,энн) успешна.

Следовательно, цель успешна и ответ Пролог-системы: true.

3.5 Отладка и трассировка

Важным аспектом в программировании на языке Prolog является процедура отладки, поэтому остановимся на этом процессе подробнее. Отладчик в Prolog следует так называемой модели «procedure box flow control model». На русский язык это можно перевести как модель процедуры управления потоком. Эта модель довольно популярна при построении отладчиков в среде Prolog и схематично она представлена на рис. 49.

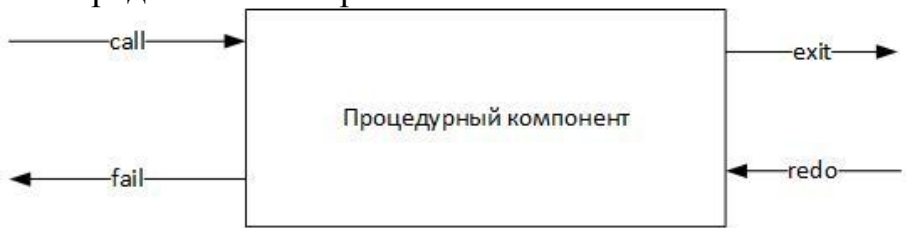


Рис. 49. Модель компоненты управления потоком выполнения

Эта модель описывает четыре события, при которых происходит передача потока управления, в конкретных реализациях этим событиям соответствуют одноименные порты – точки, где управление передается отладчику.

Отладчик выполняет программу пошагово, отслеживая вызов текущей цели (Call), и возвращение управления из нее либо при успешном сопоставлении (Exit), либо при неуспехе (Fail). Когда выполнение неуспешно, управление передается последней цели, для которой существуют альтернативные решения, тогда говорят, что цель вызывается повторно (Redo). Каждая стрелка на рисунке выше соответствует порту. Стрелка, входящая в процедуру, соответствует передаче ей потока управления, тогда как исходящая – передаче его другой процедуре. В частности, рекурсивные утверждения будут представлять собой процедуры при каждом рекурсивном вызове со своими собственными точками

входа и выхода потока управления. Цепочка вызовов, особенно рекурсивных, может приводить к затруднениям у пользователя, поэтому каждый такой компонент ассоциируется с уникальным идентификатором (номер вызова), а последовательность вызовов образуют стек.

В SWI-Prolog имеются возможности отлаживать программы в текстовом и графическом режимах.

3.5.1 Отладка в текстовом режиме

Включить опцию трассировки всех предикатов можно выполнив команду trace в окне интерпретатора или нажав пиктограмму на панели инструментов.

После появления приглашения с префиксом в виде [trace]?- можно вводить предикат для трассировки.

Трассировка заключается в выводе последовательности трассировочных сообщений, которые в SWI-Prolog имеют следующую структуру:

Port: (n) Goal ?

где Port: - имя порта (см. выше);

(n) – уникальный идентификатор компонента управления (номер вызова);

Goal – текущая цель;

В этой точке отладчик ожидает ввода пользователем одной из допустимых команд. Доступный список команд можно увидеть, введя символ «h» (рис. 50).

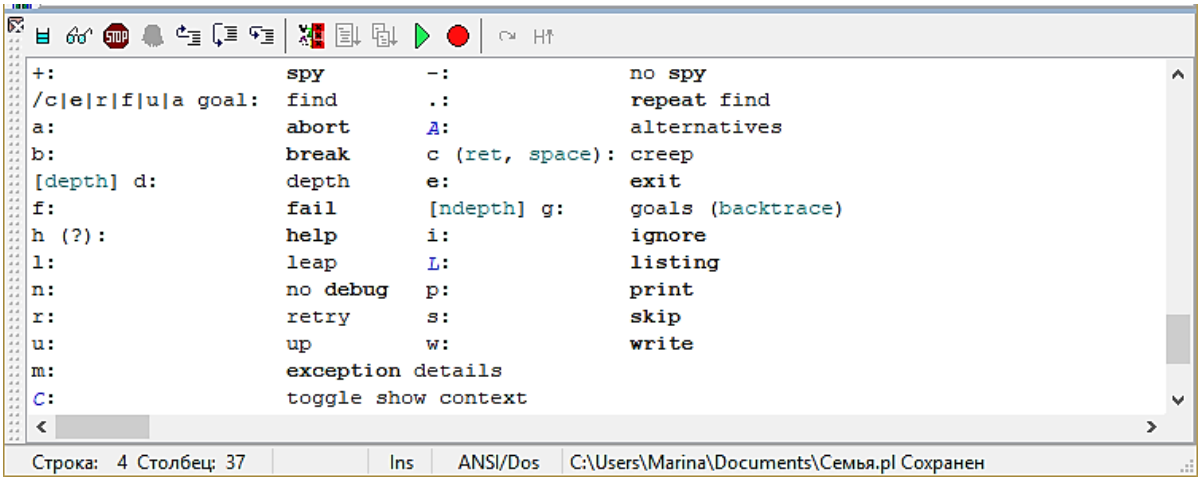


Рис. 50. Доступные команды трассировки

Наиболее часто будут использоваться две команды:

- c (ret, space) – продолжать трассировку (можно использовать пиктограмму на панели инструментов);
- a – прервать выполнение программы (можно использовать пиктограмму на панели инструментов).

Трассировка поиска Пролог-системой ответа на вопрос предок(том,энн) приведена на рис. 51. После каждого шага трассировки нажимаем Enter.

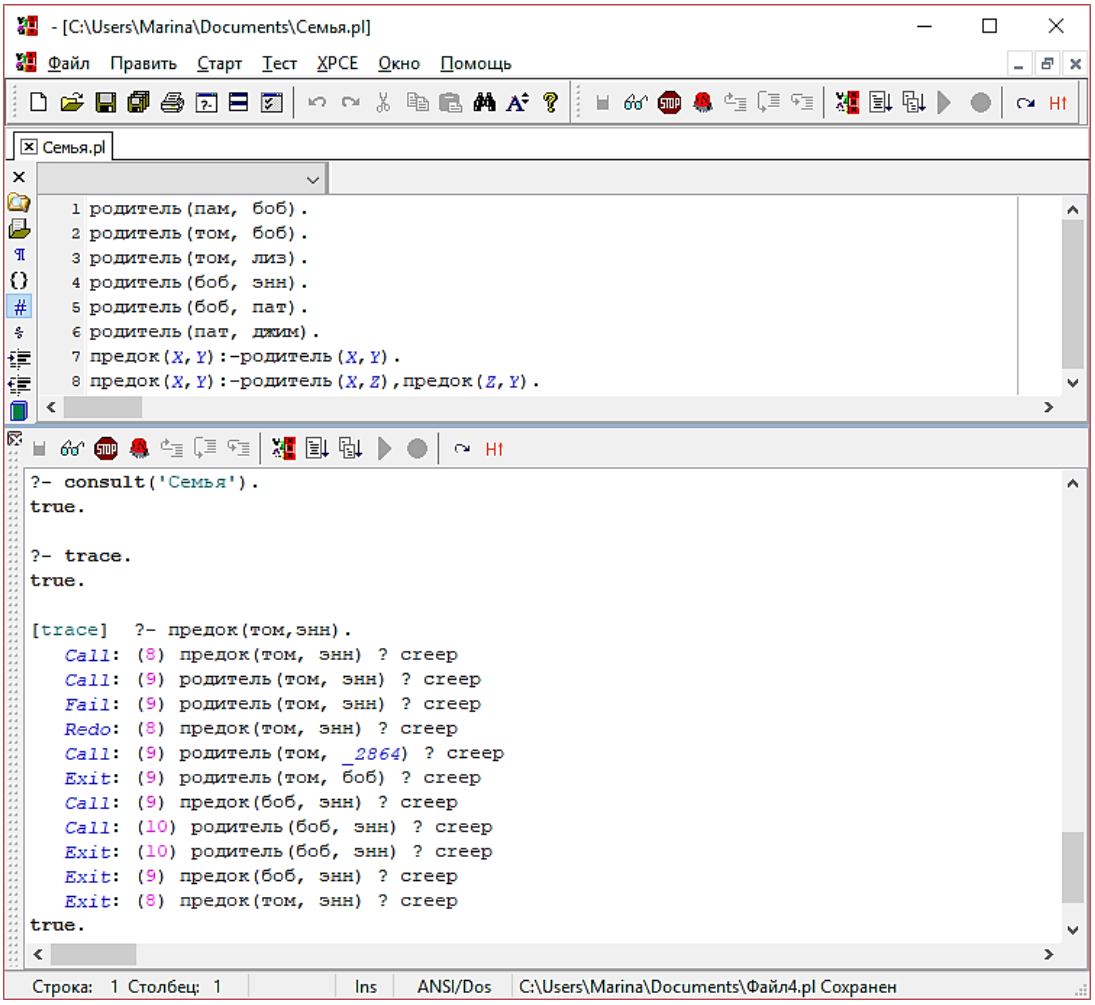


Рис. 51. Трассировка поиска Пролог-системой ответа на вопрос предок(том,энн)

Итак, прокомментируем работу отладчика на рис. 51.

1. Вызов текущей цели предок(том,энн). Эта компонента получает идентификатор 8.

2. Вызов предиката родитель(том,энн) из тела первого правила в качестве текущей цели. При этом переменные X и Y становятся связанными соответственно с том и энн, вызов получает идентификатор 9.

3. Сопоставление неуспешно – об этом сигнализирует имя порта: Fail. Как уже указывалось выше при неуспешном сопоставлении управление должно быть передано в ближайшую точку, где возможен поиск альтернатив. Такой точкой является второе правило для предиката предок.

4. Возврат к вызову первоначальной цели предок(том,энн) (об этом сигнализирует имя порта: Redo), т.к. у предка было еще одно правило, которое запомнилось в качестве точки возврата.

5. Вызов первого предиката родитель(том,_2864) из тела второго правила в качестве текущей цели. При этом переменные X и Y становятся связанными соответственно с том и _2864 (анонимной переменной, которая пока не связана ни с каким значением), вызов получает свой идентификатор 9.

6. Сопоставление успешно родитель(том,боб) (об этом сигнализирует имя порта: Exit), а переменная Y становится связанной со значением боб.

7. Появляется новая цель – второй предикат из тела второго правила предок(боб,энн), вызов получает идентификатор 9.

8. Вызов предиката родитель(боб,энн) из тела первого правила в качестве текущей цели. При этом переменные X и Y становятся связанными соответственно с боб и энн, вызов получает идентификатор 10.

9. Предикат родитель(боб,энн) успешен (об этом сигнализирует имя порта: Exit).

10. Управление передается по стеку выше, т.е. предикату с идентификатором 9, предок(боб,энн) также успешен (об этом сигнализирует имя порта: Exit).

11. Управление передается по стеку выше, т.е. предикату с идентификатором 8, предок(том,энн) также успешен (об этом сигнализирует имя порта: Exit).

12. Возвращаемое значение исходной цели true.

Отказ от ранее включенной трассировки осуществляется командой notrace или дважды нажатием пиктограммы  на панели инструментов.

3.5.2 Отладка в графическом режиме

Существует более наглядный способ трассировки выполнения с помощью графического отладчика SWI-Prolog. Для вызова графического отладчика, перед вводом цели необходимо выполнить команду gtrace или нажать пиктограмму  на панели инструментов, затем включить трассировку (командой trace или нажав пиктограмму  на панели инструментов). Затем вводим предикат для трассировки (рис. 52), открывается графический отладчик в новом окне (рис.53).

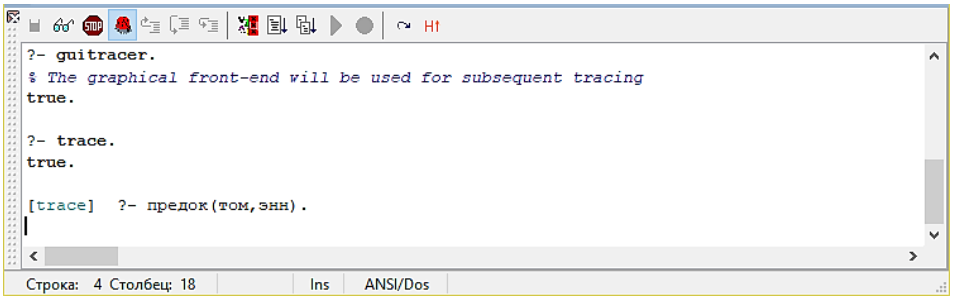


Рис. 52. Запуск графического отладчика

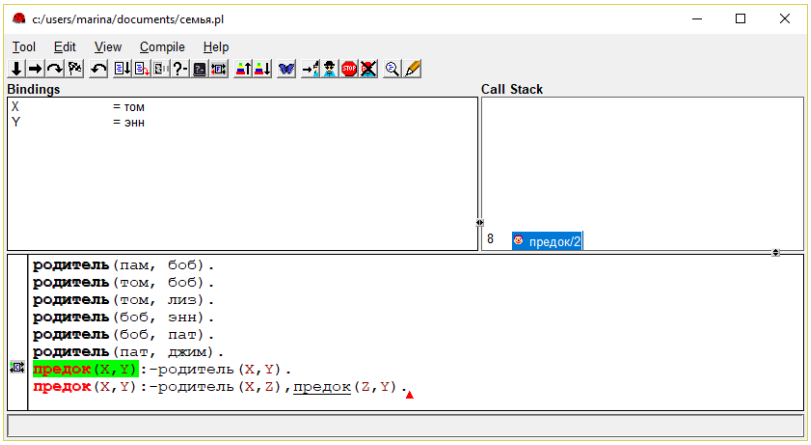


Рис. 53. Окно графического отладчика

Окно графического отладчика разделено на 3 части: верхнее левое окно показывает текущие значения переменных (окно связанных переменных), верхнее правое окно показывает стек вызовов, а нижнее – текст программы.

