Поволжский государственный университет
телекоммуникаций и информатики

Акчурин Э.А.

Программирование на языке Visual Prolog в ИСР Visual Prolog

Учебное пособие для студентов направления
«Информатика и вычислительная техника»

Самара
2012

Факультет информационных систем и технологий

Кафедра «Информатика и вычислительная техника»

Автор - д.т.н., профессор Акчурин Э.А.

Другие материалы по дисциплине Вы найдете на сайте

www.ivt.psuti.ru

1 Введение

Учебное пособие может использоваться по дисциплинам:

- "Программирование на языке высокого уровня". Направление 210700.
- "Программирование на языках высокого уровня". Специальность 230105.
- «Программирование на языке Visual Prolog». Направление 200600.
- «Искусственный интеллект». Направление 200600.

Список литературы

1. Costa E. Visual Prolog 7.1 для начинающих, 2007 Перевод с английского, 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
6. Цуканова Н., Дмитриева Т. Логическое программирование на языке Visual Prolog. Уч. пособие для вузов. М.: Горячая линия: Телеком. 2008 – 144с.
7. Солдатова О., Лёзина И. Программирование на языке ПРОЛОГ: метод. указания / Самара: Изд-во Самарского государственного аэрокосмического университета, 2008 - 52 с.

1.1 История языка Visual Prolog

Язык программирования Visual Prolog - декларативный язык программирования общего назначения. Это усовершенствование языка Prolog (от “PROgramming in LOGic”) Prolog был создан в 1972 с целью сочетания использования логики с представлением знаний. С тех пор у него появился ряд диалектов, расширяющих основу языка различными возможностями.

Первыми исследователями в Прологе были Роберт Ковальски из Эдинбурга (теоретические основы), Маартен ван Эмден из Эдинбурга (экспериментальная демонстрационная система) и Ален Колмерое из Марселя (реализация).

Стандарт языка дан в ISO/IEC 13211-1 (1995 год).

Prolog — один из старейших языков логического программирования, хотя он значительно менее популярен, чем основные императивные языки. Он используется в системах обработки естественных языков, исследованиях искусственного интеллекта, экспертных системах, онтологиях и других предметных областях, для которых естественно использование логической парадигмы.

Prolog был создан под влиянием более раннего языка Planner и позаимствовал из него следующие идеи:

- обратный логический вывод (вызов процедур, исходя из целей);
- структура управляющей логики в виде вычислений с откатами;
- принцип “отрицание как неудача”;
- использование разных имен для разных сущностей и т.д.

Prolog использует один тип данных - терм, который бывает нескольких типов:

- **атом** — имя, используемое для построения составных термов;
- **числа и строки** такие же, как и в других языках;
- **переменная,** обозначается именем, начинающимся с **прописной** буквы, и используется как символ-заполнитель для любого другого терма;

- **составной терм**, состоит из атома-функтора, за которым следует несколько аргументов, каждый из которых в свою очередь является атомом.

Программы, написанные на Prolog, описывают **отношения** между обрабатываемыми сущностями при помощи **clauses** (предложений). Предложение — это формула вида Голова :- Тело.

Читается как: «чтобы доказать/решить Голову, следует доказать/решить Тело». Тело клаузы состоит из нескольких **предикатов** (целей клаузы), скомбинированных с помощью конъюнкции и дизъюнкции. Предложения с пустым телом называются фактами и эквивалентны предложениям вида

```
Голова :- true.
```

true — не атом, как в других языках, а встроенный предикат.

Другой важной частью Prolog являются **предикаты.**

- Унарные предикаты выражают свойства их аргументов.
- Предикаты с несколькими аргументами выражают отношения между ними.
- Ряд встроенных предикатов языка выполняют ту же роль, что и функции в других языках.

Чтобы быть языком общего назначения, Prolog должен предоставлять ряд сервисных функций, например, процедур ввода/вывода. Они реализованы как предикаты без специального логического смысла, которые всегда оцениваются как истинные и выполняют свои сервисные функции как побочный эффект оценивания.

Целью выполнения программы на Prolog является оценивание одного целевого предиката. Имея этот предикат и набор правил и фактов, заданных в программе, Prolog пытается найти привязки (значения) переменных, при которых целевой предикат принимает значение истинности.

Структура программы на Прологе отличается от структуры программы, написанной на процедурном языке. Пролог-программа является собранием правил и фактов. Решение задачи достигается интерпретацией этих правил и фактов. При этом пользователю не требуется обеспечивать детальную последовательность инструкций, чтобы указать, каким образом осуществляется управление ходом вычислений на пути к результату. Вместо этого он только определяет возможные решения задачи и обеспечивает программу фактами и правилами, которые позволяют ей отыскать требуемое решение.

Во всех других отношениях Пролог не отличается от традиционных языков программирования.

Пролог относится к так называемым декларативным языкам, требующим от автора умения составить формальное описание ситуации. Поэтому программа на Прологе не является таковой в традиционном понимании, так как не содержит управляющих конструкций типа if … then, while … do; нет даже оператора присваивания. В Прологе задействованы другие механизмы. Задача описывается в терминах фактов и правил, а поиск решения Пролог берет на себя посредством встроенного механизма логического вывода.

Перечень возможных синтаксических конструкций Пролога невелик, и в этом смысле язык прост для изучения. С другой стороны, декларативный стиль программирования оказывается столь непривычным и новым для опытных программистов.

Пролог реализован практически для всех известных операционных систем и платформ. В число операционных систем входят всё семейство Unix, Windows, OS для мобильных платформ.

Разработчик Visual Prolog – Prolog Development Center (Дания).

Глава команды разработчиков – Томас Линдер Пулс .

Учебные материалы также создают

- Томас дэ Боер из университета Гронинген (Голландия).
- Эдуардо де Коста из университета штата Юта (США).

1.2 Основы языка Visual Prolog

В основе Visual Prolog лежит традиционный Пролог.

Различия между традиционным Прологом и Visual Prolog можно разделить на эти категории

- Есть отдельные, но легко понимаемые различия между **структурой**, используемой в традиционном языке Prolog, и в Visual Prolog. Различаются форматы деклараций и определений, способ указания главной цели **goal** (что программа должна искать) с помощью особых ключевых слов.
- Файловые соображения: Visual Prolog дает удобства в организации структуры программы – проект содержит не один файл, а набор файлов различных типов.
- Область доступа: программа Visual Prolog может использовать функциональность, разработанную в других модулях, применяя концепцию под названием **область идентификации**.
- Объектная ориентация: программа на Visual Prolog может быть записана как объектно-ориентированная, используя классические объектно-ориентированные функции.

1.2.1 Объявления и определения

В Прологе, когда нам нужно использовать предикат, мы просто делаем это без каких-либо предварительных намеков движку о нашем намерении. Например, в более ранних учебниках предикат grandFather (дед) в clause (предложении) был прямо записан с использованием предиката головы и тела в стиле традиционного Пролога. Мы не должны сообщить движку явно в коде, какой предиката ожидается позднее.

Аналогично когда составной домен должен использоваться в традиционном Прологе, мы можем использовать его без предварительного уведомления движку о нашем намерении использовать домен. Мы просто используем домен в момент, когда есть потребность в нем.

Однако в Visual Prolog, **прежде чем записывать код** для тела положения предиката, необходимо сначала **объявить существование такого предиката** для компилятора. Аналогично перед использованием каких-либо доменов они должны быть объявлены, и их присутствие является информацией для компилятора.

Причина, по которой в Visual Prolog требуются такие уведомления, заключается в обеспечении защиты от исключений времени выполнения еще во время компиляции, насколько это возможно.

В термине «Исключения времени выполнения» мы имеем в виду вопросы, которые появляются только во время запуска программы, которая была скомпилированы с ними. Например, если вы намеревались использовать целое число в качестве аргумента функтора, и вместо целого числа вы ошибочно использовали вещественное число, то возникнет ошибка выполнения (в программах, написанных с использованием других компиляторов, но не Visual Prolog) и программа будет там неверной.

Когда вы также объявляете предикаты и домены, которые определены, такого рода позиционная грамматика (какой аргумент принадлежит к какому домену) и т.д. доступна для компилятора. Поэтому когда Visual Prolog выполняет компиляцию, он проверяет программу довольно тщательно, чтобы отсеять такие грамматические и другие ошибки.

Из-за этой особенности Visual Prolog повышает общую эффективность работы программиста. Программист не должен ждать, когда программа будет фактически выполняться, чтобы обнаружить ошибку. В самом деле, те из вас, кто имеют опыт в программировании, поймут, что опасение действительно может быть. Часто особая последовательность событий, которые вызывают исключения во время выполнения, могут быть так неуловимыми. Ошибка может фактически проявиться после нескольких лет, или она может возникнуть в некоторых критически важнейших или иных затруднительных ситуациях!

Все это означает, что компилятор должен получить четкие инструкции относительно предикатов и доменов, которые существуют в коде, используя соответствующие объявления, **прежде** чем они определены.

1.2.2 Ключевые слова

Программа Visual Prolog состоит из кода Prolog, который перемежается соответствующими ключевыми словами, которые информирует компилятор кода, что он должен генерировать. Например, есть ключевые слова, которые различают для предикатов и доменов объявления от определений. Как правило, каждый раздел предваряется ключевым словом. Существует правило - нет ключевого слова, которое означает окончание определенной секции. Наличие другого ключевого слова указывает на окончание в предыдущем разделе и начало следующего.

Исключением из этого правила являются ключевые слова **implement** (реализация) и **end implement** (конец реализации) Код, содержащийся между этими двумя ключевыми словами, используется для определенного класса. Те из вас, кто не понимает концепцию «class», может сейчас (то есть в этом руководстве) посмотреть модуль или секцию общего кода программы.

Ключевое слово **open** используется для расширения области видимости класса. Оно должна использоваться только после ключевого слова **implement** (реализация).

Ключевое слово **constants** используется для отметки раздел кода, который определяет некоторые часто используемые значения в коде программы. Например, если строка является литералом «PDC Prolog», который должен использоваться в нескольких местах по всему коду, то можно для него определить мнемонический образ (краткая форма, легко запоминаемое слово):

```
constants
        pdc = "PDC Prolog".
```

Обратите внимание, что определение константы оканчивается **точкой** (.). В отличие от переменной Пролога **constant** - слово, начинающееся с буквы **нижнего** регистра.

Ключевое слово **domains** используется для отметки секции объявления доменов, которые будут использоваться в коде. Есть много вариантов для синтаксиса объявлений таких доменов, и они удовлетворяют все возможные виды доменов, которые будут использоваться позже в коде.

Резюме, вы должны объявить функтор, который будет использовать домены и домены потомки, образующие ее аргументы.

Ключевое слово **facts** обозначает раздел, который объявляет факты, которые будут использоваться позже в коде программы. Каждый факт объявляется с именем, которое используется для обозначения факта, и аргументами вместе с доменами, которым эти аргументы принадлежат.

Ключевое слово **predicates** обозначает раздел, который содержит только объявления предикатов, которые будут позднее определены в разделе **clauses** (предложения) кода. Однажды объявленные имена, используемые для этих предикатов вместе с аргументами и доменами, к которым принадлежат аргументы, будут указываться в этом разделе.

Ключевое слово **clauses** обозначает раздел, который содержит определения ранее объявленных предикатов. Из всех разделов, которые присутствуют в коде Visual Prolog, этот раздел наиболее близко имитирует традиционную программу Prolog.

Ключевое слово **goal** обозначает раздел, который определяет главную точку входа в программу Visual Prolog.

В традиционном Прологе всякий раз, когда предикат определяется в коде, движку Пролога может быть поручено начать выполнение кода из этого преди-

ката с начала. Однако **это не так** в Visual Prolog. Будучи компилятором, он отвечает за производство эффективно исполняемого кода для **всей** программы, которую вы пишете.

Код не будет выполняться в то время, когда компилятор делает свою работу. Следовательно, компилятору необходимо заранее знать точный предикат, с которого начнется выполнение кода. Позже, когда программа вызывается для выполнения, она может сделать это с правильной отправной точки. Скомпилированная программа может работать самостоятельно без Visual Prolog компилятора или ИСР.

Для того, чтобы сделать это, есть специальный раздел, указываемый с ключевым словом **goal** (цель). Это специальный предикат, который надо писать без аргументов. Этот предикат является тем, с которого начнется выполнение всей программы.

1.2.3 Создание файлов

Если разместить все части программы в одном файле, он может стать громоздким, программу оказывается нечитаемой и иногда некорректной.

Visual Prolog имеет возможность разделения кода программы на отдельные файлы, используя средства IDE (интегрированной среды разработки). Можно написать аккуратно куски кода в отдельные файлы с помощью среды **ИСР**. Затем каждый файл отдельно компилируется в объектные файлы, которые затем компонуются линкером, создающий исполняемый в операционной системе файл.

1.2.4 Доступ к области

В Visual Prolog общий программный код делится на отдельные части, каждая часть определяет один класс. В объектно-ориентированных языках класс представляет собой пакет кода и связанных с ним данных. Как правило, Visual Prolog определяет каждый класс в собственном отдельном файле.

Во время выполнения программы часто случается, что программа может потребовать вызова предиката, который фактически определен в другом классе (файле). Аналогично данные (константы) или домены, определенные в классе, могут требоваться из другого файла. В Visual Prolog позволяет такие перекрестные ссылки данных кода класса, используя концепцию под названием **область доступа**.

Область видимости определения – это класс, реализованный в определенном файле. Предикаты, определяемые в нем, могут вызывать друг друга без квалификатора класса и маркеров двойного двоеточия (::), предшествующих ему.

Область видимости класса может быть расширена с помощью ключевого слова **open**. Это ключевое слово сообщает компилятору о необходимости принести имена (предикатов / констант / доменов), которые были определены в других файлах. Если область расширена, то не нужно писать квалификатор класса с двойным двоеточием.

2 Традиционный Пролог

2.1 Введение в логическое программирование

В Прологе мы получаем решение задачи логическим выводом из ранее известных положений. Обычно программа на Прологе не является последовательностью действий, она представляет собой набор фактов с правилами, обеспечивающими получение заключений на основе этих фактов. Поэтому Пролог известен как **декларативный язык**.

Пролог включает механизм вывода, который основан на сопоставлении образцов. С помощью подбора ответов на запросы он извлекает хранящуюся (известную) информацию. Пролог пытается проверить истинность гипотезы, запрашивая для этого информацию, о которой уже известно, что она истинна. В Прологе знание о мире — это ограниченный набор фактов (и правил), заданных в программе.

Одной из важнейших особенностей Пролога является то, что, в дополнение к логическому поиску ответов на поставленные вами вопросы, он может иметь дело с альтернативами и находить все возможные решения. Вместо обычной работы от начала программы до ее конца, Пролог может возвращаться назад и просматривать более одного "пути" при решении всех составляющих задачу частей.

2.2 Предикаты

Логика предикатов была разработана для наиболее простого преобразования принципов логического мышления в записываемую форму. Пролог использует преимущества синтаксиса логики для разработки программного языка. В логике предикатов вы, прежде всего, исключаете из своих предложений все несущественные слова. Затем вы преобразуете эти предложения, ставя в них на первое место отношение, а после него — сгруппированные объекты. В дальнейшем объекты становятся аргументами, между которыми устанавливается это отношение.

Отношение в Прологе называется **предикатом**. **Аргументы** — это объекты, которые связываются этим отношением, в факте

```
любит (саша, маша).
```

отношение **любит** — это предикат, а объекты **саша** и **маша** — аргументы.

Предикаты могут вовсе не иметь аргументов.

2.3 Факты

На Прологе описываются **объекты, отношения** и **правила**, при которых эти отношения являются истинными. Например, предложение

Саша любит собак

устанавливает отношение между объектами (Саша и собаки), этим отношением является **любит**. Ниже представлено правило, определяющее, когда предложение "Саша любит собак" является истинным:

Саша любит собак, если собаки хорошие.

