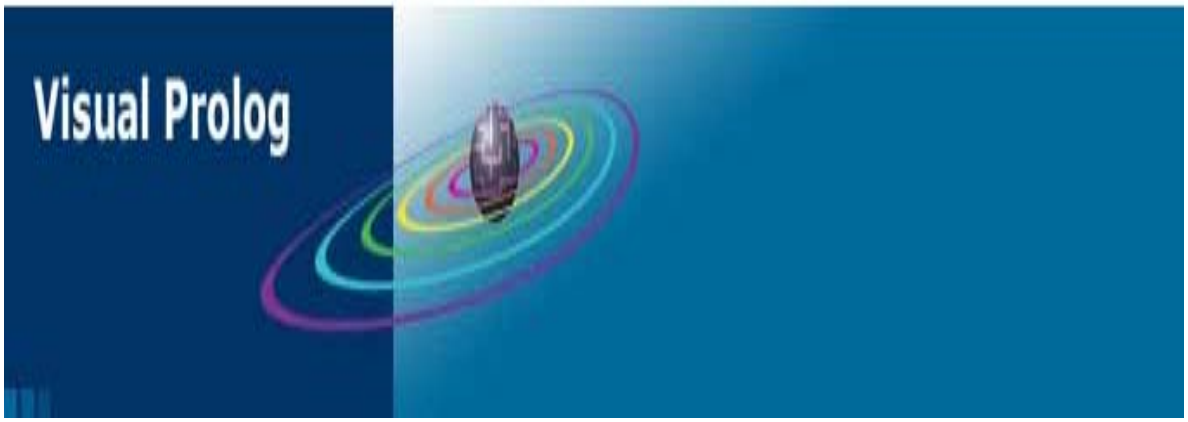
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ростовский государственный экономический университет (РИНХ)

Шполянская И.Ю.

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ В ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМАХ. ЯЗЫК ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ VISUAL PROLOG 7.5

Учебно-методическое пособие



Ростов-на-Дону
2014

УДК 004 (075) + 330.4 (075)
Ш 84

Шполянская И.Ю. Представление знаний в интеллектуальных системах. Язык логического программирования Visual Prolog 7.5: Учеб.-методич. пособие / РГЭУ (РИНХ). – Ростов н/Д.,- 2014. – 80 с. - ISBN

В пособии рассмотрены теоретические основы представления и использования знаний с помощью логики предикатов первого порядка и вопросы их практической реализации. Приводится описание логического языка программирования Visual Prolog 7.5 как инструментального средства решения интеллектуальных задач. Пособие содержит большое количество примеров создания интеллектуальных приложений в среде Windows и пояснений к ним. Рассмотрены базовые приемы построения экспертных систем с помощью методов логического программирования.

Учебное пособие предназначено для студентов и аспирантов экономических и технических специальностей вузов, изучающих дисциплины «Интеллектуальные информационные системы», «Представление и использование знаний».

Рецензенты:

Тищенко Е.Н., доктор экономических наук, доцент, зав. каф. Информационных технологий и защиты информации Ростовского государственного экономического университета

ISBN

ВВЕДЕНИЕ

Пролог известен как декларативный язык. Это означает, что при заданных необходимых фактах и правилах, Пролог будет использовать дедуктивные умозаключения для решения задач программирования. Эта его особенность выгодно отличает от традиционных процедурных языков, таких как C, Basic и Pascal. В процедурном языке компьютеру следует задавать пошаговый алгоритм решения конкретной задачи, иными словами, программист должен заранее знать, как решить данную задачу. Пролог-программисту нужно предоставить только описание задачи и основные правила для ее решения. Таким образом, система Пролог предназначена, в том числе, и для определения того, как найти необходимое решение.

Пролог имеет ряд преимуществ по сравнению с процедурными языками программирования, вот некоторые из них:

- для определенных задач программа на Прологе требует в несколько раз меньше строк кода по сравнению с аналогичной программой на языке C++;
- благодаря декларативному (в большей степени, чем процедурному) подходу, такие хорошо известные источники ошибок, как зацикливания, устраняются с самого начала;
- Пролог "заставляет" программиста начинать с хорошо структурированного описания задачи, поэтому язык может использоваться и как средство создания спецификации, и как средство реализации продукта.

Сферы применения Пролога:

•разработка быстрых прототипов прикладных программ;

•создание естественно-языковых интерфейсов;

•реализация экспертных систем и оболочек экспертных систем;

управление производственными процессами;

•создание динамических реляционных баз данных;

•перевод с одного языка на другой;

•создание пакетов символьных вычислений;

•доказательства теорем и интеллектуальные системы, в которых возможности Пролога по

обеспечению дедуктивного вывода применяются для проверки различных теорий.

Используя Visual Prolog, разработчик может:

- создавать приложения искусственного интеллекта и экспертные системы: консультативные системы, средства поддержки принятия решений, диагностические средства, оболочки экспертных систем или приложения естественного языка для различных прикладных областей (банковское дело, авиация, здравоохранение, страхование, медицина, промышленность и т. д.).
- решать традиционные задачи ведения баз данных, потому что Visual Prolog имеет полноценный и легкий в использовании механизм поддержки баз данных.
- использовать конкурентоспособную, универсальную среду разработки, как то:

- ✓ Оптимизированный компилятор Visual Prolog, который обеспечивает высокую скорость работы приложений.
- ✓ Средства разработки Web-приложений — новый, важный элемент Visual Prolog.
- ✓ Visual Prolog поддерживает объектно-ориентированный подход.

1. **Структура программы на Visual Prolog**

ПРОЛОГ - язык логического программирования (ПРОграммирование в ЛОГике), используемый для представления и манипулирования знаниями в системах искусственного интеллекта. Логическое программирование - программирование, основанное на использовании механизма доказательства теорем в логике, который позволяет выяснить, является ли противоречивым некоторое множество логических формул. При этом программа рассматривается как набор логических формул, описывающих предметную область, совместно с теоремой, которая должна быть доказана. Логическое программирование избавляет программиста от необходимости определения точной последовательности шагов выполнения вычислений.

При рассмотрении Пролог-программы можно выделить два уровня ее смысла: декларативный и процедурный. Декларативный смысл Пролог-программы связан с отношениями, объявленными (декларируемыми) в программе, он определяет, достижимо ли целевое утверждение, и при каких значениях переменных оно будет верным. Процедурный смысл определяет, как Пролог-система обрабатывает отношения пролог-программы, каким образом пролог-система отвечает на вопросы.

Пролог является языком программирования, который обеспечивает решение задач, выраженных в терминах объектов и отношений между ними.

Программа на языке Пролог состоит из фактов, правил и целевых утверждений. Факт представляет собой истинное утверждение. Правило представляет собой утверждение, которое истинно при определенных условиях. Совокупность фактов и правил образует базу данных Пролога. Когда база данных загружена в память Пролог-системы, к ней можно обращаться с вопросами, формулируемыми в виде целевых утверждений.

Отличительные особенности языка Пролог состоят в следующем :

1. Для представления знаний используются фразы Хорна.
2. Программа описывает логическую модель предметной области в виде фактов относительно свойств предметной области и отношений между этими свойствами, а также правила вывода новых свойств и отношений из уже заданных.
3. Отсутствуют операторы присваивания, ветвлений, безусловных переходов, а также указатели, которые используются при определении динамических структур данных в традиционных процедурно ориентированных языках.
4. В качестве единообразной структуры данных используется терм.

Написание программы на языке Пролог включает следующие этапы:
1) Объявление некоторых фактов об объектах и отношениях между ними. 2) Определение некоторых правил, касающихся объектов и отношений между ними. 3) Формулировка вопросов об объектах и отношениях между ними.

Основные определения.

Пролог (Prolog) — язык логического программирования, основанный на логике дизъюнктов Хорна, представляющей собой подмножество логики предикатов первого порядка. Это язык программирования, предназначенный для представления и использования знаний о некоторой предметной области в виде множества рассматриваемых объектов и

взаимосвязей между объектами и (или) их свойствами, называемых **отношениями**. Отсюда ключевым понятием Пролога является понятие **предиката** – отношения, связывающего некоторые объекты описываемой предметной области.
В логике теории задаются при помощи аксиом и правил вывода. То же самое имеет место и в Прологе. Аксиомы здесь принято называть ***фактами***, а правила вывода в форме так называемых "дизъюнктов Хорна" - ***утверждений*** вида A <= B1& ...& Bn.
В Прологе такие утверждения (клозы) принято записывать так:

a :- b1, ..., bn
а факты, (аксиомы), представлять как правила с пустой "посылкой": **a**.

Программа на языке ПРОЛОГ - набор утверждений, составляющих базу фактов и базу правил, к которым допустимо обращение с запросами, касающимися их содержимого.
Объект данных в Прологе называется термом. ***Терм*** - это либо константа, либо переменная, либо структура.
Отношения в Visual Prolog называются ***предикатами*** и записываются они так же как структуры. Для того, чтобы Пролог мог отличить где объект данных, а где – отношение, необходимо их явно описать.

Примеры предикатов:

получил_оценку("Петров",5)
отличная(Оценка)
обучает(профессор, студент, дисциплина)

Утверждение (предложение) языка ПРОЛОГ - линейная конструкция из термов, заканчивающаяся точкой.
Запросы называются также целевыми утверждения.
Константами являются атомы и числа. Константы используются для обозначения (именования) конкретных объектов предметной области и конкретных отношений между ними.
Атом языка ПРОЛОГ - это :

1. последовательность букв, цифр и знака подчеркивание, обязательно начинающаяся со строчной буквы;
2. последовательности специальных знаков :-, ?-, =, > и других.
3. произвольную последовательность символов, заключенную в кавычки.

Примеры атомов: машина, иван, “ кто-то ”, “Иван”, “иван”.

Переменная языка ПРОЛОГ – это:

1. последовательность букв, цифр и знака подчеркивание, обязательно начинающаяся с прописной буквы;
2. знак подчеркивание.

Примеры переменных: Иван, Студент, Z34, _test.

Переменная может находиться в свободном(free) или связанном (bound) состоянии. Причем переменные не предназначены для хранения значений, т.е. им нельзя явно присвоить какое-либо значение. А связанными они становятся в процессе проверки утверждений Пролога.
Область видимости переменных - одно предложение. Поэтому одно и то же имя в двух предложениях обозначает две разные переменные.

Структура языка ПРОЛОГ - конструкция из функтора и компонент, имеющая следующий вид
<функтор>(<компонент-1>,<компонент-2>, ... ,<компонент-n>)

где в качестве ***функтора*** должен выступать атом, а компонентом может быть любой терм (в том числе и структура).

Примеры структур:

- graf (a,b,c,d,u,v,x)
- студент(иванов, иван, возраст)
- имеет("Иван", Машина)
- green(дерево)
- наша_семья (дедка, бабка, внучка, собака(жучка), кошка (мурка), Мышка)

Основными утверждениями (clauses) Пролога являются факт и правило. Отличительной чертой утверждения является точка в конце.

Факт - это предикат, завершенный символом "точка".

Факт описывает свойство объекта или отношение между объектами, значение которого истинно.

Форма записи факта :

имя_предиката(аргумент {, аргумент}).

Примеры фактов:

Свойства	**Отношения**
собака (бобик).	parent(“Bill”,”Tom”).
человек (сократ).	владеет(иван,ipad).
человек(студент).	любит(петя,оля).
студент(“Петя”).	любит(оля,мороженое).

При этом :

1)Все имена предикатов и аргументов должны начинаться со строчной латинской буквы. 2) Перечисление аргументов — через запятую. 3) Каждый факт должен заканчиваться точкой. 4) Количество аргументов и вид отношений (направления отношений) определяются программистом и не меняются при выполнении программы. 5)При описании фактов переменные не используются.

К примеру, факт :

who_likes_what(ivan, programming).

не эквивалентен факту :

who_likes_what(programming, ivan).

При описании фактов и правил в разделе **clauses** программы на Visual Prolog'е одноименные предикаты должны быть сгруппированы :

```
clauses

who_likes_what(ivan, programming).

who_likes_what(ivan, reading).

who_likes_what(mary, reading).
```

База фактов в языке ПРОЛОГ - последовательность утверждений, описывающих факты предметной области в виде структур.

База правил - совокупность правил в программе на языке ПРОЛОГ.

Правило - это конструкция вида **A: — B, C, D.**

где A, B, C, D - предикаты.

A называется заголовком или целью, **B, C, D** - телом или подцелями. Цель правила – всегда единственна, число подцелей - произвольно.

Интерпретация правила выглядит следующим образом: A истинно, если одновременно истинны B, C и D.

Примеры правил:

- имеет_стипендию(петров): — экзамен(петров, Оценка), удовлетворительная(Оценка).
- удовлетворительная(Оценка): — Оценка >=3.

Правило записывается следующим образом

<структура-0>:-<структура-1>, ... ,<структура-N>.

Правило может трактоваться следующим образом: предикат, являющийся заголовком правила доказан (удовлетворен), когда доказан каждый предикат тела правила.

В качестве предиктов, составляющих тело правила, могут выступать

- предикаты, фигурирующие в базе фактов;
- предикаты, совпадающие с заголовком других правил;
- встроенные предикаты систем программирования ПРОЛОГ.

Любая совокупность фактов (и правил) в программе Пролога называется Базой Данных (БД).

Visual Prolog в отличие от стандартного ПРОЛОГА реализован в виде типизированного компилятора, и, в частности, требует обязательного описания типов аргументов для всех структур и предикатов. Это позволяет обеспечить высокую скорость исполнения программ на Visual Prolog, сравнимую с C и Паскалем.

Кроме того при описании предикатов могут быть заданы такие их свойства как

- неоднозначность результата – nondeterm, или одно решение determ;
- схема использования аргументов – вход(i), выход(o), сложный объект.

В Прологе существует множество встроенных специальных предикатов. Например, предикат без аргументов «nl» определяет печать с новой строки, а предикаты free(var) и bound(var) проверяют является ли переменная var, соответственно, свободной или связанной.

Другим сложным объектом данных Пролога является список.

◊ ***Список*** - это объединение произвольного количества объектов, разделенных запятыми и заключенных в квадратные скобки.

Список не имеет имени, а количество его элементов может меняться.

Примеры списков:

- [3,5,4]
- [собака, кошка, хомяк, крокодил]
- [петров, [иванов, сидорова], сидоров, козлова]
- [[4,5,3],[4,4,5],[2,2],[ ],[5,5,5]]
- [ ] – пустой список

Следует подчеркнуть, что атом “иван” и список, состоящий из единственного элемента “иван” – это разные объекты данных.

Можно создавать список из любых элементов, включая другие списки. Но все элементы одного списка должны принадлежать одному и тому же домену.

Список представляет собой рекурсивный объект. Поэтому любой список (кроме пустого) можно разбить на «голову», совпадающую с первым элементов этого списка, и «хвост», который включает все его остальные элементы кроме первого.

Голова списка – элемент, а хвост – всегда список. Например, голова списка [s] – это “s”, а хвост равен [ ]. Если достаточное число раз отделить первый элемент списка, то получим в конце концов пустой список.

Для отделения головы от хвоста Пролог использует специальный выделенный символ “|”. Например, [a,b,c] эквивалентно [a | [b,c]] и далее [a | [b | [c]]] и [a | [b | [c | [ ]]]].

Основная операция над объектами в Прологе - это унификация (иногда называют сопоставление, согласование, конкретизация). По своей сути ***унификация*** - это сравнение объектов или их совокупностей.

Правила унификации:

- число сопоставляется только с равным ему числом,
- атом сопоставляется только с равным ему атомом,
- переменная сопоставляется с любым объектом и получает значение того, с чем сопоставляется,
- структура сопоставляется с другой структурой, если число их компонент и функторы (имена) совпадают, а компоненты попарно сопоставимы.

Очевидно, что операция сопоставления может кончиться неудачно.

Примеры унификации:

- 9 сопоставляется с 9
- "иван" сопоставляется с иван
- "иван" не сопоставляется с "петр"
- *имеет (иван, машина)* не сопоставляется с *имеет (иван, канарейка)*

- *имеет (иван, машина)* сопоставляется с *имеет (иван, X)* $\Rightarrow$ *X=машина*
- [1,2] не сопоставляется [3, X]
- {X,Y,Z] сопоставляется [петя, вася, нина] $\Rightarrow$ X= петя, Y = вася, Z = нина

Структура программы на Visual Prolog в упрощенном виде:

В разделе доменов (областей) **DOMAINS** объявляются любые нестандартные домены (области), используемые для параметров предикатов. Домены (области) в Прологе подобны типам данных на других языках программирования.

Visual Prolog имеет множество примитивных типов данных, в том числе:
integer: Например: 3, 45, 65, 34, 0x0000FA1B, 845.
real: 3.45, 3.1416, 2.18E-18, 1.6E-19 и пр.
string: "pencil", "John McKnight", "Cornell University" и пр.
symbol : "Na", "Natrium", "K", "Kalium"
char — Символы в одинарных кавычках: 'A', 'b', '3', ' ', '.', ….

Элементы типа *symbol* выглядят так же, как строки: и те, и другие представляют собой последовательности Unicode-символов. Однако хранятся они по-разному. Элементы типа *symbol* хранятся в таблице идентификаторов, и Пролог для их внутреннего представления использует адреса в этой таблице. Таким образом, если элемент типа *symbol* встречается в программе много раз, он будет занимать меньше места, чем строка.

Программа на Visual Prolog, представляемая кодом, разделяется ключевыми словами на секции разного вида путем использования ключевых слов, предписывающих компилятору, какой код генерировать. Обычно, каждой секции предшествует ключевое слово. Ключевых слов, обозначающих окончание секции, нет. Наличие другого ключевого слова обозначает окончание предыдущей секции и начало другой.
Исключением из этого правила являются ключевые слова **implement** и **end implement**. Код, содержащийся между этими ключевыми словами, есть код, который относится к конкретному классу (модуль).

implement и end implement Среди всех ключевых слов Visual Prolog это единственные ключевые слова, используемые парно. Visual Prolog рассматривает код, помещенный между этими ключевыми словами, как код, принадлежащий одному классу. За ключевым словом implement обязательно **ДОЛЖНО** следовать имя класса. Например:

*implement **main***
……
*end implement **main***

open Это ключевое слово используется для расширения области видимости класса. Оно должно быть помещено в начале кода класса, сразу после ключевого слова implement (с именем класса и, возможно, именем интерфейса). Например:

open core

constants Это ключевое слово используется для обозначения секции кода, которая определяет неоднократно используемые значения, применяемые в коде. Например, если строковый литерал "PDC Prolog" предполагается использовать в различных местах кода, тогда можно один раз определить константу . Например:
constants
pdc="PDC Prolog".

Определение константы завершается точкой (.). В отличие от переменных Пролога константы должны начинаться со строчной буквы (нижний регистр).

domains Это ключевое слово используется для обозначения секции, объявляющей домены, для определения типов данных, которые будут использованы в коде. Например:
domains
gender = female(); male().

class facts Это ключевое слово представляет секцию, в которой объявляются факты, которые будут в дальнейшем использоваться в тексте программы. Каждый факт объявляется как имя, используемое для обозначения факта, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену. Например:
class facts - familyDB
person : (string Name, gender Gender).
parent : (string Person, string Parent).

class predicates Эта секция содержит объявления предикатов, которые далее определяются в тексте программы в разделе **clauses**. Объявление предиката - это имя, которое присваивается предикату, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену. Например:
class predicates
father : (string Person, string Father) nondeterm anyflow.

Если в разделе clauses программы на Прологе вы описали собственный предикат, то вы обязаны объявить его в разделе predicates (предикатов); в противном случае Пролог не поймет, о чем вы ему "говорите". В результате объявления предиката вы сообщаете, к каким доменам (типам) принадлежат аргументы этого предиката.

clauses Этот раздел содержит утверждения - конкретные определения объявленных в разделе **class predicates** предикатов, причем синтаксически им соответствующие. Например:
clauses
father(Person, Father) :-
parent(Person, Father),
person(Father, male()).

В раздел **clauses** (предложений) вы помещаете все факты и правила, составляющие вашу программу. Все предложения для каждого конкретного предиката в разделе **clauses** должны располагаться вместе. Последовательность предложений, описывающих один предикат, называется **процедурой**.

Пытаясь разрешить цель, **Пролог** (начиная с первого предложения раздела **clauses**) будет просматривать каждый факт и каждое правило, стремясь найти сопоставление.

Тело правила состоит из одной или более подцелей. Подцели разделяются запятыми, определяя конъюнкцию, а за последней подцелью правила следует точка. Каждая подцель выполняет вызов другого предиката ***Пролога***, который может быть истинным или ложным. После того, как программа осуществила этот вызов, ***Пролог*** проверяет истинность вызванного предиката, и если это так, то работа продолжается, но уже со следующей подцелью. Если же в процессе такой работы была достигнута точка, то все правило считается истинным; если хоть одна из подцелей ложна, то все правило ложно.

Для успешного разрешения правила ***Пролог*** должен разрешить все его подцели и создать последовательный список переменных, должным образом связав их. Если же одна из подцелей ложна. ***Пролог*** вернется назад для поиска альтернативы предыдущей подцели, а затем вновь двинется вперед, но уже с другими значениями переменных. Этот процесс называется ***поиском с возвратом***.

goal Этот раздел определяет главную точку входа в программу на языке системы Visual Prolog. Ключевое слово Goal можно представлять как особый предикат, не имеющий аргументов, с которого вся программа начинает исполняться.

Раздел goal (цели) аналогичен телу правила: это просто список подцелей. Цель отличается от правила лишь следующим:
•за ключевым словом goal не следует ":-";
•при запуске программы Пролог автоматически выполняет цель.

Это происходит так, как будто Пролог вызывает goal, запуская тем самым программу, которая пытается разрешить тело правила goal. Если все подцели в разделе goal истинны, - программа завершается успешно. Если же какая-то подцель из дела goal ложна, то считается, что программа завершается неуспешно - программа просто завершит свою работу.