Для пошагового выполнения программы необходимо последовательно нажимать кнопку с графическим изображением стрелки вправо (Step) или пробел. При нажатии можно видеть изменение стека вызовов, проход отладчика по тексту программы, моменты, когда переменная становится связанной (рис. 54).

Наличие альтернативных путей выполнения предиката обозначается на графических обозначениях предикатов раздвоенной стрелкой. Прямые стрелки, соединяющие различные предикаты, обозначают направление передачи потока управления между предикатами. После того, как верхняя цель будет успешно или неуспешно достигнута, отладчик перейдет в режим ожидания команды пользователя. Если вы хотите увидеть другие решения для вашей цели, вам необходимо вернуться в основное окно SWI-Prolog и ввести «;» - это заставит отладчик продолжить работу.

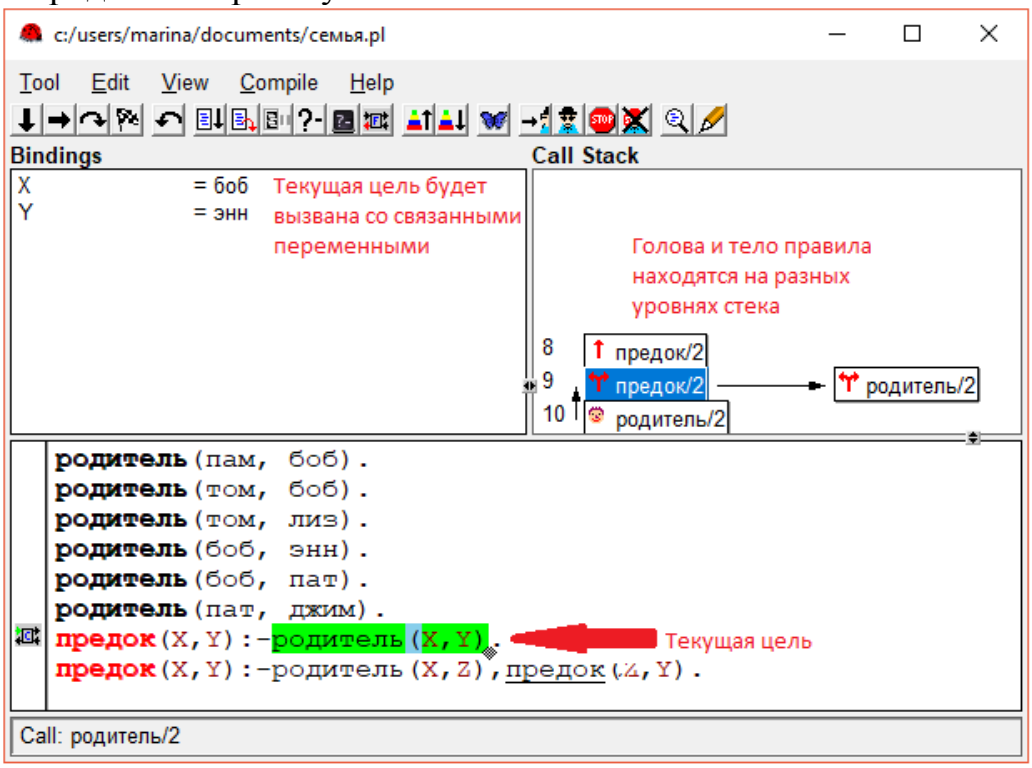


Рис. 54. Фрагмент выполнения отладки

3.6 Данные в Прологе

В соответствии с теорией предикатов первого порядка единственная структура данных в логических программах - *термы*.

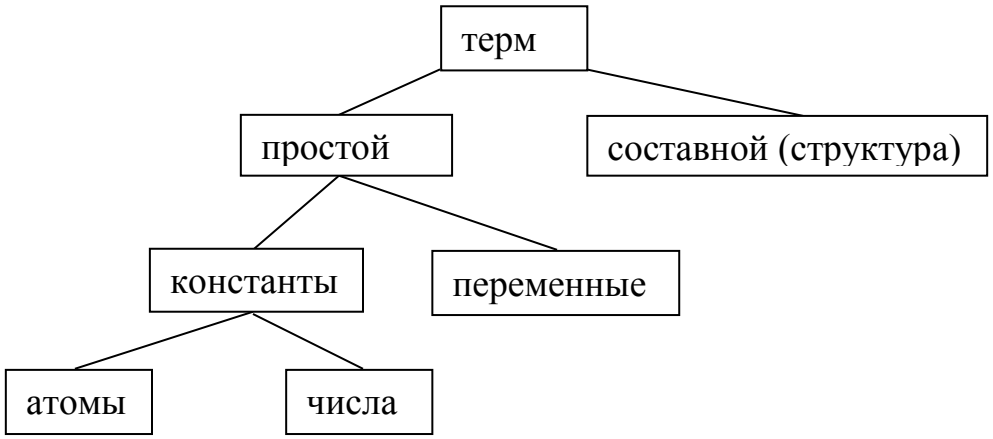


Рис. 55. Виды термов

Определение термов индуктивно. Константы (числа и атомы) и переменные являются термами. Атом – последовательность латинских букв, цифр и символа подчеркивания, начинающаяся со строчной буквы, любая последовательность символов, заключенных в апострофы или специальных символов (+ - * / < > = : . & _ ~). Переменная – последовательность латинских букв, цифр и символа подчеркивания, начинающаяся с прописной буквы или символа подчеркиванияю

Кроме того, термами являются составные термы, или структуры. *Составной терм* содержит *функтор* и последовательность из одного или более аргументов, являющихся термами. Функтор задается своим *функциональным именем* и своей *арностью* (числом аргументов). Синтаксически составные термы имеют вид:

$$f(t_1, t_2, ..., t_n),$$

где f - имя n-арного функтора, а t_i - термы.

Например, для представления точки можно ввести структуру с функтором точка, которая будет объединять объекты – координаты точки в пространстве.

Поскольку предикаты синтаксически выглядят так же, как составные термы, то программы и данные в Прологе имеют одинаковую форму, как в Лиспе.

В Пролог-программах допускается использование одного и того же предикатного (или функционального символа) с разным числом аргументов. Это возможно, поскольку каждый функтор определяется двумя параметрами: именем и арностью.

Структуры используются в Прологе для конструирования сложных типов данных. Например, для представления даты естественно использовать терм вида:

дата(1,май,1998).

Для проверки типа терма используются встроенные предикаты:

var(Term) - свободная переменная
nonvar(Term) - несвободная (конкретизированная) переменная
integer(Term) - целое число
float(Term) - вещественное число
number(Term) - целое или вещественное число
atom(Term) – атом
atomic(Term) - атом или число
ground(Term) -терм не содержит свободных переменных

3.7 Семантика Пролога

3.7.1 Порядок предложений и целей

Программу на Прологе можно понимать по-разному: с декларативной и процедурной точки зрения. Декларативная семантика касается только отношений, описанных в программе, и определяет, что является ли поставленная цель достижимой и если да, то определяются значения переменных, при которых эта цель достижима. Процедурная семантика определяет, как должен быть получен результат, т.е. как Пролог-система отвечает на вопросы. Для правила вида P:–Q,R. декларативная семантика определяет, что из истинности Q и R следует истинность P, а процедурная семантика определяет, что для решения P следует сначала решить Q, а потом R (важен порядок обработки целей).

Если рассмотреть верное с декларативной точки зрения правило P:–P, то можно заметить, что с процедурной точки зрения оно бесполезно, более того, такое правило приводит к бесконечному циклу.

Рассмотрим насколько порядок предложений и целей, который не затрагивает декларативную семантику, может изменить процедурную семантику.

Пример Рассмотрим определение предиката предок из примера 3 п.3.1:

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

Попробуем поменять порядок следования правил и подцелей в правиле. Рассмотрим еще три варианта этого отношения предок1, предок2, предок3. Эти предикаты являются правильными с декларативного смысла.

```
предок1(X,Y):–родитель(X,Z),предок1(Z,Y);
                родитель(X,Y).
предок2(X,Y):–родитель(X,Y);
                предок2(Z,Y), родитель(X,Z).
предок3(X,Y):– предок3(Z,Y), родитель(X,Z);
                родитель(X,Y).
```

При ответе на вопрос предок(том, боб) получим ответ true в результате применения первого правила.

При ответе на вопрос предок1(том, боб) получим ответ true в результате применения второго правила, но прежде, чем применить это правило Пролог-система будет пытаться применить первое правило для Боба и всех его потомков. Решение будет найдено, но его поиск будет идти очень долго.

При ответе на вопрос предок2(том, боб) получим сразу ответ true в результате применения первого правила. Но заметим, что этот предикат не всегда будет давать верный ответ. Например, на вопрос предок2(лиз, боб) не будет получено ответа, т.к. на первой подцели второго правила, Пролог-система уйдет в бесконечную рекурсию.

Ответ на вопрос предок3(том, боб) не будет получен, т.к. на первой подцели первого правила, Пролог-система уйдет в бесконечную рекурсию.

Таким образом, правильные с декларативной точки зрения программы могут работать неправильно. При составлении правил следует руководствоваться следующим:

- более простое правило следует ставить на первое место;
- по возможности избегать левой рекурсии.

3.7.2 Пример декларативного создания программы

Рассмотрим задачу об обезьяне и банане. Возле двери комнаты стоит обезьяна. В середине комнаты к потолку подвешен банан. Обезьяна голодна и хочет съесть банан, но не может до него дотянуться, находясь на полу. Около окна этой комнаты находиться ящик, которым обезьяна может воспользоваться. Обезьяна может предпринимать следующие действия: ходить по полу, залазить на ящик, двигать ящик (если обезьяна находится возле ящика), схватить банан (если обезьяна находится на ящике под бананом). Необходимо ответить на вопрос, сможет ли обезьяна добраться до банана?

Будем считать, что мир обезьяны всегда находится в некотором состоянии, которое может изменяться со временем. Состояние обезьяньего мира определяется четырьмя компонентами: горизонтальная позиция обезьяны, вертикальная позиция обезьяны (на ящике или на полу), позиция ящика, наличие у обезьяны банана (есть или нет). Объединим эти компоненты в структуру функтором состояние.

Задачу можно рассматривать как игру для одного игрока – обезьяны. Формализуем правила этой игры. Начальное состояние игры: состояние(удвери, наполу, уокна, нет), цель игры: состояние(_,_,_, да). Определим возможные ходы обезьяны: перейти в другое место, подвинуть ящик, залезть на ящик, схватить банан. Не всякий ход допустим для каждого состояния. Например, ход «схватить банан» допустим только для состояния, когда обезьяна стоит на ящике в середине комнаты и еще не имеет банана, т.е. состояние(середина, наящике, середина, нет). В результате хода система переходит из одного состояния в другое. Для отражения таких переходов определим предикат ход(состояние, действие, состояние).

С учетом вышесказанного декларативная программа будет иметь следующий вид:

```
goal:- может_достать(состояние(удвери, наполу, уокна, нет)).
ход(состояние(середина, наящике, середина, нет),
        схватить,
        состояние(середина, наящике, середина,да)).
ход(состояние(P, наполу, P, H),
        залезть,
        состояние(P, наящике, P, H)).
ход(состояние(P1, наполу, P1, H),
        подвинуть(P1, P2),
        состояние(P2, наполу, P2, H)).
ход(состояние(P1, наполу, P, H),
        перейти(P1, P2),
```

```
        состояние(P2, наполу, P, H)).
может_достать(состояние(_,_,_,да)).
может_достать(S1):-ход(S1, _, S2),
                    может_достать(S2).
```

На рис. 56 представлено, как для такой декларативной программы Пролог-система будет искать решение.

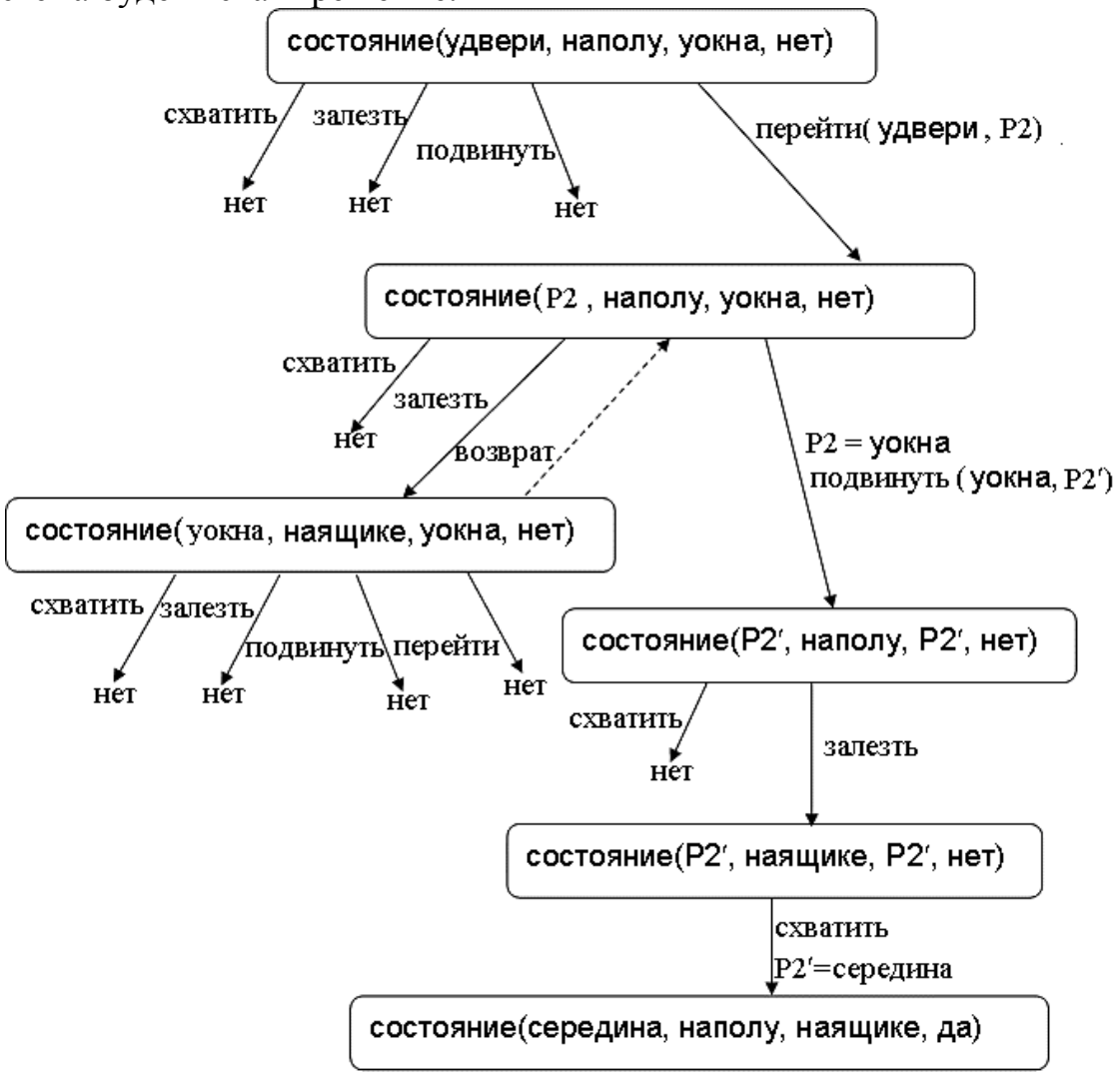


Рис. 56. Поиск решений в задаче «Обезьяна и банан»

Для того чтобы ответить на вопрос, Пролог-системе пришлось сделать лишь один возврат. Причина такой эффективности – правильно выбранный порядок следования предложений, описывающих ходы. Однако возможен и другой порядок, когда обезьяна будет ходить туда-сюда, не касаясь ящика, или бесцельно двигать ящик в разные стороны. Порядок предложений и целей важен в программе.