В Прологе отношение между объектами называется **фактом** (fact). В естественном языке отношение устанавливается в предложении. В логике предикатов, используемой Прологом, отношение соответствует простой фразе (факту), состоящей из имени отношения и объекта или объектов, заключенных в круглые скобки. Как и предложение, **факт завершается точкой (.)**.

Ниже представлено несколько предложений на естественном языке с отношением "любит":

Саша любит Машу.
Маша любит Сашу.

А теперь перепишем эти же факты, используя синтаксис Пролога:

любит (саша, маша).
любит (маша, саша).

Допустимы русские слова. Вместо них можно использовать английские синонимы. Иногда с применением русскоязычных слов возникают проблемы. В частности исполняемый файл для Windows не может иметь русское имя.

2.4 Правила

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

Формат предложения:

<Заголовок> :- <список предикатов>.

Символ <:-> означает, что для определения истинности заголовка нужно выполнить последовательно его предикаты. Если они все истинны, то истинным будет и заголовок.

Предложение завершается **точкой**.

Программа Visual Prolog последовательно анализирует предложения раздела **clauses**, пытаясь найти логический путь к решению. По мере продвижения вниз по разделу clauses, он устанавливает внутренний указатель на первое предложение, являющееся частью пути, ведущего к решению. Если следующее предложение не является частью этого логического пути, то Visual Prolog возвращается к установленному указателю и ищет очередное подходящее сопоставление, перемещая указатель на него (этот процесс называется **поиск с возвратом***).*

Правила позволяют вам вывести один факт из других фактов. Другими словами, можно сказать, что **правило** — это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными. Ниже представлены правила, соответствующие связи "любить".

Маша любит то, что любит Саша.
Маша любит зеленое.

Чтобы перевести эти правила на Пролог, нужно изменить синтаксис:

```
любит(маша, нечто):- любит (саша, нечто).   % Да, если ее любит Саша
любит(маша, нечто):- зеленая (нечто).       % Да, если оно зеленое
```

Символ (:-) имеет смысл "если", и служит для разделения двух частей правила: заголовка и тела. Можно рассматривать правило и как процедуру. Другими словами представленные правила означают: "Чтобы доказать, что Маша любит нечто, докажите, что Саша любит это же" и "Чтобы доказать, что Маша любит нечто, докажите, что оно зеленое".

2.5 Запросы (Цели)

Описав в Прологе несколько фактов, можно задавать вопросы, касающиеся отношений между ними. Это называется **запросом (query)** системы языка Пролог. Можно задавать Прологу такие же вопросы, которые мы могли бы задать вам об этих отношениях. Основываясь на известных, заданных ранее фактах и правилах, вы можете ответить на вопросы об этих отношениях, в точности так же это может сделать Пролог. На естественном языке мы спрашиваем:

Саша любит Машу?

По правилам Пролога мы спрашиваем:

любит(саша, маша).

Получив такой запрос, Пролог ответит:

yes (да)

потому что Пролог имеет факт, подтверждающий, что это так. Немного усложнив вопрос, можно спросить на естественном языке:

(Что любит Саша?)

По правилам Пролога мы спрашиваем:

любит(саша, Что).

Необходимо отметить, что второй объект — **Что** - начинается с большой буквы, тогда как первый объект — **саша** — нет. Это происходит потому, что саша— фиксированный, постоянный объект — известная величина, а **Что** — переменная.

Переменные начинаются с **заглавной** буквы или символа подчеркивания (_)

Пролог всегда ищет ответ на запрос, начиная с первого факта, и перебирает все факты, пока они не закончатся.

3 Программы на Visual Prolog

Синтаксис Visual Prolog разработан для того, чтобы отображать знания о свойствах и взаимосвязях.

В отличие от других версий Пролога, Visual Prolog - это компилятор, контролирующий типы: для каждого предиката объявляются типы объектов, которые он может использовать. Это объявление типов позволяет программам Visual Prolog быть скомпилированными непосредственно в машинные коды, при этом, скорость выполнения сравнима, а в некоторых случаях — и превышает скорости аналогичных программ на языках C и Pascal.

3.1 Основные разделы Visual Prolog

Предметная область в Visual Prolog включает:

- **Множество объектов**. Конкретный объект описывается константой. Это число или символическое имя, называется атом (**atom**). Представляет собой неразрывную цепочку букв, цифр и символа подчеркивания, начинается со **строчной** буквы. В Visual Prolog разрешены латинские и русские буквы. Допускаются имена в виде строк, заключенных в двойные кавычки. Для описания объектов, принадлежащих классу, используется пе-

ременная. Имя переменной - неразрывная цепочка букв, цифр, начинается с **прописной** латинской буквы.

- **Отношения** между объектами. Объекты могут объединяться в классы.
- **Утверждения** программы, делятся на факты и вопросы.

Программа Visual Prolog может содержать разделы:

- Домены **domains**. Содержат описания типов данных. Если используются только стандартные типы, то этого раздела нет.
- Константы **constants**. Имеют символические имена, которым соответствуют значения. При компиляции имя заменяется значением.
- Факты **facts**. Содержит предикаты внутренней базы данных, которые истинны.
- Предикаты **predicates**. Сюда помещают предикаты, отсутствующие в разделе фактов. Если их нет, то раздел не нужен. Вам не нужно объявлять предикаты, встроенные в Visual Prolog.
- Предложения **clauses**. Сюда записываются факты и правила, которыми будет оперировать Visual Prolog, пытаясь разрешить цель программы.
- Цель – **goal. Раздел обязателен.** Здесь формулируется цель Visual Prolog программы: запуск исполняемого файла.

3.1.1 Раздел доменов

Вам могут потребоваться типы данных (доменов), отличающиеся от стандартных для Visual Prolog. В этом случае вам необходимо определить новые типы данных, которые вы поместите в ваш код.

Форма задания домена:

```
domains
    <имя1>, <имя2> = <имя известного домена>
```

Каждое строка задания доменов завершается символом **новой строки**.

Полезно описать новый домен, когда вы хотите прояснить отдельные части раздела predicates. Объявление собственных доменов, благодаря присваиванию осмысленных имен типам аргументов, помогает документировать описываемые вами предикаты. Рассмотрим пример, показывающий, как объявление доменов помогает документировать предикаты:

Петр — мужчина, которому 35 лет.

Используя стандартные домены, вы можете так объявить соответствующий предикат:

```
person(symbol, symbol, integer).
```

Первый аргумент – имя, второй – пол, третий – возраст. В большинстве случаев такое объявление будет работать хорошо, но не наглядно для чтения программы. Более правильным было бы следующее описание:

```
domains
    name, sex = symbol
    age = integer
predicates
    person(name, sex, age).
```

В раздел domains добавлены именованные типы. В разделе predicates тот же предикат имеет другие аргументы.

Домены позволяют задавать разные имена различным видам данных, которые, в противном случае, будут выглядеть абсолютно одинаково. В программах Visual Prolog объекты в отношениях (аргументы предикатов) принадлежат доменам, причем это могут быть как стандартные, так и описанные пользователем специальные домены. Раздел **domains** служит двум полезным целям. Во-первых, можно задать доменам осмысленные имена, даже если внутренне эти домены аналогичны уже имеющимся стандартным. Во-вторых, объявление специальных доменов используется для описания структур данных, отсутствующих в стандартных доменах.

Иногда очень полезно описать новый домен — особенно, когда вы хотите прояснить отдельные части раздела predicates. Объявление собственных доменов, благодаря присваиванию осмысленных имен типам аргументов, помогает документировать описываемые вами предикаты. Рассмотрим пример, показывающий, как объявление доменов помогает документировать предикаты:

Саша — мужчина, которому 45 лет.

Используя стандартные домены, вы можете так объявить соответствующий предикат:

```
person(symbol, symbol, integer).
```

В большинстве случаев такое объявление будет работать хорошо, но оно не наглядно для чтения программы. Более правильным было бы следующее описание:

```
domains
    name, sex = symbol
    age = integer
predicates
    person(name, sex, age).
```

Одним из главных преимуществ объявления собственных доменов является то, что Visual Prolog может отслеживать ошибки типов, например, такие:

```
same_sex(X,Y):-
    person(X, Sex, _),
    person(Sex, Y, _).
```

Несмотря на то, что и name и sex описываются как symbol, они не эквивалентны друг другу. Это и позволяет Visual Prolog определить ошибку, если вы перепутаете их. Это полезно в тех случаях, когда ваши программы очень велики и сложны.

Аргументы с типами из специальных доменов не могут смешиваться между собой, даже если эти домены одинаковы.

Если переменная в предложении используется более чем в одном предикате, она должна быть одинаково объявлена в каждом из них.

3.1.2 Раздел констант

В своих программах на Visual Prolog вы можете объявлять и использовать символические (с именем) константы. Раздел для объявления констант обозначается ключевым словом **constants**, за которым следуют сами объявления, использующие следующий синтаксис:

```
constants
    <имя> = <Макроопределение>.
```

<имя>— имя символической константы (символы в нижнем регистре), а <макроопределение> — это то, что вы присваиваете этой константе.

Каждое <макроопределение> **завершается точкой.** В Прологе использовался символ новой строки.

Следовательно, на одной строке может быть только одно описание константы. Объявленные таким образом константы могут позже использоваться в программах.

Рассмотрим следующий фрагмент программы:

```
constants
    zero = 0.
    one = 1.
    two = 2.
    hundred = (10*(10-1)+10.
```

Перед компиляцией программы Visual Prolog заменит каждую константу на соответствующее значение.

На использование символических констант накладываются следующие ограничения:

- Описание константы не может ссылаться само на себя, это приведет к сообщению об ошибке "Recursion in constant definition" (Рекурсия в описании константы).
- В описаниях констант система не различает верхний и нижний регистры. Следовательно, при использовании в разделе программы clauses идентификатора типа constants, его первая буква должна быть **строчной** для того, чтобы избежать путаницы между константами и переменными.
- В программе может быть несколько разделов constants, однако объявление константы должно производиться перед ее использованием.
- Идентификаторы констант являются глобальными и могут объявляться только один раз. Множественное объявление одного и того же идентификатора приведи к сообщению об ошибке "Constant identifier can only be declared once" (Идентификатор константы может объявляться только один раз).

3.1.3 Раздел предикатов

Отношение между объектами в Visual Prolog называется предикатом. Аргументы — это объекты, которые связываются этим отношением. В Visual Prolog в классы встроено множество предикатов, которые объявлять не надо.

Предикатами являются функции, домены которых отображаются в множество с двумя значениями: {true - истина, false -ложь}. Существует несколько предикатов, известных любому, кто пробовал себя в программировании. Вот они:

X > Y есть true, если X больше, чем Y, иначе возвращается false;
X < Y есть true, если X меньше, чем Y, иначе false;
X = Y есть true, если X равен Y, иначе false.

Если в разделе **clauses** программы на Visual Prolog описан **собственный предикат**, то его необходимо **предварительно** объявить в разделе **predicates** (предикатов). В результате объявления предиката сообщается, к каким доменам (типам) принадлежат аргументы этого предиката.

Объявление предиката начинается с имени этого предиката, за которым идет круглая скобка, после чего следуют аргументы предиката с разделением запятыми. Для каждого аргументы указываются тип и имя:

имяПредиката (типАрг1 имяАрг1, типАрг2 имяАрг2,…, типАргN имяАргN).

После последнего аргумента — закрывающая скобка. Каждое задание предиката **завершается точкой.** В Прологе использовался символ новой строки.

Доменами (типами) аргументов предиката могут быть либо стандартные домены, либо домены, объявленные вами в разделе **domains**. Можно указывать осмысленные имена аргументов — это улучшает читаемость программы, и не сказывается на скорости ее исполнения, т. к. компилятор их игнорирует.

Имена предикатов. Имя предиката должно начинаться с буквы, за которой может располагаться последовательность букв, цифр и символов подчеркивания. **Буквы должны быть в нижнем регистре!** Имя предиката может иметь длину до 250 символов.

В именах предикатов запрещается использовать пробел, символ минус, звездочку и другие алфавитно-цифровые символы.

Аргументы предикатов. Аргументы предикатов должны принадлежать доменам, известным Visual Prolog. Эти домены могут быть либо стандартными, либо пользовательскими.

Примеры предикатов с различным числом аргументов:

```
class predicates
    pred : (integer, symbol).
    person : (last, first, gender).
    run().
    birthday : (firstName, lastName, date).
```

В примере показано, что предикаты могут вовсе не иметь аргументов.

Арность (размерность) предиката — это количество аргументов, которые он принимает. Вы можете иметь два предиката с одним и тем же именем, но отличающейся арностью. В разделах predicates и clauses версии предикатов с одним именем и разной арностью должны собираться вместе; за исключением этого ограничения, **различная арность всегда понимается как полное различие предикатов**.

3.1.4 Раздел предложений

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

Пытаясь разрешить цель, Visual Prolog, начиная с первого предложения раздела **clauses**, просматривает каждый факт и каждое правило, стремясь найти сопоставление.

По мере продвижения вниз по разделу **clauses**, он устанавливает внутренний указатель на первое предложение, являющееся частью пути, ведущего к ре-

шению. Если следующее предложение не является частью этого логического пути, то Visual Prolog возвращается к установленному указателю и ищет очередное подходящее сопоставление, перемещая указатель на него (этот процесс называется **поиск с возвратом**).

3.1.5 Раздел фактов

Программа на Visual Prolog представляет собой набор фактов и правил. Иногда в процессе работы программы бывает необходимо модифицировать (изменить, удалить или добавить) некоторые из фактов, с которыми она работает. В этом случае факты рассматриваются как динамическая или внутренняя **база данных**, которая при выполнении программы может изменяться. Для объявления фактов программы, рассматриваемых как части динамической (или изменяющейся) базы данных, Visual Prolog включает специальный раздел — **facts**.

Ключевое слово **facts** объявляет раздел фактов. Именно в этой секции вы объявляете факты, включаемые в динамическую базу данных. Отметим, что в ранних версиях Visual Prolog для объявления раздела фактов использовалось ключевое слово database, т. е. ключевое слово facts — синоним устаревшего ключевого слова database. В Visual Prolog есть несколько встроенных предикатов, облегчающих использование динамических фактов.

Каждое задание факта **завершается точкой.** В Прологе использовался символ новой строки.

В Visual Prolog описываются **объекты** (objects) и **отношения** (relations), а затем **правила** (rules), при которых эти отношения являются истинными. Например, предложение

Маша любит собак.

устанавливает отношение между объектами Маша и собаки; этим отношением является <любит>. Ниже представлено правило, определяющее, когда предложение "Маша любит собак" является истинным:

Маша любит собак, если собаки хорошие.

В Visual Prolog отношение между объектами называется фактом (**fact**). В естественном языке отношение устанавливается в предложении. В логике предикатов, используемой Visual Prolog, отношение соответствует простой фразе (факту). Форма задания факта:

```
class facts
    <имя отношения> : (список объектов).
```

Факт **завершается точкой** (.).

Ниже представлено предложение на естественном языке с отношением "любит":

Маша любит собак.

Перепишем факт, используя синтаксис Visual Prolog:

```
class facts
    любит : (маша, собак).
```

Факты, помимо отношений, могут выражать и свойства. Так, например, предложение естественного языка «Маша — девочка» на Visual Prolog, выражая те же свойства, выглядят следующим образом:

```
class facts
    девочка:  (маша).
```

3.1.6 Раздел цели

Раздел **goal** - цель программы. В Visual Prolog программа перед исполнением компилируется в **исполняемый файл main.exe**. Поэтому цель программы – запуск этого файла. В конце файла main.pro должны быть строки:

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

3.2 Синтаксис правил

Правила используются в случае, когда какой-либо факт зависит от истинности другого факта или группы фактов. В правиле есть две части: заголовок и тело. Ниже представлен обобщенный синтаксис правила в Visual Prolog:

```
    Заголовок:-
        <Подцель>,
        <Подцель>,
        ... ,
        <Подцель>.
```

Тело правила состоит из одной или более подцелей. Подцели разделяются **запятыми**, определяя их конъюнкцию (правило И, заголовок - истина, если все подцели - истина), 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, Visual Prolog 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 хотя бы 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 (если она есть), а затем вновь двинется вперед, но уже с другими значениями переменных. Этот процесс называется **поиск с возвратом.**

Как упоминалось выше, в качестве разделителя заголовка и тела правила Пролог использует знак (:-), который читается как "если" (if). Однако if Пролога отличается от if, написанного в других языках, где условие, содержащееся в операторе if, должно быть указано перед телом оператора, который может быть выполнен.