Файловое структурирование программ. Расширение Области Видимости

Visual Prolog предусматривает возможность деления текста программы на отдельные файлы, используя среду IDE (Integrated Development Environment) и, кроме того, IDE позволяет помещать логически связанные фрагменты текста программы в отдельные файлы. Тогда, если имеется домен, который используется несколькими файлами, то объявление домена делается в отдельном файле и к этому файлу из других файлов существует доступ.
В Visual Prolog текст программы разделен на отдельные части, каждая часть определяется как класс (class). В объектно-ориентированных языках программирования, класс - это пакет кода и ассоциированные с ним данные. В Visual Prolog каждый класс обычно помещается в отдельный файл.
В процессе исполнения программы, часто бывает так, что программе может потребоваться вызвать предикат, который определен в другом классе (файле). Аналогично, данные (константы) или домены могут быть востребованы в другом файле. Visual Prolog позволяет делать такие взаимные ссылки на предикаты или данные используя концепцию области видимости (**scope access**).
Например. Предположим имеется предикат pred1, определенный в классе class1 (который помещается в другом файле, согласно стратегии среды IDE), и мы хотим вызвать этот предикат в теле **clauses** некоторого другого предиката pred2, находящегося в другом файле (скажем, class2). Тогда предикат pred1 должен быть вызван в теле клауза предиката pred2 следующим образом:

```
clauses
  pred3:-
    ...
```

```
  !.

pred2:-
  class1::pred1, % pred1 неизвестен в этом файле.
            % Он определен в другом файле,
            % поэтому требуется квалификатор класса.
  pred3,
  ...
```

Область видимости конкретного класса лежит внутри границ, где класс определен (то есть в интервале между ключевыми словами implement - end implement). Предикаты, определенные здесь, могут вызывать друг друга без квалификатора класса и двойного двоеточия (::).
Область видимости класса может быть расширена использованием ключевого слова **open**. Это ключевое слово информирует компилятор о том, что в этот класс должны быть "доставлены" имена (предикатов / констант / доменов), которые были определены в других файлах. Если область видимости расширена, то необходимости использования квалификатора класса (с двойным двоеточием) нет.

```
open class1
 ...

clauses
 pred3:-
   ...
   !.

 pred2:-
   pred1, % Внимание: квалификатор "class1::" больше не нужен,
        % поскольку область видимости расширена
        % использованием ключевого слова 'open'
   pred3,
   ...
```

Объектная ориентированность

Visual Prolog является полностью объектно-ориентированным языком. Весь код, который разрабатывается для программы, помещается в класс. Например, код помещается в класс, называемый "family1", несмотря на то, что мы не используем объекты, порождаемые этим классом. Кроме того, в этом классе мы используем общедоступные предикаты, находящиеся в других классах.

Пример: family1.prj6

```
implement family1
open core

constants
  className = "family1".
  classVersion = "$JustDate: $$Revision: $".
```

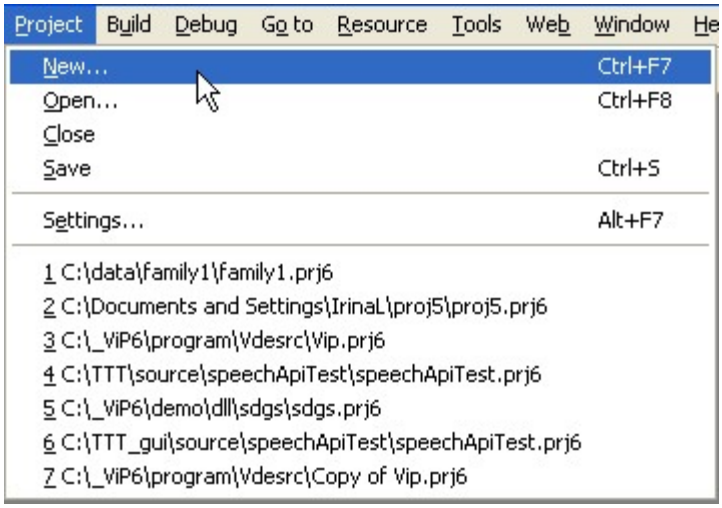
```
clauses
  classInfo(className, classVersion).

domains
  gender = female(); male().

class facts - familyDB
  person : (string Name, gender Gender).
  parent : (string Person, string Parent).

class predicates
  father : (string Person, string Father) nondeterm anyflow.
clauses
  father(Person, Father) :-
    parent(Person, Father),
    person(Father, male()).

class predicates
  grandFather : (string Person, string GrandFather) nondeterm anyflow.
clauses
  grandFather(Person, GrandFather) :-
    parent(Person, Parent),
    father(Parent, GrandFather).

class predicates
  ancestor : (string Person, string Ancestor) nondeterm anyflow.
clauses
  ancestor(Person, Ancestor) :-
    parent(Person, Ancestor).
  ancestor(Person, Ancestor) :-
    parent(Person, P1),
    ancestor(P1, Ancestor).

class predicates
  reconsult : (string FileName).
clauses
  reconsult(FileName) :-
    retractFactDB( familyDB),
    file::consult(FileName, familyDB).

clauses
  run():-
    console::init(),
    stdIO::write("Load data\n"),
    reconsult("fa.txt"),
    stdIO::write("\nfather test\n"),
    father(X, Y),
      stdIO::writef("% is the father of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\ngrandFather test\n"),
    grandFather(X, Y),
```

```
      stdIO::writef("% is the grandfather of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\nancestor of Pam test\n"),
    X = "Pam",
    ancestor(X, Y),
      stdIO::writef("% is the ancestor of %\n", Y, X),
    fail.
  run():-
    stdIO::write("End of test\n").
end implement family1

goal
mainExe::run(family1::run).
```

Лабораторная 1. Создание консольных приложений в среде Visual Prolog

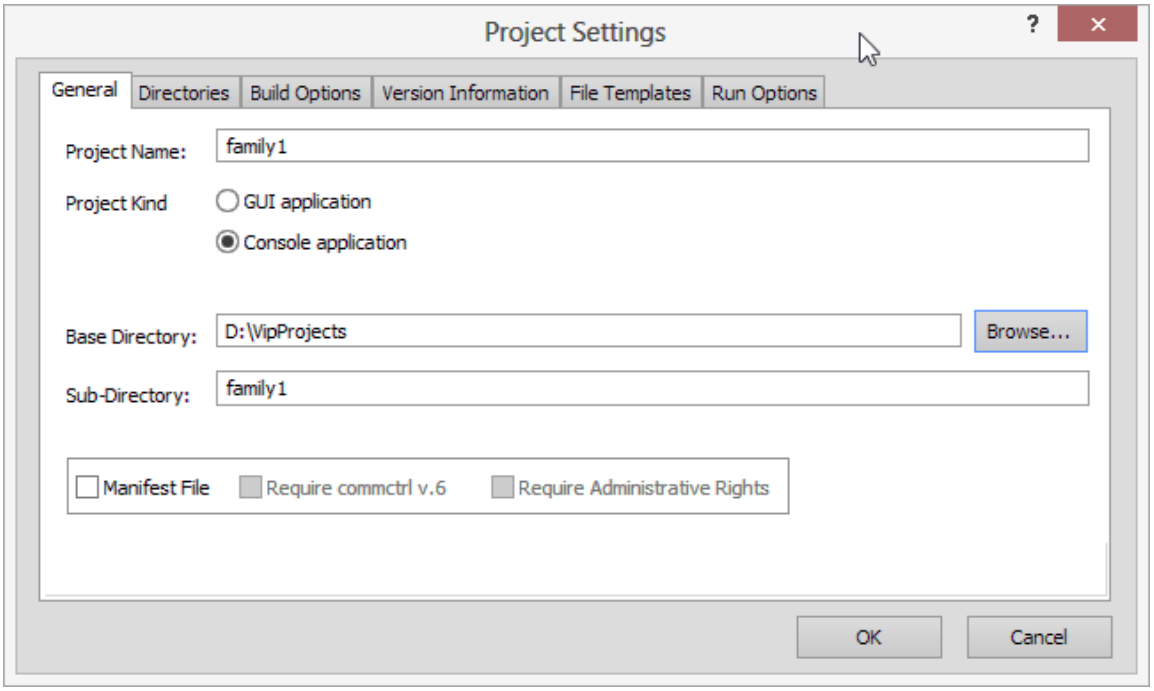
Шаг 1: Создайте в IDE новый консольный проект

Шаг 1a. После старта среды программирования Visual Prolog, выберите *New* из меню *Project*.



Шаг 1b. Появляется диалог Project Settings, включающий имя, вид и тип проекта, его расположение на диске. Удостоверьтесь, что вид проекта (Project Kind - UI Strategy) **Console**, а **HE** (*GUI-MDI,SDI)*.

Нажмите кнопку Finish (или OK).



Шаг 2: Постройте пустой проект

Шаг 2a. Когда проект только что создан, среда будет показывать проектное окно, как показано ниже. В этот момент пока никакой информации о том, от каких файлов зависит проект, нет. Однако основные тексты программ проектных файлов уже созданы.

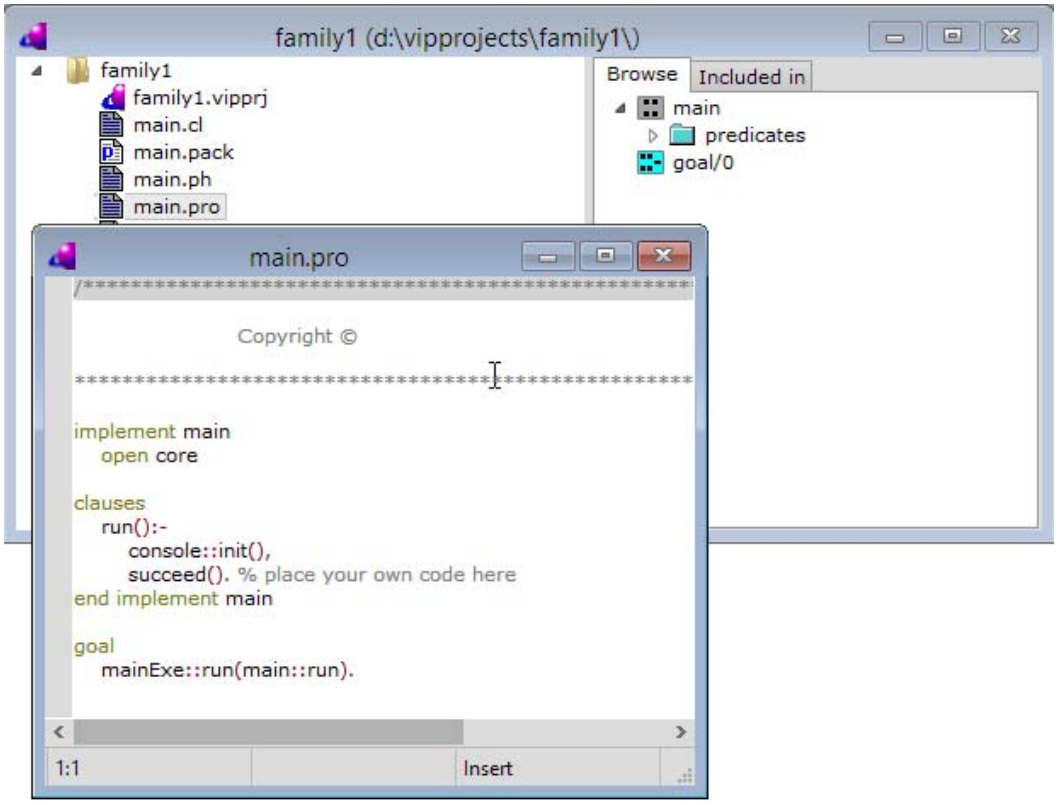
Шаг 2b. Выберите из меню *Build* позицию *Build*. Пустой проект, который ничего не делает, будет построен. (В этот момент времени никаких сообщений о ошибках быть не должно).

В процессе построения проекта посмотрите на сообщения, которые динамически появляются в окне **Messages** среды IDE. (Если окно **Messages** не видно, его можно вызвать на передний план, используя меню Window в среде IDE). Вы заметите, что среда IDE включит в Ваш проект все необходимые модули PFC (Prolog Foundation Classes). Классы PFC являются теми классами, которые содержат поддержку функционирования программ, которые Вы строите.

Шаг 3: Поместите в файл main.pro проекта family1 актуальный код

```
implement main
    open core

    domains
        gender = female(); male().

    class facts - familyDB
        person : (string Name, gender Gender).
        parent : (string Person, string Parent).

    class predicates
        father : (string Person, string Father) nondeterm anyflow.
    clauses
        father(Person, Father) :-
            parent(Person, Father),
            person(Father, male()).

    class predicates
        grandFather : (string Person, string GrandFather) nondeterm (o,o).
    clauses
        grandFather(Person, GrandFather) :-
            parent(Person, Parent),
            father(Parent, GrandFather).

    class predicates
        ancestor : (string Person, string Ancestor) nondeterm (i,o).
    clauses
        ancestor(Person, Ancestor) :-
            parent(Person, Ancestor).
        ancestor(Person, Ancestor) :-
            parent(Person, P1),
            ancestor(P1, Ancestor).

    class predicates
        reconsult : (string FileName).
    clauses
        reconsult(FileName) :-
            retractFactDB(familyDB),
            file::consult(FileName, familyDB).

    clauses
        run():-
            console::init(),
            stdIO::write("Load data\n"),
            reconsult("..\\fa.txt"),
            stdIO::write("\nfather test\n"),
            father(X, Y),
                stdIO::writef("% is the father of %\n", Y, X),
            fail.
        run():-
```

```
            stdIO::write("\ngrandFather test\n"),
            grandFather(X, Y),
                stdIO::writef("% is the grandfather of %\n", Y, X),
            fail.
        run():-
            stdIO::write("\nancestor of Pam test\n"),
            X = "Pam",
            ancestor(X, Y),
                stdIO::writef("% is the ancestor of %\n", Y, X),
            fail.
        run():-
            stdIO::write("End of test\n"),
            programControl::sleep(9000),
        succeed().
end implement main

goal
    mainExe::run(main::run).
```



Шаг 4: Перестроение кода проекта

Повторите вызов команды меню **Build**. IDE теперь уведомит, что исходный текст поменялся, и выполнит все необходимые действия для перекомпиляции соответствующих разделов. Если Вы вызовите команду меню Rebuild All, тогда *все* модули проекта будут перекомпилированы. В случае больших проектов это может занимать значительное время. Rebuild All обычно используется в качестве финальной фазы после ряда небольших изменений и соответствующих компиляций, чтобы удостовериться в том, что все в полном порядке.

В процессе выполнения команды *Build* среда IDE выполняет не только компиляцию. В это же время выясняется, нужны ли проекту другие модули из набора PFC (Prolog Faundation Classes), и, если это так, то такие модули добавляются и вызывается повторная перекомпиляция, если необходимо. Этот процесс можно наблюдать по сообщениям, появляющимся в окне сообщений (Message Window). Можно увидеть как IDE вызвало построение проекта дважды, поскольку были обнаружены директивы "include", повлиявшие на это.

Шаг 5: Исполнение Программы

Когда приложение, успешно откомпилированное с помощью Visual Prolog. построено, можно запустить его выполнение, вызвав команду из меню *Build* | *Run in Window*. Но для работы программы нужен текстовый файл с исходными данными *fa.txt*. Этот текстовый файл содержит записи относительно лиц, которые программа должна обработать.

Текстовый файл создается с использованием текстового редактора. Его нужно поместить в ту же директорию, где располагается сама исполняемая программа. Обычно исполняемая программа помещается в поддиректории проектной директории, имеющей имя EXE.

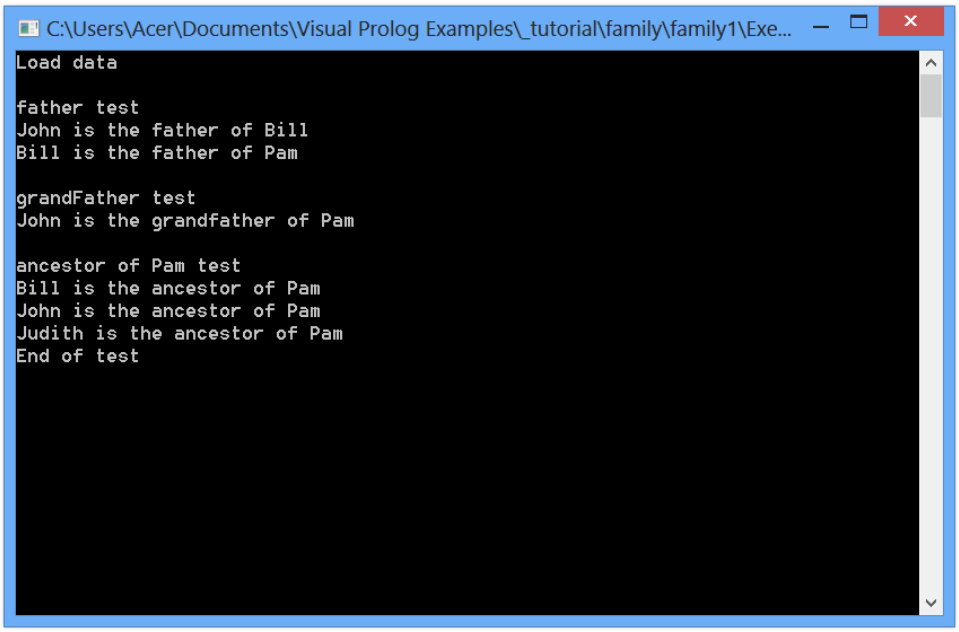
Создадим такой файл с использованием среды IDE. Вызовите команду меню *File* | *New*. Появляется окно создания новой сущности *Create Project Item*. Выберите Текстовый файл (Text File) в списке слева. Затем выберите директорию, где будет размещаться файл (та же директория, где располагается исполняемое приложение family1.exe). Присвойте теперь имя *fa.txt* файлу и нажмите кнопку Create (создать). До тех пор, пока имя файла не введено, кнопка Create (создать) будет неактивна (надпись серого цвета). Удостоверьтесь, что флажок *Add to project as module* (добавить в проект на правах модуля) включен.

Файл следует заполнить текстом следующего содержания:

```
clauses
    person("Judith",female()).
    person("Bill",male()).
    person("John",male()).
    person("Pam",female()).
    parent("John","Judith").
    parent("Bill","John").
    parent("Pam","Bill").
```

Синтаксис файла данных похож на обычный код Пролога. Синтаксис файла fa.txt ДОЛЖЕН быть совместим с определениями доменов в программе. Например, функтор, применяемый для определения персоны ДОЛЖЕН быть *person* и никак иначе. В противном случае соответствующий составной домен не будет инициализирован.

Теперь, запустив программу по команде меню **Build | Run in the Window** вы увидите:



Программа обрабатывает данные, помещенные в файл fa.txt и порождает логические связи персон, данные о которых там помещены.

Как работает программа

При старте непосредственно будет вызван главный предикат. Здесь все построено вокруг предиката *run*. Предикат *run* прежде всего считывает данные, записанные в файле *fa.txt*. После загрузки данных, программа систематически переходит от одной проверки к другой, обрабатывая данные, и выводит на консоль выводы на базе этих проверок.

Обработка данных основывается на стандартных механизмах отката и рекурсивных вызовов. Когда предикат *run* вызывает предикат *father*, он не ограничивается первым удовлетворяющим решением предиката *father*. Применение предиката fail в конце последовательности понуждает механизм Пролога к поиску следующего прохода предиката *father*. Такое поведение, называется *backtracking* (откат), поскольку кажется, что механизм Пролога перепрыгивает назад, минуя ранее исполненный код.

Это происходит рекурсивно (то есть как повторяющийся или циклический процесс), и в каждом цикле предикат *father* вырабатывает новый результат. В итоге все возможные определения "отцов", представленные данными, исчерпываются, и у предиката *run* нет другого выхода, как перейти к следующему своему утверждению (клаузу, **clauses**).

Предикат *father* (так же как и ряд других предикатов) объявлен как нетерминированный (**nondeterm)**, обозначающее, что предикат может вырабатывать множество решений посредством бэктрекинга (отката).

Если никакое ключевое слово в объявлении предиката не используется, то предикат получает режим процедуры, которая может выработать только одно решение, и предикат *father* прекратил бы работу после получения первого же результата и не смог бы быть вызван повторно. Следует быть осторожным в использовании отсечения (**cut**), обозначаемого как **!**, в клаузах таких недетерминированных предикатов. Если в конце клаузы предиката *father* помещено отсечение, то предикат выработает только одно решение (даже если он объявлен как недетерминированный).

Квалификатор направлений ввода-вывода ***anyflow*** говорит компилятору о том, что параметры, назначенные предикату, могут быть как связанными, так и свободными, без каких-либо ограничений. Это означает, что используя одну и ту же декларацию предиката *father*, можно будет получать как имя отца (в случае, если имя потомка связано), так и имя потомка (в случае, если имя отца связано). Ситуация, когда оба параметра связаны, также обрабатывается - и в этом случае проверяется, существует ли отношение между данными, представленными параметрами.

Квалификатор ***anyflow*** включает ситуацию, когда оба параметра предиката *father* являются свободными переменными. В этом случае, предикат *father* вернет в ответ на запрос различные комбинации отношений родитель-потомок, предусмотренные в программе (представленные в загружаемом файле ***fa.txt***). Последний вариант как раз и использован в предикате ***run***, как видно во фрагменте, приведенном ниже. Можно заметить, что переменные X и Y, переданные предикату *father* , не связаны при его вызове.

```
clauses
   run():-
      console::init(),
      stdIO::write("Load  data\n"),
      reconsult("fa.txt"),
      stdIO::write("\nfather test\n"),
      father(X,Y),
      stdIO::writef("% is the father of %\n", Y, X),
      fail.
```

Ввод и вывод данных.

Для ввода с клавиатуры данных используется встроенный предикат **read** из класса **console**, но аналогичные предикаты есть в других классах, например в классе **stdio** пакета **stream**.

console::init() - инициализация консоли

console::readChar() - чтение одиночного символа с консоли

console::readLine() - чтение строки с консоли

console::nl - перевод строки

Если во фрагменте заголовка (строка **open core**) открыть и класс консоли (**console**), то это позволяет в предикате **run**() ссылку на консоль не упоминать.

Для вывода используются предикат ***write*** из класса **console** или аналогично в классе **stdio** пакета **stream**.

Форматированный вывод реализуется предикатом stdIO::writef.

```
stdIO::writef("% is the father of %\n", Y, X).
```

Отсечение

Пролог предусматривает возможность отсечения, которая используется для прерывания поиска с возвратом; отсечение обозначается восклицательным знаком (!). Действует отсечение просто: через него невозможно совершить откат (поиск с возвратом).

Отсечение помещается в программу таким же образом, как и подцель в теле правила. Когда процесс проходит через отсечение, немедленно удовлетворяется обращение к cut и выполняется обращение к очередной подцели (если таковая имеется). Однажды пройдя через отсечение, уже невозможно произвести откат к подцелям, расположенным в обрабатываемом предложении перед отсечением, и также невозможно возвратиться к другим предложениям, определяющим обрабатывающий предикат (предикат, содержащий отсечение).