3.8 Рекурсия

Рекурсия в Прологе может быть алгоритмическая и по данным. Основное отличие от Лиспа заключается в том, что возвращаемое значение должно находиться в аргументе предиката. При этом следует помнить, что передача параметров осуществляется по значению.

Пример 1 Напишем программу, которая *n* раз выводит на экран строку **********.

```
вывод_строки(N):-N>0,
                writeln('**********'),
                N1 is N-1,
                вывод_строки(N1).
```

На (рис. 57) приведены результаты ответа на вопрос вывод_строки(3).

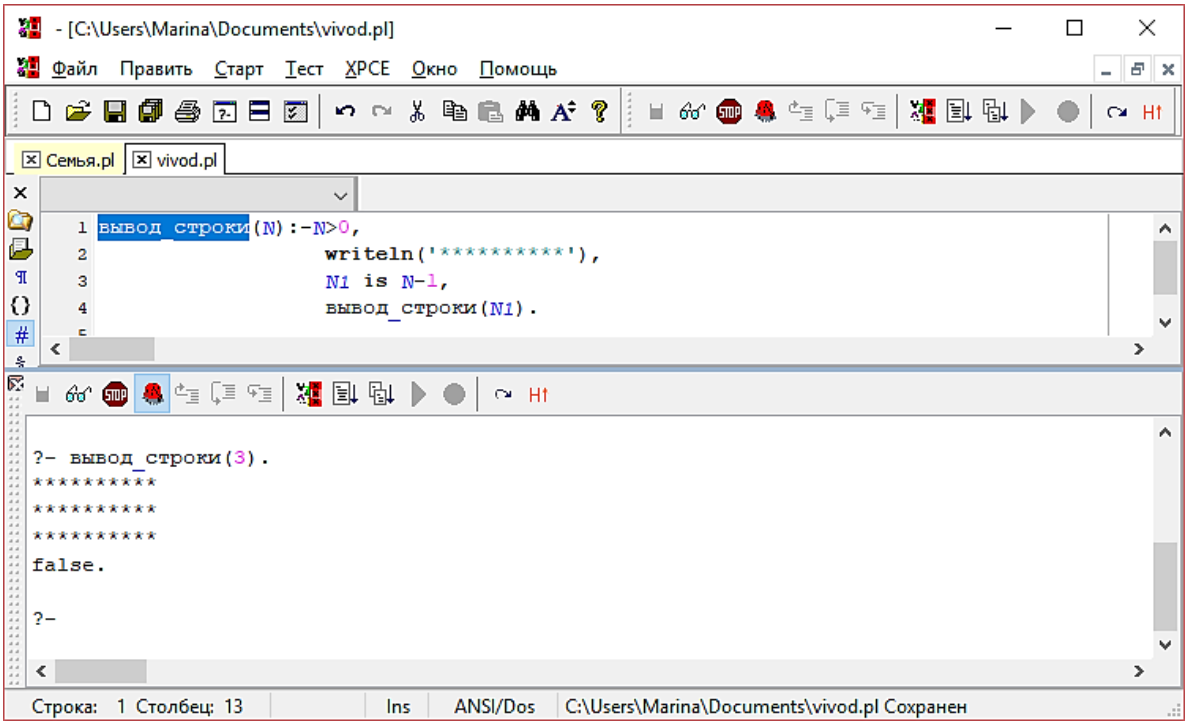


Рис. 57. Результаты ответа Пролог-системы

Несмотря на то, что строка напечаталась, цель не была достигнута. Таким образом, если бы за обращением к предикату вывод_строки(3) следовало продолжение предложения цели, то это продолжение не стало бы выполняться. Это связано с тем, что предикат вывод_строки(N) неудачно завершился при N=0 (не нашлось подходящего факта или правила), т.е. не было правила выхода из рекурсии.

Добавим правило остановки рекурсии для успешного завершения предиката: вывод_строки(0) (рис. 58).

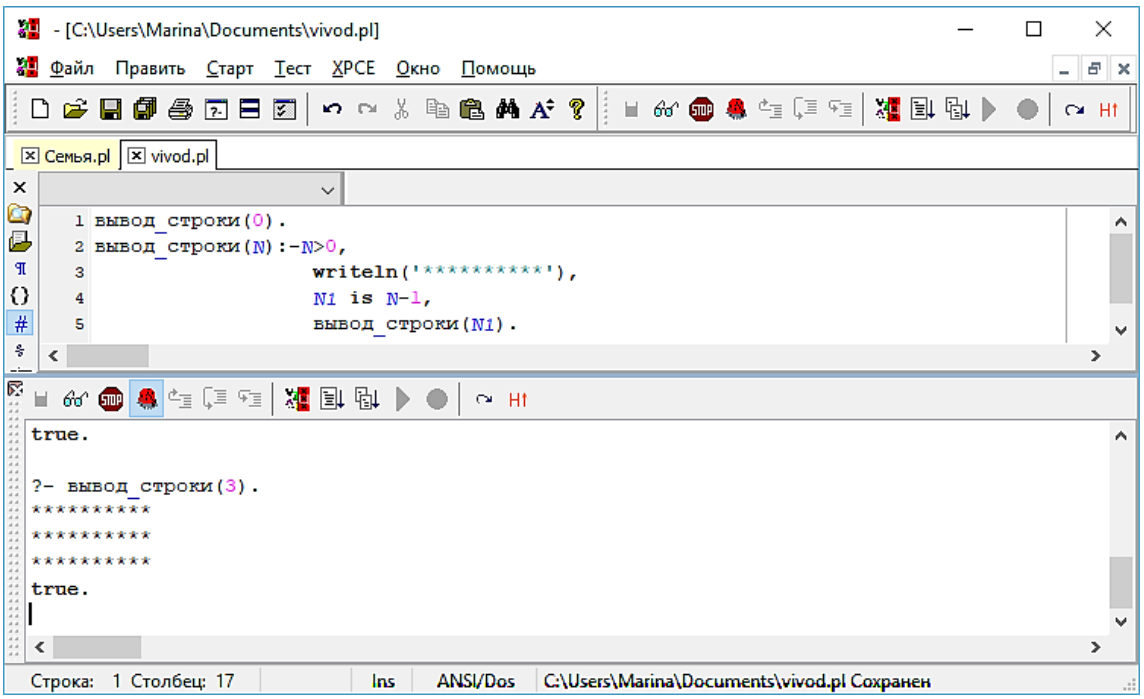


Рис. 58. Результаты ответа Пролог-системы после добавления правила

Пример 2 Напишем программу вычисления *n*!, где *n* вводится с клавиатуры.

```
факториал(0,1):-!.
факториал(N,P):-N>0,
                N1 is N-1,
                факториал(N1,P1),
                P is P1*N.
goal:-writeln('Введите N'),read(N),факториал(N,P),writeln(P).
```

Результаты работы предиката goal приведены на рис. 59.

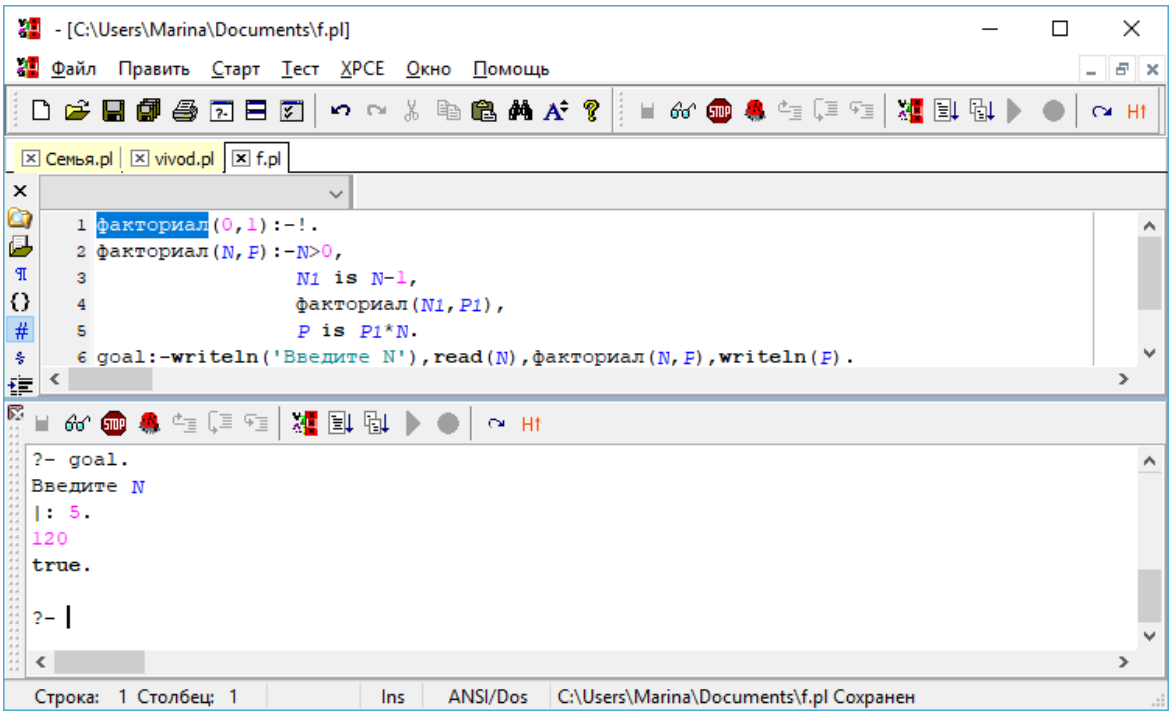


Рис. 59. Результаты работы предиката goal

3.9 Внелогические предикаты управления поиском решений

В процессе достижения цели Пролог-система осуществляет автоматический перебор вариантов, делая возврат при неуспехе какого-либо из них. Такой перебор является полезным программным механизмом, поскольку он освобождает пользователя от необходимости программировать такой перебор. С другой стороны, неограниченный перебор может стать источником неэффективности программы. Поэтому полный перебор часто требуется либо ограничить, либо исключить совсем.

3.9.1 Откат после неудач

Встроенный предикат fail всегда неудачен, поэтому вызывает неудачное завершение цели и инициализирует откат в точки поиска альтернативных решений.

Пример 1 Пусть имеются факты относительно предиката кандидат, которые определяют претендентов для участия в выборах депутатов областного совета. Требуется найти всех кандидатов в возрасте до 40 лет.

```
кандидат('Иванова Анна Федоровна',36).
кандидат('Петров Иван Сергеевич',41).
……
все_до_40:- кандидат(X,Y),
            Y<40,
            writeln(X),
            fail.
все_до_40.
```

После вывода каждого претендента до 40 лет, Пролог-система считает, что цель не достигнута и продолжает поиск решений (точки альтернативных решений находятся в предикате кандидат). Когда будут перебраны все кандидаты, произойдет откат на второе правило все_до_40 (там осталась точка возврата), оно успешно, поэтому выведется true (цель достигнута).

3.9.2 Ограничение перебора – отсечение

Пример 2 Рассмотрим ступенчатую функцию, заданную графически (рис. 60).

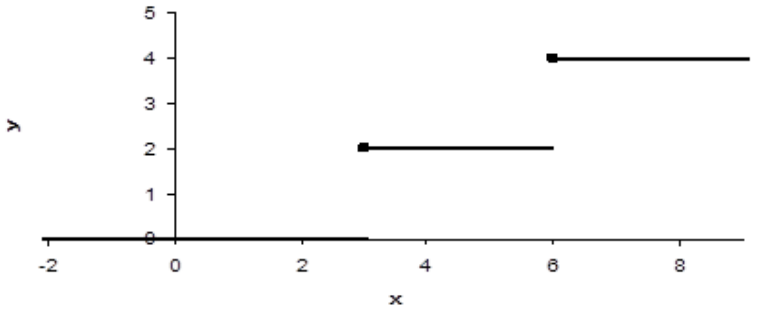


Рис. 60. Задание функции

Аналитически эту функцию можно представить следующим образом:

$$f(x)=\begin{cases}0, x\ge 0\\ 2, 3<x\le 6\\ 0, x>6\end{cases}$$

На Прологе можно описать двуместный предикат *f*.

```
f(X,0):-X=<3.
f(X,2):-X>3,X=<6.
f(X,4):-X>6.
```

Рассмотрим поиск решения Пролог-системой при задании в качестве цели предложения *f* (1,Y),Y>2. Поиск ответа на этот вопрос отражен на рис. 61.