Пролог использует другую форму логики в таких правилах. Вывод об истинности заголовка правила Пролога делается, если (после того, как) тело этого правила истинно.

Учитывая вышесказанное, правило Пролога соответствует условной форме тогда/если (then/if).

3.3 Автоматическое преобразование типов

Совсем не обязательно, чтобы при сопоставлении двух Visual Prolog-переменных они принадлежали одному и тому же домену. Переменные могут быть связаны с константами из различных доменов. Такое (избирательное) смешение допускается, т. к. Visual Prolog автоматически выполняет преобразование типов (из одного домена в другой), но только в следующих случаях:

- Между строками (string) и идентификаторами (symbol).
- Между целыми, действительными и символами (char). При преобразовании символа в числовое значение этим значением является величина символа в коде ASCII.

Аргумент из домена my_dom, который объявлен следующим образом:

```
domains
    my_dom = <base domain> % <base domain> — стандартный домен
```

Он может свободно смешиваться с аргументами из этого основного домена и с аргументами всех совместимых с ним стандартных доменов. Если основной домен — string, то с ним совместимы аргументы из домена symbol; если же основной домен integer, то с ним совместимы домены real, char, word и др. Такое преобразование типов означает, например, что вы можете:

- вызвать предикат с аргументами типа string, задавая ему аргументы типа symbol, и наоборот;

- передавать предикату с аргументами типа real параметры типа integer;
- передавать предикату с аргументами типа char параметры типа integer;
- использовать в выражениях и сравнениях символы без необходимости получения их кодов в ASCII.

Существует набор правил, определяющих, к какому домену принадлежит результат смешивания разных доменов. Эти правила будут детально рассмотрены далее.

3.4 Директивы компилятора

Visual Prolog поддерживает несколько **директив**, которые можно добавлять в программу для сообщения **компилятору** специальных инструкций по обработке программы при ее компиляции. Кроме этого, вы можете устанавливать большинство директив компилятора с помощью команды меню среды визуальной разработки Visual Prolog **Options/Project/Compiler Options.**

Директива **include**

Для того чтобы избежать многократного набора повторяющихся процедур, вы можете использовать директиву include. Ниже приведен пример того, как это делается.

- Создаете файл (например, MYSTUFF.PRO), в котором объявляете свои наиболее или часто используемые предикаты (с помощью разделов domains и predicates) и даете их описание в разделе clauses.
- Пишете исходный код программы, которая будет использовать эти процедуры.
- В "допустимых областях" исходного текста программы размещаете строку: include "mystuff.pro".

*"***Допустимые области***"* — это любое место программы, в котором вы можете расположить декларацию разделов domains, facts, predicates, clauses или goal.

При компиляции исходных текстов программы Visual Prolog вставит содержание файла MYSTUFF.PRO прямо в окончательный текст файла для компиляции.

Директиву include можно использовать для включения в исходный текст (практически любого) часто используемого фрагмента. Кроме того, любой включаемый в программу файл может, в свою очередь, включать другой файл (однако каждый файл может быть включен в вашу программу только один раз).

3.5 Унификация и поиск с возвратом

3.5.1 Сопоставление и унификация

Пытаясь выполнить целевое утверждение Visual Prolog должен проверить каждое предложение в программе. Сопоставляя аргументы заголовка утверждения с аргументами каждого проверяемого предложения, Visual Prolog выполняет поиск от начала программы до ее конца. Обнаружив предложение, соответствующее целевому утверждению, Visual Prolog присваивает значения свободным переменным таким образом, что целевое утверждение и предложение становятся идентичными. Говорят, что целевое утверждение **унифицируется** с предложением. Такая операция сопоставления называется **унификацией**.

Когда Visual Prolog пытается выполнить целевое утверждение, он проверяет, действительно ли обращение может соответствовать факту или заголовку правила.

Когда Visual Prolog проверяет предложение, он пытается завершить сопоставление унификацией аргументов.

3.5.2 Поиск с возвратом

Часто при решении реальной задачи мы придерживаемся определенного пути для ее логического завершения. Если полученный результат не дает искомого ответа, мы должны выбрать другой путь.

Так, вам, возможно, приходилось играть в лабиринт. Один из верных способов найти конец лабиринта — это поворачивать **налево** на каждой развилке лабиринта до тех пор, пока вы не попадете в тупик. Тогда следует вернуться к последней **развилке** и попробовать свернуть **вправо**, после чего опять поворачивать **налево** на каждом встречающемся распутье. Путем методичного перебора всех возможных путей вы, в конце концов, найдете выход.

В Visual Prolog при поиске решения задачи используется именно такой метод проб и возвращений назад. Метод называется **поиск с возвратом**. Если, начиная поиск решения задачи (или целевого утверждения), Visual Prolog должен выбрать между альтернативными путями, то он ставит маркер у места ветвления (называемого **точкой отката**) и выбирает первую подцель, которую и станет проверять. Если данная подцель не выполнится, Visual Prolog вернется к точке отката и попробует проверить другую подцель.

3.5.3 Процесс повторения

Программисты на языках Pascal, Basic или C, которые начинают использовать Visual Prolog, часто испытывают разочарование, обнаружив, что язык не имеет

конструкций FOR, WHILE или REPEAT. В Прологе не существует прямого способа выражения повтора. Пролог обеспечивает только два вида повторения

- **откат**, с помощью которого осуществляется поиск многих решений в одном запросе,
- **рекурсия**, в которой процедура вызывает сама себя.

Однако этот недостаток не снижает мощи Пролога. Фактически, Visual Prolog распознает специальный случай рекурсии — хвостовую рекурсию — и компилирует ее в оптимизированную итерационную петлю. Это означает, что хотя программная логика и выражается рекурсивно, скомпилированный код так же эффективен, как если бы программа была написана на Pascal или Basic.

3.5.4 Поиск с возвратом для повторов

Когда выполняется процедура поиска с возвратом (откат), происходит поиск другого решения целевого утверждения. Это осуществляется путем возврата к последней из проверенных подцелей, имеющей альтернативное решение, использования следующей альтернативы этой подцели и новой попытки движения вперед. Очень часто для этого используется директива fail.

Пример.

```
predicates
    country(symbol).
    print_countries.
clauses
    country("England").
    country("France").
    country("Germany").
    country("Denmark").
    print_countries:-
        country(X),
        write(X),        % записать значение X
        nl,              % начать новую строку
        fail.
    print_countries.
```

3.5.5 Использование отката с петлями

Поиск с возвратом является хорошим способом определить все возможные решения целевого утверждения. Но даже если ваша задача не имеет множества решений, можно использовать поиск с возвратом для выполнения итераций Просто определите предикат с двумя предложениями

```
    repeat
    repeat - repeat
```

Этот прием демонстрирует создание структуры управления Пролога (см листинг), которая порождает бесконечное множество решений. Цель предиката **repeat** — допустить бесконечность поиска с возвратом (бесконечное количество откатов).

Пример.

```
/* Использование repeat для сохранения введенных символов и печати их до
тех пор, пока пользователь не нажмет Enter (Ввод)*/

predicates
    repeat.
    typewriter.
clauses
    repeat.
    repeat -repeat.
typewriter :-
    repeat,
    readchar(C),        % Читать символ, его значение присвоить C
    write(C),
    C = '\r',           % Символ возврат каретки (Enter)? или неуспех
```

Программа показывает, как работает repeat. Правило typewriter - описывает процесс приема символов с клавиатуры и отображения их на экране, пока пользователь не нажмет клавишу <Enter>. Правило typewriter работает следующим образом

1. Выполняет repeat (который ничего не делает, но ставит точку отката).
2. Присваивает переменной C значение символа.
3. Отображает значение переменной C.
4. Проверяет, соответствует ли C коду возврата каретки.
5. Если соответствует, то — завершение. Если нет — возвращается к точке отката и ищет альтернативы, так как ни write, ни readchar не являются альтернативами,

3.5.6 Управление поиском решений

Встроенный механизм поиска с возвратом в Прологе может привести к поиску ненужных решений, в результате чего теряется эффективность, например, когда желательно найти только одно решение. В других случаях может оказаться необходимым продолжать поиск дополнительных решений, даже если целевое утверждение уже согласовано.

Visual Prolog имеет 2 инструментальных средства, которые дают возможность управлять механизмом поиска с возвратом:

- предикат **fail**, используется для инициализации поиска с возвратом,
- **cut** (отсечение), обозначается (!), для запрета возможности возврата.

Использование предиката fail

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

Прерывание поиска с возвратом: отсечение

Visual Prolog предусматривает возможность отсечения, которое 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).

3.5.7 Детерминизм и отсечение

Если предикат не содержит отсечений, то это **недетерминированный** предикат (способный производить множественные решения при помощи поиска с возвратом). В предыдущих реализациях Пролога программисты должны были обращать особое внимание на недетерминированные предложения из-за сопутствующих им дополнительных требований к ресурсам памяти. Теперь Visual Prolog сам выполняет проверку на недетерминированные предложения, облегчая вашу работу.

В Прологе существует директива компилятора **check_determ**. Если вставить эту директиву в самое начало программы, то Visual Prolog будет выдавать предупреждение в случае обнаружения недетерминированных предложений в процессе компиляции.

Можно превратить недетерминированные предложения в детерминированные, вставляя отсечения в тело правил, определяющих данный предикат.

3.6 Объекты данных

3.6.1 Типы данных

Язык Visual Prolog является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение.

Visual Prolog имеет множество встроенных типов данных (доменов).

Тип	Ключ	Значения	Примечание
Целые числа	integer	–32768 … 32767	
	byte	0 …255	
	цord	0 … 65535	
	dword	0	
Действительные числа	real	1E-307 …1E+308	формат с ПТ
	short		16 бит со знаком
	ushort		16 бит без знаком
	long		32 бит со знаком
	ulong		32 бит без знаком
	unsigned		16 или 32 бита без знака
!6-ричные числа			Первые два символа 0x
Символы	char		в апострофах
Строки	string	Unicode-символ	в двойных кавычках
Символическое имя	symbol	набор символов	для имен

К сведениям, хранимым в типе, может относиться следующее:

- Место для хранения переменной типа.
- Максимальное и минимальное значения, которые могут быть представлены.
- Содержащиеся члены (методы, поля, события и т. д.).
- Базовый тип, которому он наследует.
- Расположение, в котором будет выделена память для переменных во время выполнения.
- Разрешенные виды операций.

Компилятор использует сведения о типе, чтобы убедиться, что все операции, выполняемые в коде, являются строго типизированными. Например, при объявлении переменной численного типа **integer**, компилятор позволяет исполь-

зовать переменную и операции вычитания. При попытке выполнить эти же операции с переменной строкового типа компилятор вызовет ошибку.

Программа Visual Prolog работает со строго типизированными данными. Это означает, что компилятор проверяет, принадлежат ли данные, подаваемые на вход предикату или процедуре, правильному типу. Например, арифметические операции (x + y, x*y, a–b, p/q) работают с целыми или действительными числами. Поэтому компилятор удостоверяется, что ваша программа передает этим операциям числа, а не что-нибудь другое. Если в вашем коде имеется логическая ошибка (например, вы пытаетесь поделить строку на число), то компилятор генерирует ошибку.

Целые числа - **integer**. Возможны представления:

- десятичные числа с символами 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Целые числа имеют диапазон от –2147483648 до 2147483647.
- 16-ричные числа (0x0000FA1B) с символами 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Первые два символа 0x.

Вещественные десятичные числа – **real**. Возможны представления:

- с фиксированной точкой – 3.45, 3.1416,
- с плавающей точкой - 2.18E-18, 1.6E-19 (E – разделитель, слева мантисса, справа порядок).

Строки **string**. Это последовательность **Unicode**-символов в двойных кавычках – например, "pencil".. Элементы типа **string** хранятся в таблице данных, и для их внутреннего представления используются адреса каждого появления строки в программе. Таким образом, если элемент типа **string** встречается в программе много раз, он будет занимать больше места, чем символ.

Символы **symbol.** Это тоже набор символов в двойных кавычках "Na", "Natrium". Элементы типа **symbol** хранятся в таблице идентификаторов, и для их внутреннего представления используются адреса в этой таблице. Таким образом, если элемент типа **symbol** встречается в программе много раз, он будет занимать меньше места в памяти, чем строка.

3.6.2 Простые объекты данных

Простой объект данных — это переменная или константа. Не путайте это значение слова "константа" с символьными константами, которые вы определяете в разделе constants программы. То, что мы здесь называем константой, это нечто, идентифицирующее объект, который нельзя изменять: символ (char), число (integer или real) или атом (symbol или string).

Константы включают числа, символы и атомы. Числа и символы были рассмотрены ранее.

Атомы имеют тип идентификатор (symbol) или строка (string). Отличие между ними — главным образом вопрос машинного представления и реализации, и, в основном, оно синтаксически не заметно. Когда атом передается в качестве аргумента при вызове предиката, то к какому домену принадлежит атом — symbol или string -определяется по тому, как описан этот аргумент в декларации предиката.

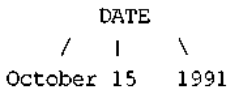
Visual Prolog автоматически преобразует типы между доменами string и symbol, поэтому вы можете использовать атомы symbol в доменах string и наоборот. Однако принято считать, что объект в двойных кавычках принадлежит домену string, а объект, не нуждающийся в кавычках, домену symbol. **Атомы** типа symbol — это имена, начинающиеся со строчной буквы и содержащие только буквы, цифры и знак подчеркивания.

Атомы типа string выделяются двойными кавычками и могут содержать любую комбинацию литер, кроме ASCII-нуля (0, бинарный нуль), который обозначает конец строки атома.

Так как string/symbol взаимозаменяемы, их отличие не существенно. Однако имена предикатов и функторы для составных объектов должны соответствовать синтаксическим соглашениям домена symbol.

3.6.3 Составные объекты данных и функторы

Составные объекты данных позволяют интерпретировать некоторые части информации как единое целое таким образом, чтобы затем можно было легко разделить их вновь. Возьмем, например, дату "октябрь 15, 1991". Она состоит из трех частей информации — месяц, день и год. Представим ее, как древовидную структуру.



Можно объявить домен, содержащий составной объект date:

```
domains
    date_cmp = date(string,unsigned,unsigned)
```

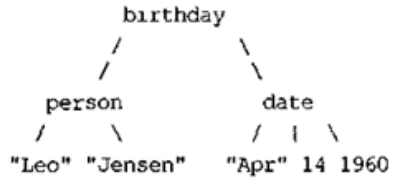
а затем просто записать:

```
    D = date("0ctober",15,1991) .
```

Такая запись внешне выглядит как факт Пролога, но **это не так** — это объект данных, который вы можете обрабатывать наряду с символами и числами. Он начинается с имени, называемого **функтором** (в данном случае date), за которым следуют три аргумента.

Функтор в Visual Prolog — не то же самое, что функция в других языках программирования; это просто имя, которое определяет вид составного объекта данных и объединяет вместе его аргументы. Функтор не обозначает, что будут выполнены какие-либо вычисления.

Аргументы составного объекта данных могут сами быть составными объектами. Например, вы можете рассматривать чей-нибудь день рождения, как информацию со следующей структурой:



На языке Пролог это выглядит следующим образом:

```
birthday(person("Leo","Jensen"),date("Apr",14,1960))
```

3.6.4 Унификация составных объектов

Составной объект может быть унифицирован с простой переменной или с составным объектом (возможно, содержащим переменные в качестве частей во внутренней структуре), который ему соответствует. Это означает, что составной объект можно использовать для того, чтобы передавать целый набор значений как единый объект, и затем применять унификацию для их разделения. Например:

```
date("April",14,2010)
```

сопоставляется с date(Mo, Da, Yr) и присваивает переменным Mo = "April", Da=14 и Yr = 2010.