Существуют два основных случая применения отсечения.
•Если вы заранее знаете, что определенные посылки никогда не приведут к осмысленным решениям (поиск решений в этом случае будет лишней тратой времени), - примените отсечение, - программа станет быстрее и экономичнее. Такой прием называют зеленым отсечением.
•Если отсечения требует сама логика программы для исключения из рассмотрения альтернативных подцелей. Это - красное отсечение.

Объявление режимов детерминизма

Объявление режимов детерминизма предиката, которые указывают на то, имеет ли он единственное решение или может иметь много решений, осуществляется с помощью следующих ключевых слов.
determ Выполнение детерминированного предиката может завершиться либо неудачно (*fail*), либо успешно (*succeed*) и при этом иметь одно решение.
procedure Предикаты этого вида всегда завершаются успешно и имеют одно решение.
multi Такой предикат не может завершиться неудачно, при этом он имеет множество решений.
nondeterm Недетерминированный предикат может завершиться либо неудачно, либо успешно, и при этом иметь множество решений.
Если предикат имеет много решений и одно из его решений не в состоянии удовлетворить другой предикат в той же конъюнкции, то Пролог откатывается (*backtrack*) и предлагает другое решение в попытке удовлетворить эту конъюнкцию.

Предикаты с режимом детерминизма procedure, называют процедурами. Если предикат не может порождать более одного решения, то он называются *детерминированным*, а если может, то *недетерминированным*.

По умолчанию используется режим procedure, если предикат объявляется в разделе (class) predicates, и режим nondeterm, если предикат объявляется в разделе (class) facts.

В объявлении предиката требуется указывать не только режим детерминизма, но и поток параметров (flow pattern). В нем описывается, какие аргументы предиката при его вызове являются входными, а какие выходными. Используется обозначение (i) для входного аргумента и обозначение (o) для выходного. Произвольный поток параметров обозначается с помощью ключевого слова anyflow. По умолчанию все аргументы предиката являются входными. Поток параметров указывается в виде последовательности (i,o,o,...) символов i или o, соответствующих аргументам предиката, либо с помощью слова [out], которое ставится после имени домена аргумента предиката.

Пример 1. “Hello World”.

```
implement main
    open core

clauses
    run ():-
        console::init (),
        stdio::write ("Hello, World!"),
        programControl::sleep(5000),
        succeed ().
end implement main
goal
mainExe::run (main::run).
```

Или так:

```
implement main
    open core, console

clauses
    run ():-
        %console::init (),
        %stdio::
        write ("Hello, World!"),
        programControl::sleep(5000),
        succeed ().
end implement main
goal
mainExe::run (main::run).
```

Пример 2 – Вычисление суммы двух переменных

Вводятся X и Y. Найти их сумму.

Решение

Создайте новое консольное приложение аналогично тому, как было описано выше. Перейдите к редактированию файла main.pro двойным щелчком мыши по файлу в дереве проекта и замените его содержимое приведенным кодом.

```
implement main
open core,console

clauses

        run():- console::init(),
        stdIO::write("Введите X= "),
         X= stdIO::read(),
        stdIO::write("Введите Y= "),
           Y=stdIO::read(),
         Z=X+Y,
     stdIO::write("X+ Y= "),
```

```
    stdIO::write(Z, "\n"),
    programControl::sleep(5000),

 succeed().

          end implement main
goal
   mainExe::run(main::run).
```

Или в следующем виде:

```
implement main
open core, console

clauses

       run():- console::init(),        write("Введите X= "),
        X=  read(),
       write("Введите Y= "),
         Y=read(),          Z=X+Y,
   write("X+ Y= "),
   write(Z, "\n"),
   write("Конец программы").

          end implement main
goal
   mainExe::run(main::run).
```

После построения программы (команда Build/ Build) для запуска программы выберите команду Build/Execute (или Run in Window), появится приглашение ввести X и Y. Пример результата выполнения указан на рисунке 7.

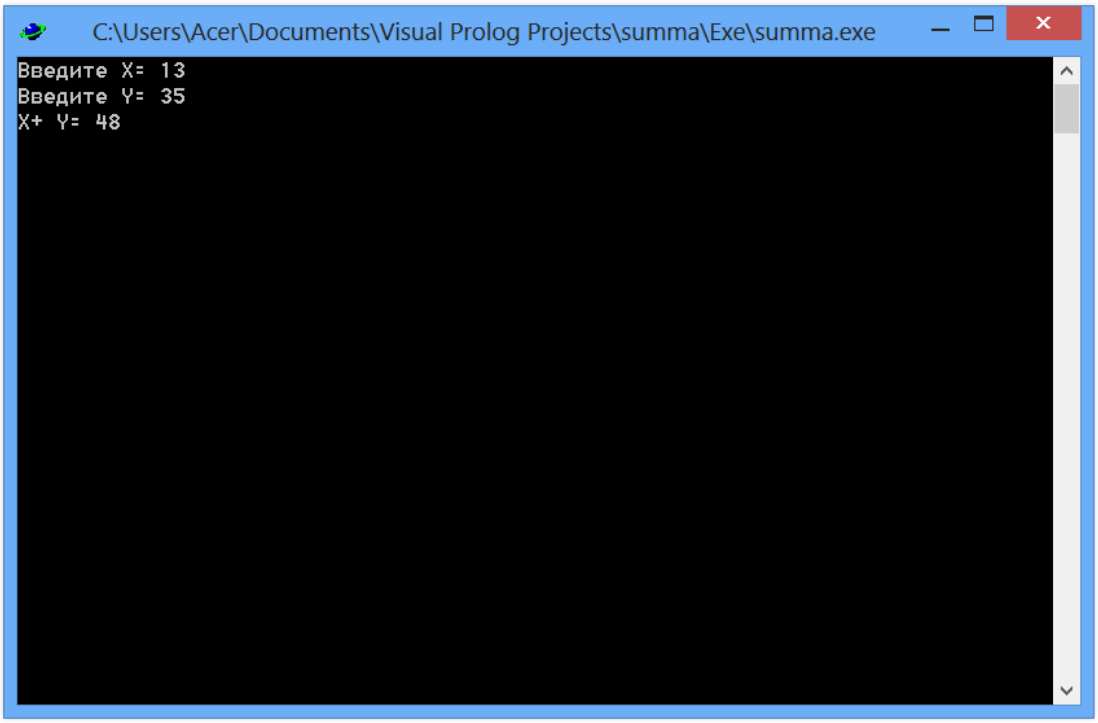


Рис. 7 Результат выполнения программы из примера 2

Пример 3. Число положительных чисел.

```
implement main
    open core
%constants
  %className = "main".
 %classVersion = "".
%clauses
  % classInfo(className, classVersion).
class predicates
dataInput : () nondeterm().
compare: (integer A, integer B, integer C) nondeterm(i,i,i).
clauses
compare(A,B,C) :-  (A<0), (B<0), (C<0), stdIO::write("Ни одного положительного числа").
compare(A,B,C) :-
  ((A>0, B<0, C<0);(A<0, B>0, C<0);(A<0, B<0, C>0)), stdIO::write("Одно положительное число").
 compare(A,B,C) :-
  ((A>0, B>0, C<0);(A>0, B<0, C>0);(A<0, B>0, C>0)), stdIO::write("Два положительных числа").
 clauses
 dataInput():- stdIO::write("Input a:"), stdIO::nl, hasDomain(integer,NUM1),  NUM1 = stdIO::read(),
stdIO::write("Input b:"), stdIO::nl,    hasDomain(integer,NUM2),  NUM2 = stdIO::read(),

%встроенный предикат hasDomain(тип,переменная)  даёт информацию VIP на этапе компиляции о том,
%какого типа будет вводиться переменная.

stdIO::write("Input c:"), stdIO::nl,  hasDomain(integer,NUM3),   NUM3 = stdIO::read(),
compare(NUM1, NUM2, NUM3),
% NUM3 = stdIO::read(),
 stdIO::nl.
 clauses
 run():-
  console::init(),  dataInput(),  fail.
 run():- stdIO::write("End of test...").
 end implement main
 goal     mainExe::run(main::run).
```

Пример 4. **Факториал**

implement main
open core, console

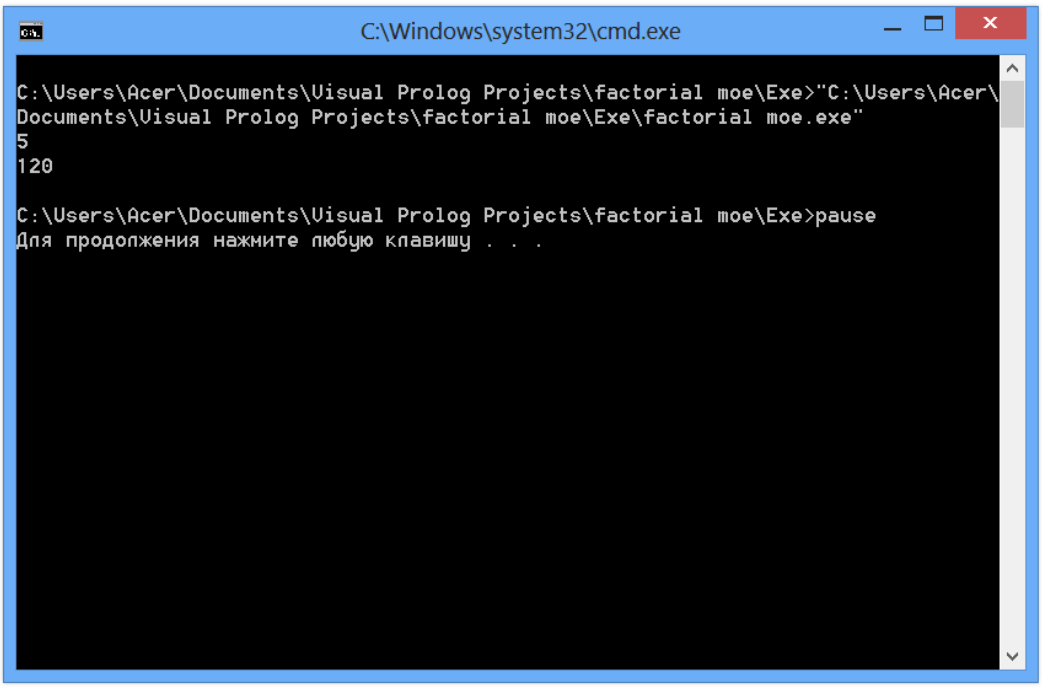
class predicates

*% определяется бинарный предикат **fact** с известным первым и неизвестным вторым*
*%аргументами. Ключевое слово **procedure** описывает поведение предиката, указывая,*
%что его вычисление всегда будет успешным и будет найдено ровно одно решение, так
%что откаты не понадобятся:
fact : (integer N, integer F) procedure (i,o).

% procedure (i,o). Объявленная схема утверждает, что 1-й аргумент N обязан быть входным, то есть, что он является константой, а не свободной переменной. В этом случае предикат принимает данные через свой аргумент. Вообще говоря, вы можете не указывать все схемы входа-выхода аргументов. Их самостоятельно выведет компилятор. Если он не сможет это сделать, то выдаст сообщение об ошибке, и вы сможете добавить необходимую схему.
clauses

%вызов предиката с нулевым первым аргументом.
fact(N, 1) :- N<1, !.   / отсечение, предотвращает откат ко второй формуле*/*
%вызов предиката с произвольным первым аргументом.

```
        fact(N, N*F) :- fact(N-1, F).
        %рекурсивное вычисление факториала

         run():- console::init(),
         fact(stdio::read(), F), /* Ввод числа N
        stdio::write(F,"\n").    /*  Вывод F
end implement main
goal
mainExe::run(main::run).
```



Пример 5. П**рограмма выводит имена всех трёх персон.**

```
implement main
  open core, console


class facts
  person:(string).

clauses
  person("Иван").
  person("Петр").
  person("Сергей").

class predicates
  man:(string) nondeterm (o).

clauses
  man(X) :- person(X).

clauses
  run() :- init(),
    man(X), write(X), nl, fail.
  run().
end implement main

goal
  mainExe::run(main::run).
```

Использование предиката fail

Пролог начинает поиск с возвратом, когда вызов завершается неудачно. В определенных ситуациях бывает необходимо инициализировать выполнение поиска с возвратом, чтобы найти другие решения. Пролог поддерживает специальный предикат **fail**, вызывающий неуспешное завершение, и, следовательно, инициализирует возврат.

Программа 6. Выводит название книги, с числом страниц > 300

```
implement main
  open core, console

class facts
  book:(symbol Title, integer Pages).
written_by:(symbol Author, symbol Title).

class predicates
  long_novel:(symbol Title) nondeterm (o).

clauses

  written_by("fleming","DR NO").
  written_by("melville","MOBY DICK").

  book("MOBY DICK",250).
  book("DR NO",310).

clauses

  long_novel(Title):-
written_by(_,Title),
book(Title,Length),
Length > 300.
clauses
   run():- init(),
   long_novel(Title), write(Title, "\n"),nl, fail.
  run().


end implement main

goal
  mainExe::run(main::run).
```

Дополните программу информацией о книгах, изданных на русском языке. Добавьте в программу правила :

– отыскивающее самую тонкую книгу в БД;
– отыскивающее самую толстую книгу в БД;
– отыскивающее все книги в БД, не являющиеся самой тонкой и самой толстой;
– определяющее наличие самой тонкой и самой толстой книги в Базе Данных среди произведений заданного автора;
– определяющее самую тонкую и самую толстую книгу для заданного автора;
– отыскивающее в БД для заданного автора все книги, которые он не писал.

Стандартные предикаты, предназначенные для ввода и вывода информации.

В *Пролог* включены три стандартных предиката для вывода. Это:

- предикат write;
- предикат nl;
- предикат writef.

Предикат write может быть вызван с произвольным числом аргументов:

write(Param1,Param2,Param3,...,ParamN) *% (i,i,i,...,i)*

Эти аргументы могут быть либо константами из стандартных доменов, либо переменными. Если это переменные, то они должны быть входными параметрами.

Стандартный предикат nl (от англ. new line - новая строка) всегда используется вместе с предикатом write. Он обеспечивает переход на новую строку на экране дисплея. Например, следующие подцели:

```
pupil(PUPIL,CL),

write(PUPIL,"is in the",CL,"class"),

nl,

write("----------------------------").
```

могут привести к выводу на экран такого результата:

Helen Smith is in the fourth class

Пролог включает в себя несколько стандартных предикатов для чтения. Из них четыре основных:

- readln - для чтения всей строки символов;
- readint - для чтения целых значений;
- readreal - для чтения вещественных значений;
- readchar - для чтения символьных значений.

Предикат readln читает текстовую строку, используя формат: readln(Line) % (o)

Домен для переменной Line должен быть строкового типа. Перед тем как вы вызовете readln, переменная Line должна быть свободна, readln считывает до 254 символов.

readint(X) *% (o)*

Домен для переменной X должен быть целого типа, а X перед вызовом должна быть одна. Предикат readint будет считывать целое значение с текущего входного устройства (возможно, с клавиатуры), пока не будет нажата клавиша Enter.

Пример 7. Чтение целых чисел в список.

```
……..
domains
    list=integer*
  predicates
    readlist(list)
  clauses
    readlist([H|T]):-
      write("> "),
      readint(H),!,
      readlist(T).
    readlist([ ]).
……..
```

Пример 8. **Наибольшее из двух чисел**

```
implement main
open core, console
 class predicates
  clauses
  run():-
  init(),
  write("Введите X:"), X= read(),

    write("Введите Y:"), Y= read(),

   clearInput,
  (     X>Y,write("Максимальное число: ",X);     write("Максимальное число: ",Y)    ),
 _=readchar(),
fail;
  write("\nПрограмма завершена"),
     _=readchar().
end implement main
goal
  mainExe::run(main::run).
```

*Очистка буфера клавиатуры с помощью **console::clearInput()** нужна перед предикатом ожидания Enter _=readline(). Дело в том, что предикат read не очищает буфер клавиатуры,*

откуда он читает. Поэтому если перед _=readline() не очистить буфер, то никакого ожидания не будет и _=readline() считает, что есть в буфере от последнего ввода, и закроет окно приложения.

Пример 9. Квадрат числа

```
implement main
  open core, console, programControl
class predicates
  f : (real) -> real.
  clauses     f(X) =X*X.
 run():-init(),
  write("X = "),         X = read(),
    write("Y = ", f(X)),
   sleep(1000).
end implement main
 goal
   mainExe::run(main::run).
```

Пример 10. Корни квадратного уравнения

Задание. Реализовать представленную программу на языке Пролог в среде Visual Prolog

```
корни():-
A= read(),
(A<>0,B= read(),C= read(),
V=B^2-4*A*C,
( V>0, D=sqrt(abs(V)), write((-B+D)/2/A, (-B-D)/2/A),!;
V=0, write(-B/2/A),!;
write("комплексные корни") ),!;
write("нельзя A=0")).
```

Пример 11 – Решение логических задач

Так как Prolog является языком логического программирования, то основная задача данного языка – решать логические задачи. Следующая программа определяет, является ли утверждение верным:

```
implement main
 open core

class predicates
likes:(string,string) determ (i,i).

clauses

likes("ellen","tennis").
```

```
likes("john","football").
likes("tom","baseball").
likes("eric","swimming").
likes("mark","tennis").
likes("bill",Activity):-likes ("tom", Activity). %bill любит то же, что и tom

 run():-
 console::init(),
 if (likes("bill","baseball")) then  %Выясняется, любит ли bill бейсбол,

%выясняется, любит ли tom бейсбол.

%Так как существует предикат likes("tom","baseball"),

% то tom любит бейсбол, значит и bill любит бейсбол.
 stdio::write("true", "\n")
 else stdio::write("false", "\n")

 end if,
        programControl::sleep(5000),

 succeed().

end implement main

goal
 mainExe::run(main::run).
```

В результате получили, что bill любит бейсбол и на консоль выводится true.

Логические связки

Если в примере конструкцию **if…end if** записать следующим образом:

```
if (likes("bill","baseball"),likes("john","tennis")) then
stdio::write("true")
else stdio::write("false")
end if
```

то программа определит, верно ли, что bill любит бейсбол **и** john любит теннис (то есть запятая “**,**” используется как “**И**”).

Если в примере предикат

likes("bill",Activity):-likes ("tom", Activity)

заменить предикатом

likes("bill",Activity):-not (likes ("tom", Activity)),

то получим, что bill не любит то, что любит tom (то есть “**not**” используется как “**не**”).

Задание

Измените программу из примера , чтобы проверить, являются ли верными утверждения:

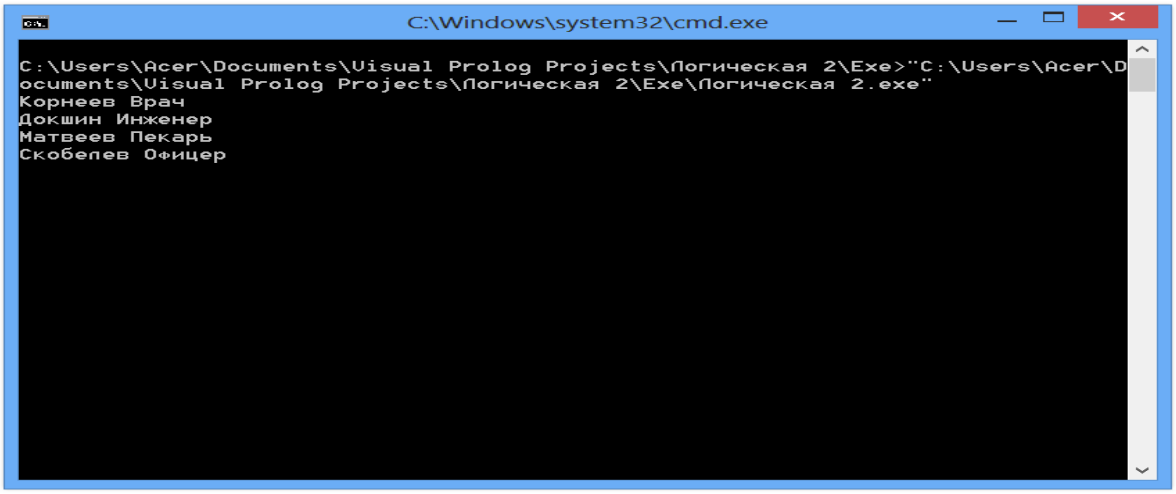
1. tom любит бейсбол, и ellen не любит футбол, и mark любит теннис.
2. eric любит плавание, и bill не любит теннис, и ellen любит плавание.
3. ellen любит футбол, при этом установить, что ellen не любит то увлечение, которое любит mark.

Пример 12. Логическая задача

Корнеев, Докшин, Матвеев и Скобелев – жители одного города. Их профессии – пекарь, врач, инженер и офицер полиции. Корнеев и Докшин – соседи и всегда на работу ездят вместе. Докшин старше Матвеева. Корнеев регулярно обыгрывает Скобелева в пинг-понг. Пекарь на работу всегда ходит пешком. Офицер полиции не живет рядом с врачом. Инженер и офицер полиции встречались единственный раз, когда последний оштрафовал инженера за нарушение правил уличного движения. Офицер полиции старше врача и инженера. Определите, кто и чем занимается.

```
implement main
    open core, console, string
domains

class facts
man:(string).
prof:(string).

class predicates
solve:(string, string) nondeterm (i,o).

clauses
man("Корнеев").
 man("Докшин").
man("Матвеев").
man("Скобелев").
prof("Офицер").
prof("Пекарь").
prof("Врач").
prof("Инженер").

%Корнеев не пекарь и не офицер, потому что пекарь ходит пешком, а офицер не живет с врачом.

solve(X, Y):- man(X), prof(Y),
X="Корнеев"
, Y<>"Пекарь", Y<>"Офицер".

%Докшин не офицер потому что младше его и не пекарь потому что старше его.
solve(X, Y):- man(X), prof(Y),
X="Докшин", Y<>"Офицер", Y<>"Пекарь".

%Скобелев не инженер, потому что инженер либо Корнеев, Матвеев или Докшин
solve(X, Y):- man(X), prof(Y),
X="Скобелев", Y<>"Инженер".

%мареев не офицер потому что докшин старше мареева и Кореева
```

```
solve(X, Y):- man(X), prof(Y),X="Матвеев", Y<>"Офицер".

solve(X, Y):- man(X), prof(Y).

clauses
run() :-init(),
solve("Корнеев", Y1),
solve("Докшин", Y2),
solve("Матвеев", Y3),
solve("Скобелев", Y4),
Y1<>Y2, Y2<>Y3, Y1<>Y4,Y3<>Y4,Y2<>Y4,Y1<>Y3,
write("Корнеев ",  Y1), nl,write("Докшин ", Y2), nl,write("Матвеев ", Y3),nl,write("Скобелев ", Y4),_=readLine(),!
;
_=readLine().
end implement main
goal
console::run(main::run).
```



Списки.