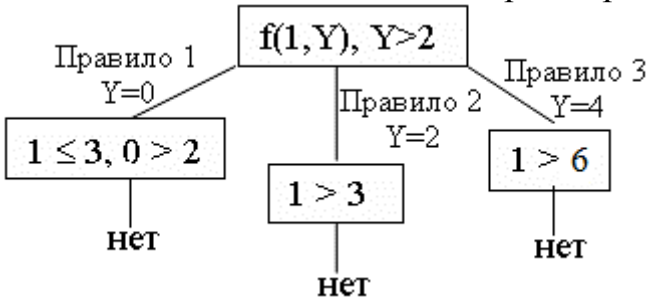


Рис.61. Поиск решения для предложения *f* (1,Y),Y>2.

При вычислении первой цели по правилу 1 переменная Y конкретизируется нулем и цель успешна. Далее вторая цель Y>2 терпит неудачу, поэтому весь список целей терпит неудачу. Однако, Пролог-система при помощи возвратов попытается проверить еще две бесполезные в данном случае альтернативы.

Все имеющиеся правила являются взаимоисключающими, поэтому только одно из них может быть успешным. Мы знаем (но не Пролог-система), что, если одно из правил успешно, нет смысла проверять остальные, поскольку они обречены на неудачу. О том, что правило 1 успешно становится известно в точке, обозначенной словом «отсечение» на рис.62.

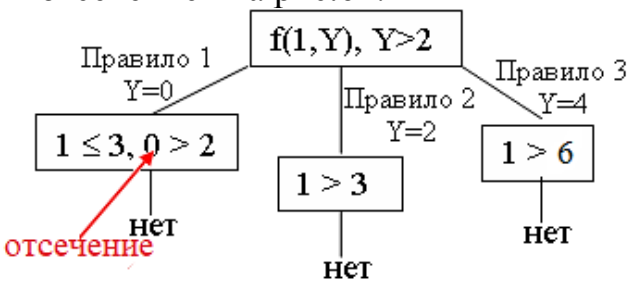


Рис.62. Место для отсечения

Для предотвращения бесполезного перебора мы должны явно указать Пролог-системе, что не нужно осуществлять возврат из этой точки. Для запрета возврата используется предикат cut, который можно записать так же как «!». Этот предикат всегда успешен, он вставляется между целями и предотвращает возврат из тех точек программы, где он стоит. Перепишем программу с использованием отсечения.

```
f(X,0):-X=<3,!.
f(X,2):-X>3,X=<6,!.
f(X,4):-X>6.
```

Теперь при поиске решения альтернативные ветви, соответствующие правилам 1 и 2, порождены не будут. Программа станет эффективнее. Если убрать отсечения, программа выдаст тот же результат, хотя на его получение она затратит, скорее всего, больше времени. Получается, что в данном случае отсечения изменили только процедурный смысл программы (теперь проверяется только левая часть дерева решений), не изменив ее декларативный смысл. Далее будет показано, что отсечения могут затрагивать и декларативный смысл программы.

Теперь рассмотрим поиск решения Пролог-системой при задании в качестве цели предложения *f*(7,Y) (рис. 63).

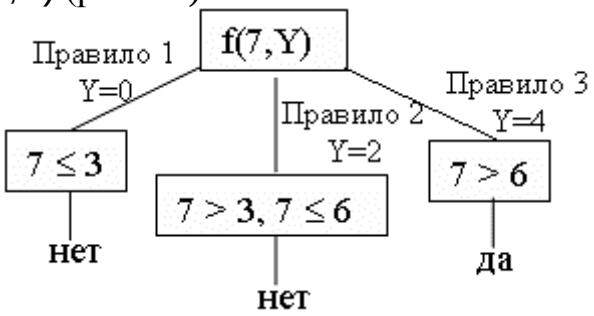


Рис.63. Поиск решения для предложения *f*(7,Y).

Здесь стал заметен еще один источник неэффективности. После того, как выяснилось, что цель 7 ≤ 3 потерпела неудачу, следующей целью становится проверка 7 > 3. Но эту проверку можно опустить, т.к. известно, утверждение Y > 3 является отрицанием утверждения Y ≤ 3. То же самое можно сказать и о цели X > 6 в правиле 3. Эти рассуждения приводят к более эффективной программе:

```
f(X,0):-X<=3,!.
f(X,2):-X<=6,!.
f(X,4).
```

Но теперь, если из этой программы убрать отсечения, то она будет не всегда правильно работать. Например, для цели *f*(1,Y) будут найдены три решения: Y=0, Y=2, Y=4. Таким образом, теперь отсечения затрагивают декларативный смысл программы.

Так же на декларативный смысл программы может повлиять перестановка правил, содержащих отсечения.

Отсечения, которые не затрагивают декларативный смысл программы, называются *зелеными*. Отсечения, меняющие декларативный смысл программы называются *красными*. Их следует применять с большой осторожностью.

Часто отсечение является необходимым элементом программы - без него она правильно не работает.

Сформулируем более точно, как работает механизм отсечений. Пусть имеется правило вида: $H:-B_1,\ldots,B_k,!,\ldots,B_n$. Если цели $B_1,\ldots,B_k$ успешны, то это решение замораживается, и другие альтернативы для этого решения больше не рассматриваются (отсекается правая часть дерева решений, которая находится выше $B_1,\ldots,B_k$).

Можно выделить 3 основных случая использования отсечения:

1. Указание интерпретатору Пролога, что найдено *необходимое правило* для заданной цели.
2. Указание интерпретатору Пролога, что необходимо *немедленно прекратить* доказательство конкретной цели, не пытаясь рассматривать какие-либо альтернативы.
3. Указание интерпретатору Пролога, что в ходе перебора альтернативных вариантов найдено *необходимое решение*, и нет смысла вести перебор далее.

Пример 3 Пусть необходимо написать правила для вычисления суммы ряда натуральных чисел 1, 2, ... *N*. Запишем правила:

```
sum(1,1).
sum(N,S):-N1 is N-1, sum(N1,S1), S is S1+N.
```

На вопрос:

```
?-sum(2,X).
```

получим ответ

```
X=3
```

При попытке продолжить поиск других решений после нажатия клавиши Enter получим: «ERROR: Out of local stack» – бесконечную рекурсию! (рис. 64).

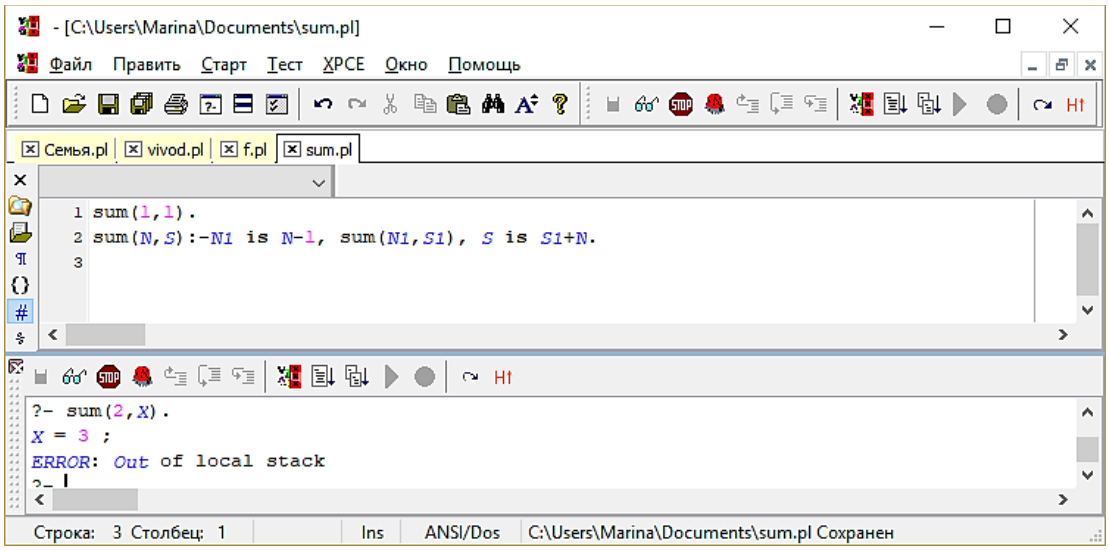


Рис.64. Бесконечная рекурсия при попытке продолжить поиск решений

Для исправления ситуации необходимо модифицировать первое правило следующим образом (это пример первого случая использования отсечения):

```
sum(1,1):-!.
```

Теперь попробуем такой вопрос:

```
?-sum(-5,X).
```

В результате - опять бесконечная рекурсия! (рис. 65).

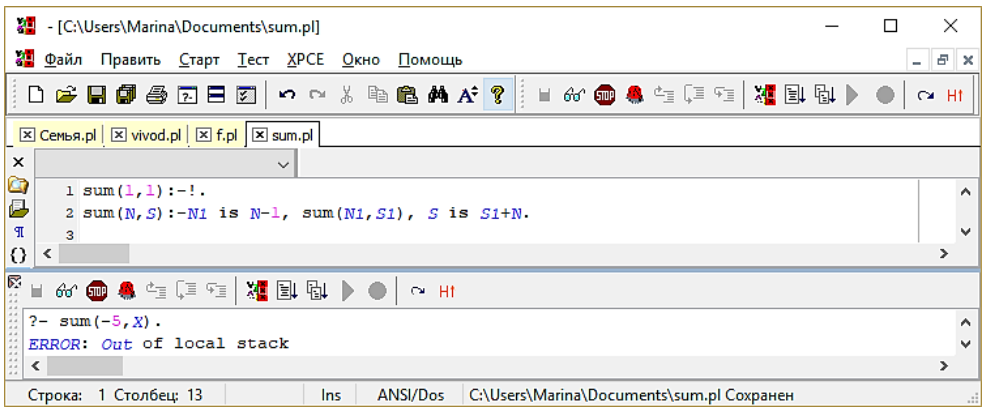


Рис.65. Бесконечная рекурсия при попытке продолжить поиск решений

Поэтому перед имеющимися двумя правилами добавляем еще одно утверждение (это пример второго случая использования отсечения). Окончательный вариант для нахождения суммы:

```
sum(N,_):-N=<0,!, fail.
sum(1,1):-!.
sum(N,S):-N1 is N-1, sum(N1,S1), S is S1+N.
```

3.10 Циклы, управляемые отказом

В SWI Prolog определен встроенный предикат repeat без аргументов с помощью двух правил:

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

Первое правило всегда успешно, поскольку не содержит подцелей. Однако, поскольку имеется второй вариант правила для этого предиката, Пролог-система запоминает в качестве точки возврата второе правило. Попытка применить второе правило так же всегда успешна, т.к. первое правило удовлетворяет подцели второго правила. Таким образом, предикат repeat всегда успешен. С помощью этого предиката легко организовать цикл «до тех пор, пока», записав правило вида:

```
<голова правила>:- repeat,
                   <тело цикла>,
                   <условие выхода>,!.
```

Отсечение в качестве завершающей подцели гарантирует остановку цикла в случае выполнения условия выхода из цикла.

Пример: Напишем программу, которая считывает слово, введенное с клавиатуры, и дублирует его на экран до тех пор, пока не будет введено слово «stop».

```
цель:- write('Введите слово для повтора: '),nl, эхо.
эхо:- repeat,
      read(Slovo),
      write(Slovo),nl,
      проверка(Slovo),!.
проверка(stop):-write('Конец '),nl.
проверка(_):-fail.
```

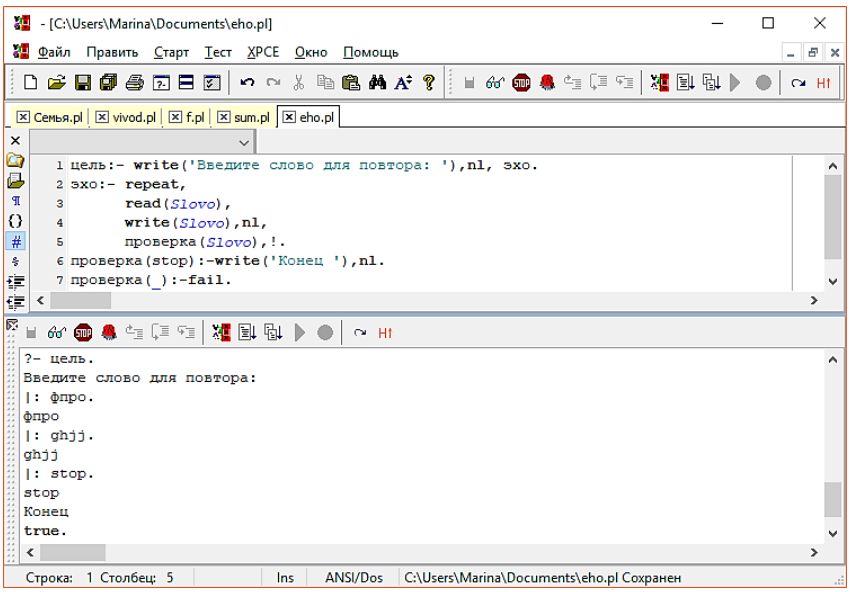


Рис.67. Предикат с циклом с отказами

Такие циклы используются для описания взаимодействия с внешней системой путем повторяющегося ввода или вывода. В таком цикле обязательно должен быть предикат (в примере – это repeat), приводящий к безуспешным вычислениям. Рекурсивные циклы предпочтительнее циклов, управляемых отказами, поскольку последние не имеют логической интерпретации. Но на практике такие циклы необходимы при выполнении большого объема вычислений, поскольку рекурсия требует много памяти.