3.6.5 Объявление составных доменов

Рассмотрим, как определяются составные домены. После компиляции программы, которая содержит следующие отношения:

```
owns(john, book("From Here to Eternity", "James Jones")).
```

и

owns (John, horse (blacky)).

вы можете послать системе запрос в следующем виде:

owns (John, X)

Переменная X может быть связана с различными типами объектов: книга, лошадь и, возможно, другими объектами, которые вы определите. Отметим, что теперь **вы не можете** более использовать старое определение предиката owns:

owns (symbol, symbol).

Второй элемент более не является объектом типа symbol. Вместо этого вы можете дать новое определение этого предиката

owns(name, articles).

Домен articles в разделе domains можно описать так

```
domains
    articles = book(title, author); horse(name)
```

Точка с запятой читается как "или" В этом случае возможны 2 варианта:

- книга будет определяться своим заглавием и автором,
- лошадь будет распознаваться своим именем.

Домены title, author и name имеют стандартный тип symbol.

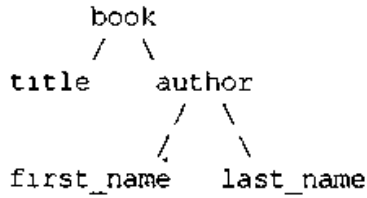
К определению домена легко могут быть добавлены другие варианты.

3.6.6 Многоуровневые составные объекты

Visual Prolog позволяет конструировать составные объекты на нескольких уровнях. Например:

```
domains
    articles = book(title, author)              %Первый уровень
    author= author(first_name, last_name)       %Второй уровень
    title, first_name, last_name = symbol       %Третий уровень
```

При использовании составных объектов со многими уровнями часто помогает такое "дерево":



Компьютеры способны повторять одно и то же действие снова и снова, Visual Prolog может выражать повторение, как в процедурах, так и в структурах данных. Идея повторяющихся структур данных может показаться странной, но Пролог позволяет создавать структуры данных, размер которых не известен во время создания.

3.7 Рекурсивные процедуры

3.7.1 Понятие рекурсии

Одним из способов организации повторений — рекурсия. **Рекурсивная процедура** — это процедура, которая вызывает сама себя. В рекурсивной процедуре нет проблемы запоминания результатов ее выполнения, потому что любые вычисленные значения можно передавать из одного вызова в другой как аргументы рекурсивно вызываемого предиката.

Логика рекурсии проста для осуществления. Представьте себе ЭВМ, способную найти факториал числа N по правилам:

- Если N равно 1, то факториал равен 1.
- Иначе найти факториал N-1 и умножить его на N.

Этот подход означает следующее:

- Первое («закручиваете» стек), чтобы найти факториал 3, вы должны поместить 3 в стек и найти факториал 2.
- Чтобы найти факториал 2, вы должны поместить 2 в стек и найти факториал 1.
- Факториал 1 ищется без обращения к другим факториалам, т.к. он равен 1, поэтому повторения не начнутся.
- Второе («раскручиваете» стек), если у вас есть факториал 1, то умножаете его на 2, чтобы получить факториал 2, а затем умножаете полученное на 3, чтобы получить факториал 3.

Информация хранится в области памяти, называемой стековым фреймом или просто стеком (**stack**), который создается каждый раз при вызове правила.

Когда выполнение правила завершается, занятая его стековым фреймом память освобождается (если это не недетерминированный откат), и выполнение продолжается в стековом фрейме правила-родителя.

3.7.2 Преимущества рекурсии

Рекурсия имеет 3 основных преимущества:

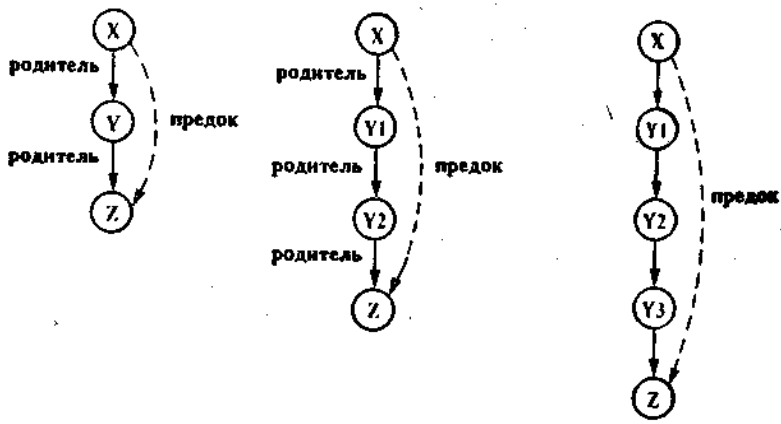
- она может выражать алгоритмы, которые нельзя удобно выразить никаким другим образом;
- она логически проще метода итерации;
- она широко используется в обработке списков.

Рекурсия — хороший способ для описания задач, содержащих в себе подзадачу такого же типа. Например, поиск в дереве (дерево состоит из более мелких деревьев) и рекурсивная сортировка (для сортировки списка, он разделяется на части, часть сортируются и затем объединяются вместе).

Логически рекурсивным алгоритмам присуща структура индуктивного математического доказательства. Приведенная выше рекурсивная программа вычисления факториала описывает бесконечное множество различных вычислений с помощью всего лишь двух предложений. Это позволяет легко увидеть правильность этих предложений. Кроме того, правильность каждого предложения может быть изучена независимо от другого.

3.7.3 Пример рекурсивного определения правил

Давайте добавим к программе о родственных связях еще одно отношение - предок. Определим его через отношение родитель. Все отношение можно выразить с помощью двух правил. Первое правило будет определять непосредственных (ближайших) предков, а второе - отдаленных. Будем говорить, что X является отдаленным предком Z, если между X и Z существует цепочка людей, связанных между собой отношением родитель-ребенок.



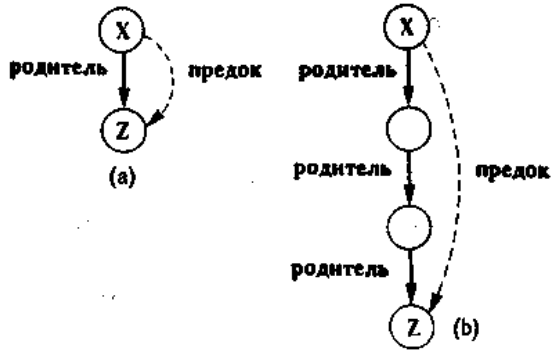
Первое правило простое и его можно сформулировать так:

Для всех X и Z,
X - предок Z, если X - родитель Z.

Это непосредственно переводится на Пролог как

предок(X, Z) :- родитель(X, Z).

Второе правило сложнее, поскольку построение цепочки отношений родитель может вызвать некоторые трудности. Один из способов определения отдаленных родственников мог бы быть таким, как показано на рисунке Пример отношения **предок:**(a) X - **ближайший** предок Z; (b) X - **отдаленный** предок Z.

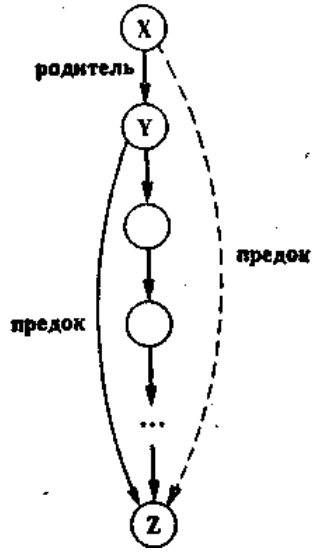


В соответствии с ним отношение предок определялось бы следующим множеством предложений:

```
предок( X, Z) :-
    родитель( X, Z).
предок( X, Z) :-
    родитель( X, Y), родитель( Y, Z).
предок( X, Z) :-
    родитель( X, Y1),
    родитель( Y1, Y2),
    родитель( Y2, Z).
предок (X, Z) :-
    родитель( X, Y1),
    родитель( Y1, Y2),
    родитель( Y2, Y3),
    родитель( Y3, Z).
```

Эта программа длинна и, что более важно, работает только в определенных пределах. Она будет обнаруживать предков лишь до определенной глубины фамильного дерева, поскольку длина цепочки людей между предком и потомком ограничена длиной наших предложений в определении отношения.

Существует, однако, корректная и элегантная формулировка отношения предок - корректная в том смысле, что будет работать для предков произвольной отдаленности. Ключевая идея здесь - определить отношение предок через него самого. Рисунок иллюстрирует эту идею:



Для всех X и Z, X - предок Z, если существует Y, такой, что

- X - родитель Y и
- Y - предок Z.

Предложение Пролога, имеющее тот же смысл, записывается так:

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

Теперь мы построили полную программу для отношения предок, содержащую два правила: одно для ближайших предков и другое для отдаленных предков. Здесь приводятся они оба вместе:

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

Ключевым моментом в данной формулировке было использование самого отношения предок в его определении. Такие определения называются **рекурсивными**. Логически они совершенно корректны и понятны. Рекурсия - один из фундаментальных приемов программирования на Прологе. Без рекурсии с его помощью невозможно решать задачи сколько-нибудь ощутимой сложности.

3.8 Списки

3.8.1 Определение списка

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]
```

Каждая составляющая списка называется **элементом**. Чтобы оформить списочную структуру данных, надо отделить элементы списка запятыми и заключить их в квадратные скобки. Вот примеры:

```
[собака, кошка, птица]
["саша", "маша", "даша"]
```

3.8.2 Объявление списков

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. Однако все его элементы должны принадлежать одному домену. Декларация домена для элементов должна быть следующего вида:

```
domains
    elementlist = elements*
    elements = ....
```

Здесь elements имеют единый тип (например: integer, real или symbol) или являются набором отличных друг от друга элементов, отмеченных разными функторами. В Visual Prolog нельзя смешивать стандартные типы в списке. Например, следующая декларация неправильно определяет список, составленный из элементов, являющихся целыми и действительными числами или идентификаторами:

```
    elementlist = elements*
    elements = integer; real; symbol          % Неверно
```

Чтобы объявить список, составленный из целых, действительных и идентификаторов, надо определить один тип, включающий все три типа с функторами, которые покажут, к какому типу относится тот или иной элемент. Например:

```
    elementlist = elements*
    elements = i(integer); r(real); s(symbol)      % здесь i, r и s - функторы
```

3.8.3 Головы и хвосты

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:

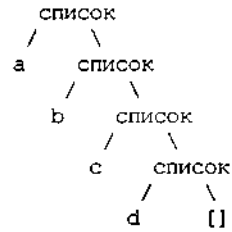
```
    голова [a, b, c] есть a
    хвост [a, b, c] есть [b, c]
```

Что происходит, когда вы доходите до одноэлементного списка? Ответ таков:

голова [c] есть c
хвост [c] есть []

Если выбирать первый элемент списка достаточное количество раз, вы обязательно дойдете до пустого списка []. Пустой список нельзя разделить на голову и хвост.

В концептуальном плане это значит, что список имеет структуру дерева, как и другие составные объекты. Структура дерева [a, b, c, d].



Одноэлементный список, как, например [a], не то же самое, что элемент, который в него входит, потому что [a] на самом деле составная структура данных.

3.8.4 Работа со списками

В Прологе 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| []]]].

Можно использовать оба вида разделителей в одном и том же списке при условии, что вертикальная черта есть последний разделитель. При желании можно набрать

[a, b, c, d] как [a, b|[c, d]].

В таблице. приведены несколько примеров на присвоение в списках.

Список 1	Список 2	Присвоение переменным
[X, Y, Z]	[маша, ест, мороженое]	X=маша, У=ест, Z=мороженое
[7]	[X | Y]	X=7, Y=[]
[1, 2, 3, 4]	[X, Y | Z]	X=l, Y=2, Z=[3,4]
[1, 2]	[3 | X]	fail% неудача

3.8.5 Использование списков

Список является рекурсивной составной структурой данных, поэтому нужны алгоритмы для его обработки. Главный способ обработки списка — это просмотр и обработка каждого его элемента, пока не будет достигнут конец.

Алгоритму этого типа обычно нужны два предложения. Первое из них говорит, что делать с обычным списком (списком, который можно разделить на голову и хвост), второе — что делать с пустым списком.

Вывод списков

Если нужно вывести элементы списка, это делается так, как показано в листинге.

```
domains
    list = integer*          % Или любой тип, какой вы хотите
predicates
    write_a_list(list).
clauses
    write_a_list([ ]),       % Если список пустой — ничего не делать
    write_a_list([H|T]):-    % Присвоить H – голова списка ,T - хвост, затем...
        write(H),nl,
        write_a_list(T).
```

Вот два целевых утверждения write_a_list, описанные на обычном языке:

- выводить пустой список — ничего не делать.
- Иначе, выводить список — вывести его голову (которая является одним элементом), затем его хвост (список) .

Подсчет элементов списка

Рассмотрим, как можно определить число элементов в списке. Что такое длина списка? Вот простое логическое определение:

Длина [] — 0.
Длина любого другого списка = 1 плюс длина его хвоста.

Можно ли применить это? В Прологе — да. Для этого нужны два предложения

Листинг

```
domains
    list = integer*
predicates
    length_of(list,integer).
```

```
clauses
    length_of ( [ ] , 0).
    length_of ( [ _|T],L) :-
        length_of(T,TailLength),
        L = TailLength + 1.
```

Посмотрим сначала на второе предложение. Действительно, [_|T] можно сопоставить любому непустому списку, с присвоением L хвоста списка. Значение головы не важно, главное, что оно есть, и компьютер может посчитать его за один элемент.

Таким образом, целевое утверждение