В Прологе список - это объект, который содержит конечное число других объектов. Списки можно грубо сравнить с массивами в других языках, но, в отличие от массивов, для списков нет необходимости заранее объявлять их размер.

Список, содержащий числа 1, 2 и 3, записывается так: [1,2,3]

Или [dog,cat,canary] или ["valerie ann","jennifer caitlin","benjamin thomas"]

Каждая составляющая списка называется элементом.

Чтобы объявить домен для списка целых, надо использовать декларацию домена, такую как:

domains

integerlist=integer*

Символ **(*)** означает "список чего-либо"; таким образом, **integer*** означает "список целых".

Элементы списка могут быть любыми, включая другие списки.

Список является рекурсивным составным объектом. Он состоит из двух частей - головы, которая является первым элементом, и хвоста, который является списком, включающим все последующие элементы. ***Хвост списка - всегда список, голова списка - всегда элемент***. Например:

голова [a,b,c] есть a

хвост [a,b,c] есть [b,c]

Если выбирать первый элемент списка достаточное количество раз, вы обязательно дойдете до пустого списка **[ ]**. Пустой список нельзя разделить на голову и хвост. Это значит, что список имеет структуру дерева, как и другие составные объекты. Структура дерева **[a,b,c,d]** представлена на рис. 1.

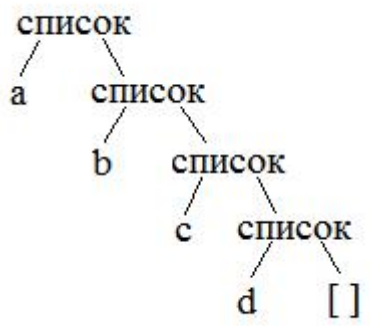


Рис.1. Структура дерева

Есть способ явно отделить голову от хвоста. Вместо разделения элементов запятыми, это можно сделать вертикальной чертой "|". Например: **[a,b,c] эквивалентно [a| [b,c]]** и, продолжая процесс, **[a| [b,c]] эквивалентно [a | [b| [c]]], что эквивалентно [a| [b| [c| []]]]**

Пример 13. Списки

```
implement main
    open core, console

%constants
   % className = "main".
   % classVersion = "$JustDate:  $$Revision:  $".

class predicates
member:(integer,integer*) nondeterm (i,i) (o,i).

%clauses
%classInfo(className, classVersion).

clauses
member(A,[A|_]).
member(A,[_|B]):- member(A,B).

clauses
    run():-init(),
        if member(3,[1,2,3,4,5]),! then write("3 - элемент списка"),nl
        else write("не элемент списка"),nl
        end if,
```

```
      % (member(3,[1,2,3,4,5]),write("3 - элемент списка"),nl,!;succeed),        % логический аналог if-then-else
       nl,

       foreach member(A,[1,2,3,4,5]) do write(A," - элемент списка   ") end foreach,
       nl,
       %(member(A,[1,2,3,4,5]),write(A," - элемент списка   "),fail;succeed),        % логический аналог foreach

       _=readline().
end implement main

goal
   mainExe::run(main::run).
```

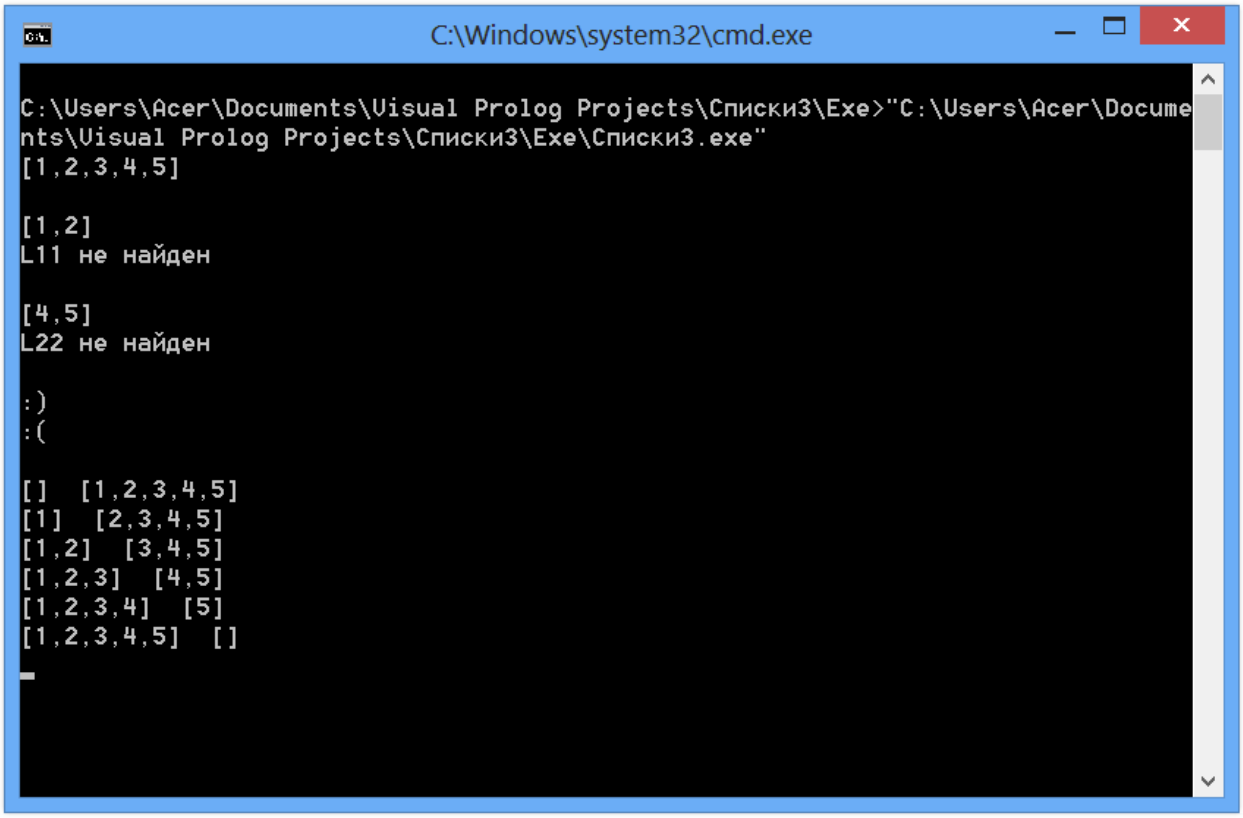
Пример 14. Список 1

```
implement main
open core, console

constants
%className = "main". Считает сумму элементов списка целых чисел и число элементов
%classVersion = "". При вводе данных после последнего элемента введите 0.

class predicates
ввод_списка: (integer*) procedure (o).
sum1: (integer*,integer,integer,integer,integer) procedure (i,i,o,i,o).
result:().

clauses
%classInfo(className, classVersion).

result():-
ввод_списка(L),
sum1(L,0,S1,0,C1),write("Summa1="),write(S1),nl,
write("Count1="),write(C1),nl.

ввод_списка([H|T]):- H=read(),H<>0,!,ввод_списка(T).
ввод_списка([]).

sum1([X|T],S1,S,C1,C):-!,sum1(T,S1+X,S,C1+1,C).
sum1([],S,S,C,C).

run():-
console::init(),
result(),
clearInput(),
_=readchar().
end implement main

goal
mainExe::run(main::run).
```

Списки 3.

```
implement main
 open core, console
 %constants    className = "main".
 %classVersion = "$JustDate:  $$Revision:  $".
 class predicates
 append:(Z*,Z*,Z*) nondeterm (i,i,i) (o,i,i) determ (i,o,i) multi (o,o,i) procedure (i,i,o).
 clauses
 %classInfo(className, classVersion).
 clauses
 append([],L,L).append([A|L1],L2,[A|L]):- append(L1,L2,L).
 clauses
 run():-init(),
 append([1,2],[3,4,5],L),
 write(L),nl,nl,       % procedure (i,i,o)
 (append(L1,[3,4,5],[1,2,3,4,5]),
```

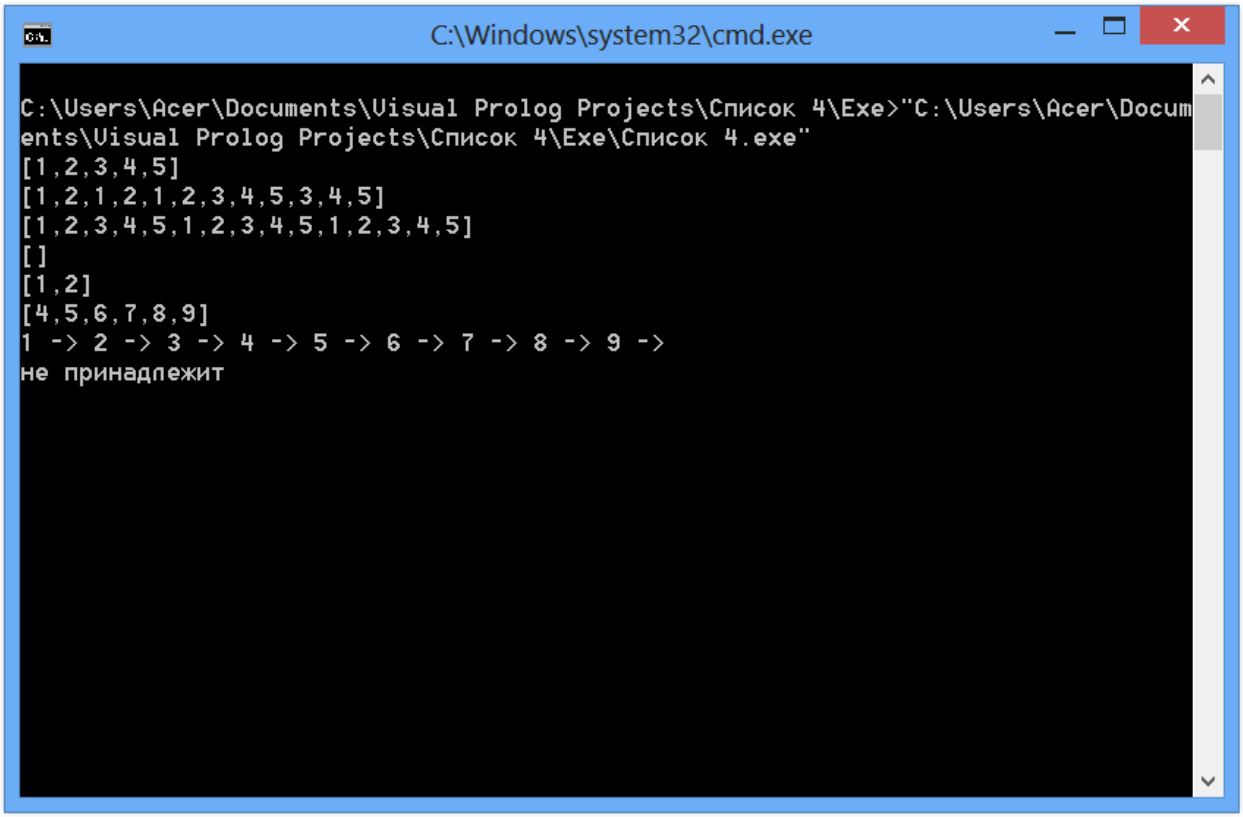
```
write(L1),nl,!; write("L1 не найден"),nl),   %nondeterm (o,i,i)
(append(L11,[3,4],[1,5]),
write(L11),nl,!; write("L11 не найден"),nl),nl,   %nondeterm (o,i,i)
(append([1,2,3],L2,[1,2,3,4,5]),
write(L2),nl,!; write("L2 не найден"),nl),    %determ (i,o,i)
(append([9],L22,[4,5]), write(L22),nl,!; write("L22 не найден"),nl),nl,       %determ (i,o,i)
(append([1,2,3],[4,5],[1,2,3,4,5]), write(":)"),nl,!; write(":("),nl),        %nondeterm (i,i,i)
(append([1,0],[5],[9,8,4,5]), write(":)"),nl,!; write(":("),nl),nl,           %nondeterm (i,i,i)
(append(X,Y,[1,2,3,4,5]), write(X,"  ",Y),nl,fail; succeed),                  %multi (o,o,i)
_=readline().
end implement  main
goal    mainExe::run(main::run).
```



Пример 15. Списки 4

```
implement main
  open core, console, list
  %constants    className = "main".    classVersion = "$JustDate:  $$Revision:  $".
  class predicates
  clauses
  %classInfo(className, classVersion).
  run():-init(),
  L=[1,2], L1=[3,4,5],
  write(append(L,L1)),nl, % соединение двух списков
  write(append(L,L,L,L1,L1)),nl,  % соединение пяти списков
  write(appendList([L,L1,L,L1,L,L1])),nl, % соединение списка списков
  L2=[1,2,3,4,5,6,7,8,9],
  write(difference(L,L2)),nl,  % разность множеств(списков): L\L1
  write(intersection(L,L2)),nl, % пересечение двух множеств(списков)
  write(drop(3,L2)),nl,  % отрезаем первые три элемента и возвращаем остаток
  foreach  E=getMember_nd(L2) do write(E," -> ") end foreach,nl,
  if isMember(1,L1) then write("принадлежит") else write("не принадлежит")
```

```
end if,nl,
_=readline().
end implement main
goal    mainExe::run(main::run).
```



Пример 16. Простая экспертная система.

```
implement main
 open core,console
  class facts
  fact: (integer,string).
  class predicates
  rule: (integer,string,string,integer*) nondeterm (o,i,o,o).
  ask: (integer) determ.
  recognition: (string) .
  discover: (integer*)  nondeterm.
  complete: (integer) determ.
 % constants

  %className = "ЭС".
  %classVersion = "001".
%  clauses
 % classInfo(className, classVersion).

  clauses
rule(1,"блюдо","первое",[1,2]).
 rule(2,"блюдо","фастфуд",[1,3]).
 rule(3,"блюдо","второе",[1]).
 rule(4,"блюдо","салат",[4]).
 rule(5,"блюдо","десерт",[5]).
 rule(6,"первое","суп щавелевый с галушками",[6,10,12,18,27]).
 rule(7,"первое","рассольник",[6,12,16,27]).
 rule(8,"первое","борщ",[6,12,14]).
 rule(9,"первое","куриный суп",[7,11,12]).
 rule(10,"первое","суп с фрикадельками",[8,11,12]).
 rule(11,"первое","молочный суп",[11,26]).
 rule(12,"первое","уха",[9,12]).
 rule(13,"второе","пюре с котлетой",[8,12,29]).
 rule(14,"второе","голубцы",[8,13,14]).
 rule(15,"второе","плов",[6,13]).
 rule(16,"второе","манты",[8,10,28]).
 rule(17,"второе","пельмени",[8,10]).
 rule(18,"второе","мясо по-французски",[6,12,20,22]).
 rule(19,"второе","рыба фаршированная",[9,24]).
 rule(20,"фастфуд","пицца",[10,15,17,20,21]).
 rule(21,"фастфуд","бутерброд",[10,17,20,22]).
 rule(22,"салат","зимний",[6,12,16,18,22,29]).
 rule(23,"салат","греческий",[15,16,20,21,22]).
 rule(24,"салат","хе",[6,19,27]).
 rule(25,"салат","красная шапочка",[12,18,20,22,23]).
 rule(26,"салат","летний",[15,16,22]).
 rule(27,"салат","нежность",[7,12,16,18,20,22]).
 rule(28,"десерт","блин сладкий",[10,24,26]).
 rule(29,"десерт","блин",[10,26]).
 rule(30,"десерт","шоколадные конфеты",[24,25]).
 rule(31,"десерт","торт",[10,24]).
 rule(32,"десерт","мороженое",[26,24,30]).
 rule(33,"десерт","мороженое",[26,25,30]).
 rule(34,"десерт","мороженое",[26,30]).
 rule(35,"десерт","молочный коктейль",[26]).

ask(X):-fact(X,"y"),!.
 ask(X):-fact(X,"n"),!,fail.
 ask(1):-write("блюдо должно подаваться горячим/теплым ?"),!,complete(1).
 ask(2):-write("блюдо жидкое?"),!,complete(2).
 ask(3):-write("блюдо быстрого приготовления?"),!,complete(3).
 ask(4):-write("блюдо легкое?"),!,complete(4).
 ask(5):-write("блюдо сладкое?"),!,complete(5).
 ask(6):-write("из мяса?"),!,complete(6).
 ask(7):-write("из курицы?"),!,complete(7).
 ask(8):-write("из мясного фарша?"),!,complete(8).
 ask(9):-write("из рыбы?"),!,complete(9).
```

```
ask(10):-write("с тестом / из теста?"),!,complete(10).
ask(11):-write("с лапшой?"),!,complete(11).
ask(12):-write("с картофелем?"),!,complete(12).
ask(13):-write("с рисом?"),!,complete(13).
ask(14):-write("с капустой?"),!,complete(14).
ask(15):-write("с помидорами?"),!,complete(15).
ask(16):-write("с солеными/свежими огурцами?"),!,complete(16).
ask(17):-write("с колбасой?"),!,complete(17).
ask(18):-write("с яйцом?"),!,complete(18).
ask(19):-write("с морковью?"),!,complete(19).
ask(20):-write("с сыром?"),!,complete(20).
ask(21):-write("с оливками?"),!,complete(21).
ask(22):-write("с майонезом?"),!,complete(22).
ask(23):-write("с гранатами?"),!,complete(23).
ask(24):-write("с начинкой?"),!,complete(24).
ask(25):-write("с шоколадом?"),!,complete(25).
ask(26):-write("из молока?"),!,complete(26).
ask(27):-write("кисловатое?"),!,complete(27).
ask(28):-write("на пару?"),!,complete(28).
ask(29):-write("традиционное?"),!,complete(29).
ask(30):-write("очень холодное?"),!,complete(30).

 recognition(X):- rule(N, X, Y, Z), discover(Z),!,
 write("       это более всего похоже на ", X, " - ", Y, " (правило ", N,")"), nl,recognition(Y).
 recognition("блюдо"):- write("это блюдо системе не известно!"),nl,!.
 recognition("первое"):-
write("это блюдо системе не известно, но его состав очень оригинален!"),nl,!.
 recognition("второе"):-
write("это блюдо системе не известно, но его состав очень оригинален!"),nl,!.
 recognition("фастфуд"):- write("это блюдо системе не известно!"),nl,!.
 recognition("салат"):-
write("это блюдо системе не известно, но его состав очень оригинален!"),nl,!.
 recognition("десерт"):- write("это блюдо системе не известно!"),nl,!.
 recognition(_).
 discover([X|Y]):- ask(X),discover(Y).
 discover([]).

 complete(X):-
write(" y/n"),nl, Y=readline(), ( (Y="y";Y="Y"),assert(fact(X, "y")),!; assert(fact(X, "n")),!,fail).
 run():-init(),
 recognition("блюдо"),
 _=readchar().
 end implement main
 goal
 mainExe::run(main::run).
```

Факты для предикатов базы фактов могут быть определены во время компиляции в разделе clauses, как это показано в последнем примере. Во время выполнения факты могут быть добавлены и удалены, используя описанные ниже предикаты. Обратите внимание, что факты, определенные во время компиляции в разделе clauses, также могут быть удалены, они ничем не отличаются от фактов, добавленных во время выполнения.

Стандартные предикаты *Пролога* для работы с фактами: assert, asserta, assertz, retract, retractall, consult и save - могут иметь один или два аргумента. Необязательный второй аргумент представляет собой имя внутренней базы фактов. Обозначение /1 и /2 после каждого имени предиката указывает необходимое число аргументов для данной версии предиката. Комментарии (такие как */* *(i)* */* и */* *(o,i)* */*) показывают поток(и) параметров для этого предиката.

Занесение фактов во время выполнения программы

Но время выполнения факты могут быть добавлены во внутреннюю базу данных фактов посредством предикатов: assert, asserta и assertz, или путем загрузки фактов из файла с помощью consult.

Существует три предиката для добавления одного факта во время выполнения:

asserta(the fact) *% (i)*

asserta(the fact,facts_sectionName) *% (i,i)*

assertz(the fact) *% (i)*

assertz(the fact,facts_sectionName) *% (i,i)*

assert(the fact) *% (i)*

assert(the fact,facts_sectionName) *% (i,i)*

Предикат asserta вставляет новый факт в базу данных фактов перед имеющимися фактами для данного предиката, а assertz вставляет факты после имеющихся фактов данного предиката. Использование предиката assert дает результат, аналогичный использованию assertz.

Поскольку имена предикатов базы фактов уникальны внутри программы, для предикатов asserta и assertz всегда известно, в какую базу данных фактов нужно добавлять факт. Однако для того, чтобы обеспечить работу с требуемой базой данных фактов, в целях проверки типа можно использовать необязательный второй аргумент.

Здесь предикат discover задаёт вопросы по одному:

```
discover([X|Y]):- ask(X),discover(Y).
discover([]).
```

Организация проектов в среде программирования Visual Prolog

Проект и пакеты

Целью построения проекта является связывание объектных файлов **(.obj и .lib)** и генерация исполняемого файла **(.exe или .dll).** Объектный файл **(.obj)** является результатом компиляции пакета - файла с расширение **.pack**.

Пакет, как правило, является функционально законченной единицей программирования, выполняющей заданные функции. Пакет является текстовым файлом и включает (с использованием директивы *#include или #requires*) все файлы, необходимые для его успешной компиляции.

Файлы, перечисленные в файле пакета **(.pack**), могут использоваться только файлами этого же пакета и не видны другим пакетам. Пакет в других пакетах представляет объявление пакета - файл с расширением **.ph**.

Файлы деклараций классов пакета и интерфейсов **(.cl и .i)**, к которым могут обращаются классы других пакетов, должны указываться в объявлении пакета (файл **.ph**). Если эти декларации используют домены, константы или интерфейсы других пакетов, то представители этих пакетов (файлы **.ph**) должны указываться в файле **.ph**.

Проект является множеством пакетов, внешних библиотек и некоторых других вспомогательных файлов.