3.11 Списки

Список – это упорядоченный набор объектов. Объектами списка могут быть любые термы: целые числа, действительные числа, символы, строки, структуры. Список может содержать объекты разных типов. Список может быть объектом списка. Элементы списка разделяются запятой и заключаются в квадратные скобки. Пустой список не содержит элементов и обозначается [].

Пример: [1,2,-3,4] – список целых чисел;
[энн,боб,лиз] – список символических имен;
[[a,b],[c,g,l],[ ]] – список из списков символических имен.

3.11.1 Голова и хвост списка

Для более удобной работы со списком, так же как в Лиспе, непустой список делится на хвост и голову. *Головой* называется первый элемент списка, *хвостом* – часть списка без первого элемента.

Пример 1 Деление списка на хвост и голову.

Список	Голова	Хвост
[1,2,3,4]	1	[2,3,4]
[a]	a	[ ]
[ ]	не определена	не определен

Для деления списка на хвост и голову в Прологе имеется специальная форма представления списка: [Head|Tail]. При сопоставлении конкретного списка с

такой формой, Head сопоставляется с головой списка, а Tail – с хвостом списка. Таким образом, одновременно определяются голова и хвост списка.

Пример 2 Рассмотрим результаты сопоставления списков.

Список 1	Список 2	Результаты сопоставления
[X,Y,Z]	[1,2,3]	X=1, Y=2, Z=3
[5]	[X|Y]	X=5, Y=[ ]
[1,2,3,4]	[X,Y|Z]	X=1, Y=2, Z=[3,4]
[1,2,3]	[X,Y]	нет решений
[a,X|Y]	[Z,a]	Z=a, X=a, Y=[ ]

3.11.2 Предикаты для работы со списками

Принадлежность элемента списку

Определим предикат member1, определяющий принадлежность терма списку. Определение будет таким же, как в Лиспе: терм совпадает с головой списка, либо принадлежит хвосту списка.

```
member1(X,[X|_]).
member1(X,[_|Tail]):-member1(X,Tail).
```

Предикат member1 имеет много интересных приложений, приведенных на рис. 68 (а, б, в).

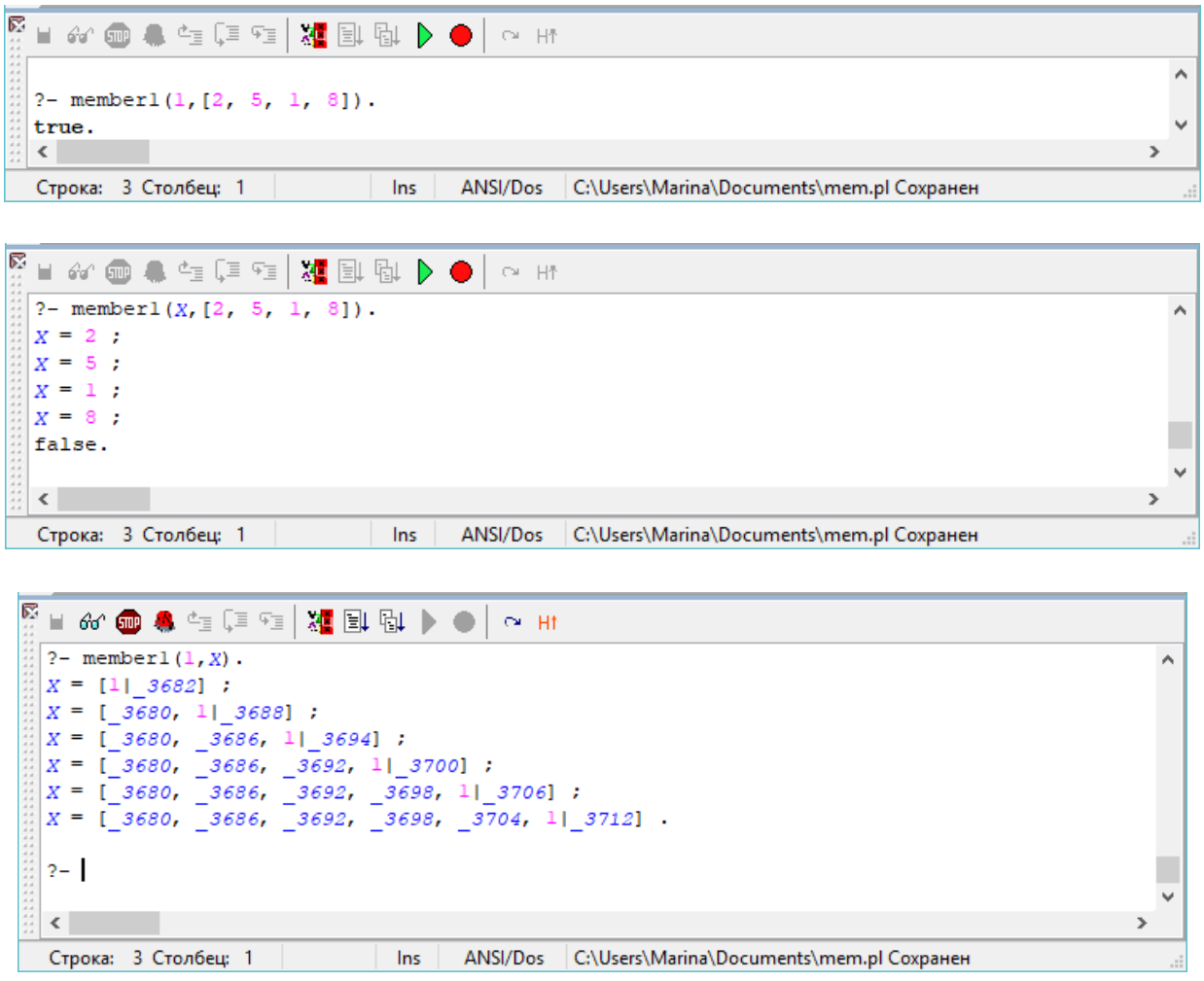


Рис.68. Применение предиката member1

Последний вопрос выглядит несколько странным (какому списку принадлежит 1?), но, тем не менее, имеется ряд программ, основанных на таком применении отношения.

В SWI Prolog имеется встроенный предикат member.

Соединение двух списков

Определим предикат append1, соединяющий два списка в один путем добавления после элементов первого списка всех элементов второго списка. Определим этот предикат следующим образом: головой результирующего списка будет голова первого списка, а хвостом - результат соединение хвоста первого списка со вторым; если первый список пуст, то результат соединения – второй список.

```
append1([ ],L,L).
append1([Head|Tail],L,[Head|Tail1]):-append1(Tail,L,Tail1).
```

Как и в случае с member1 существуют разнообразные применения предиката append1:

- слияние двух списков;
- получение всех возможных разбиений списка;
- поиск подсписков до и после определенного элемента;
- поиск элементов списка, стоящих перед и после определенного элемента;
- удаление части списка, начиная с некоторого элемента;
- удаление части списка, предшествующей некоторому элементу.

Примеры использования предиката append1 приведены в таблице.

Вопрос в Пролог-системе	Ответ Пролог-системы
append1([1,2],[3],L).	L=[1,2,3].
append1(L1,L2,[1,2,3]).	L1=[ ],L2=[1,2,3]; L1=[1],L2=[2,3]; L1=[1,2],L2=[3]; L1=[1,2,3],L2=[ ]; false
append1(Before,[3|After],[1,2,3,4,5]).	Before=[1,2],After=[4,5]; false
append1(_,[Before,3,After|_],[1,2,3,4,5]).	Before=2,After=4; false
append1(L1,[3|_],[1,2,3,4,5]).	L1=[1,2]; false
append1(_,[3|L2],[1,2,3,4,5]).	L2=[4,5]; false

Добавление и удаление элемента из списка

Определим предикат delete1, который будет удалять первое вхождение указанного элемента в список. Определим предикат следующим образом: если элемент совпадает с головой списка, то результат удаления – хвост списка, иначе результатом удаления является список, у которого голова совпадает с исходным

списком, а хвост - результат удаления указанного элемента из хвоста исходного списка.

Определим предикат insert, который будет добавлять указанный элемент в голову списка. Определим предикат следующим образом: головой результирующего списка является указанный элемент, а хвостом – исходный список.

```
insert(X,L,[X|L]).
delete1([ ], _, [ ]):-!.
delete1 ([Y|Tail], Y,Tail):-!.
delete1( [Z|Tail], Y, [ Z|Tail1]):-delete1(Tail, Y,Tail1).
```

Заметим, что если необходимо удалить все вхождения указанного элемента, то второе правило для предиката delete1 следует изменить следующим образом:

```
delete1 ([Y|Tail], Y, L):-delete1(Tail, Y, L),!.
```

В SWI Prolog имеется встроенный предикат delete, который удаляет все вхождения заданного параметром элемента.

Для удаления одного вхождения элемента можно использовать предикат

select(<элемент>,<список1>,<список2>),

который удаляет одно вхождение заданного элемента из списка1, результат – список2.

Пример 3

```
?- select(1,[2,1,3,1,4],L).
L = [2, 3, 1, 4];
L = [2, 1, 3, 4];
false
```

Так же этот предикат можно использовать для добавления элемента в список.

Пример 4

```
?- select(0,L,[2,1,3]).
L = [0, 2, 1, 3];
L = [2, 0, 1, 3];
L = [2, 1, 0, 3];
L = [2, 1, 3, 0];
false
```

Деление списка на два списка по разделителю

Определим предикат split, который будет делить список на две части, используя разделитель M. Определим предикат следующим образом: если элемент исходного списка меньше разделителя, то он помещается в первый результирующий список, иначе – во второй результирующий список.

```
split(M,[Head|Tail],[Head|L1],L2):-Head@<M,!,split(M,Tail,L1,L2).
split(M,[Head|Tail],L1,[Head|L2]):-split(M,Tail,L1,L2).
split(_,[ ],[ ],[ ]).
```

Пример 5 Применение предиката split.

```
?- split(4,[5,1,6,3,4],L1,L2).
L1 = [1, 3]
L2 = [5, 6, 4] ;
false
```

Предикат split так же можно использовать для слияния списков.

Подсчет количества элементов в списке

Определим предикат count, который находит количество элементов в списке. Определим предикат следующим образом: количество элементов в списке на 1 больше количества элементов в хвосте списка, в пустом списке содержится 0 элементов.

```
count([ ],0).
count([_|Tail],N):-count(Tail,N1),N is N1+1.
```

В SWI Prolog имеется встроенный предикат length, который находит количество элементов в списке.

Может оказаться полезным использование предиката reverse(<список1>,<список2>) для обращения любого из двух аргументов.

3.11.3 Сортировка списков

До сих пор средства Пролога использовались при записи простых процедур и примеров. Рассмотрим более сложные примеры написания программ на Прологе для различных методов сортировки. Сортировку списка будем выполнять по неубыванию.

Сортировка вставкой

Этот вид сортировки наиболее удобно реализуется на Прологе. Алгоритм этой сортировки можно описать следующим образом: сортируем хвост и добавляем голову списка в нужное место отсортированного хвоста. Нам понадобится определить дополнительный предикат для добавления элемента в отсортированный список таким образом, чтобы не нарушалась его упорядоченность.

```
in_sort([ ],[ ]).
in_sort([X|Tail],Sort_list):-in_sort(Tail,Sort_tail), insert(X,Sort_tail,Sort_list).
insert(X,[Y|Sort_list],[Y|Sort_list1]):-X@>Y,!,insert(X,Sort_list,Sort_list1).
insert(X,Sort_list,[X|Sort_list]).
```

Пузырьковая сортировка

При пузырьковой сортировке меняем местами соседние элементы до тех пор, пока есть неверно упорядоченные пары. Если таких пар нет, то список отсортирован. Для перестановки пары соседних элементов в списке будем использовать предикат swap (он успешен, если такая пара нашлась).

```
pu_sort(L,Sort_list):-swap(L,L1),!, pu_sort(L1,Sort_list).
pu_sort(L,L).
swap([X,Y|Tail],[Y,X|Tail]):-X@>Y.
swap([X|Tail],[X|Tail1]):-swap(Tail,Tail1).
```

Быстрая сортировка

Процедуры пузырьковой сортировки и сортировки вставкой просты, но не эффективны. Среднее время сортировки для этих процедур пропорционально n^2. Поэтому, для длинных списков используют алгоритм быстрой сортировки. При быстрой сортировке сначала разбиваем список на два списка по разделителю – голове списка, затем упорядочиваем новые списки и соединяем их. Если новые списки получаются примерно одинаковой длины, то временная сложность алгоритма быстрой сортировки будет примерно $n \cdot \log n$. Если же длины списков сильно различаются, то сложность будет порядка n^2. В среднем время быстрой сортировки ближе к лучшему случаю, чем к худшему. В программе будем использовать определенный в п.3.6.4 предикат split.

```
q_sort([Head|Tail],Sort_list):- split(Head,Tail,Less,More),
                                q_sort(Less,Sort_less),
                                q_sort(More,Sort_more),
                                append(Sort_less,[Head|Sort_more],Sort_list).
q_sort([ ],[ ]).
```

В SWI-Prolog имеются встроенные предикаты sort и msort, которые сортируют список по возрастанию, причём первый предикат удаляет повторяющиеся элементы.

3.11.4 Компоновка данных в список

Иногда при программировании на Прологе возникает необходимость собрать данные из предикатов в список для их последующей обработки.

1. Предикат bagof (набор), обращение к которому имеет вид: bagof(X,P,L) порождает список L всех объектов X, удовлетворяющих цели P.

Обычно предикат bagof имеет смысл применять только тогда, когда X и P содержат общие переменные.