```
length_of([1, 2, 3], L).
```

подходит второму предложению при T=[2, 3]. Следующим шагом будет подсчет длины T. Когда это будет сделано (не важно как), TailLength будет иметь значение 2, и компьютер добавит к нему 1 и затем присвоит L значение 3.

Итак, как компьютер выполнит промежуточный шаг? Это шаг, в котором определяется длина [2, 3] при выполнении целевого утверждения

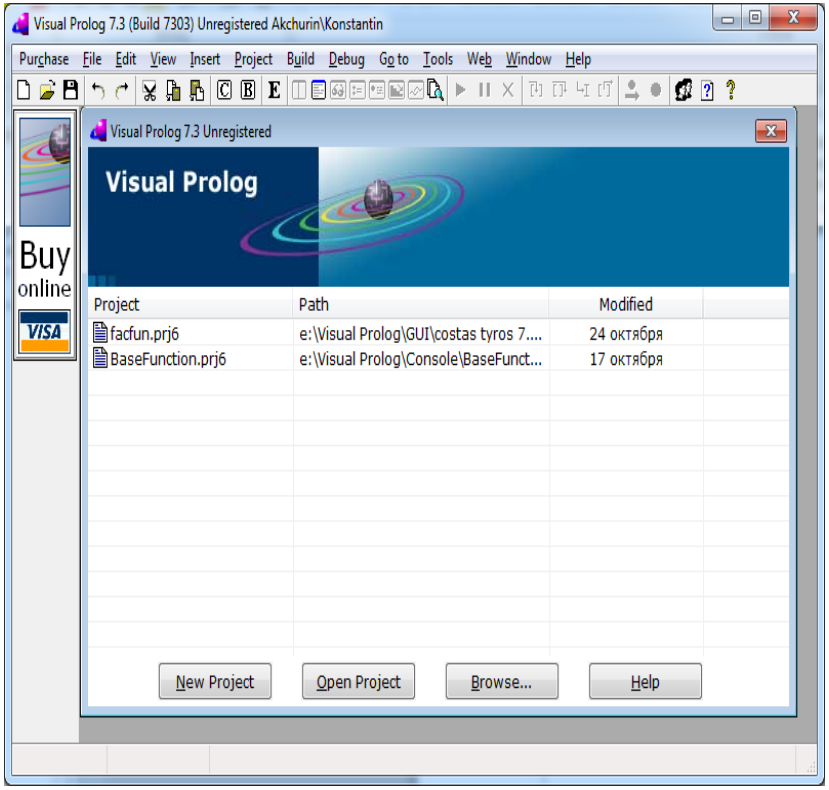
```
length_of([2, 3], TailLength).
```

Другими словами, length_of вызывает сама себя **рекурсивно**.

4 ИСР Visual Prolog

Среда которую вы будете использовать для разработки программ, называется ИСР (Интегрированная Среда Разработки), на английском языке IDE (**Integrated Development Environment**).

При запуске ИСР отображается стартовая страница.



В центре может размещаться окно приглашения Visual Prolog, содержащее список ранее реализованных проектов и кнопки управления.

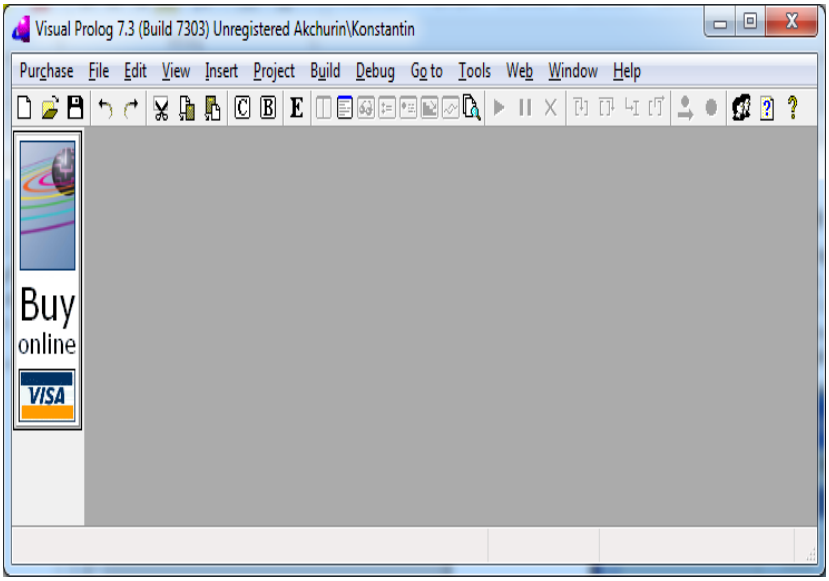
- New Progect – новый проект.
- Open Project – открыть проект, выделенный в списке.
- Browse – поиск и выбор каталога для проекта.
- Help – справка. Открыть учебник, встроенный в ИСР.

Из этого окна можно выбрать проект для работы.

ИСР имеет две реализации:

- Полная платная версия.
- Сокращенная бесплатная версия. В ней в левой позиции имеется пункт Purchase (покупка), позволяющий при желании перейти к платной версии.

4.1 Главное окно

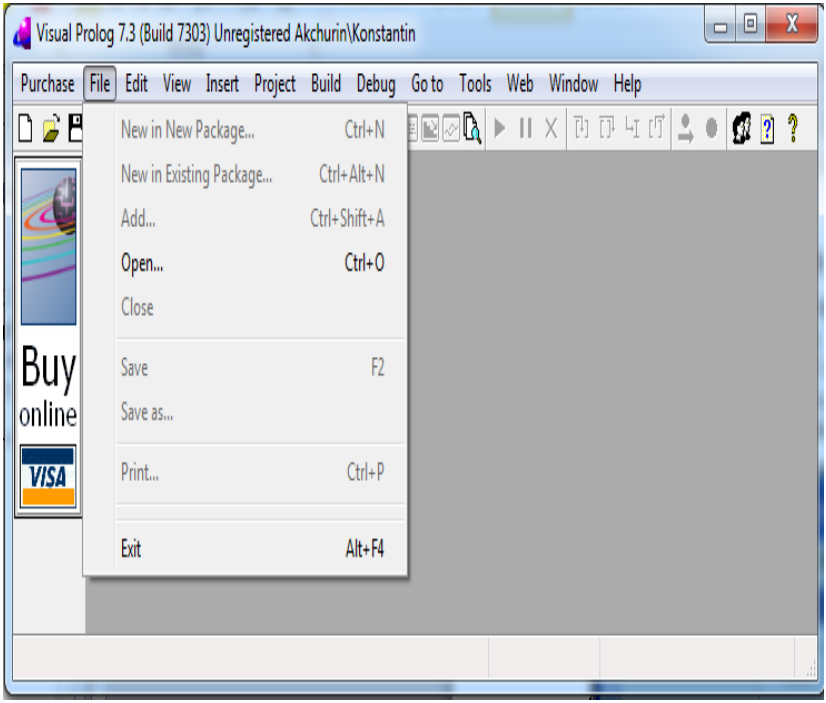


Окно содержит меню и панель инструментов.

Меню содержит следующие пункты

File (файл)	Операции с файлами	
Edit (правка)	Редактирование файла	
View (вид)	Что показывать	
Insert (ввести)	Что ввести в файл	
Project (проект)	Операции с проектом	
Build (построить)	Построить проект	
Debag (отладка)	Отладка	
Go to (идти к)	Выбор перехода	
Tools (инструменты)	Выбор инструментального средства	
Web (Интернет)	Работа в Интернете	
Window (окно)	Отображение окон	
Help (справка)	Получение справки	

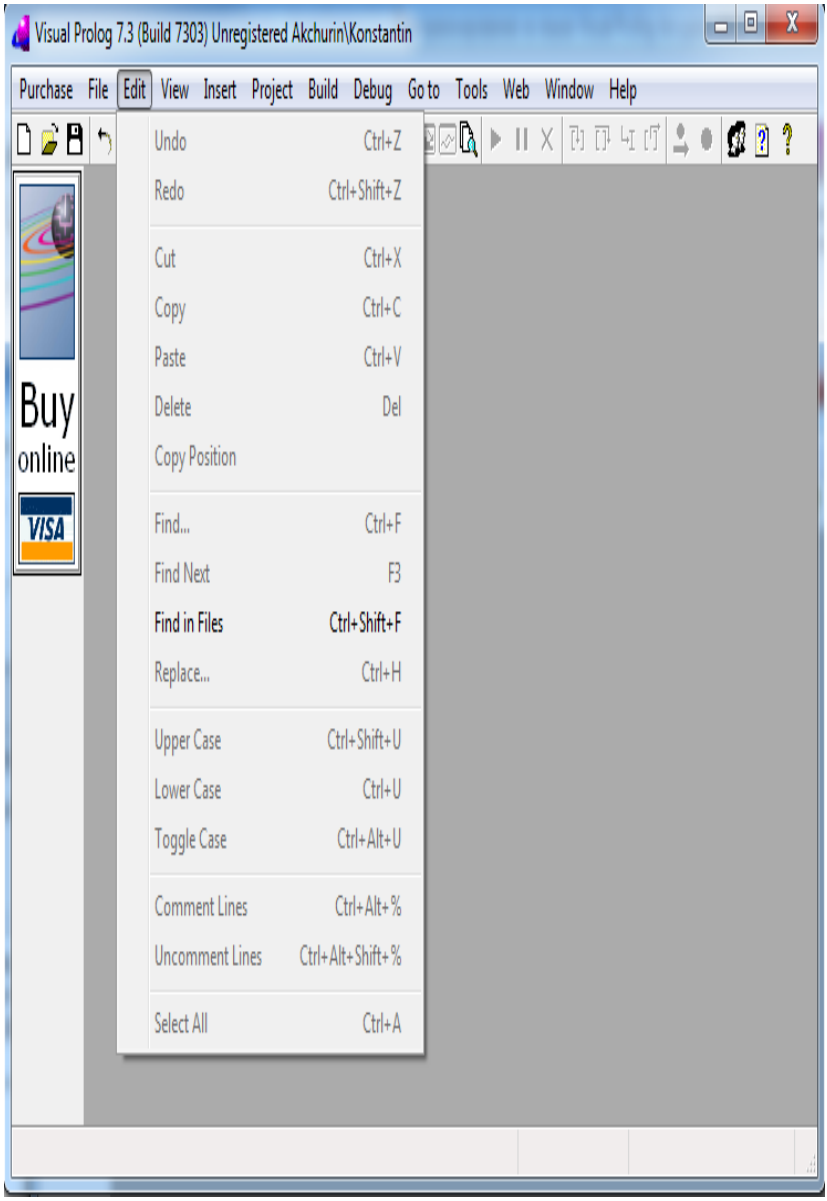
4.1.1 File (файл)



Пункт File (файл) содержит команды работы с файлами:

Команда	Описание
New in New Package.	Создать новую тему в новом пакете.
New in Existing Package	Создать новую тему в существующем пакете.
Add	Добавить.
Close	Закрыть.
Open	Открыть.
Save as…	Сохранить как …
Print …	Печать…
Exit	Выход из ИСР

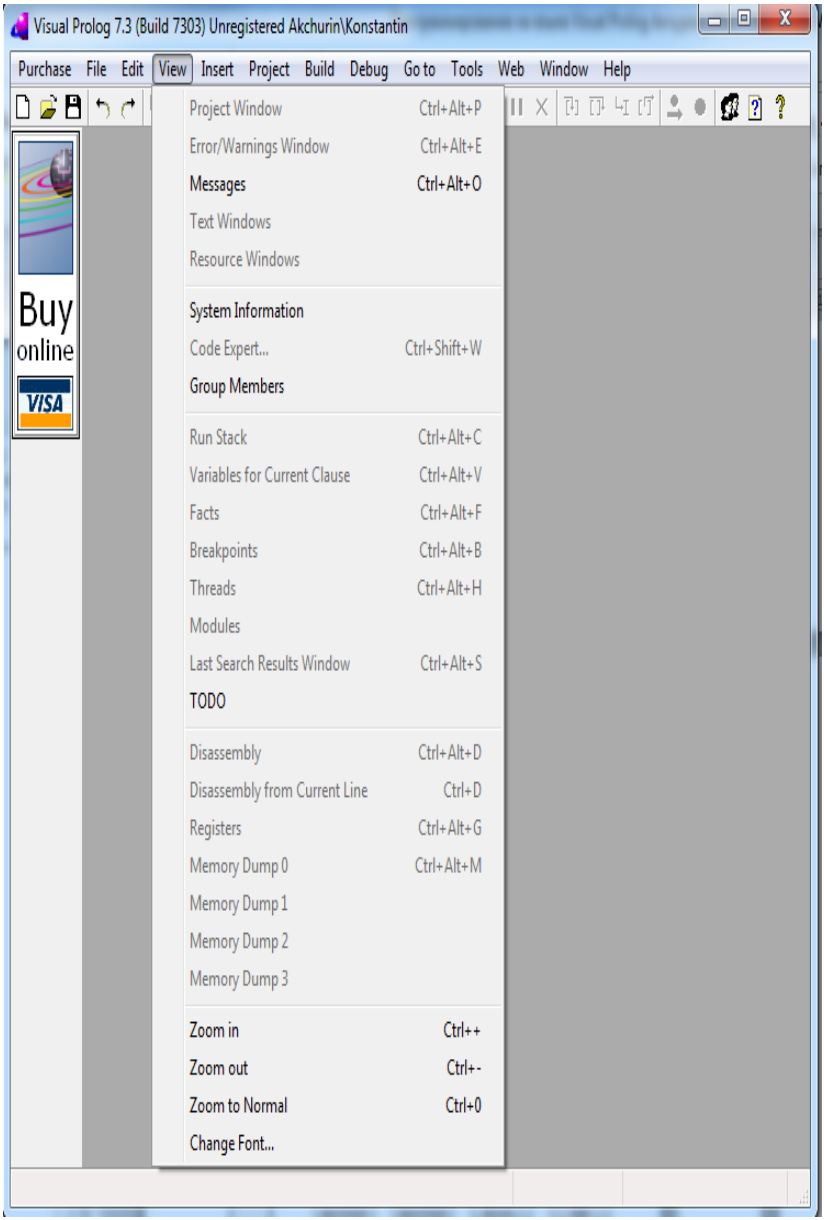
4.1.2 Edit (правка)



Пункт Edit (правка) содержит команды редактирования:

Команда	Описание
Undo	Отменить операцию
Redo	Восстановить операцию
Cut	Вырезать выделение
Copy	Копировать выделение
Paste	Открыть.
Delete	Удалить выделение
Copy Position	Копировать позицию выделенного
Find…	Найти…
Find Next…	Найти следующее…
Find in Files…	Найти в файлах…
Replace	Заменить выделенное…
Upper Case	Верхний регистр…
Lower Case	Нижний регистр…
Toggle Case	Переключить регистр…
Comment Lines	Сделать выделенные строки комментарием
Uncomment Lines	Сделать выделенные строки не комментарием
Select All	Выделить все

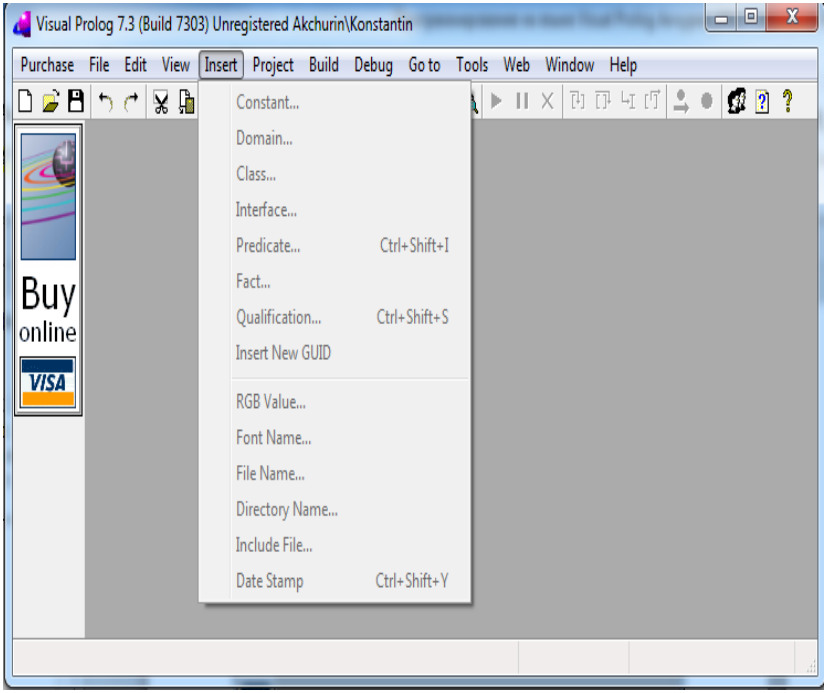
4.1.3 View (вид)


Visual Prolog 7.3 (Build 7303) Unregistered Akchurin\Konstantin
Purchase File Edit View Insert Project Build Debug Go to Tools Web Window Help
Project Window Ctrl+Alt+P
Error/Warnings Window Ctrl+Alt+E
Messages Ctrl+Alt+O
Text Windows
Resource Windows
System Information
Code Expert... Ctrl+Shift+W
Group Members
Run Stack Ctrl+Alt+C
Variables for Current Clause Ctrl+Alt+V