Видимость и использование пакетов

При создании пакета средствами среды создается полный набор файлов, необходимых для компиляции пакета, но в проект включается только файл пакета. При добавлении существующего пакета в проект добавляется только файл пакета (**.pack**).

Если проект не откомпилирован, то в проектом окне среды видны только файлы пакетов и библиотек. Результаты компиляции пакетов проекта помещаются по умолчанию в директорию OBJ (при этом может быть назначена любая другая директория). Проект становится полностью не откомпилированным, если полностью очистить директорию OBJ или удалить ее.

Файлы пакетов, составляющие сущность программы **(.i, .cl, .pro**) и объявление пакета **(.ph**), становятся видны в проектом окне только в результате компиляции проекта или соответствующего пакета.

Если необходимо использовать в классе данного пакета какой-либо класс другого пакета, то этот другой пакет должен быть включен в проект и либо необходимо вручную прописать включение этот другого пакета в данный пакет написать использование нужного класса. Либо можно в файле **.pro** написать использование нужного класса, но этот другой пакет должен быть предварительно откомпилирован (среда "увидит" его **.ph** файл). Тогда при компиляции будет предложено автоматическое включение другого пакета в данный пакет.

Для удаления пакета из проекта необходимо удалить из проекта сам файл пакета и все ссылки на этот пакет (вручную путем редактирования файлов **.pack и .ph**).

Рекомендуется после этого выполнить операцию RebuildAll, которая очищает всю директорию OBJ и производит полное построение проекта.

Среда разработки

Интегрированная Среда Разработки (Integrated Development Environment - IDE) используется для создания, разработки и поддерки Visual Prolog - проектов.

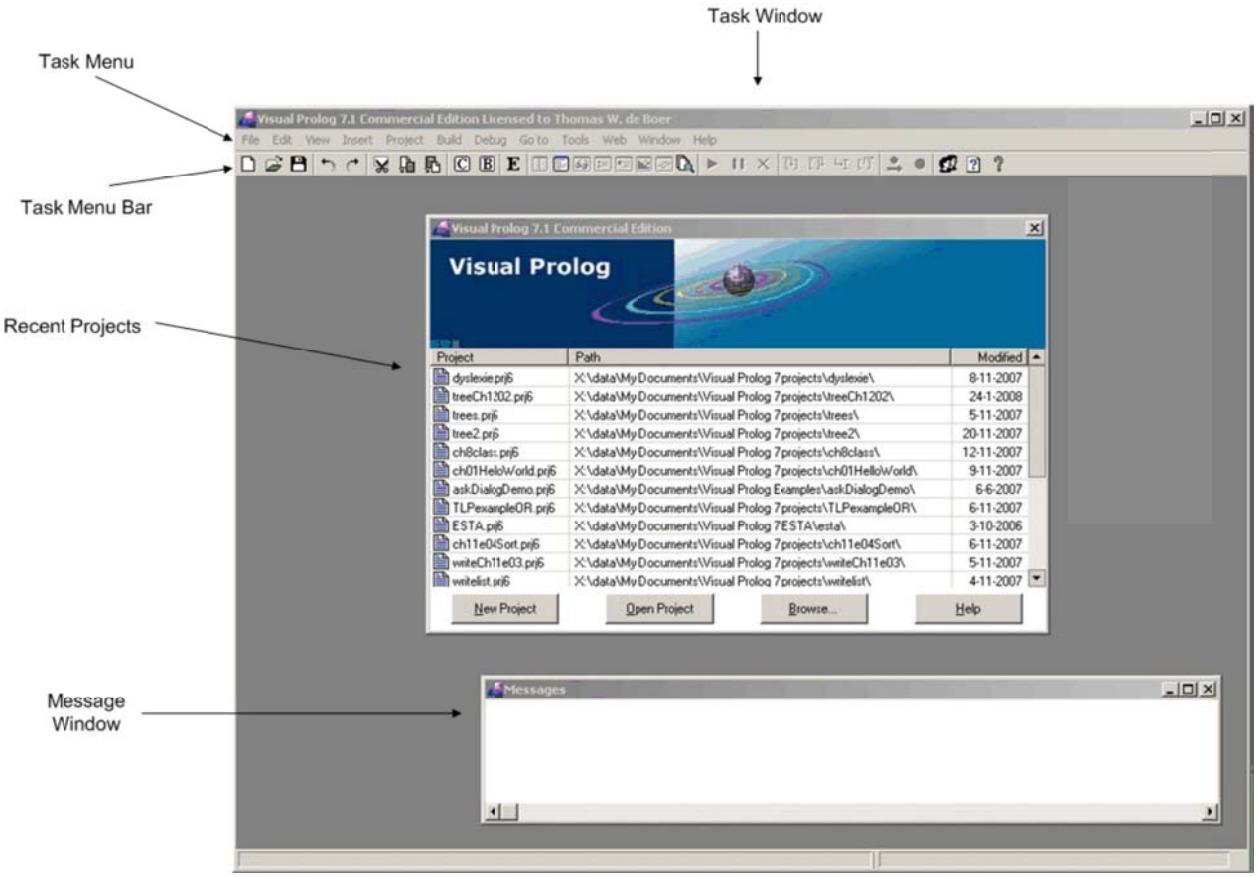
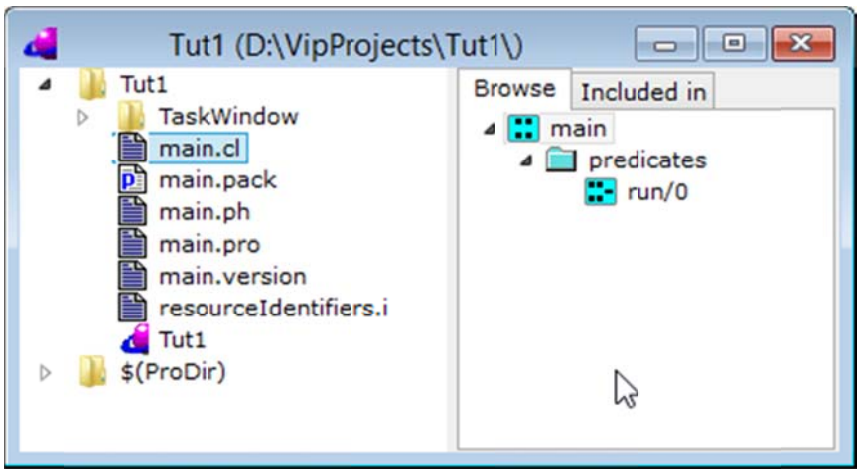


Рис. Среда разработки Visual Prolog

Visual Prolog использует следующие соглашения:

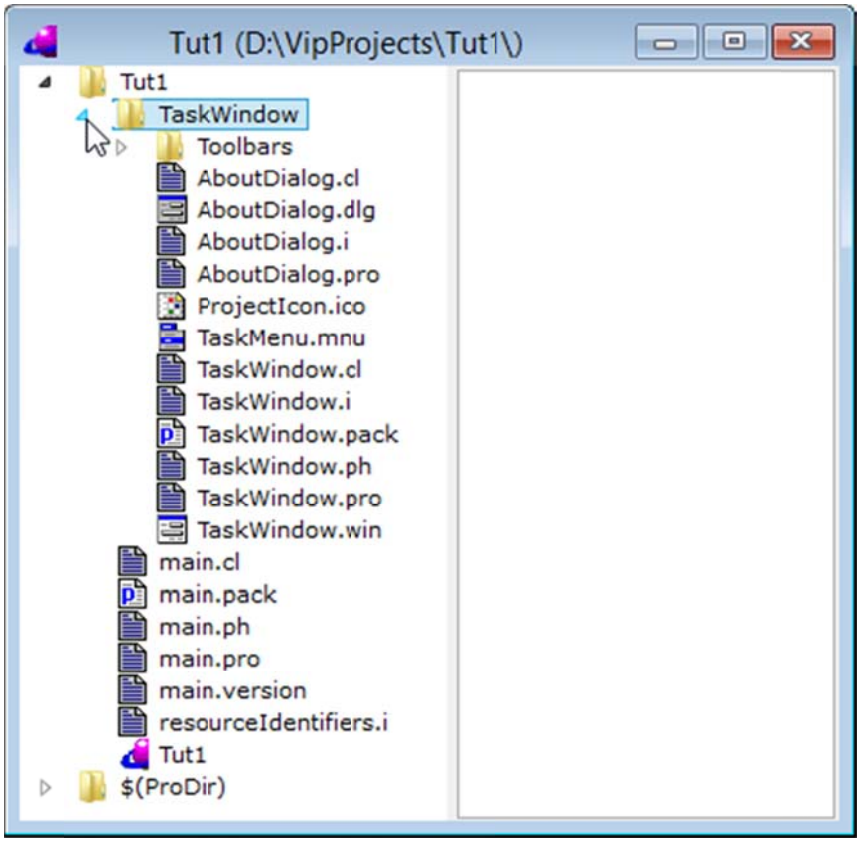
> .**ph** файлы есть **заголовки пакетов (package headers)**. Пакет является набором классов и интерфейсов, которые используются совместно.
> .**pack** файлы есть **packages**. Они содержат исполняемые разделы или конкретизации файлов, перечисленных в соответствующих .ph файлах.
> .**i** файл содерижт **интерфейс (interface)**.
> .**cl** файл содержит**декларацию класса (class declaration)**
> .**pro** файл содерижт **имплементацию класса (class implementation)**.

Если Вы теперь установите указатель на узел с классом main.cl, то вы увидите следующее:



Это поддерево показывает, что файл **main.cl** содержит класс, называемый **main**, который содержит два предиката, с именами classinfo и run, соответственно.

Если теперь Вы раскроете узел **TaskWindow**, то увидите дерево:



Здесь видны несколько новых типов узлов:

.**dlg** файл содержит **диалог (dialog)**

.**frm** файл содержит **форму (form)**

.**win** файл содержит **окно (window)** (окно приложения или обычное окно класса window из PFC GUI)

.**mnu** файл содержит **меню (menu)**

.**ico** файл содержит **иконку (icon)**

.**ctl** файлы, содержащие **элементы управления (controls)**

.**tb** файлы, содержащие **панели инструментов (toolbars)**

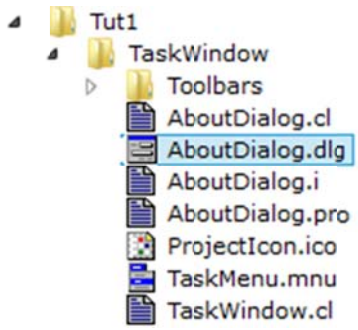
.**cur** файлы, содержащие **курсоры (cursors)**

.**bmp** файлы, содержащие **картинки (bitmaps)**

.**lib - библиотеки (libraries)**

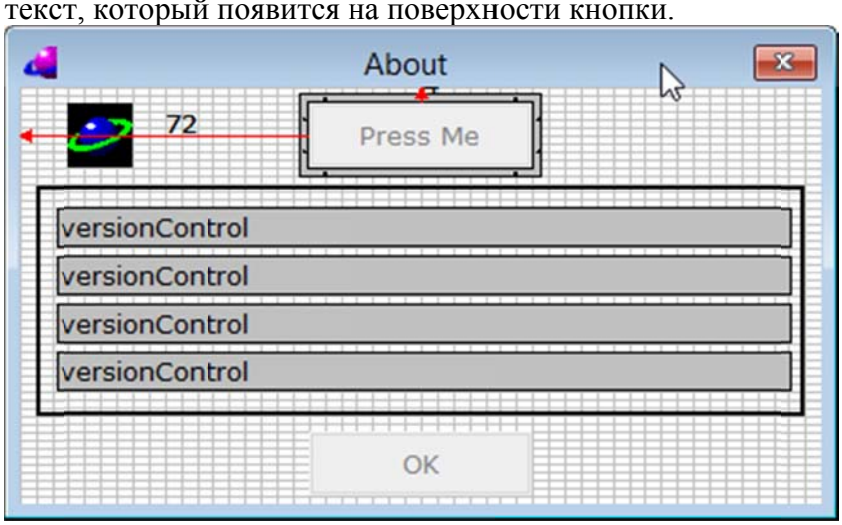
Правая кнопка мышки вызывает контекстное меню с операциями, которые можно выполнить над текущим узлом. Двойной щелчок на узле вызывает редактор соответствующего элемента.

Внесите изменение в **about dialog** (информационный диалог). Откройте диалог **About** в **Dialog Editor (Редакторе Диалогов)**. Для этого сделайте двойной щелчок на диалоге в проектном окне.

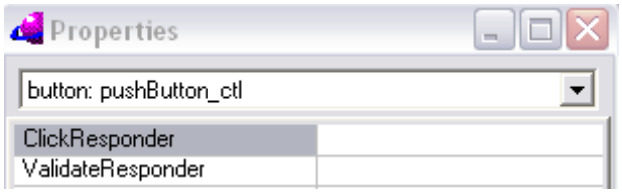


Вы увидите диалог в **Dialog Editor (Редакторе Диалога)**, две панели инструментов и окно свойств. Добавьте кнопку в диалог. Кликните на "**button** (кнопку)" в панели инструментов с элементами управления, а затем кликните в редактируемом диалоге **AboutDialog** где-то после проектной иконки. В результате Вы увидите свойства нового элемента управления, которые Вы сможете редактировать. Поменяйте текст **Text** на "***Press Me*** (Нажми Меня)" -

текст, который появится на поверхности кнопки.



Теперь надо предусмотреть обработку события по нажатию кнопки. Для этого в том же редакторе свойств выберите закладку **Events (События)**.



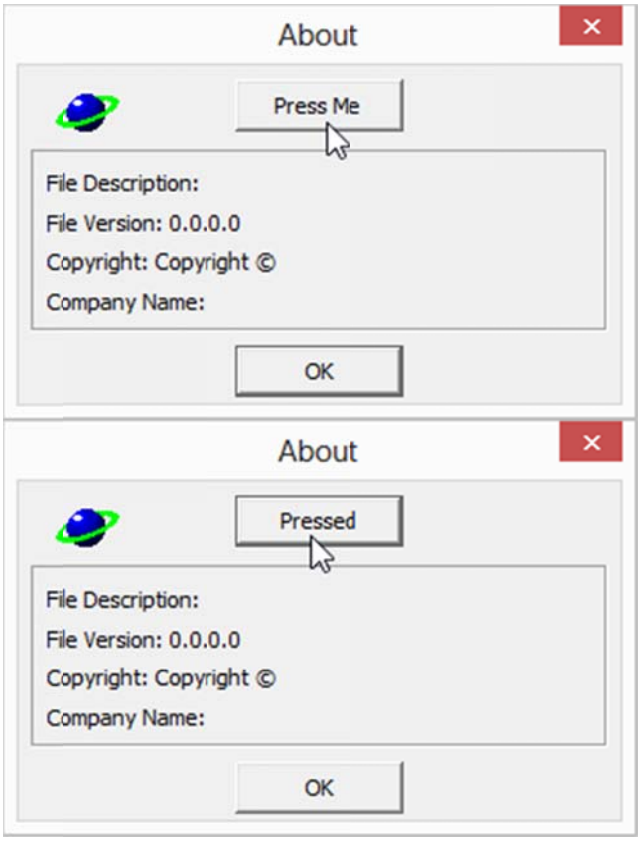
В поле для редактирования свойства с именем **ClickResponder** (приемник события Click) наберите имя предиката, который будет обрабатывать это событие. Поле можно оставить и пустым, при этом IDE сама создаст это имя. Сделайте теперь двойной щелчок, находясь в этом поле. Управление теперь вернется к текстовому редактору (он откроется, если не был открыт) и указатель будет установлен на код, созданный средой (IDE).

Меняйте текст на кнопке, когда она нажимается. Для этого модифицируйте код, как показано:

```
predicates
    onPushButtonClick : button::clickResponder.
clauses
    onPushButtonClick(_Source) = button::defaultAction() :-
        pushButton_ctl:setText("Pressed").
```

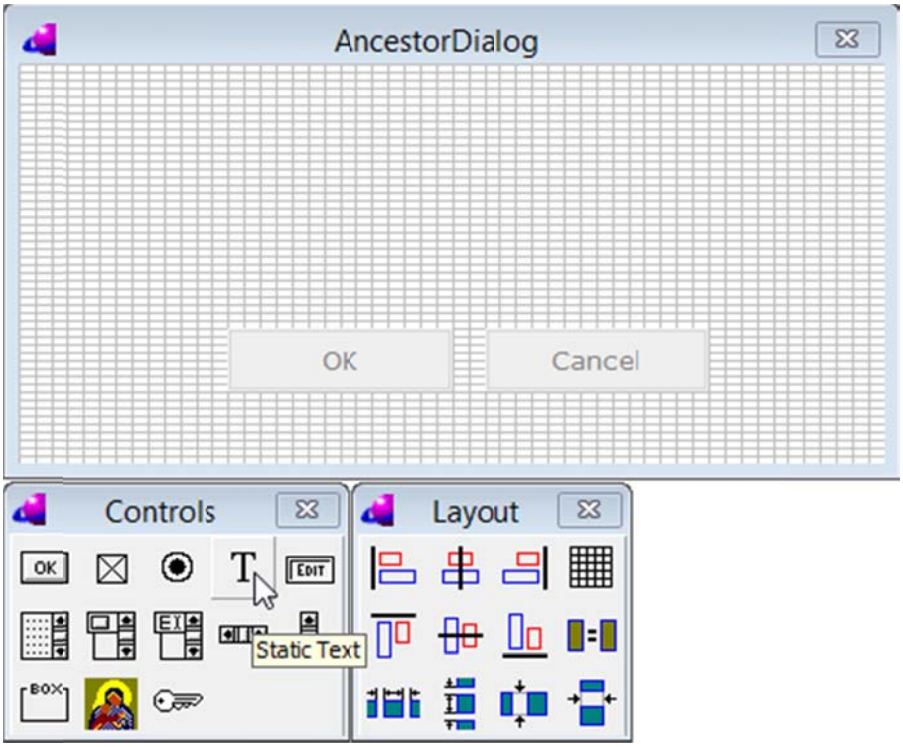
Теперь постройте и запустите приложение опять (нажав **F9**). При запуске приложения нужно открыть диалог **About**, выбрав в меню **Help -> About,** и затем нажав эту вновь созданную кнопку.

Это будет выглядеть таким образом:



Элементы Управления графического интерфеса

Пакет элементов управления содержит набор элементов управления, наиболее используемых в GUI-проектах.



Button (кнопка) - это элемент управления, который дает возможность программе производить некоторое действие, когда пользователь нажимает на эту кнопку. Это обеспечивается ответчиками нажатия (click responders). Ответчик нажатия - это фрагмент

программного кода, который выполняется в ответ на нажатие пользователя на кнопку. Существует три вида кнопок: обычные кнопки, кнопки Выполнить (Ok) и кнопки Отменить (Cancel). Разница между этими типами кнопок связана с использованием кнопок в диалогах dialog и в формах formWindow.

CheckButton (флажок) - это элемент управления, который имеет два (чаще) или три (реже) состояния: установлен, неустановлен и неопределен. Пользователь может менять состояния флажка с помощью щелчка мышью, а также с помощью нажатия клавиши "Пробел" в случае если элемент управления обладает фокусом ввода. Когда это происходит, программа уведомляется при помощи приемника изменения состояния (state changed listeners). Кроме того, в любой момент времени программа может получить состояние каждого флажка.

ContainerControl (контейнер) - это элемент управления, который обеспечивает функциональность интерфейса containerWindow в виде элемента управления PFC/GUI. Этот элемент управления обычно объединяет в группу другие элементы управления.

EditControl (поле редактирования) - это элемент управления для получения от пользователя текстовых данных. Существуют однострочные поля редактирования с фиксированной высотой и многострочные поля редактирования, которые принимают тексты с переносами строк. Поле редактирования уведомляет программу, когда изменяются его текстовые данные, при помощи приемников изменений (modified listeners).

GroupBox (групповой блок) - это элемент управления, который обычно используется для объединения набора элементов управления в группу. Интерфейс GroupBox поддерживает интерфейс containerControl, поэтому он является контейнером-родителем для элементов управления, которые в нем содержатся. Кроме этого, он имеет визуальное представление и обладает некоторым количеством визуальных стилей.

IconControl (пиктограммы) - это элемент управления, который позволяет выводить на экран ресурсы изображений пиктограмм.

IntegerControl (поле целых) - это элемент управления, который позволяет программе получать числовые (а именно, целые) данные от пользователя. Этот элемент управления выглядит в точности как поле редактирования и обеспечивает аналогичную функциональность, за исключением того, что принимает только целые данные. Данный элемент управления имеет внутренний ответчик проверки достоверности (validate responder), который обеспечивает соответствие данных домену integer.

ListBox (список) - это элемент управления, который служит для того, чтобы выводить на экран списки текстовой информации и предоставить возможность выбора одного элемента списка (реже нескольких элементов). Данный элемент управления может сообщить программе о том, что выбор изменился или что пользователь дважды щелкнул мышью на элементе в этом списке.

ListButton (раскрывающийся список) - это элемент управления, который используется для вывода на экран выпадающихся списков и дает возможность выбирать только один элемент из списка. Он обладает приемниками изменения выбора (selection changed listeners).

ListEdit (редактируемый список) - это элемент управления, который используется для вывода на экран выпадающих списков и предоставляет пользователю возможность ввода любой текстовой информации в дополнение к элементам списка.

ListControl является основой для других элементов управления, относящихся к спискам (*ListBox, ListButton, ListEdit*). Его не следует использовать вручную.

RadioButton (переключатель) - это элемент управления, который обычно используется в группах, для того чтобы обеспечить возможность пользователю выбора ровно одной из нескольких альтернатив. Этот элемент управления обладает приемниками изменения выбора (selection changed listeners).

RadioButtonGroup не является элементом управления. Это абстракция, которая служит для того, чтобы было легче поддерживать несколько переключателей. Каждый контейнер типа containerWindow обладает ровно одним объектом группы переключателей, который с ним связан. Это означает, что внутри контейнера может быть только одна группа переключателей.

RealControl (поле действительных) - это элемент управления, который позволяет программе получать числовые данные (а именно, числа с плавающей точкой) от пользователя. Он наследует функциональность поля редактирования, обеспечивающую проверку содержания для того, чтобы принимать только числа с плавающей точкой.

ScrollControl (полоса прокрутки) - это элемент управления, который позволяет пользователю прокручивать данные на экране. Большинство элементов управления может при необходимости иметь собственные полосы прокрутки. Элемент управления ScrollControl может быть использован для создания собственных элементов управления.