Пример 6 Определим предикат класс для разбиения букв из некоторого множества на гласные и согласные.

```
класс(a,глас).
класс(b,согл).
класс(c,согл).
класс(d,согл).
класс(e,глас).
класс(f,согл).
```

Тогда мы можем получить список всех согласных, упомянутых в этих предложениях:

```
?- bagof( Буква, класс(Буква,согл), Буквы).
Буквы = [d, c, d, f].
```

Если же мы в указанной цели оставим класс букв неопределенным, то, получим два списка букв: гласные и согласные.

```
?- bagof( Буква, класс(Буква,Класс), Буквы).
Класс = глас
Буквы = [a, e] ;
```

```
Класс = согл
Буквы = [b, c, d, f].
```

Если bagof(X, P, L) не находит ни одного решения для P, то цель bagof просто терпит неуспех. Если один и тот же X найден многократно, то все его экземпляры будут занесены в L, что приведет к появлению в L повторяющихся элементов.

2. Предикат setof работает аналогично предикату bagof. Цель setof(X,P,L) как и раньше, порождает список L объектов X, удовлетворяющих P. При этом, список L упорядочен и не содержит повторяющихся элементов. Упорядочение происходит по алфавиту или по отношению '<', если элементы списка - числа. Если элементы списка - структуры, то они упорядочиваются по своим главным функторам. Если же главные функторы совпадают, то решение о порядке таких термов принимается по их первым несовпадающим функторам, расположенным выше и левее других (по дереву). На вид объектов, собираемых в список, ограничения нет. Например, для составления списка пар вида Класс/Буква составим цель:

```
?- setof( Класс/Буква, класс( Буква, Класс), Спис).
Спис = [глас/a, глас/e, согл/b, согл/d, согл/f, согл/c]
```

3. Предикат findall, аналогичен предикату bagof. Он тоже порождает список объектов, удовлетворяющих P. Отличие от bagof в том, что findall собирает в список все объекты X, не обращая внимание на возможно отличающиеся для них конкретизации тех переменных из P, которых нет в X. Это различие видно из следующего примера:

```
?- findall( Буква, класс(Буква,Класс), Буквы).
Буквы = [a, b, c, d, e, f].
```

Если не существует ни одного объекта X, удовлетворяющего P, то findall все равно успешен и возвращает L = [ ].

Пример 7 Пусть имеются сведения о количестве детей у определенных лиц (предикат information). Требуется найти общее количество детей в этой группе людей.

```
goal:- findall(Kolvo, information(_,Kolvo),L),
       /*Элементы списка L – количества детей */
       all(L,S),
       write(S),nl.
information (иванов,2).
…….
information (петров,1).
all([ ],0).
all([Head|Tail],S):-all(Tail,S1),S is S1+Head.
```

3.11.5 Решение логических задач с использованием списков

Пример 1 Рассмотрим широко известную логическую задачу о фермере, волке, козе и капусте. Задача заключается в следующем. Фермер (**farmer**), волк (**wolf**) , коза (**goat**) и капуста (**cabbidge**) находятся на одном берегу. Всем надо

перебраться на другой берег на лодке. Лодка перевозит только двоих. Нельзя оставлять на одном берегу козу и капусту (коза съест капусту), козу и волка (волк съест козу).

Главная проблема в формировании алгоритма - найти эффективное представление структурой данных информации о задаче. Процесс перевозки может быть представлен последовательностью состояний state с 4 аргументами, каждый из которых может принимать два значения: right – на левом берегу, left – на правом берегу и отражает соответственно нахождение фермера, волка, козы и капусты. Например, state(left,right,left,right) означает, что фермер и коза находятся на левом берегу, а волк и капуста – на правом.

Фермер может перевести на другой берег кроме себя либо волка, либо козу, либо капусту. Для описания возможных вариантов перевозки (переходов из одного состояния в другое) напишем предикат move. Для того чтобы можно было переправляться с левого берега на правый и наоборот, определим предикат opposite, который определяет сторону, противоположную исходной. Например, перевозка фермером на другой берег волка будет иметь вид:

```
move(state(X,X,G,C),state(Y,Y,G,C)):-opposite(X,Y).
```

С помощью предиката danger определим опасные состояния, когда коза может съесть капусту или волк съесть козу.

Наконец, определим предикат path, который сформирует список из последовательности состояний, решающих поставленную задачу. Добавляем в список новое состояние, если в него возможен переход из текущего состояния, оно не опасно и не содержится в уже сформированной последовательности состояний (чтобы не было зацикливания)

Итак, окончательный вариант программы будет иметь вид:

```
goal:-S=state(left,left,left,left),
        G=state(right,right,right,right),
        path(S,G,[S],L),reverse(L,L1),
        nl,writeln('A solution is:'),
        writeln(L1).
move(state(X,X,G,C),state(Y,Y,G,C)):-opposite(X,Y).
move(state(X,W,X,C),state(Y,W,Y,C)):-opposite(X,Y).
move(state(X,W,G,X),state(Y,W,G,Y)):-opposite(X,Y).
move(state(X,W,G,C),state(Y,W,G,C)):-opposite(X,Y).
opposite(right,left).
opposite(left,right).
danger(state(F,_,X,X)):-opposite(F,X).
danger(state(F,X,X,_)):-opposite(F,X).
path(G,G,T,T):-!.
path(S,G,L,L1):- move(S,S1),
                 not(danger(S1) ),
                 not(member(S1,L)),
                 path(S1,G,[S1|L],L1),!.
```

Данная задача имеет два решения. Например, одно из решений будет иметь вид:

[state(left,left,left,left), state(right,left,right,left), state(left,left,right,left), state(right,right,right,left), state(left,right,left,left), state(right,right,left,right), state(left,right,left,right), state(right,right,right,right)]

Это соответствует следующему алгоритму:

1. Фермер перевозит козу с левого берега на правый.
2. Фермер перемещается с правого берега на левый.
3. Фермер перевозит волка с левого берега на правый.
4. Фермер перевозит козу с правого берега на левый.
5. Фермер перевозит капусту с левого берега на правый.
6. Фермер перемещается с правого берега на левый.
7. Фермер перевозит козу с левого берега на правый.

Пример 2 Решение числового ребуса.

```
  AFAFA
+ SLEEPY
 ETUDES
```

Задача состоит в том, чтобы заменить все буквы цифрами. Одинаковым буквам должны соответствовать одинаковые цифры, а разным буквам – разные цифры.

Определим предикат sum(N1,N2,N), который истинен, если существует такая замена букв цифрами в словах N1, N2, N, что N1+N2=N. Числа будем представлять списком цифр, из которых состоит число. Без ограничения общности можно считать, что все три списка, представляющие числа имеют равную длину. Если числа разной длины, то всегда можно приписать нужное количество нулей более «коротким» числам. В правилах для предиката sum необходимо описать правила суммирования в десятичной системе счисления. Суммирование производится справа налево с учетом цифры переноса с предыдущего разряда. Следовательно, при суммировании очередных цифр нужна дополнительная информация о переносе из предыдущего разряда и о переносе в следующий разряд. Поскольку разным буквам должны соответствовать разные цифры, то при суммировании цифр нужна информация о цифрах, доступных до и после сложения (эта информация будет представляться двумя списками). В связи с вышесказанным определим вспомогательный предикат sum1 с дополнительными параметрами: sum1(N1,N2,N, P_before,P_after, Cifri_before,Cifri_after), где P_before – перенос из правого разряда до сложения, P_after - перенос в левый разряд после сложения, Cifri_before – список цифр, которые могут быть использованы для конкретизации переменных до сложения, Cifri_after – список цифр, которые могут быть использованы для конкретизации переменных после сложения. Тогда можно описать связь предиката sum и sum1 следующим образом:

```
sum(N1,N2,N):-sum1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_).
```

Определим отношение sum1 следующим образом:

– Если у всех трех чисел имеется хотя бы одна цифра (в списках можно выделить головы), то суммируем числа без самой левой цифры и, используя список оставшихся цифр и перенос из предыдущего разряда, суммируем самые левые

цифры и добавляем полученную сумму в результат суммирования без самой левой цифры. Для суммирования цифр определим предикат sumc.

– Если все три списка пустые, то переносов разрядов нет, и списки цифр до и после сложения совпадают. Это соответствует правилу остановки рекурсии: sum1([],[],[],0,0,Cifri,Cifri).

Получаем следующие правила:

```
sum1([],[],[],0,0,L,L).
sum1([D1|N1],[D2|N2],[D|N],C1,C,L1,L):- sum1(N1,N2,N,C1,C2,L1,L2),
                                        sumc(D1,D2,D,C2,C,L2,L).
```

Осталось определить правила для предиката sumc. Первые три аргумента предиката должны быть цифрами. Если какая-нибудь из этих переменных не конкретизирована, то ее необходимо конкретизировать какой-нибудь цифрой из списка L2. После конкретизации цифру следует удалить из списка доступных цифр. Если переменная уже имела значение, то удалять из списка доступных цифр ничего не надо.

```
sumc(D1,D2,D,C2,C,L2,L):-
                        delete(D1,L2,L3),
                        delete(D2,L3,L4),
                        delete(D,L4,L),
                        S is D1+D2+C2,
                        D is S mod 10,
                        C is S//10.
```

Для удаления цифры из списка или для получения значения неозначенной переменной будем использовать предикат delete.

```
delete(X,L,L):-nonvar(X),!.
delete(X,[X|L],L).
delete(X,[Y|L],[Y|L1]):-delete(X,L,L1).
```

Итак, окончательный вариант программы будет иметь вид:

```
sum(N1,N2,N):-sum1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_).
sum1([],[],[],0,0,L,L).
sum1([D1|N1],[D2|N2],[D|N],C1,C,L1,L):-sum1(N1,N2,N,C1,C2,L1,L2),
                                        sumc(D1,D2,D,C2,C,L2,L).
sumc(D1,D2,D,C2,C,L2,L):-delete(D1,L2,L3),
                         delete(D2,L3,L4),
                         delete(D,L4,L),
                         S is D1+D2+C2,
                         D is S mod 10,
                         C is S//10.
delete(X,L,L):-nonvar(X),!.
delete(X,[X|L],L).
delete(X,[Y|L],[Y|L1]):-delete(X,L,L1).
```

Для решения заданного в начале пункта ребуса следует ввести цель:

```
?- sum([0,A,P,A,P,A], [S,L,E,E,P,Y], [E,T,U,D,E,S]).
A = 2
P = 8
```

```
S = 5
L = 7
E = 6
Y = 3
T = 0
U = 4
D = 9 ;
false
```

Решение другого ребуса:

```
   БАЙТ
+  БАЙТ
 СЛОВО
?- sum([0,Б,А,Й,Т], [0,Б,А,Й,Т], [С,Л,О,В,О]).
Б = 3
А = 6
Й = 4
Т = 1
С = 0
Л = 7
О = 2
В = 8;
…….
false
```

Этот ребус имеет 14 решений.

3.12 Строки

Под строкой (в SWI-Prolog) будет пониматься последовательность символов, заключенная в апострофы. Заметим, что текст в двойных кавычках является списком кодов символов, а не строкой.

Рассмотрим предикаты для работы со строками:

1. string_length(S, L) – определяет длину строки S.
2. string_concat(S1,S2,S3) – соединяет две строки S1 и S2 в третью S3. Предикат можно использовать для разбиения строки S3.
3. sub_string(S,K,N,R,L) – выделяет в строке S подстроку L, которая начинается с K-го элемента и содержит N символов. R – количество символов, стоящих в S после подстроки L. Нумерация элементов строки начинается с 0.
4. string_chars(S,L) – преобразует строку S в список символов.
5. name(S,L) – преобразует строку S в список кодов символов и наоборот.
6. char_code(C,K) – преобразует символ C в его код K и наоборот.
7. atomic_list_concat(L,R,S) – преобразует список L в строку S, используя R как разделитель и наоборот.
8. split_string(S,R,D,L) – преобразует строку S в список подстрок L, используя R как разделитель, удаляя из начала и конца подстрок символы строки D.

9. atom_string(A,S) – преобразует атом A в строку S и наоборот.

10.number_string(N,S) – преобразует число N в строку S и наоборот.

Пример 1 Напишем предикат str_list(S,L), который будет преобразовывать строку S в список символов L, удаляя пробелы.

```
str_list(S,L):- string_chars(S,L1),
                delete(L1, ' ',L).
```

Пример 2 Напишем предикат, который считает количество элементов, равных C в списке L.

```
goal:-read(S),read(C),name(S,L),
      char_code(C,K),char_count(L,K,N),writeln(N).
char_count([ ],_,0):-!.
char_count([K|T],K,N):- char_count(T,K,N1),
                        N is N1+1,!.
char_count([_|T],K,N):- char_count(T,K,N).
```

3.13 Предикаты для работы с файлами

В каждый момент выполнения программы лишь два файла могут быть «активными»: один для ввода, другой - для вывода. Эти два файла называются *текущим входным потоком* и *текущим выходным потоком* соответственно. Текущим входным потоком данных по умолчанию является клавиатура, а выходным - экран.

Рассмотрим предикаты для работы с файлами:

1. exists_file(<‘имя файла’>) завершается успешно, если файл с указанным именем существует.

2. open(<’имя файла’>, <режим>,<файловая переменная>) – открытие файла для чтения, записи или добавления. Файл можно открывать в следующих режимах:

- для чтения (режим read);
- для записи (режим write);
- для добавления (режим append).

Обработка файлов осуществляется последовательно. Следует иметь в виду, что предикаты для работы с файлами являются внелогическими, т.к. не дают альтернативных решений при откате (повторно считать то же самое из файла в случае неуспешной обработки данных нельзя). Поэтому чтение из файла и обработка считанного должна производиться отдельно!

3. set_input(<файловая переменная>),
   set_output(<файловая переменная >) – для перенаправления ввода в файл или вывода из файла.

4. close(<файловая переменная >) – закрытие файла.

5. see(<'имя файла'>),
   tell(<'имя файла'>) – используют для открытия и перенаправления ввода из файла или вывода в файл вместо 2 и 3.