Facts Ctrl+Alt+F
Breakpoints Ctrl+Alt+B
Threads Ctrl+Alt+H
Modules
Last Search Results Window Ctrl+Alt+S
TODO
Disassembly Ctrl+Alt+D
Disassembly from Current Line Ctrl+D
Registers Ctrl+Alt+G
Memory Dump 0 Ctrl+Alt+M
Memory Dump 1
Memory Dump 2
Memory Dump 3
Zoom in Ctrl++
Zoom out Ctrl+-
Zoom to Normal Ctrl+0
Change Font...
Buy online
VISA

Пункт View (вид) содержит команды просмотра:

Команда	Описание
Project Window	Окно проекта
Error/Warnings Window	Окно ошибок/предупреждений
Messages	Сообщения
Text Windows	Текстовые окна
Resource Windows	Окна ресурсов
System Information	Системная информация
Code Expert	Эксперт Кода
Group Members	Члены группы
Run Stack	Запуск стека
Variables for Current Clause	Переменные для текущей Клаузы
Facts	Факты
Breakpoints	Точки останова
Threads	Нити
Modules	Модули
Last Search Results Window	Окно результатов последнего ноиска
TODO	Делать
Disassembly	Дизассемблер
Disassembly from Current Line	Дизассемблер из текущей строки
Registers	Регистры
Memory Dump 0	Дамп памяти 0
Memory Dump 1	Дамп памяти 1
Memory Dump 2	Дамп памяти 2
Memory Dump 3	Дамп памяти 3
Zoom In	Увеличить масштаб
Zoom Out	Уменьшить масштаб
Zoom Nornal	Нормальный масштаб
Change Font	Изменить шрифт

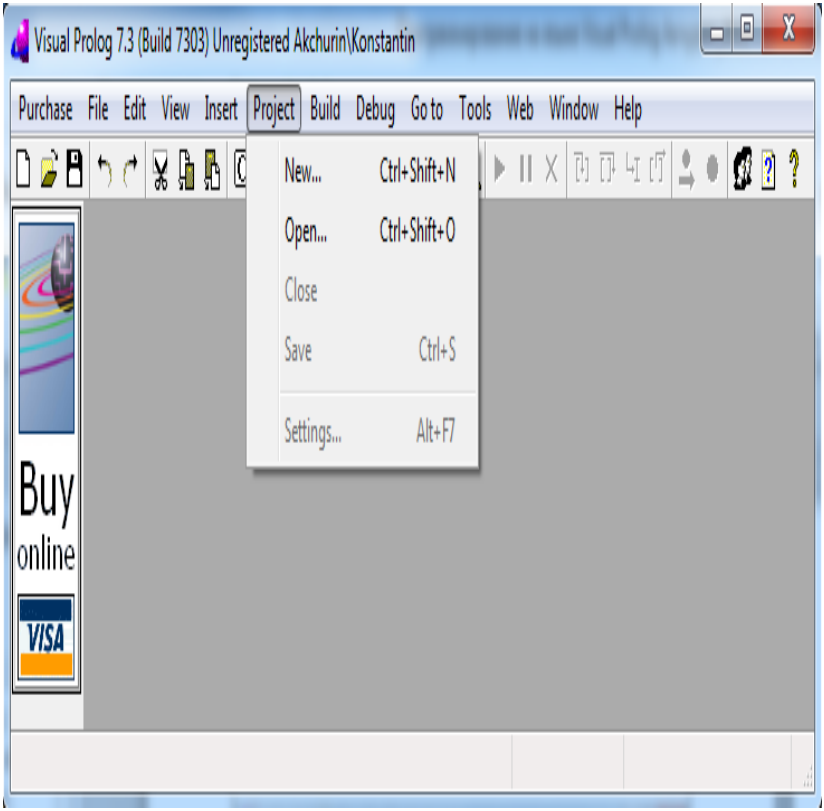
4.1.4 Insert (ввести)



Пункт Insert (ввести) содержит команды ввода в файл компонент:

Команда	Описание
Constant	Константа
Domain	Домен
Class	Класс
Interface	Интерфейс
Predicate	Предикат
Fact	Факт
Qualification	Квалификация
Insert New GUID	Ввести новый GUID
RGB Value	Значение RGB
Font Name	Имя шрифта
File Name	Имя файла
Directory Name	Имя каталога
Include File	Включить файл
Date Stamp	Дата

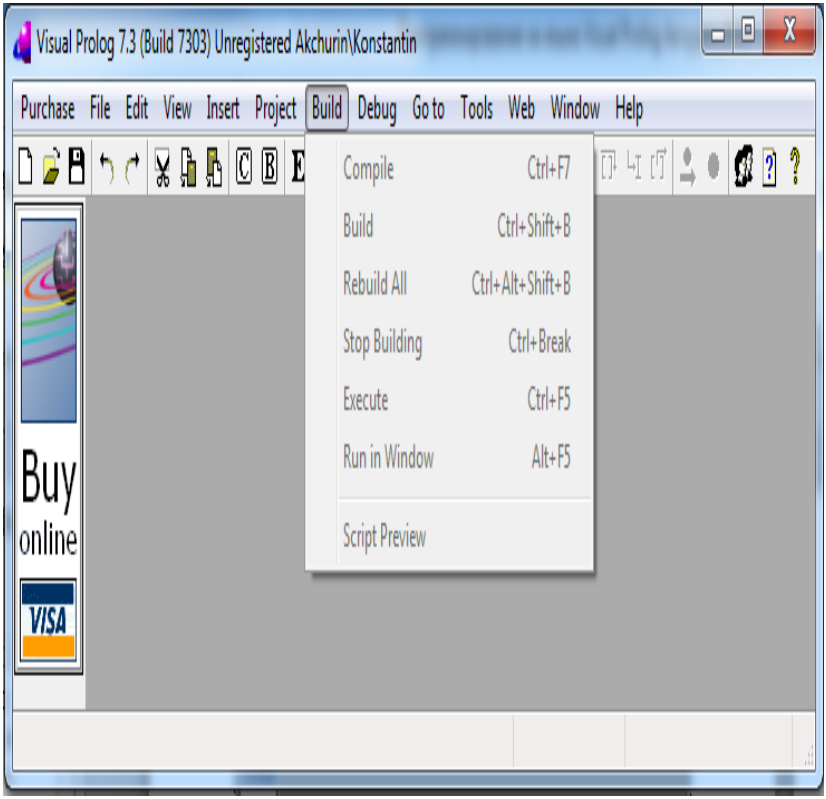
4.1.5 Project (проект)



Пункт Project (проект) содержит команды работы с проектом:

Команда	Описание
New	Новый
Open	Открыть
Close	Закрыть
Save	Сохранить
Settings	Установки

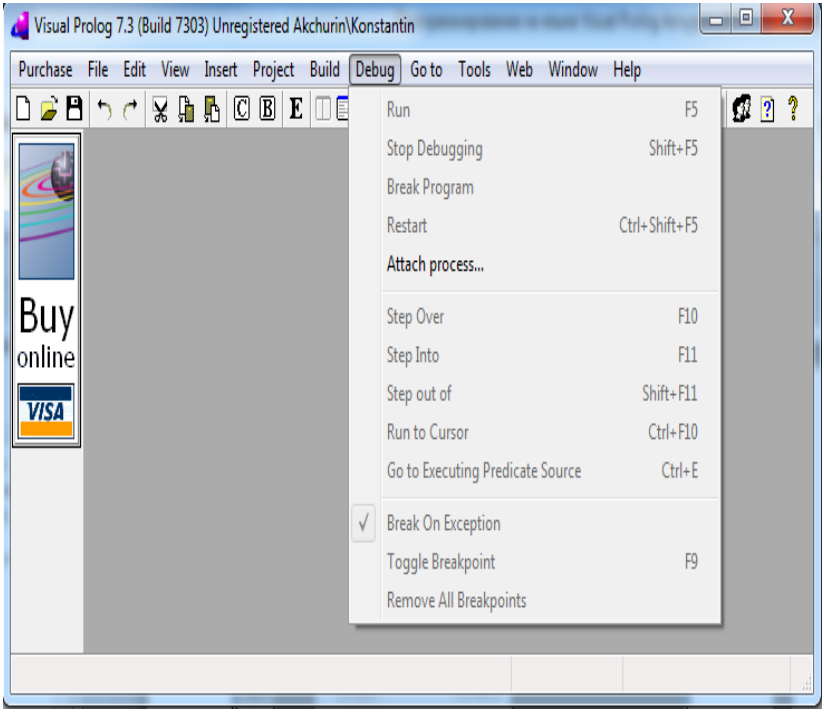
4.1.6 Build (построить)



Пункт Build (построить) содержит команды построения проекта:

Команда	Описание
Compile	Компилировать
Build	Построить
Rebuild All	Перестроить все
Step Building	Пошаговое построение
Execute	Запустить
Run in Windows	Запустить в Windows
Script Preview	Предварительный просмотр сценария

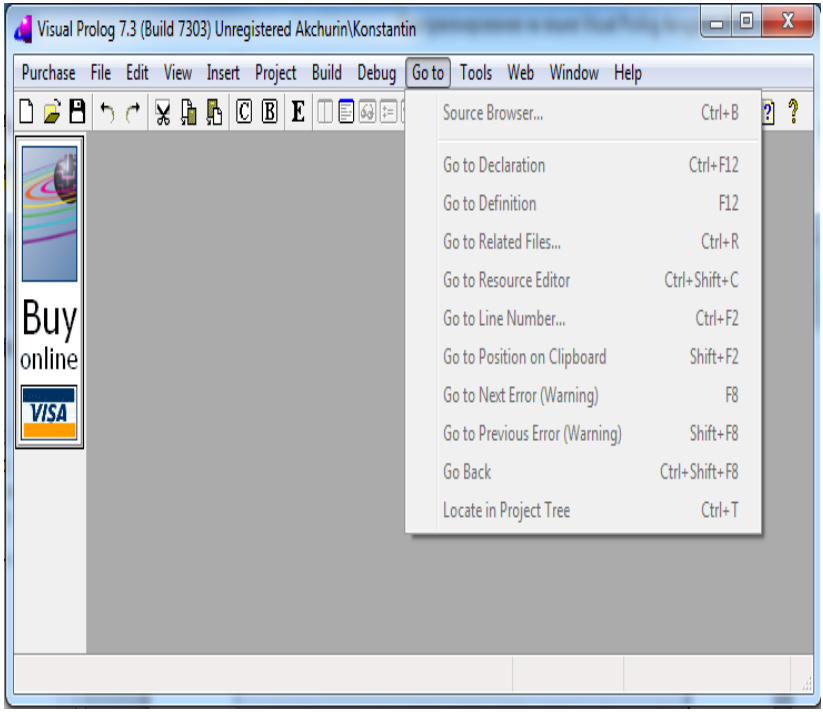
4.1.7 Debag (отладка)



Пункт Debag (отладка) содержит команды отладки проекта:

Команда	Описание
Run	Запуск
Stop Debugging	Остановить отладку
Break Program	Выйти из программы
Restart	Запуск повторный
Attach Process	Прикрепить процесс
Step Over	Шаг без останова в процедуре
Step Into	Шаг с заходом в процедуру
Step Out Of	Шаг выхода из процедуры
Run to Cursor	Прогон до позиции курсора
Go to Executing Predicate Source	Перейти к исходника предиката запуска
Break on Exception	Выход по Исключению
Toggle Breakpoint	Переключить точку прерывания
Remove All Breakpoints	Удалить все точки прерывания

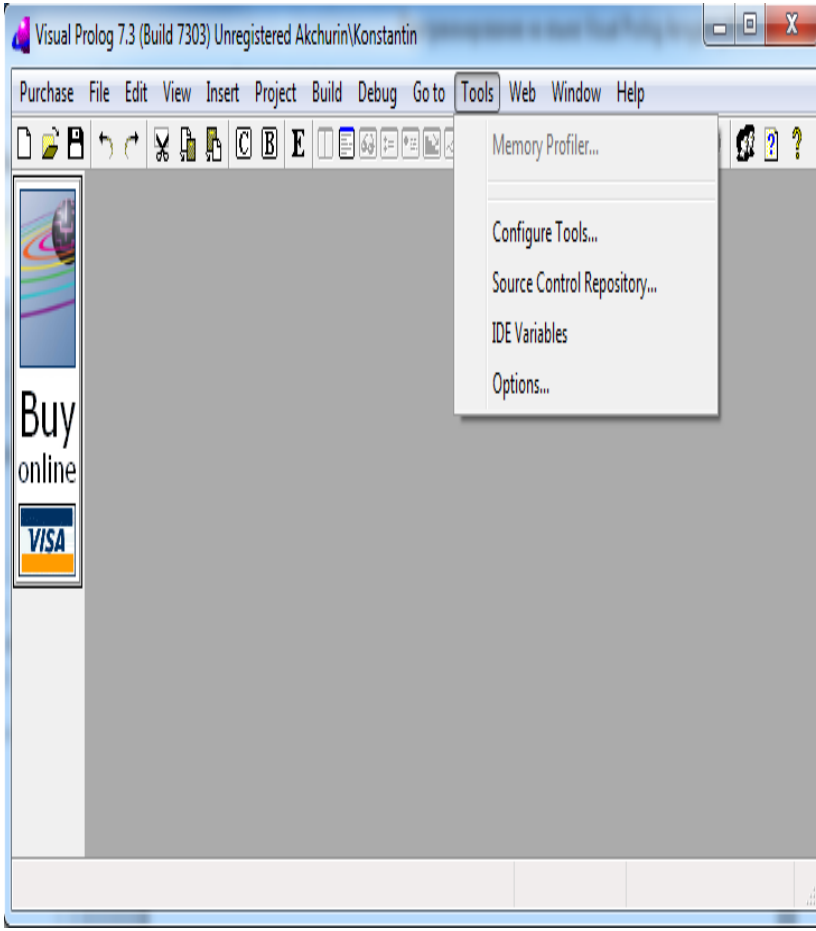
4.1.8 Go to (идти к)



Пункт Go to (идти к) содержит команды перемещения по проекту:

Команда	Описание
Source Browser	Браузер исходника
Go to Declaration	Идти к объявлению
Go to Definition	Идти к определению
Go to Related Files	Идти к связанным файлам
Go to Resource Editor	Идти к редактору исходника
Go to Line Number	Идти к остртоке номер
Go to Position on Clipboard	Идти к позиции в буфере обмена
Go to Next Error (Warning)	Идти к следующей ошибке (предупреждению)