TextControl (надпись) - это элемент управления, который выводит на экран статическую текстовую информацию. Обычно надписи используются вместе с другими элементами управления в качестве меток или заголовков.

TimerControl обеспечивает функциональность для реагирования на события с временным критерием. Он не имеет визуального представления, но должен иметь контейнер в качестве родителя. Этот элемент управления полезен для того, чтобы производить фоновые процедуры с временным критерием. Он обеспечивает оповещение при помощи приемников тиков (моментов) (tick listeners).

Создание проекта с графическим (GUI) интерфейсом

Основная цель пакета GUI - обеспечивать удобный способ работы с окнами, элементами управления, событиями и другими средствами GUI (графического интерфейса пользователя). Пакет GUI представляет объектно-ориентированный подход к работе с окнами, диалогами и элементами управления. Этот пакет встроен в верхний слой VPI.

Пакет GUI предоставляет и отвечает за следующие средства:

> Набор обычных компонентов GUI: окна, диалоги, формы, элементы управления (флажки, поля редактирования, групповые блоки, пиктограммы, списки, раскрывающиеся списки, редактируемые списки, кнопки, переключатели, полосы горизонтальной и вертикальной прокрутки, надписи, специальные элементы управления).

Система обработки событий, которая отмечает, когда пользователь взаимодействует с одним из компонентов GUI и передает эту информацию приложению.
Контейнеры - объекты, в которые могут добавляться элементы управления (контейнерами могут быть диалоги, контейнерные элементы управления, формы).
Управление размещением - возможность изменять размер, задавать положение и перемещать объекты GUI.
Поддержка графических операций: нарисовать дугу, залить многоугольник, обрезать прямоугольник и т.д.

Пример создания GUI-проекта.

Создавая GUI-проект в Visual Prolog IDE, убедитесь, что вид проекта (**Project Kind)** установлен в режим **MDI mode.**

IDE создает начальный набор модулей, требуемых для управления GUI в директории **TaskWindow**, которая является поддиректорией проектной директории. Эта директория содержит все необходимые коды для создания главного окна приложения (Task Window), его меню (Menu), панель инструментов (Toolbar) и диалог информации (about dialog).

После создания проекта можно сразу же его скомпилировать, получив пустой каркас GUI-программы, которая поначалу ничего делать не будет, но будет обладать требуемым функционалом .

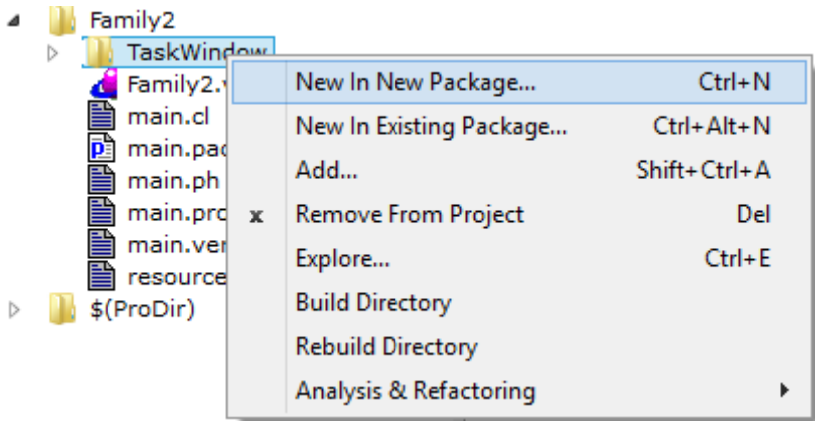
Например, внутри главного окна (Task Window) приложения Visual Prolog выводит другое окно сообщений Messages. Это окно работает как консоль. Когда программист использует предикат stdio::write(...) в программе, вывод будет направлен в это окно.

В консольных приложениях, консоль всегда доступна в виде черной панели, куда программист может выводить данные с помощью предиката stdio::write(...).

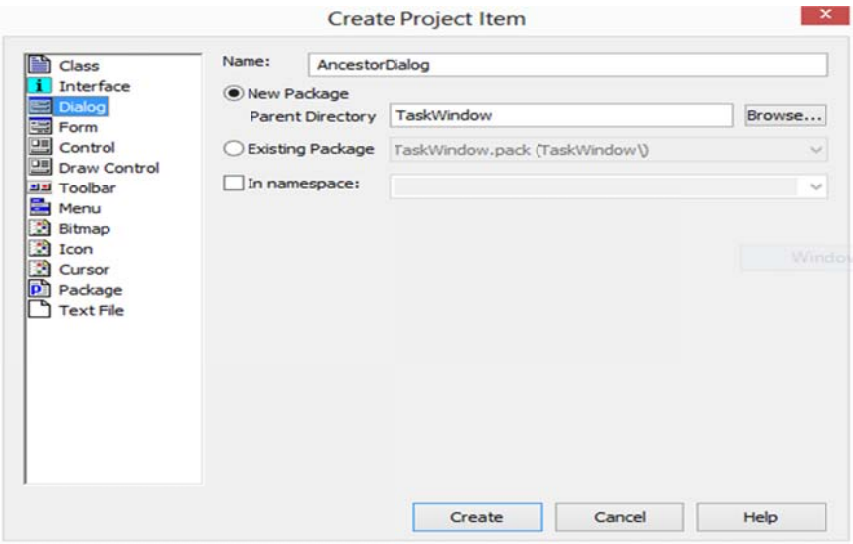
Все графические компоненты сохраняются как отдельные исходные файлы. Visual Prolog делает общую компиляцию и связывание всех файлов автоматически, без участия разработчика.

Создание диалога

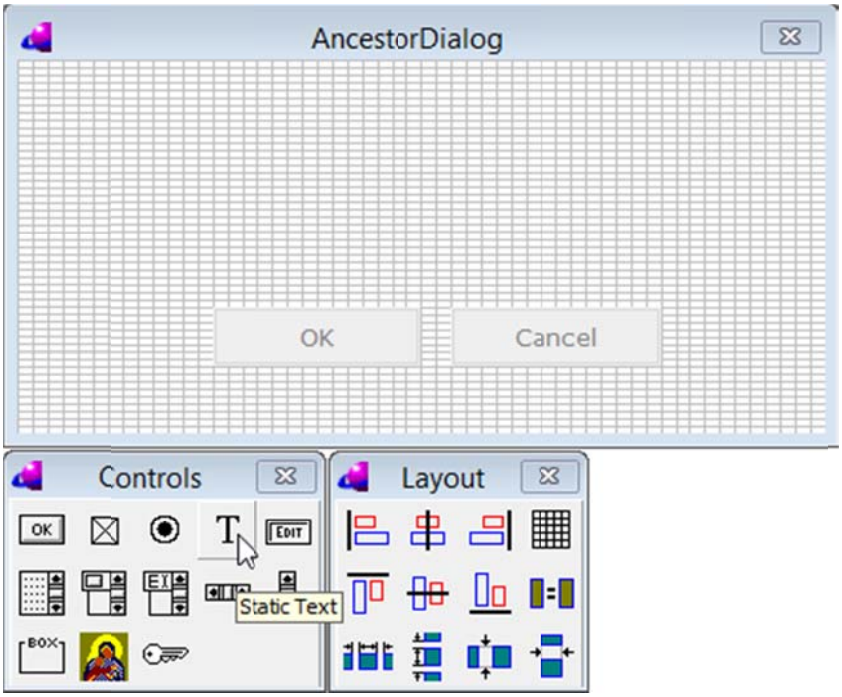
Добавим новый GUI компонент в программу – окно (панель) диалога- для того, чтобы ввести в программу имя персоны. В дереве проекта, где перечислены все модули и источники, щелкните правой кнопкой на ***Task Window*** и из контекстного меню выберите **New in New Package...**



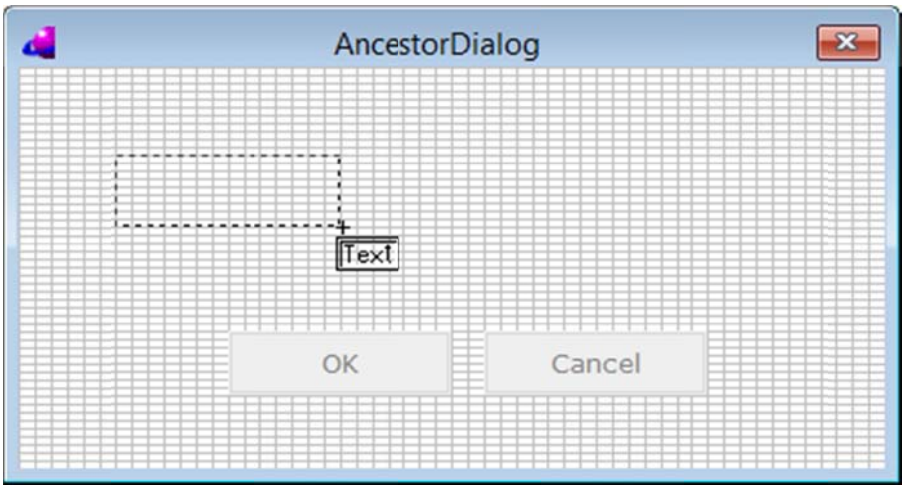
Появится окно диалога **Create Project Item**. Убедитесь, что элемент **Dialog** выбран в левом списке, и в правой части окна введите **AncestorDialog.**



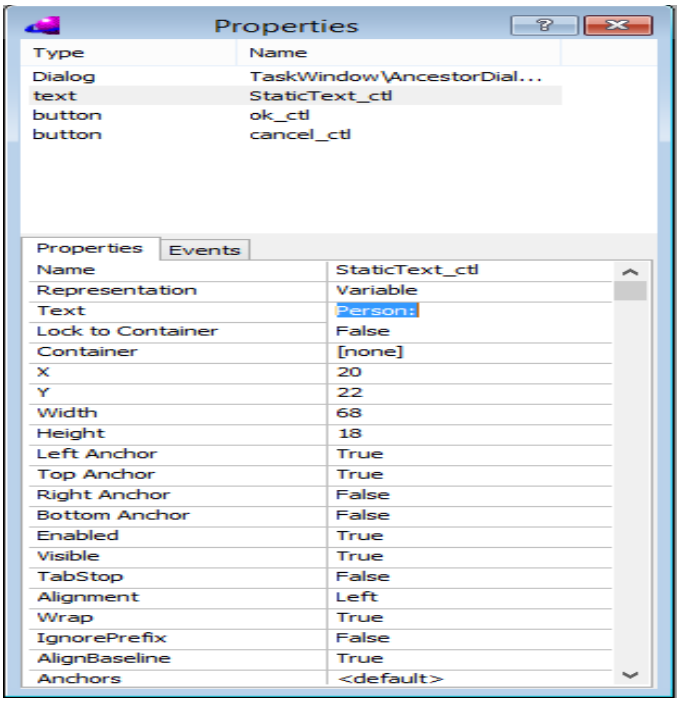
Будет создано пустое окно диалога. Т.к. нам не нужна кнопка **Help,** нужно щелкнуть по ней и нажать клавишу **DEL**. Кнопки можно передвинуть на форме:



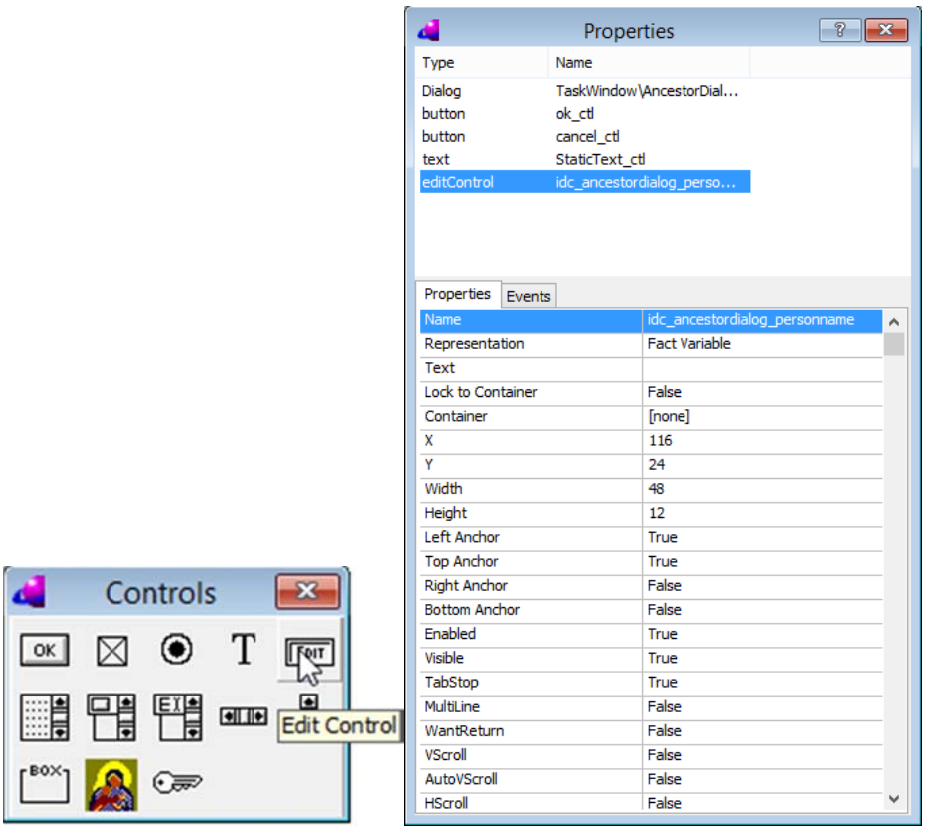
Разместите на форме статическое текстовое поле.



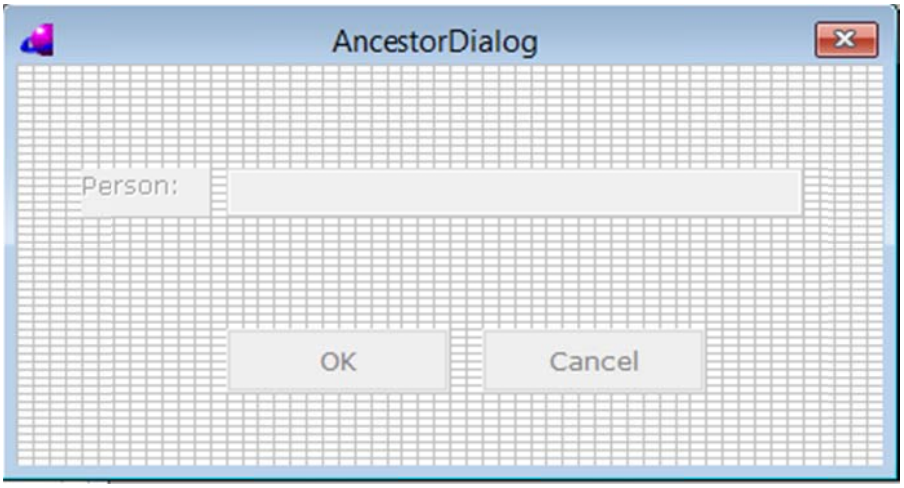
Щелкнув левой кнопкой по текстовому полю, вы увидите Окно свойств, в котором в строке Text field введите "Person".



Аналогично, разместите редактируемое текстовое поле **Edit** для ввода данных, оставив его пустым.



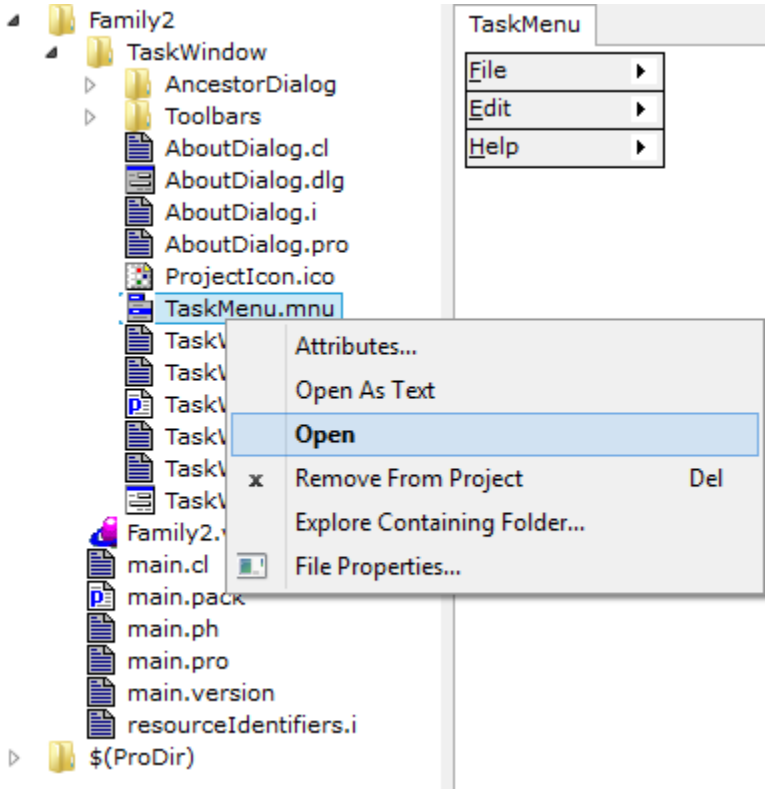
В итоге окно диалога будет выглядеть следующим образом:



Используя то же окно свойств, можно поменять название формы на *"Ancestor of ..."*.

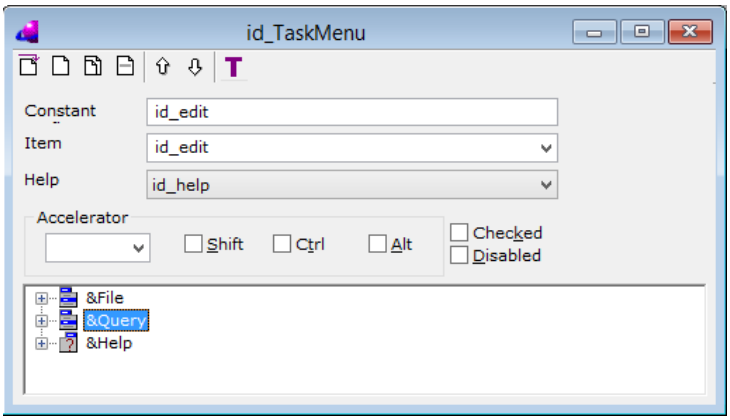
Включение пункта меню

Среда IDE создает главное меню приложения по умолчанию со стандартными разделами. Для изменения главного меню приложения необходимо, используя дерево проекта, щелкнуть правой кнопкой на элемент TaskMenu.mnu:

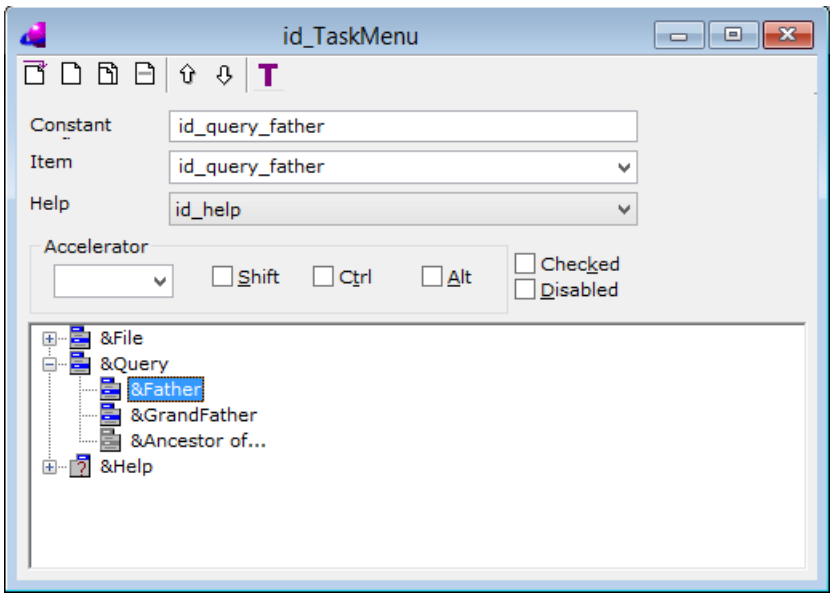


Активируется редактор меню Menu Editor. Щелкните на разделе Open, после чего откроется диалоговое окно TaskMenu. Щелкните правой кнопкой на разделе *&Edit* и замените название пункта меню *&Edit* на *&Query*. *Чтобы протестировать результат, нажмите кнопку T*.

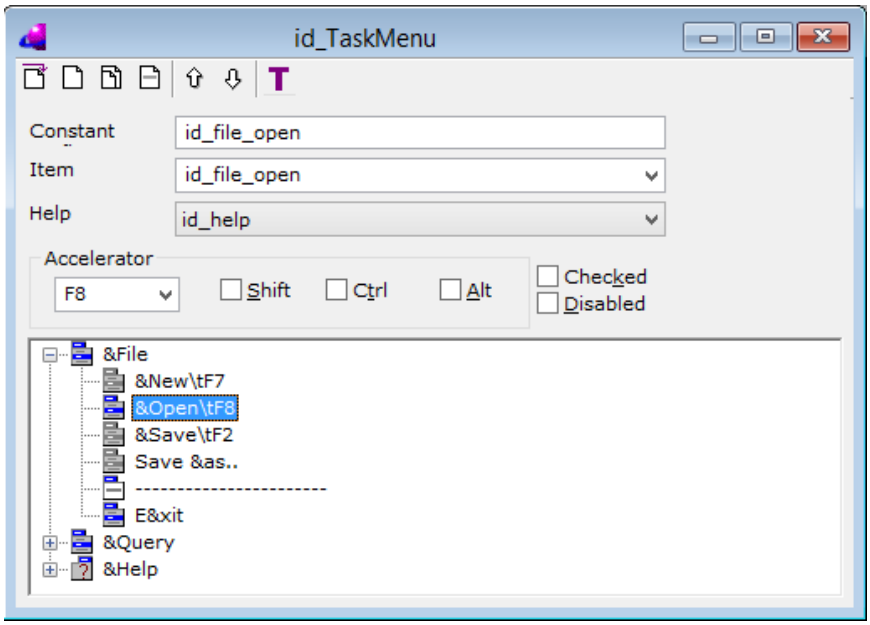
(Чтобы выйти из режима редактирования, щелкните в другом месте окна IDE.)