При перенаправлении ввода на клавиатуру, а вывода на экран в качестве имени файла используют имя user.

6. seen,
told – закрытие файлов, открытых с помощью see и tell.
7. at_end_of_stream успешно завершается, если найден конец файла.
8. read_line_to_codes(F,L) читает строку из входного потока F и преобразует ее в список кодов символов этой строки.
9. read_stream_to_codes(F,L) читает содержимое из входного потока F (до конца файла) и преобразует его в список кодов символов.

Пример 1 Напишем предикат, который выводит на экран строки из файла, начиная с некоторого номера. Имя файла и номер строки вводятся с клавиатуры.

```
goal:- write('Введите имя файла: '),
       read(Filename),
       check_exist(Filename),
       write('Введите номер строки для вывода строк из файла на экран '),
       read(N),
       open(Filename,read,F),
       set_input(F),
       read_file(F,N),
       write('Содержимое файла, начиная с '),
       write(N),writeln(',-ой строки:'),
       write_screen(F),
       close(F).
check_exist(Filename):-exists_file(Filename),!.
check_exist(_):-writeln('Такого файла нет'),
                fail.
read_file(_,N):-at_end_of_stream,!,
                write('В файле меньше, чем '),
                write(N),writeln('строк'),
                fail.
read_file(_,1):-!.
read_file(F,N):-read_line_to_codes(F,_),
                N1 is N-1,
                read_file(F,N1).
write_screen(_):- at_end_of_stream,!.
write_screen(F):- read_line_to_codes(F,L),
                  string_to_list(S,L),
                  writeln(S),
                  write_screen(F).
```

Пример 2 Напишем предикат write_to_file, который записывает вводимые с клавиатуры строки в файл t.txt. Окончание ввода – строка ‘#’.

```
write_to_file:-read(X),
               tell('t.txt'),
               write_s(X),
               told,
```

```
                write('Данные записаны в файл').
write_s('#'):-!.
write_s(X):-writeln(X),
            read(Y),
            write_s(Y).
```

3.14 Динамические базы данных

Реляционная модель базы данных описывает множество отношений. Программа на Прологе представляет собой набор фактов и правил, поэтому ее можно рассматривать как реляционную базу данных. Отношения базы данных – это предикаты, атрибутами отношений являются объекты предикатов, элементы отношений присутствуют в виде фактов (явно) и правил (неявно), мощности отношений определяются количеством фактов и правил для каждого предиката. Иногда в процессе работы программы возникает необходимость изменить, удалить или добавить некоторые предложения (факты или правила). Такие предложения являются частью динамической базы данных. Предикаты из динамической базы данных сопровождаются директивой:
:-dynamic <имя предиката>/<арность>.

Если динамическими являются несколько предикатов, то они перечисляются через запятую.

При запуске программы предложения из динамической базы данных могут быть изменены во время работы программы. Следует иметь в виду, что динамическая база данных сохраняется в оперативной памяти во время всего сеанса работы с Прологом. Поэтому, если эту память не очищать после работы программы, при повторном запуске программы можно получить неожиданные результаты.

3.14.1 Добавление и удаление предложений

Для добавления предложения (факта или правила) для предиката динамической базы данных в начало или конец базы данных используются соответственно предикаты:
asserta(<предложение>), assertz(<предложение >).

Можно использовать предикат assert(<предложение>), который работает аналогично assertz. Он добавлен для совместимости с другими версиями Пролога. Эти предикаты всегда успешны.

Например, можно выполнить добавление правила:
asserta((мать(X,Y):-родитель(X,Y),женщина(X))).

Внесенные таким образом предложения работают так же, как первоначальная программа. Новые предикаты, добавляемые с помощью asserta и assertz, становятся динамическими по умолчанию.

Для удаления из динамической базы данных первого предложения, сопоставимого с F, используется предикат retract(F).

Пример 1 На рис. 69 изображено добавление и удаление из динамической базы данных предиката кризис.

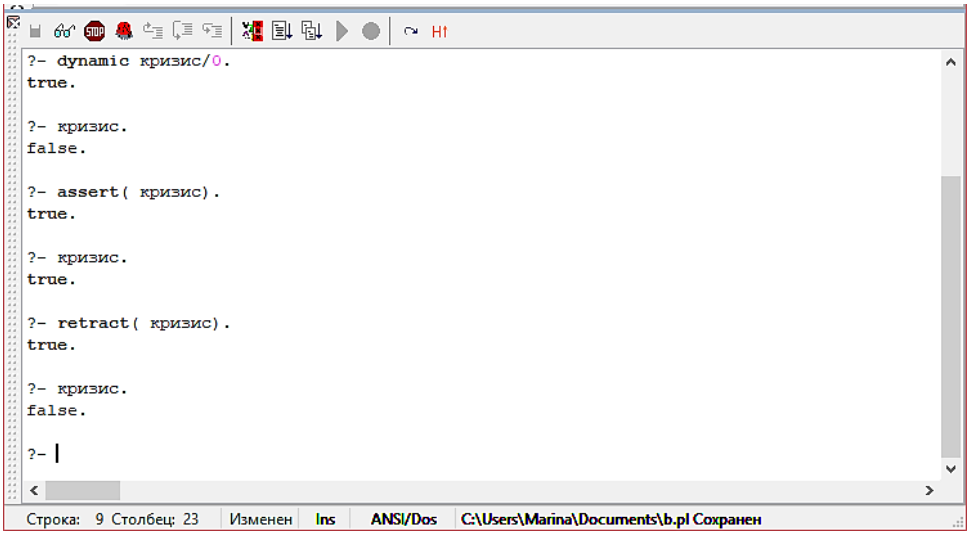


Рис.69. Работа с динамической базой данных

Предложения, добавленные к программе таким способом, ведут себя точно так же, как и те, что были в «оригинале» программы. Следующий пример показывает, как с помощью assert и retract можно работать в условиях изменяющейся обстановки.

Пример 2 Пусть имеется программа о погоде:

```
:-dynamic солнце, дождь,туман.
хорошая:- солнце, not(дождь).
необычная:- солнце, дождь.
отвратительная:- дождь, туман.
дождь.
туман.
```

На рис. 70 приведен диалог с программой, во время которого база данных постепенно меняется.

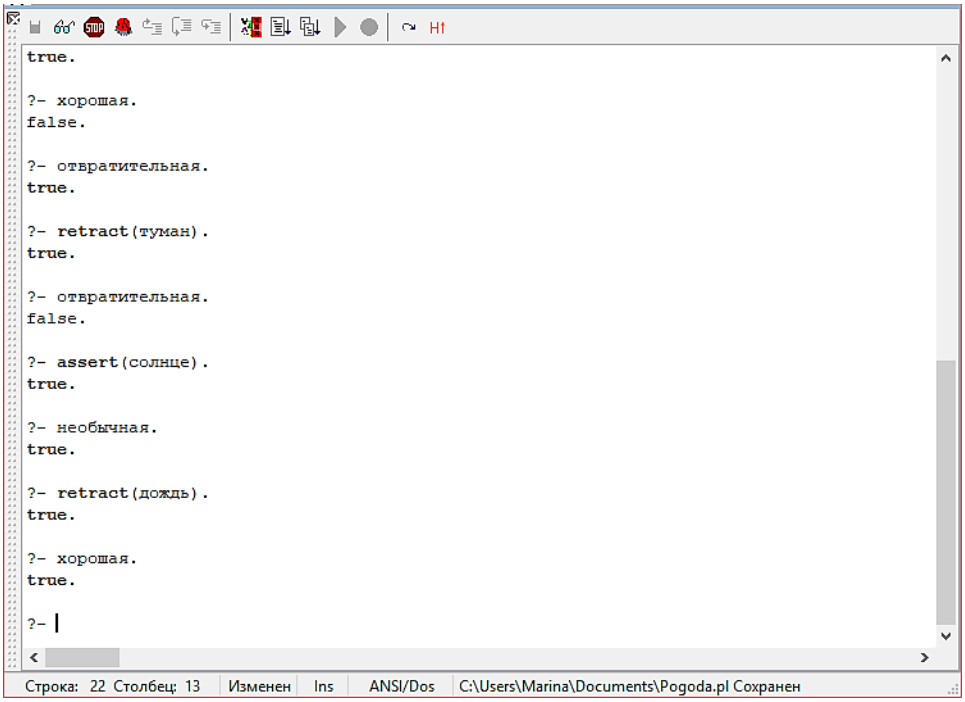


Рис.70. Изменение погоды при работе с динамической базой данных

Вывести все предложения базы данных, относящиеся к определенному предикату, в текущий выходной поток можно с помощью предиката

listing(<имя предиката>/<арность>).

Предикат listing без аргументов выводит все содержимое базы данных. В том числе, там хранится и сам код программы.

Пример 3 Напишем предикат delete_person, который будет выводить на экран фамилии тех людей, возраст которых больше 50 лет. Фамилия и возраст человека являются объектами предиката person динамической базы данных. После вывода на экран фамилии, соответствующий факт удаляется из динамической базы данных.

```
:-dynamic person/2.
person('Иванов',25).
person('Петров',52).
person('Сидоров',54).
person('Захаров',60).
person('Певцов',34).
delete_person:-person(Family,Age),
                Age>50,
                writeln(Family),
                 retract(person(Family,Age)),
                 fail.
delete_person.
```

На рис. 71 изображен вывод динамической базы данных после работы предиката delete_person. Можно видеть, что в динамической базе данных остались только факты, касающиеся сотрудников с возрастом до 50 лет.

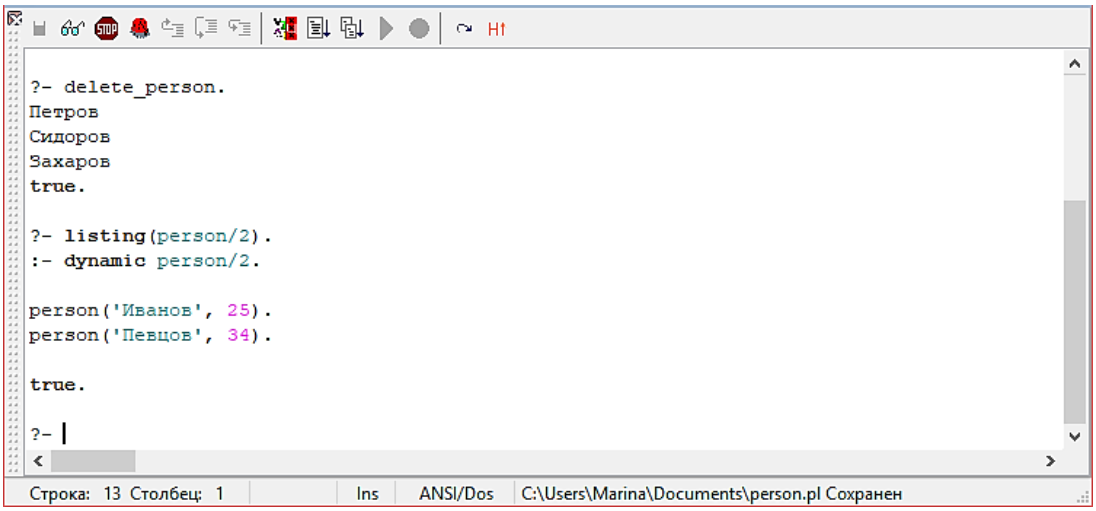


Рис.71. Вывод на экран динамической базы данных

Для модификации динамической базы данных можно использовать предложения, содержащие предикаты asserta, assertz и retract. Например, чтобы отредактировать имеющееся в динамической базе данных предложение, в программе необходимо составить отредактированное утверждение, удалить из базы данных старое утверждение и занести новое.

Для удаления всех предложений, сопоставимых с F, используется предикат retractall(F). Этот предикат всегда успешен, но в отличие от retract(F), не позволяет получить значения объектов из удаленных предложений.

Пример 4 Для удаления из динамической базы данных примера3 всех фактов, касающихся 20-летних людей, можно воспользоваться предикатом:

```
retractall(person(_,20)).
```

3.14.2 Заполнение динамической базы данных фактами из файла, сохранение динамической базы данных в файле

Предикат consult('<имя файла>') считывает из файла предложения и добавляет их в конец динамической базы данных (аналогично assertz). Между consult и assertz существует связь. Обращение к файлу при помощи consult можно в терминах assertz определить так: считать все термы (предложения) файла и добавить их в конец базы данных.

Сохранение предложений предиката динамической базы данных в файл обеспечивает предикат listing. Перед применением этого предиката следует перенаправить вывод в файл, например, с помощью предиката tell. При этом если указанный файл уже существовал, то он создается заново.

Пример 5 Формирование динамической базы данных «Читатель библиотеки» с клавиатуры и сохранение ее в файле.

```
:-dynamic читатель/3.
 goal:- writeln(‘Формирование б.д.’),
        repeat,
        writeln('Будете вводить новые факты? y/n'),
        read(A),
        ответ(A),
        tell('reader.txt'),
        listing(читатель/3),
        told.
ответ(n).
ответ(y):-запись_в_базу,
          fail.
ответ(_):-fail.
запись_в_базу:-write('Фамилия: '),
                    read(Name),
                     write('Номер билета: '),
                     read(Number),
                     write('Число посещения: '),
                     read(Data),
                     write('Месяц посещения: '),
                     read(Mounth),
                  assertz(читатель(Name,Number,дата_посещ(Data,Mounth))).
```

Пример 6 Определение количества читателей, посетивших библиотеку в мае, по информации, находящейся в файле reader.txt, сформированном в примере 5.

```
:-dynamic счетчик/1.
goal:- consult('reader.txt'),
       asserta(счетчик(0)),
       счет,
       счетчик(N),
       write('Число читателей= '),write(N).
 счет:-читатель(_,_,дата_посещ(_,may)),
        счетчик(N),
        N1 is N+1,
        retract(счетчик(N)),
        asserta(счетчик(N1)),
        fail.
счет.
```

3.15 Создание меню

Пример Напишем предикат show_menu, который создает окно с главным меню, состоящим из трех пунктов. При выборе пользователем соответствующего пункта меню выполняются необходимые действия. По окончанию действий снова можно выбирать пункты главного меню до тех пор, пока не будет выбран выход.

```
show_menu:-repeat,
            writeln('1 – процесс1'),
            writeln('2 – процесс2'),
            writeln('3 – выход'),nl,
            write('Введите Ваш выбор: (1-3) '),
            readln(X), nl,
            X<4,
            process(X),nl,
            X=3,!.
process(3).
process(1):-writeln(‘Проработал процесс 1’).
process(2):- writeln(‘Проработал процесс 2’).
```

3.16 Операции над структурами данных

3.16.1 Деревья

Списки часто используют для представления множеств. Но такое представление множеств имеет недостаток, заключающийся в том, что процедура проверки принадлежности элемента множеству оказывается неэффективной для множеств с большим количеством элементов. Если искомый элемент находится в конце списка или отсутствует в списке, процедуре придется просмотреть весь список. Для увеличения эффективности процедуры поиска применяют представление множеств в виде древовидных структур. Рассмотрим одну из таких структур – двоичное дерево.