Go to Previous Error (Warning)	Идти к предыдущей ошибке (предупреждению)
Go Back	Идти назад
Locate in Project Tree	Локализация в дереве проекта

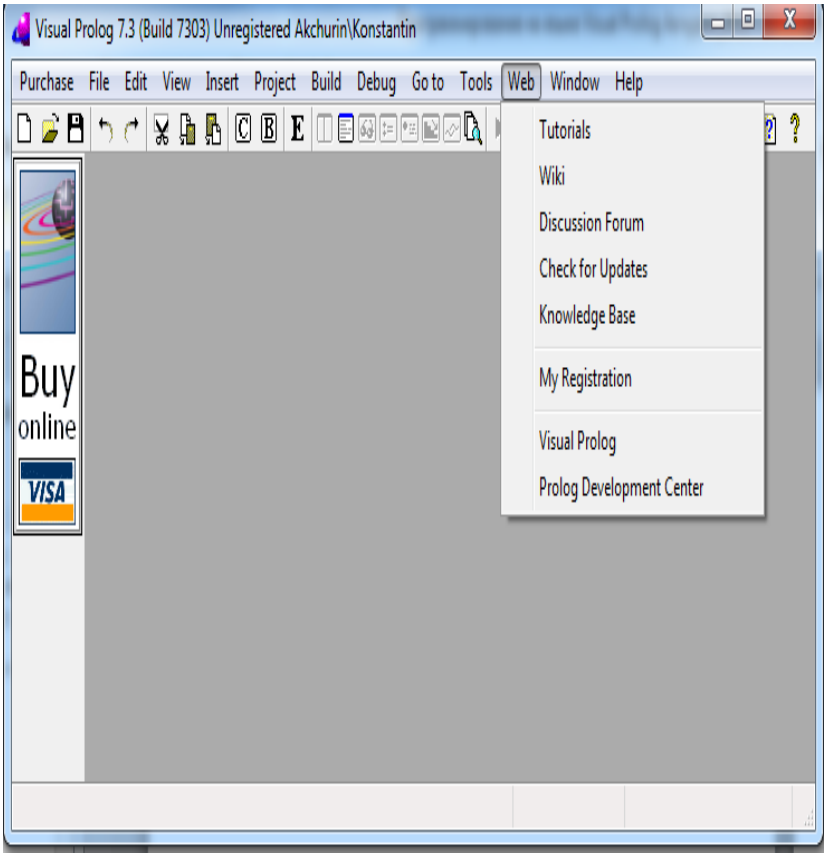
4.1.9 Tools (инструменты)



Пункт Tools (инструменты) содержит команды выбора инструментальных средств:

Команда	Описание
Memory Profilier	Профилировщик памяти
Configure Tools	Средства конфигурирования
Source Control Repository	Репозиторий управления исходником
IDE Variables	Переменные ИСР
Options	Опции

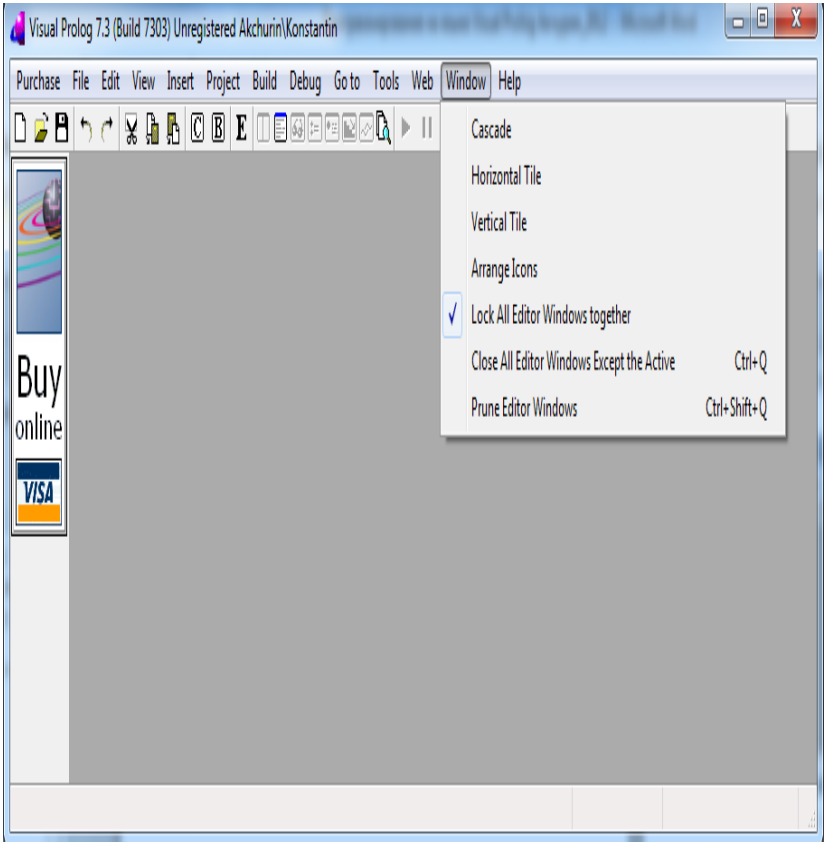
4.1.10 Web (Интернет)



Пункт Web (Интернет) содержит команды доступа в Интернет:

Команда	Описание
Tutorials	Учебники
Wiki	Wiki
Discussion Forum	Форум обсуждений
Check for Update	Проверить обновление
Knowledge Base	База знаний
My Registration	Моя регистрация
Visual Prolog	Visual Prolog
Prolog Development Center	Центр разработки Prolog

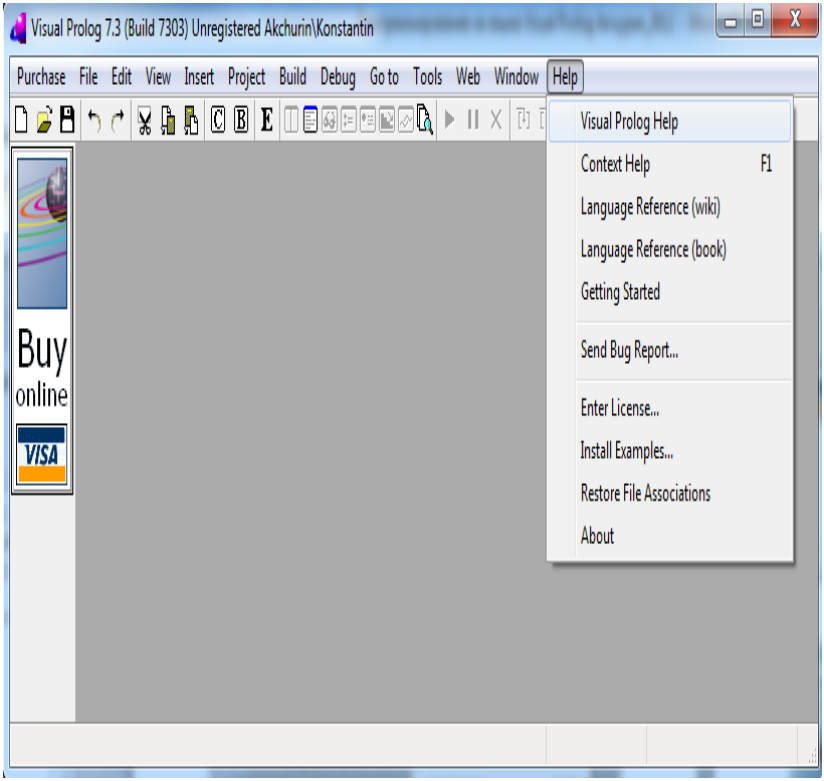
4.1.11 Window (окно)



Пункт Window (окно) содержит команды выбора представления окна.

Команда	Описание
Cascade	Каскадное
Horisontal Tile	Рядом по горизонтали
Vertical Tile	Рядом по вертикали
Arrange Icons	размещение иконок
Lock All Editor Windows together	Связать все окна редакторов
Close All Editor Windows except Active	Закрыть все окна редакторов, кроме активного
Prune Editor Windows	Окна черновиков редакторов

4.1.12 Help (справка)

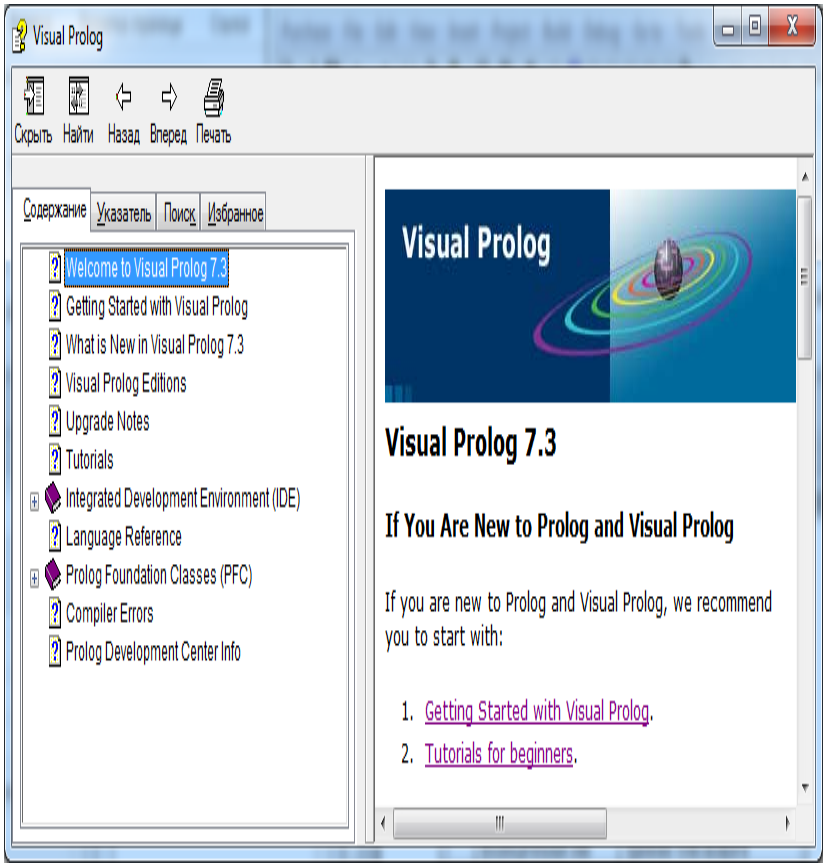


Пункт Help (справка) содержит средства получения справки:

Команда	Описание
Visual Prolog Help	Visual Prolog справка
Context Help	Контекстная справка
Language Reference (wiki)	Языковая ссылка (wiki)
Language Reference (book)	Языковая ссылка (книга)
Getting Started	Начальное обучение
Send Bug Report	Послать отчет о дефектах
Enter License	Получить лицензию
Install Examples	Установить примеры
Restore File Associations	Переустановить ИСР
About	О производителе ИСР

4.1.13 Встроенный учебник ИСР Visual Prolog

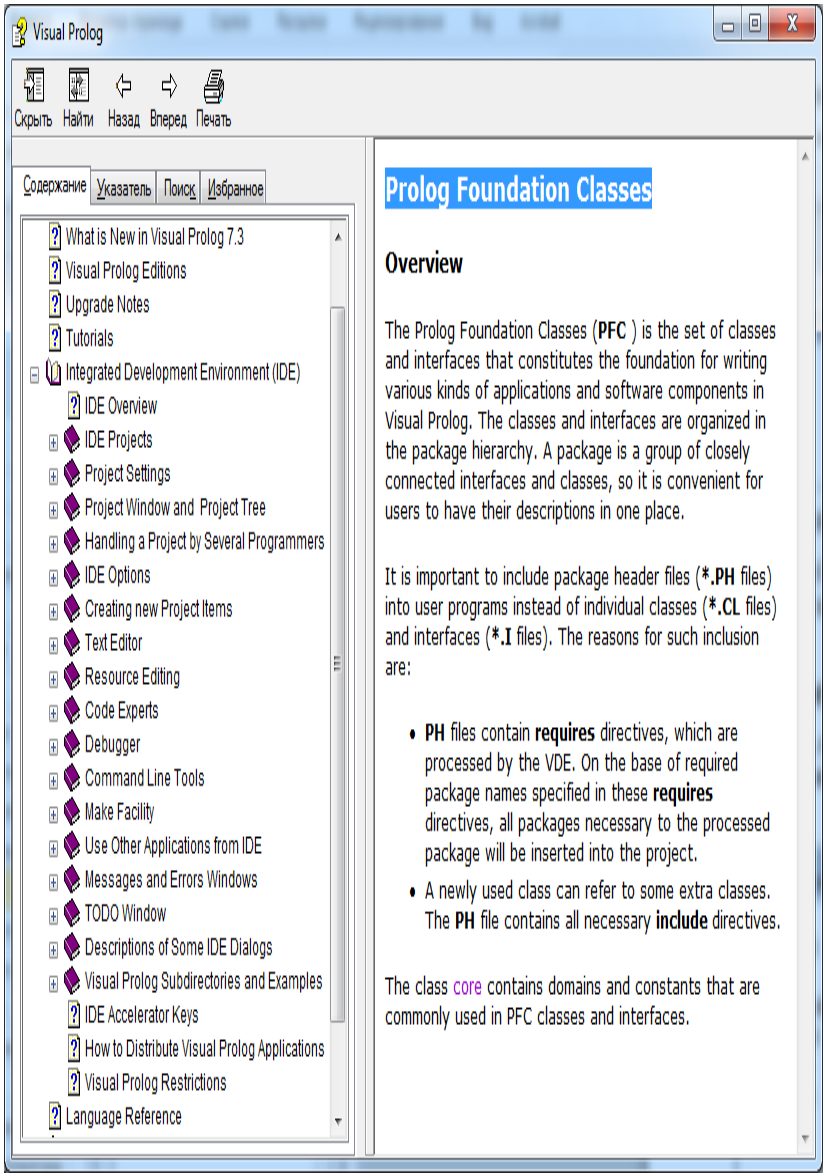
ИСР Visual Prolog содержит много классов, просмотреть которые можно, используя встроенный учебник, вызов учебника из пункта Help меню.



Окно просмотра содержит 2 вкладки: слева – содержание, справа информация о выбранном разделе. Доступны разделы:

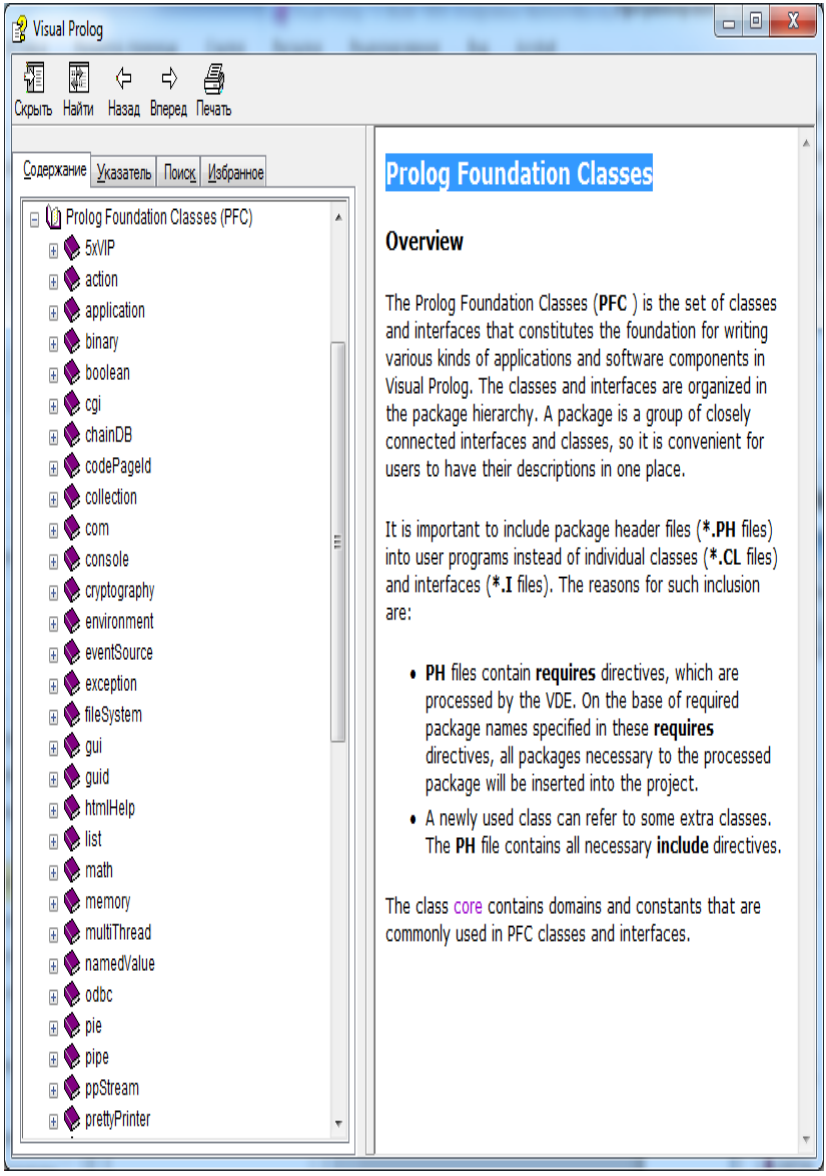
- Integrated Development Environment (IDE) – ИСР.
- Prolog Foundation Classes (PFC) – базовые классы языка Prolog.

Классы ИСР.

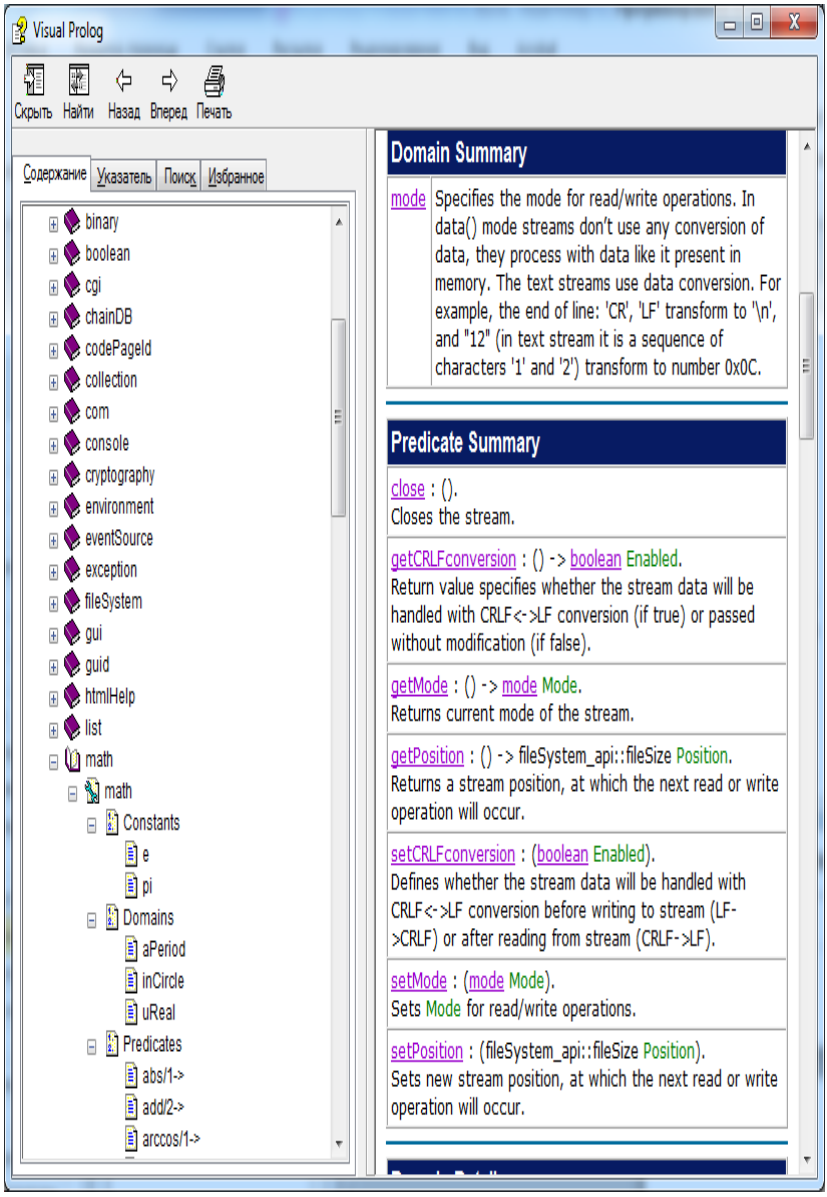


Классы PFC.

PFC – это набор базовых классов языка Пролога, встроенных в ИСР. Ниже часть классов.

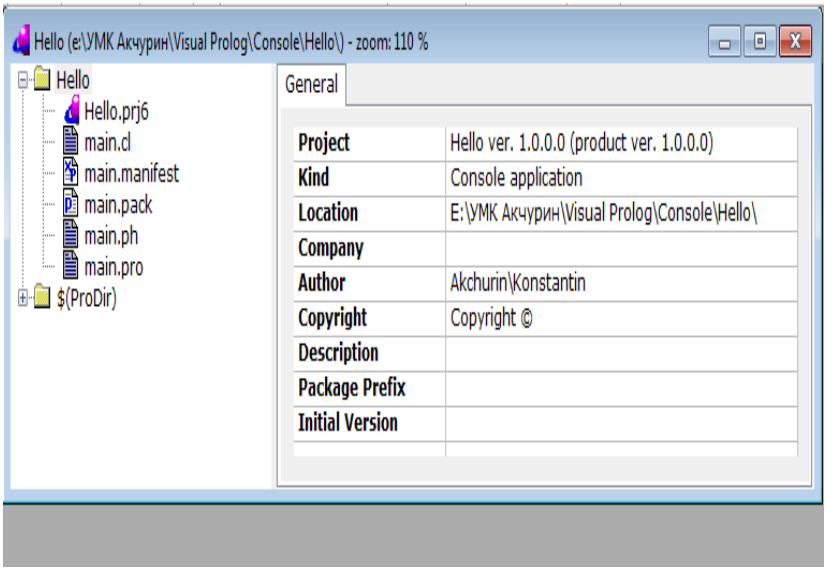


Для каждого класса представлены его объекты. Например, для класса math:



4.2 Структура программы

Чтобы понять принципы работы программы Visual Prolog, давайте рассмотрим традиционную программу "Hello World!" и разберем каждую строку ее кода на Visual Prolog.



Проект Hello,prj6 включает модули, каждый из которых хранится в отдельном файле:

Модуль	Описание
Hello.prj6	Содержит правила формирования проекта
main.cl	Объявление главного класса. Для консольного приложения здесь есть ссылка на предикат run
manifest	Содержит сведения о системных средствах для построения проекта.
main.pack	Пакет главного класса, содержит ссылки на модули проекта.
main.ph	Содержит заголовки файлов проекта.
main.pro.	Реализация главного класса. Здесь исполняемый код.

Листинг модуля объявления класса main.cl.