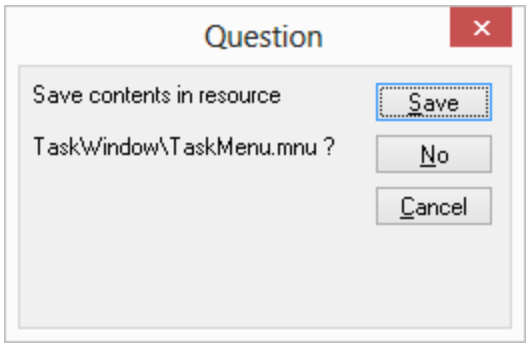
Если сделать двойной щелчок на &Query в редакторе, можно увидеть, что раздел по-прежнему содержит подпункты меню Edit (Undo, Redo, Cut, Copy and Paste). Нужно удалить все эти подпункты и ввести новые. Для ввода нового подпункта меню &Query щелкните на пиктограмме New **SubItem** и введите информацию в новый подпункт: &Father. Затем аналогично создайте второй подпункт &Grandfather и третий подпункт &Ancestor of ....



По умолчанию, при создании меню IDE пункты меню File|Open не активированы. Для их активации нужно снять флажки **Disabled** в Окне TaskMenu.



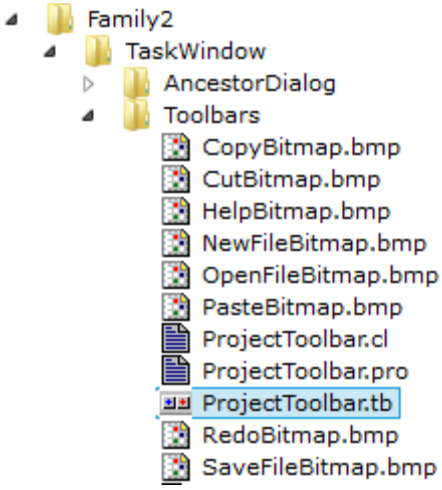
После закрытия диалога TaskMenu, IDE попросит подтвердить сохранение нового меню.



Модификация Панели инструментов.

Панель инструментов содержит кнопки, представляющие функционал различных пунктов меню. По умолчанию Visual Prolog IDE создает панель инструментов при создании проекта.

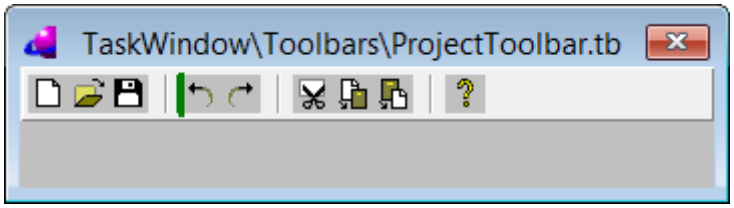
Активизируйте ProjectToolbar.tb двойным щелчком в дереве проекта.



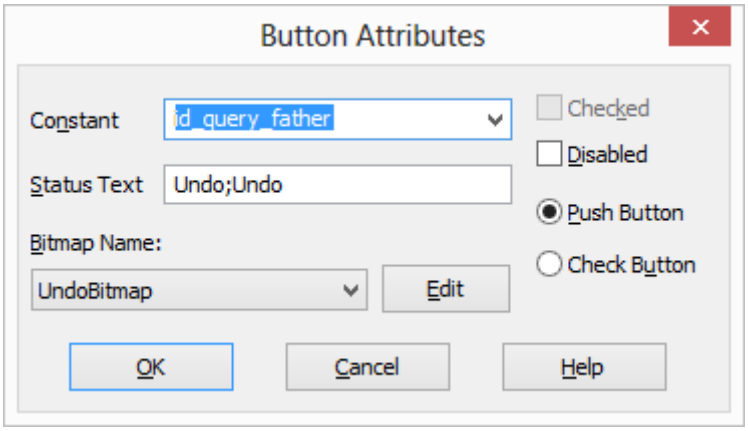
Это вызовет редактор Панели инструментов. В верхней части редактора представлена Панель, которую вы редактируете, а в нижней показаны доступные элементы управления, с помощью которых можно вносить изменения в компоненты Панели.

Пиктограммы привязаны к набору функций пунктов меню. Ввиду того, что пункты меню были ранее изменены, нужно отредактировать кнопки Панели и сделать их привязку.

Вначале нужно удалить кнопки, соответствующие функциям **cut, copy, paste** с помощью клавиши Delete**.** Затем привязать кнопки **Undo, Redo, Help** к пунктам меню : **Query|Father..., Query|Grandfather... and Query|Ancestor of ...** (без изменения их изображений).



Щелкните дважды на кнопку **Undo** и диалоге **Button Attributes** замените в поле *Constant id_edit_undo на id_query_father.*



Сделайте замену в том же окне в поле Status Text с:

Undo;Undo на **Query fathers;List of all fathers listed in the database**

Точка с запятой разделяет выражение на две части. Первая часть определяет текст всплывающей подсказки для кнопки в панели инструментов приложения, а вторая часть будет появляться в строке состояния Главного окна.

Аналогично, замените константу кнопки Redo, чтобы она имела значение ***id_query_grandfather.*** А кнопку Help замените на *id_query_ancestor_of*.

Создание главного кода программы

В отличие от консольного приложения, логика программного кода содержится не в одном модуле, а рассредоточена по нескольким модулям, в соответствии с используемыми компонентами GUI-интерфейса.

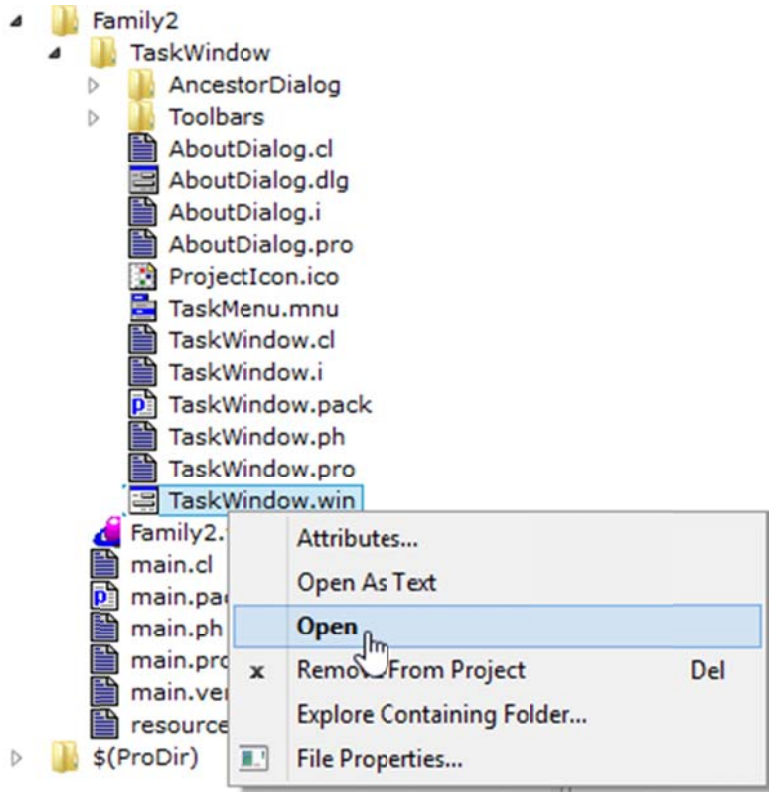
Откройте в дереве проекта **TaskWindow.pro** и вставьте в него код:

```
domains
    gender = female(); male().

class facts - familyDB
    person : (string Name, gender Gender).
    parent : (string Person, string Parent).

class predicates
    father : (string Person, string Father) nondeterm anyflow.
clauses
    father(Person, Father) :-
        parent(Person, Father),
        person(Father, male()).

class predicates
    grandFather : (string Person, string Grandfather) nondeterm anyflow.
clauses
    grandfather(Person, Grandfather) :-
        parent(Person, Parent),
        father(Parent, Grandfather).
```

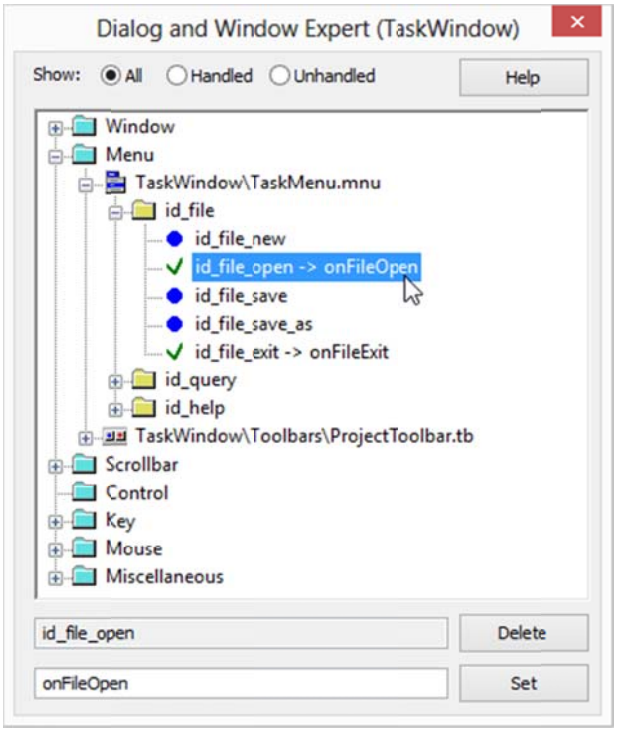
```
class predicates
    ancestor : (string Person, string Ancestor) nondeterm anyflow.
clauses
    ancestor(Person, Ancestor) :-
        parent(Person, Ancestor).
    ancestor(Person, Ancestor) :-
        parent(Person, P1),
        ancestor(P1, Ancestor).

class predicates
    reconsult : (string FileName).
clauses
    reconsult(Filename) :-
        retractFactDB(familyDB),
        file::consult(Filename, familyDB).
```

Приведенный код составляет логическое ядро программы, которое можно было бы разделить на разные модули. В первую очередь, это касается отделения логики программы от всех модулей GUI.

В данный код внесем элементы интерактивности. В дереве проекта щелкните правой кнопкой на **TaskWindow.win**. В открывшемся контекстном меню выберете пункт **Open**



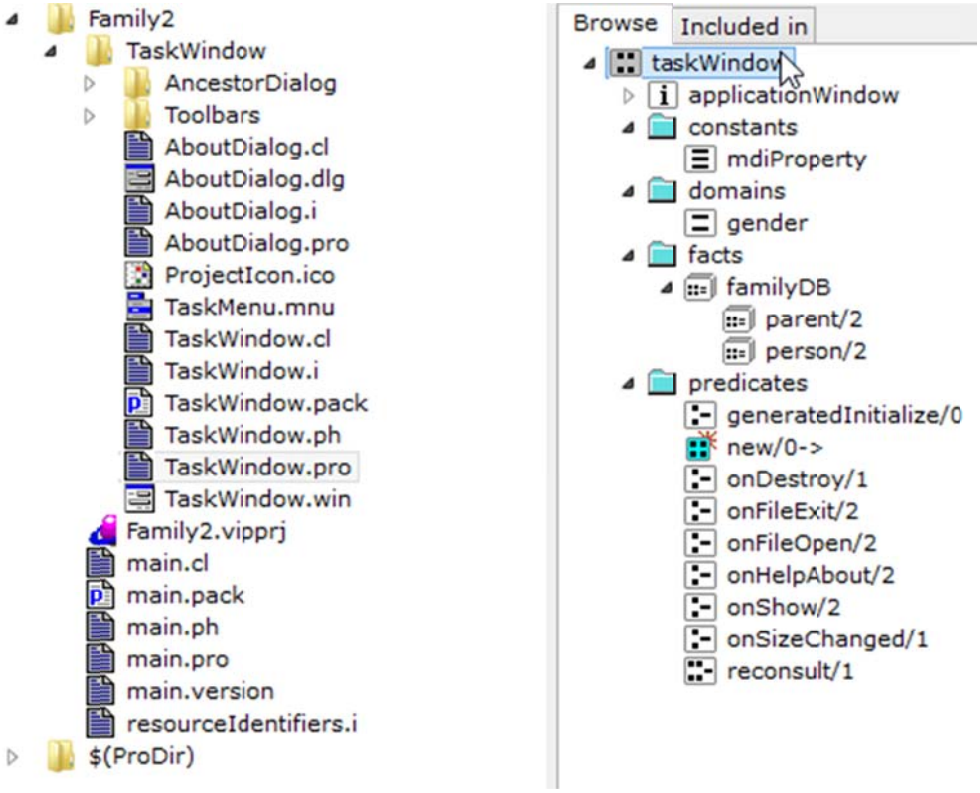
В Окне **Dialog and Window Expert** можно установить listeners и responders для всех элементов GUI-интерфейса : каждого окна и диалога. Голубая подкраска на элементе означает отсутствие listeners или responders. Зеленая галочка означает их наличие.



Так, в представленном диалоге показано, что установлен обработчик событий для пункта меню ***id_file_open***.

Теперь, в дереве проекта кликнете на модуле **TaskWindow.pro** , чтобы выбрать его, скомпилируйте его с помощью меню **Build.**

После компиляции в окне TaskWindow pro, в правой части предикатов можно увидеть все сущности, включая **onFileOpen.**



После двойного щелчка на предикате, откроется редактор точно в положении утверждения (**clause**) предиката **onFileOpen**. Предикат **onFileOpen** называется слушателем, приемником (*listener). Слушателей не надо вызывать в программном коде. Операционная система вызовет его автоматически, когда соответствующий компонент* GUI будет активирован (в данном случае выбран пункт меню).

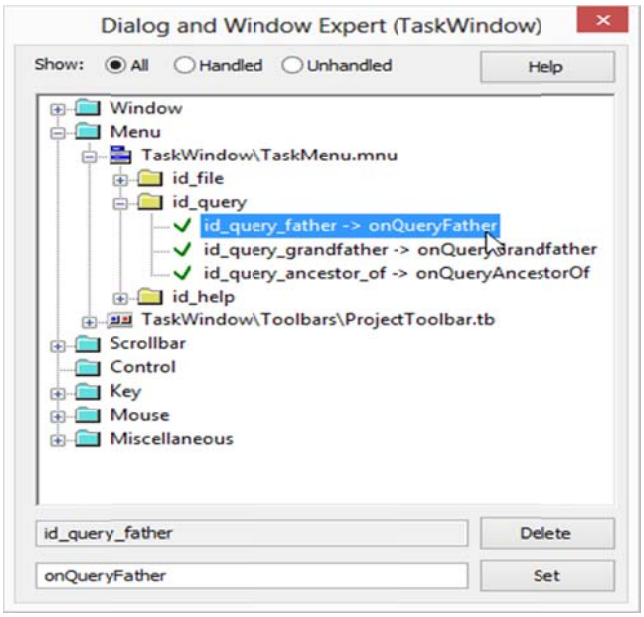
По умолчанию для этого обработчика событий будет вставлен следующий код:

```
onFileOpen(_Source, _MenuTag).
```

Этот код нужно заменить на другой код:

```
clauses
    onFileOpen(_Source, _MenuTag) :-
        Filename = vpiCommonDialogs::getFileName(
            "*.txt", ["Family data files (*.txt)","*.txt","All files", "*.*"],
            "Load family database",
             [], ".", _),
        ! ,
        reconsult(Filename),
        stdIO::writef("Database % loaded\n", Filename).
    onFileOpen(_Source, _MenuTag).
```

Вставленное в код тело утверждения **clauses** открывает стандартное диалоговое окно Windows, откуда загружается файл **с** данными. Кроме того, данное утверждение срабатывает как механизм защиты, в случае если пользователь нажмет на кнопку cancel в этом диалоге.
После компоновки программы и ее запуска можно загружать базу ***family*** в программу, используя соответствующий пункт меню **File|Open**. Можно использовать файл fa.txt, использованный ранее в консольном приложении. Если файл загружен правильно, вызывается предикат **stdIO::writef(...)** для вывода сообщения в окне Messages.
После этого можно вернуться в окно (TaskWindow) и убедиться, что слушатели для пунктов меню **Query|Father, Query|Grandfather** и **Query|Ancestor of ...** также установлены.



Теперь для пункта меню **Query|Father** добавьте следующее утверждение перед стандартным утверждением clauses, добавленным Visual Prolog автоматически:

```
predicates
   onQueryFather : window::menuItemListener.
clauses
   onQueryFather(_Source, _MenuTag):-
         stdIO::write("\nfather test\n"),
         father(X, Y),
         stdIO::writef("% is the father of %\n", Y, X),
         fail.
   onQueryFather(_Source, _MenuTag).
```

Для пункта меню **Query|Grandfather...** добавьте следующее утверждение перед стандартным утверждением **clauses**, сформированным Visual Prolog автоматически:

```
predicates
   onQueryGrandfather : window::menuItemListener.
clauses
onQueryGrandFather(_Source, _MenuTag) :-
         stdIO::write("\ngrandFather test\n"),
         grandfather(X, Y),
         stdIO::writef("% is the grandfather of %\n", Y, X),
         fail.
   onQueryGrandFather(_Source, _MenuTag).
```

Для пункта меню **Query|Ancestor of ...** добавьте следующее утверждение перед стандартным утверждением clauses, добавленным Visual Prolog автоматически:

```
predicates
   onQueryAncestorOf : window::menuItemListener.
clauses
   onQueryAncestorOf(_Source, _MenuTag) :-
    X = ancestorDialog::tryGetName(This),
        stdIO::writef("\nancestor of % test\n", X),
        ancestor(X, Y),
        stdIO::writef("% is the ancestor of %\n", Y, X),
        fail.
   onQueryAncestorOf(_Source, _MenuTag).
```

Цель этого фрагмента кода в том, чтобы ввести строку из диалогового окна, которое представляет **ancestorDialog,** который был создан ранее. Этот предикат описывает, что глобальный предикат **tryGetName** доступен в модуле ***ancestorDialog,*** *который вернет имя персоны, для которого ведется поиск родственников.*

В обработчике события **onFileOpen** мы получили строку (имя файла), введенную в диалоговом окне пользователем. Само модальное диалоговое окно было вызвано предикатом **vpiCommonDialogs::getFileName(...).** Аналогично мы получаем строку из диалогового окна **ancestorDialog.** Отличие в том только, что **vpiCommonDialogs::getFileName(...)** представляет встроенное стандартное диалоговое окно Windows для выбора файлов. А для **ancestorDialog** нужен дополнительный код.

Если откомпилировать программу, получим ошибку:

error c229: Undeclared identifier 'ancestorDialog::tryGetName/1->'

Причина в том, что предикат **onQueryAncestorOf** ожидает глобальный предикат **tryGetName** из модуля **ancestorDialog.pro.**

Предикат определен в одном модуле, а вызывается другим. Следовательно, мы должны убедиться, что объявление содержится не только в разделах **.pro** программы. Одно такое место – файл объявления класса модуля (с расширением **.cl**). Откроем теперь **ancestorDialog.cl** и вставим следующий фрагмент кода. Это утверждение **tryGetName**, выполняемое внутри модуля, и что этот предикат может быть вызван также из других модулей.

```
predicates
    tryGetName : (window Parent) -> string Name determ.
```

В файле **ancestorDialog.pro** вставим фрагмент кода с информацией об этом модуле. Также как мы делали для **Taskwindow.pro**, информацию нужно вставить сразу после строки:

classInfo(className, classVersion).

Вставляемый фрагмент кода:

```
domains
    optionalString = none(); one(string Value).
class facts
    name : optionalString := none().
clauses
    tryGetName(Parent) = Name :-
        name := none(),
        _ = ancestorDialog::display(Parent),
        one(Name) = name.
```

Теперь нужно изменить код обработчика событий для кнопки **ОК**. Это необходимо, чтобы диалог мог занести значение, введенное пользователем, в факты класса ***name.*** Иначе соответствующий предикат найдет пустую строку.

Предопределенный код, формируемый Visual Prolog, выглядит так:

```
predicates
    onOkClick : button::clickResponder.
clauses
    onOkClick(_Source) = button::defaultAction.
```

Этот код должен быть изменен на следующий:

```
predicates
    onOkClick : button::clickResponder.
clauses
    onOkClick(_Source) = button::defaultAction :-
        Name = idc_ancestordialog_personname:getText(),
```

name := one(Name).

После этого программа окончательно завершена, и ее можно компилировать и запускать на выполнение.

PIE: Приложение Prolog Inference Engine

В состав пакета Visual Prolog включен механизм логического вывода (PIE, Prolog Inference Engine) – исходный код стандартного интерпретатора Пролог, написанный на Visual Prolog. Этот интерпретатор – важный инструмент для изучения того, как работает Пролог. Изменяя его, можно добавлять возможности метаязыка к своим приложениям, создавая свои собственные специализированные языки логического программирования, механизмы логического вывода, оболочки экспертных систем или программные интерфейсы.

Приложение Prolog Inference Engine (PIE) представляет собой «классический» интерпретатор языка Пролог. Его использование освобождает разработчика от необходимости использования классов, типов и др. Он входит в собрание примеров Visual Prolog Examples, которые поставляются вместе с системой Visual Prolog ().. Для запуска приложения необходимо открыть проект в среде разработки Visual Prolog. Для этого нужно выбрать команду меню **Project > Open**, нажать кнопку **Browse…** окна Visual Prolog Environment и открыть файл Visual Prolog Examples\pie\pie.vipprj. После этого следует выбрать команду меню **Build > Execute**. В дальнейшем можно сразу запускать Exe-файл приложения PIE.

Рассмотрим задачу о родственниках в нотации «классического» Пролога. Имеется набор фактов

```
father("Bill", "John").
father("Pam", "Bill").
 father("Jack", "Bill").
```

и правил:

```
grandFather(Person, GrandFather) :-
   father(Person, Father),
   father(Father, GrandFather).
```

которые позволят нам ответить на вопросы

Является ли John отцом *Jack ?*
Кто отец Pam?
Является ли John дедушкой Pam?
...

Такого рода вопросы называются целями. Они записываются в классическом Прологе следующим образом:

```
?- father("Jack ", "John").
?- father("Pam", X).
?- grandFather("Pam", "John").
```

А в совокупности факты, правила и цели называются предложениями (клаузами) Хорна.