Дадим рекурсивное определение двоичного дерева. *Двоичное дерево* либо пусто, либо состоит из трех частей: корень, левое поддерево, правое поддерево. Корень может быть чем угодно, а поддеревья должны быть деревьями. Удобно определить двоичное дерево как функтор с тремя аргументами: корень и два поддерева и ввести обозначение для пустого дерева: nil.

Пример 1 Напишем предикат, проверяющий принадлежность элемента дереву. Дадим декларативное определение предиката. Элемент либо является корнем дерева, либо принадлежит левому поддереву, либо принадлежит правому поддереву. В программе опишем следующее двоичное дерево:

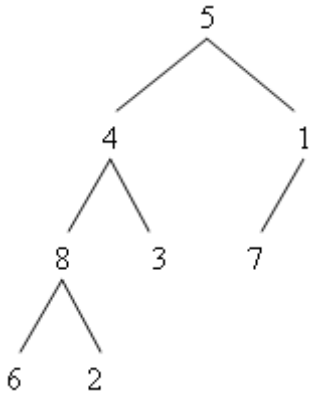


Рис.72. Дерево примера 1

```
goal:- Tree=tree(5,
                 tree(4,
                      tree(8,
                           tree(6,nil,nil),
                           tree(2,nil,nil)),
                       tree(3,nil,nil)),
                 tree(1,
                      tree(7,nil,nil),
                      nil)),
 write('Введите элемент: '),
 read(X),
 answer(X,Tree).
answer(X,Tree):-member_tree(X,Tree),!,
                 write('Да').
 answer(_,_):-write('Нет').
 member_tree(X,tree(X,_,_)):-!.
 member_tree(X,tree(_,Left,_)):-member_tree(X,Left),!.
 member_tree(X,tree(_,_,Right)):-member_tree(X,Right).
```

Очевидно, что, если элемент находится в самом низу дерева, процедура поиска становится такой же неэффективной, как и в случае списков. Введем отношение порядка между элементами множества. Тогда элементы множества можно упорядочить слева направо в соответствии с этим отношением. Выберем отношение «меньше» для упорядочивания элементов множества, содержащего

целые числа. *Двоичным справочником* назовем непустое дерево, для которого выполняется:

- все вершины левого поддерева меньше корня;
- все вершины правого поддерева больше корня;
- левое и правое поддеревья являются двоичными справочниками.

Пример 2 Приведенное ниже дерево является двоичным справочником:

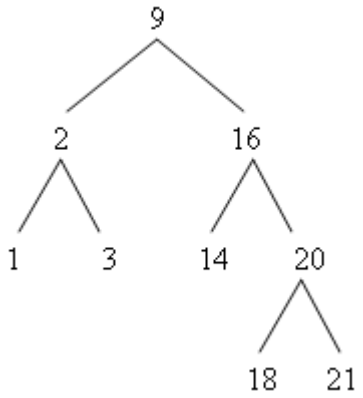


Рис. 73. Двоичный справочник

Пример 3 Модифицируем предикат поиска для двоичного справочника. Для поиска в двоичном справочнике достаточно просмотреть одно поддерево, предварительно сравнив искомый элемент с корнем. Алгоритм поиска в двоичном дереве будет иметь следующий вид:

- если элемент совпадает с корнем дерева, то элемент найден;
- если искомый элемент меньше корня, то следует искать в левом поддереве;
- если искомый элемент больше корня, то следует искать в правом поддереве;
- если справочник пуст, то поиск не удался.

```
member_tree(X,tree(X,_,_)):-!.
member_tree(X,tree(Y,Left,_)):-Y>X,!,
                                member_tree(X,Left).
member_tree(X,tree(_,_,Right)):-member_tree(X,Right).
```

Поиск элемента в двоичном справочнике эффективнее поиска в списке. Для списка в среднем будет просмотрена примерно половина элементов. Для справочника время поиска пропорционально глубине дерева – длине самого длинного пути между корнем и листом дерева. Для хорошо сбалансированных деревьев (левое и правое поддеревья должны содержать примерно одинаковое количество элементов) глубина дерева пропорциональна $\log n$, где n – количество элементов в справочнике. Если происходит разбалансировка дерева, то фактически дерево превращается в список и его глубина становится близкой к n.

Отображение деревьев

Если вывести дерево на экран предикатом write, то дерево выведется в виде прологовского терма, который сложно читать. Поэтому лучше написать

предикат вывода дерева в специальной форме, чтобы была видна его структура. Проще всего отображать дерево в повернутой на 90° форме и отображать его не сверху вниз, а слева направо. Например, дерево из примера 1 отобразится в виде:

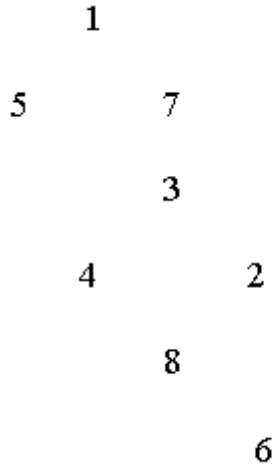


Рис. 74. Отображение двоичного дерева

Определим предикат show_tree(tree) для отображения на экране дерева слева направо. Опишем работу предиката:

- отображает правое поддерево с отступом вправо;
- выводит корень дерева;
- отображает левое поддерево с отступом вправо.

Для вывода дерева с заданным отступом определим предикат show_tree(tree, space), а для печати нужного количества пробелов – предикат print_blank(space).

```
show_tree(Tree):-show_tree(Tree,0).
show_tree(nil,_).
show_tree(tree(X,Left,Right),Space):-Space1 is Space+4,
                                     show_tree(Right,Space1),
                                     print_blank(Space),
                                     write(X),nl,
                                     show_tree(Left,Space1).
print_blank(0).
print_blank(Space):- write(‘ ‘),
                     Space1 is Space-1,
                      print_blank(Space1).
```

Попробуйте написать предикат, печатающий дерево сверху вниз.

Обходы деревьев

Во многих приложениях, использующих бинарные деревья, требуется доступ к элементам дерева. Основой этого доступа является обход дерева в предписанном порядке. Имеются три возможности линейного упорядочивания при обходе:

- сверху вниз (сначала корень дерева, затем вершины левого поддерева, после этого вершины правого поддерева);

– слева направо (сначала вершины левого поддерева, затем корень, после этого вершины правого поддерева);
– снизу вверх (сначала вершины левого поддерева, затем вершины правого поддерева, после этого корень).

Пример 4 Для дерева из примера 1 обход сверху вниз даст следующую последовательность вершин: 5, 4, 8, 6, 2, 3, 1, 7; обход слева направо даст: 6, 8, 2, 4, 3, 5, 7, 1; обход снизу вверх даст: 6, 2, 8, 3, 4, 7, 1,5.

Опишем предикат uptodownround(tree,list), формирующий список из вершин, полученных при обходе дерева сверху вниз. Для соединения обходов поддеревьев и корня в обход дерева будем использовать встроенный предикат append.

```
uptodownround(nil,[]):-!.
uptodownround(tree(X,Left,Right),S):-uptodownround(Left,Ls),
                                     uptodownround(Right,Rs),
                                     append([X|Ls],Rs,S).
```

Очевидно, что все обходы будут отличаться только порядком соединения обходов поддеревьев и корня.

Для обхода слева направо соединение обходов будет иметь вид:

```
append(Ls,[X|Rs],S).
```

А для обхода сверху вниз получим:

```
append(Rs,[X],Rs1), append(Ls,Rs1,S).
```

Перечисленные обходы используются для поиска минимального или максимального элемента в дереве, преобразовании дерева.

3.16.2 Графы

Во многих приложениях для представления отношений, ситуаций, структур задач используют графы. Граф определяется множеством вершин и ребер. Каждое ребро соединяет две вершины. Если ребра направленные, то их называют дугами. Графы с ребрами – дугами называются направленными. Ребрам могут быть приписаны стоимости, тогда граф называется взвешенным. В Прологе граф можно представить в виде отдельных предложений об имеющихся ребрах или описать граф как список ребер и список вершин, объединенных функтором graph (в случае если каждая вершина соединена хотя бы с одной вершиной, список вершин можно опустить, т.к. он неявно содержится в списке ребер).

Пример 5 Рассмотрим различные способы описания графа, представленного следующим рисунком:

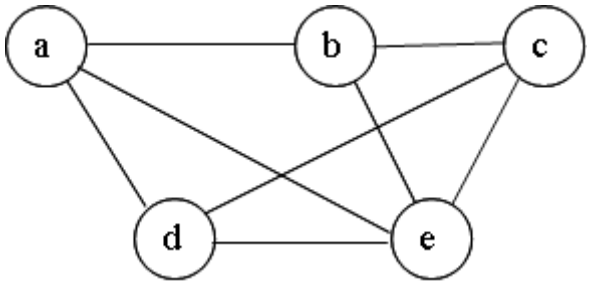


Рис. 75. Граф из примера 5

В первом случае имеем описание:

```
edge(a,b).
edge(a,d).
edge(a,e).
edge(b,e).
edge(c,d).
edge(c,e).
edge(d,e).
```

Во втором случае граф опишется как структура:

G=graph([a,b,c,d,e],[edge(a,b), edge(a,d), edge(a,e), edge(b,e), edge(c,d), edge(c,e), edge(d,e)]).

Способ представления графа зависит от конкретной задачи и от выполняемых операций над графами.

Пример 6 Определим предикат path(A,[Z],Path), который будет находить ациклический путь Path (список вершин, в котором каждая вершина присутствует только один раз) из вершины A в вершину Z. Для графа из примера 5 верны следующие предикаты:

```
path(a,[c],[a,b,c])
path(a,[c],[a,b,e,c])
path(a,[c],[a,d,c])
path(a,[c],[a,d,e,c])
path(a,[c],[a,d,e,b,c])
path(a,[c],[a,e,d,c])
path(a,[c],[a,e,b,c])
```

Опишем предикат path:

– сначала путь состоит из одной вершины Z;

– далее находим вершину, смежную с первой вершиной в списке, которая в список не была включена ранее. Процесс продолжается, пока первой вершиной в списке не станет A.

```
goal:-write('Введите начальную вершину\n'),
      read(A),
      write('Введите конечную вершину\n'),
      read(B),
      find_path(A,B).
edge(a,b).
edge(a,d).
edge(a,e).
edge(b,e).
edge(c,d).
edge(c,e).
edge(e,d).
find_path(A,B):-path(A,[B],Path),!,
                write('Путь='),
                write(Path).
```

```
find_path(_,_):-write('Пути нет').
path(A,[A|Path],[A|Path]).
path(A,[Y|Path1],Path):-neighbour(X,Y),
                        not(member(X,Path1)),
                        path(A,[X,Y|Path1],Path).
neighbour(X,Y):-edge(X,Y); edge(Y,X).
```

Заметим, что данная программа находит первый попавшийся путь из вершины A в B. Предикат path можно использовать для решения разнообразных задач. Например, для поиска всех вершин, достижимых из заданной, для поиска пути заданной длины от выделенной вершины, для нахождения диаметра графа (максимального расстояния между двумя вершинами), гамильтонова цикла (цикла, проходящего через все вершины), количества компонент связности графа. При решении этих задач удобно использовать для обмена значениями между предикатами динамическую базу данных.

Пример 7 Модифицируем приведенную в примере 6 программу для поиска пути минимальной стоимости между двумя вершинами во взвешенном неориентированном графе.

```
:-dynamic(bd_path/2).
edge(a,b,2).
edge(a,d,2).
edge(a,e,5).
edge(b,c,1).
edge(b,e,2).
edge(c,d,4).
edge(c,e,3).
edge(d,e,2).
goal:-retractall(bd_path(_,_)),
      write('Введите начальную вершину\n'),
      read(A),
      write('Введите конечную вершину\n'),
      read(B),
      find_begining(A,B),
       find_path(A,B),
      bd_path(Path,Length),
      write('Путь='),write(Path),
      write(' Длина='),write(Length),
      retractall(bd_path(_,_)).
find_begining(A,Z):-path(A,[Z],0,Path,Length),
                    assertz(bd_path(Path,Length)).
path(A,[A|Path],Length,[A|Path], Length).
path(A,[Y|Path1],Length1,Path,Length):-
                      neighbour(X,Y,Weigth),
                      not(member(X,Path1)),
                      Length2 is (Length1 + Weigth),
```

```
                        path(A,[X,Y|Path1],Length2,Path,Length).
neighbour(X,Y,W):-edge(X,Y,W); edge(Y,X,W).
find_path(A,B):-bd_path(Path,Length),
                path(A,[B],0,Path1,Length1),
                Length1<Length,
                retract(bd_path(Path,Length)),
                assertz(bd_path(Path1,Length1)),
                find_path(A,B).
find_path(_,_).
```