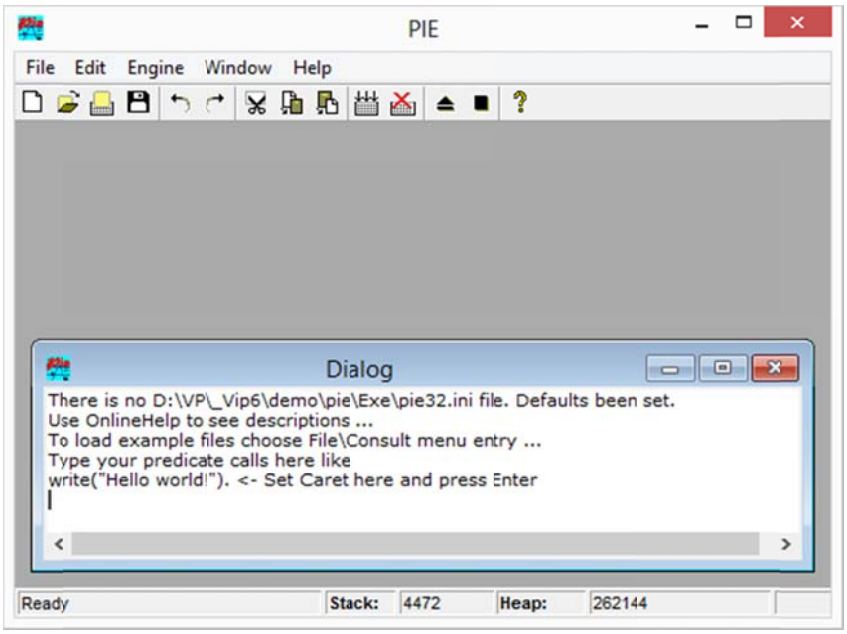
Первая и последняя цели выводят ответ в виде да или нет. Для второй цели ищется решение в виде **X = "Bill"**. Некоторые цели могут иметь много решений. Например цель

?- father(X, Y).

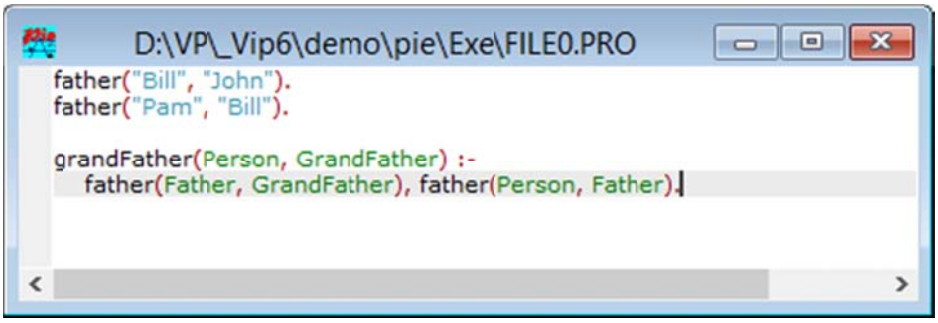
имеет два решения:

X = "Bill", Y = "John".
X = "Pam", Y = "Bill".

После запуска программы PIE на экране появится окно приложения :



Выберете **File -> New** и введите факты и правила о сущностях father и grandFather, как на рисунке. Пока окно редактирования открыто, выберите команду **Engine -> Reconsult**, которая загрузит файл в pie.



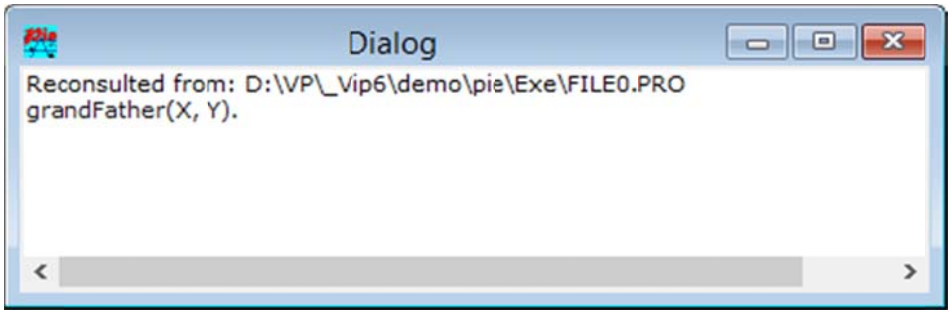
В окне ***Dialog*** вы увидите сообщение:

Reconsulted from: ....\pie\Exe\FILE0.PRO

Предикат **Reconsult** загрузит в *pie* все, что находится в редакторе, без сохранения содержимого в файл, поэтому, чтобы сохранить утверждения, используйте команду **File -> Save**.

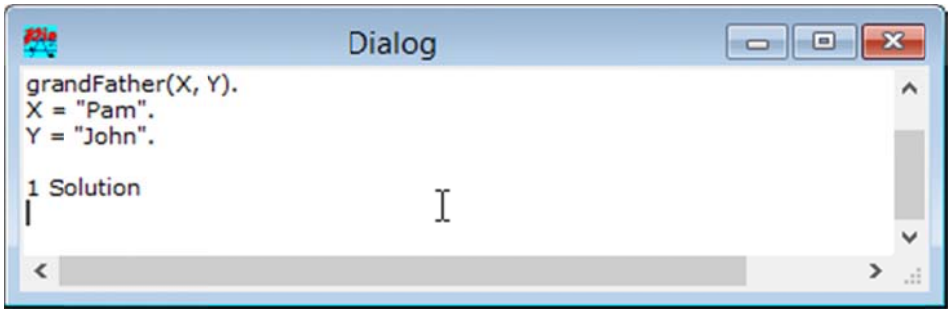
Команда **File -> Consult** будет загружать содержимое файла на диске, независимо от того, открыт файл для редактирования или нет.

После того, как правила построены, можно давать запросы с помощью целей. Для этого в пустой строке окна ***Dialog*** введите цель без знака **?-.** Например, как на рисунке:



Переместите курсор в конец строки, нажмите **Enter** на клавиатуре**.**

В качестве результата вы увидите:



Extending the family theory

Расширьте базу утверждений предикатами **mother и grandMother, sister, brother.**

Для этого введите еще предикаты **parent, fullBlodedSibling.** Например:

parent(Person, Parent) :- mother(Person, Parent).
parent(Person, Parent) :- father(Person, Parent).

Утверждения читаются как:

Parent is the parent of Person, if Parent is the mother of Person.

Утверждения можно записать и другим способом:

parent(Person, Parent) :-
mother(Person, Parent);
father(Person, Parent).

Утверждения тогда читаются как:

*Parent is the parent of Person, if Parent is the mother of Person **or** Parent is the father of Person*

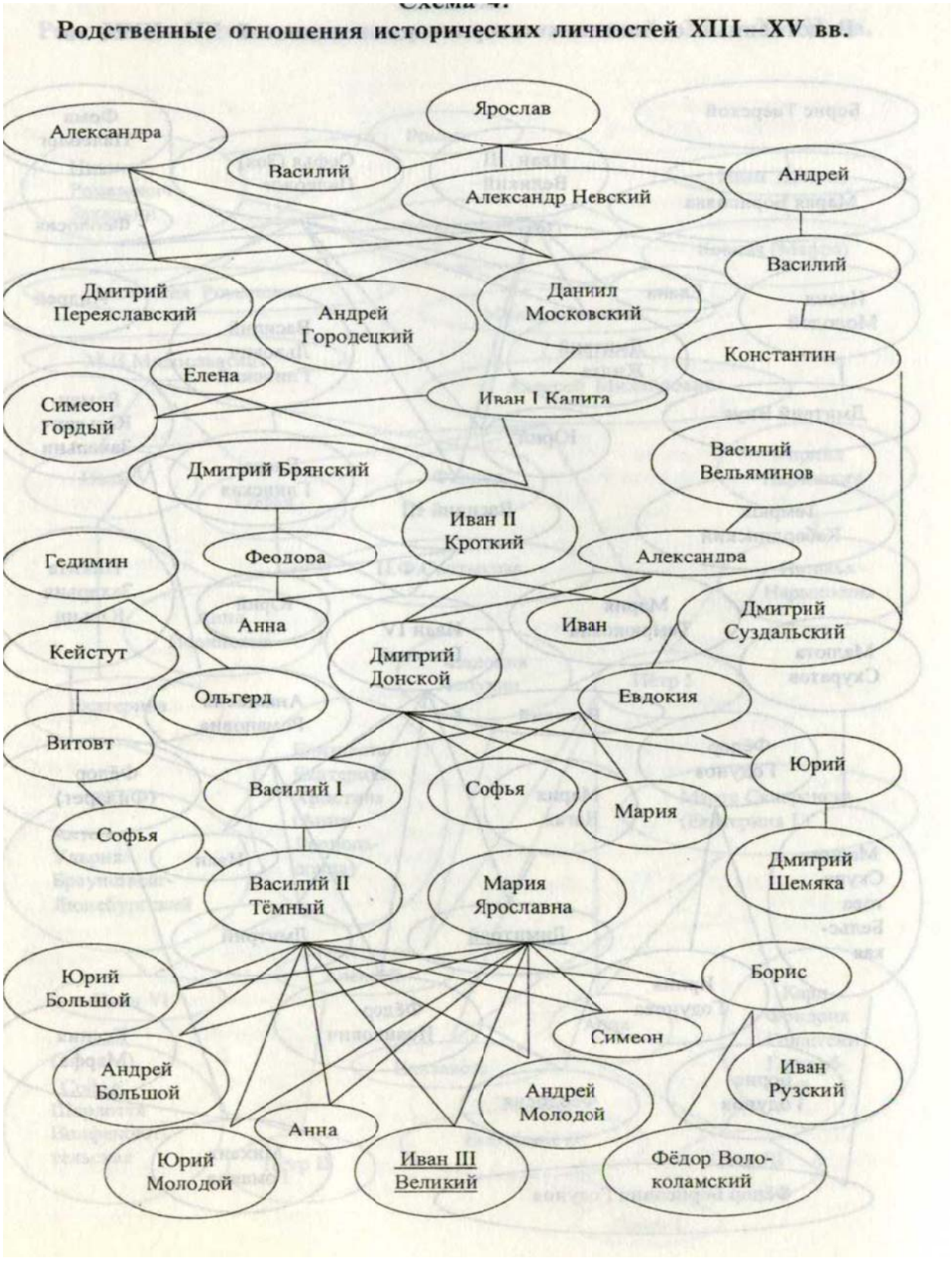
Предикат fullBlodedSibling (родные по крови) связывают отношениями персоны, если они имеют одних и тех же *mother u father.*

```
fullBlodedSibling(Person, Sibling) :-
   mother(Person, Mother),
   mother(Sibling, Mother),
   father(Person, Father),
   father(Sibling, Father).
```

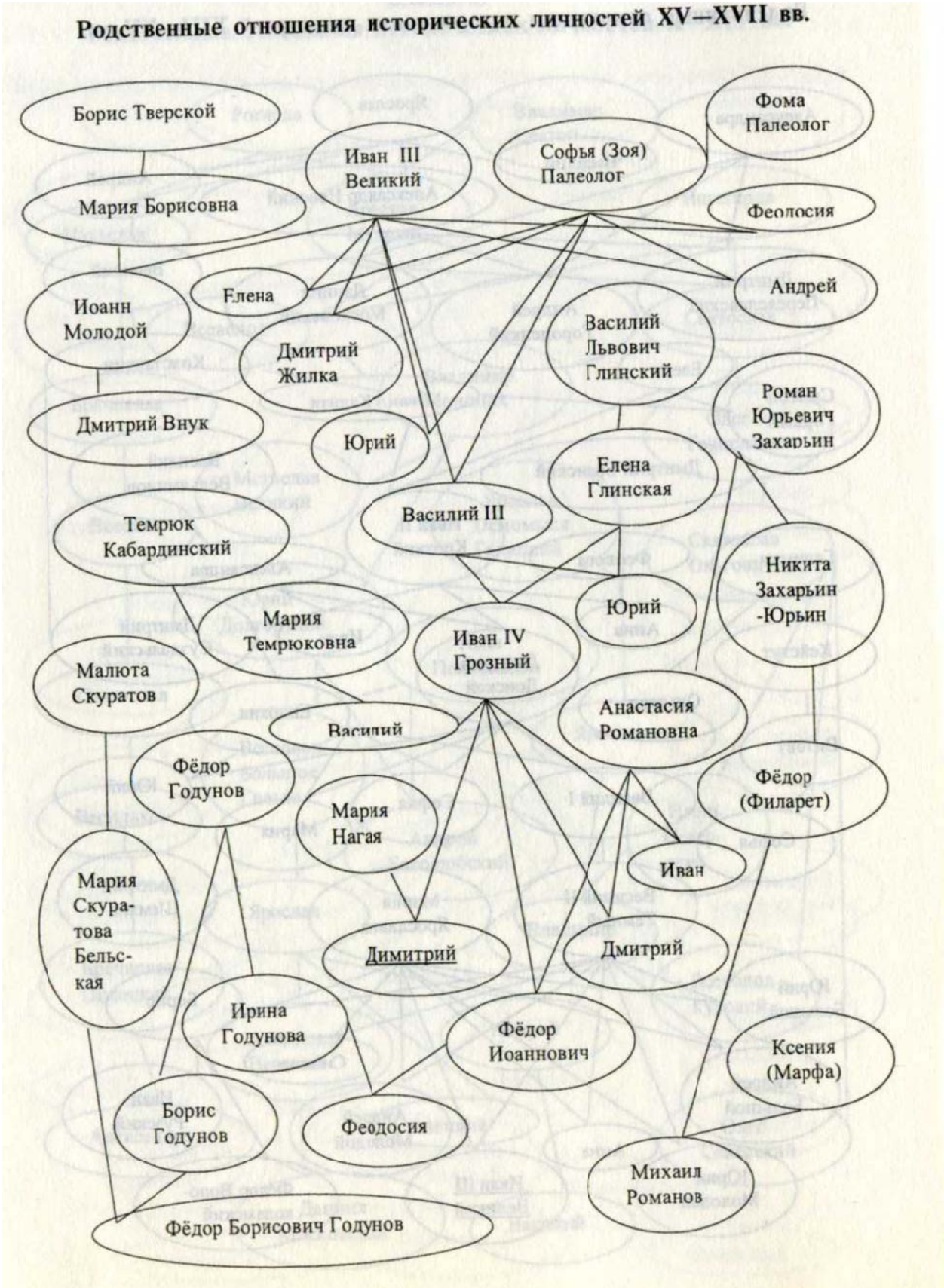
Библиографический список

1. Costa E. Visual Prolog 7.1 для начинающих, 2007 Пер. с англ., Алексеев, Ефимова, 2008 – 210с.
2. de Boer T. A Beginners' Guide to Visual Prolog, Version 7.2. 2009 – 281с.
3. Costa E. Visual Prolog 7.3 for Tyros. 2010 – 270с.
4. Visual Prolog 7.3 Language Reference. Prolog Development Center. 2010 – 88с.
5. http://www.visual-prolog.com

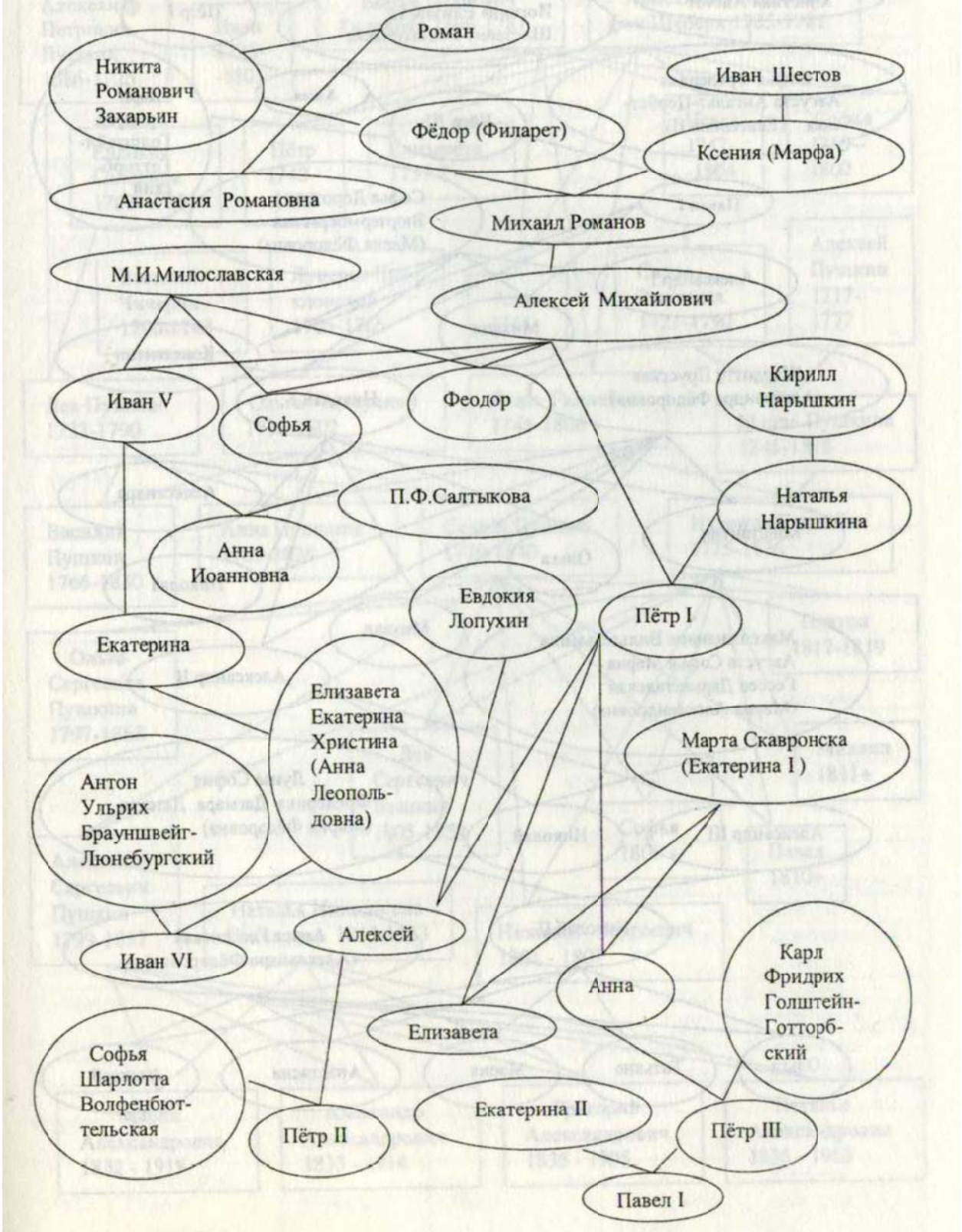
Приложение 1.



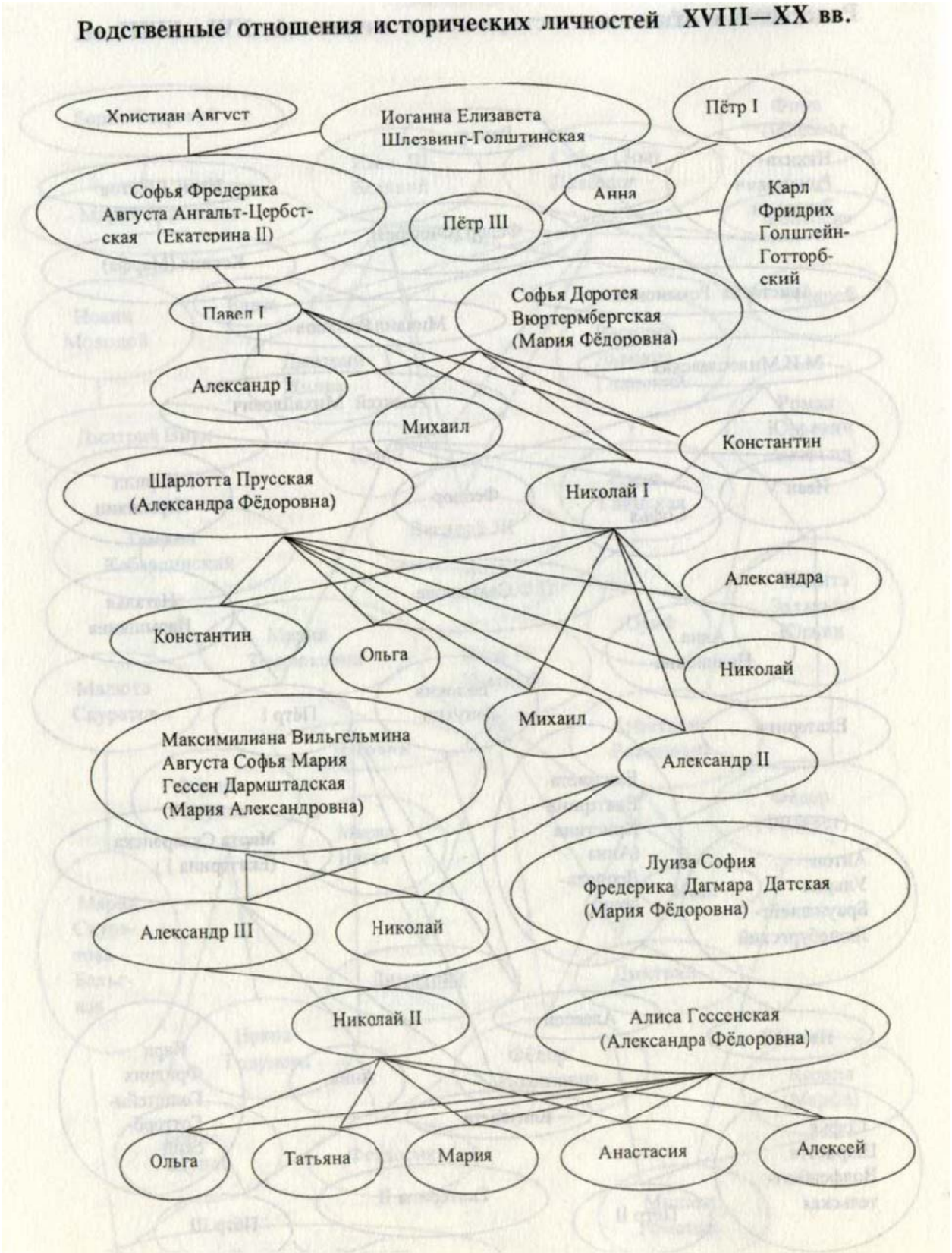
Родственные отношения исторических личностей XV—XVII вв.



Родственные отношения исторических личностей XVI—XVIII вв.



Родственные отношения исторических личностей XVIII—XX вв.



Варианты заданий

1. ЭС диагностики заболеваний
2. ЭС «Абитуриент»
3. ЭС «Авиа-диспетчер»
4. ЭС «Поиск неисправностей ПК»
5. ЭС «Выбор автомобиля»
6. ЭС «Подбор конфигурации ПК»
7. Онтология родственных отношений (Приложение)
8. Онтология учебного процесса