```
class main
    open core  % открыть класс ядра Visual Prolog - core
```

```
predicates
    classInfo : core::classInfo.        % информация о классе предикатов
    % @short Class information  predicate.
    % @detail This predicate represents information predicate of this class.
    % @end
predicates
    run : core::runnable.
end class main
```

Листинг модуля реализации main.pro.

```
implement main
open core, console                          % Добавлен класс console

constants
    className = "main".
    classVersion = "".

clauses
    classInfo(className, classVersion).

clauses
    run():-
        init(),                             % Инициализация консоли
        write("Hello, World от Акчурина!"), % Вывод текста
        _=readLine()                        % Ждем Enter
end implement main

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

При построении проекта модуль реализации формируется по шаблону, а котором задаются разделы:

- **implement main**. Это заголовок, тип – реализация (implement), модуль – главный (main).
- **constants**. Константы. В шаблоне имя класса и его версия. Можно добавить свои константы.
- **clauses**. Предложения. В шаблоне информация о классе и его версия. Можно добавить свои предложения.
- **clauses**. Предложения. Сюда программист помещает свой код.
- **end implement main**. Конец модуля.
- **goal** mainExe::run(main::run). Цель – запуск файла mainExe.

4.3 Пространства имен

Пространства имен представляют собой способ организации различных типов, присутствующих в программах Visual Prolog. Их можно сравнить с папкой в компьютерной файловой системе. Подобно папкам, пространства имен определяют для классов уникальные полные имена. Программа Visual Prolog содержит одно или несколько пространств имен, каждое из которых либо определено программистом, либо определено как часть написанной ранее библиотеки классов.

Например, пространство имен PFC содержит класс console, который включает методы для чтения и записи в окне консоли.

При написании класса вне объявления пространства имен компилятор предоставит ему заданное по умолчанию пространство имен.

Для использования метода write, определенного в классе console, без предварительного определения этого класса следует использовать строку кода

```
console::write("Hello, World!").
```

Для упрощения программирования в начало исходного файла Visual Prolog целесообразно вставить ссылку на класс, задающий пространство имен. После этого можно написать

```
write("Hello, World!").
```

4.4 Разделы программы

4.5 Реализация

Формат

```
Implement имя_класса
    ScopeQualifications (ссылка на классы)
    Разделы
end Implement
```

ScopeQualifications должен быть из:

- open – открыть.
- support – список интерфейсов, поддерживаемых данным классом.
- Inherit – наследование.
- delegate – делегирует функциональность предикатов интерфейса из предикатов БД.

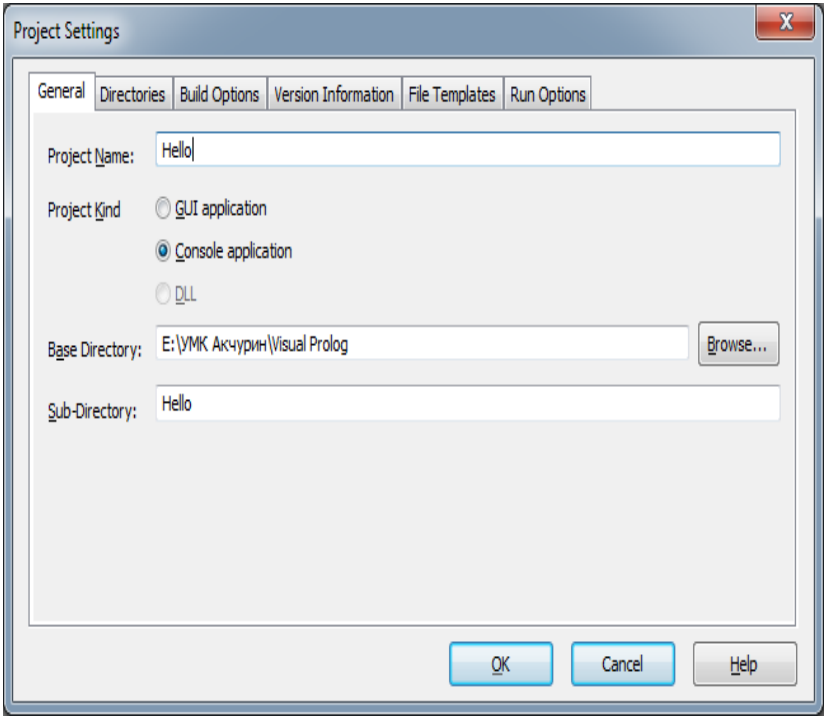
Разделы должны быть из:

- constants –константы.
- domains - домены.
- predicates - предикаты.
- properties - свойства.
- facts - факты.
- clauses - предложения.
- conditional - условия.

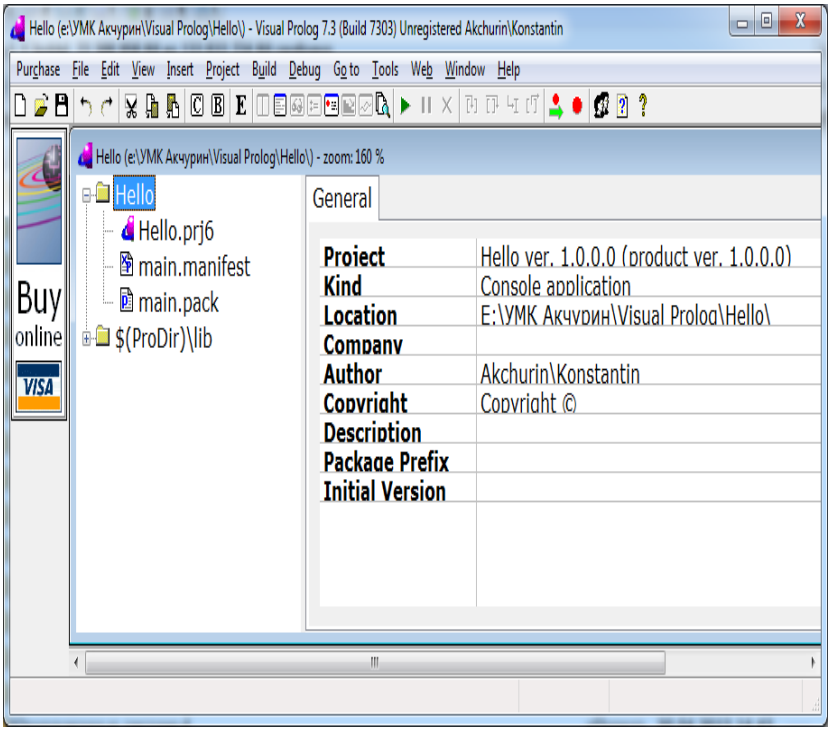
4.6 Консольное приложение Hello

Выберите команду **Project/New** меню задач. Затем заполните диалоговое окно **Project Settings**. В нем нужно определить:

- Имя проекта – Project Name.
- Тип проекта – Project Kind.
- Базовый каталог - Base Directory. Выбор осуществляется с помощью браузера Browse, вызывающего диалоговое окно файлов.
- Подкаталог - Sub Directory. Задается автоматически.



Для выбранного проекта строится дерево, которое содержит системные шаблоны и встроенные библиотеки. В левой панели - дерево проекта, в правой – браузер проекта с закладкой General.

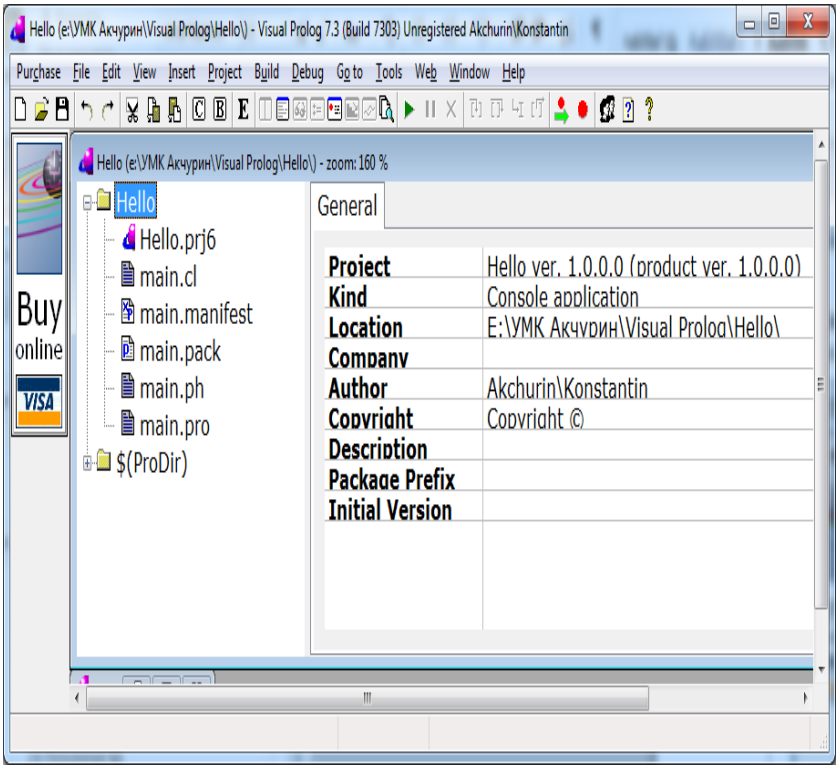


Стартовое дерево проекта Hello включает системные компоненты, которые не надо редактировать (ИСР использует их при компиляции):

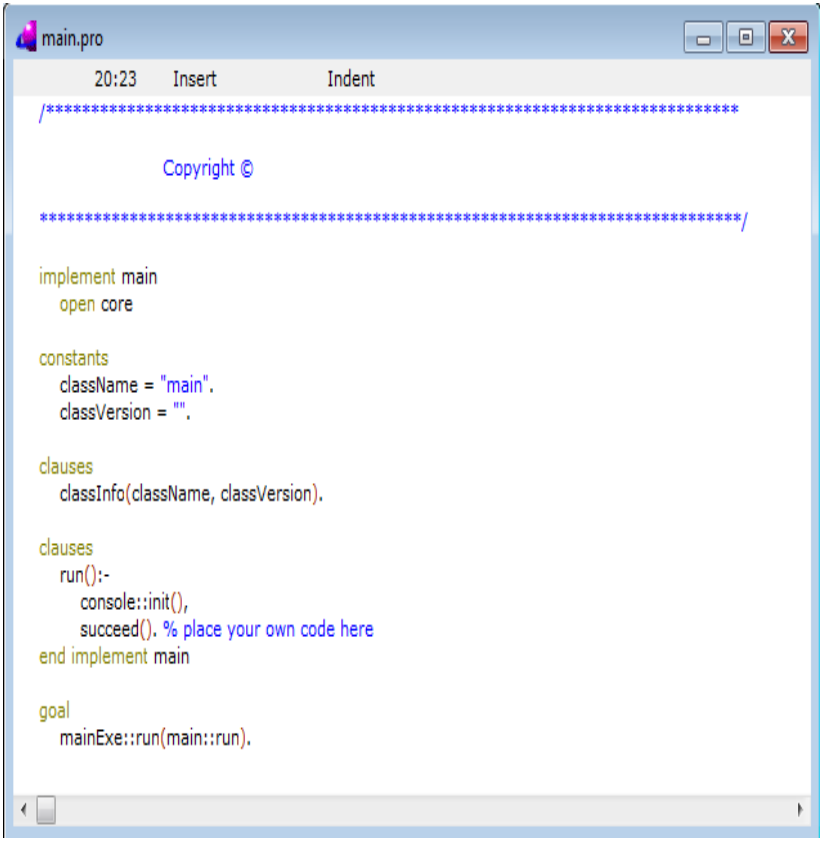
- Проект - hello.prj6.
- Манифест для приложения - main.manifest.
- Пакет для приложения - main.pack.
- Подключаемые системные библиотеки - $(ProDir)\lib.

Командой Build/Build осуществляется компиляция проекта, в дерево проекта добавляются файлы проекта **без функциональности**:

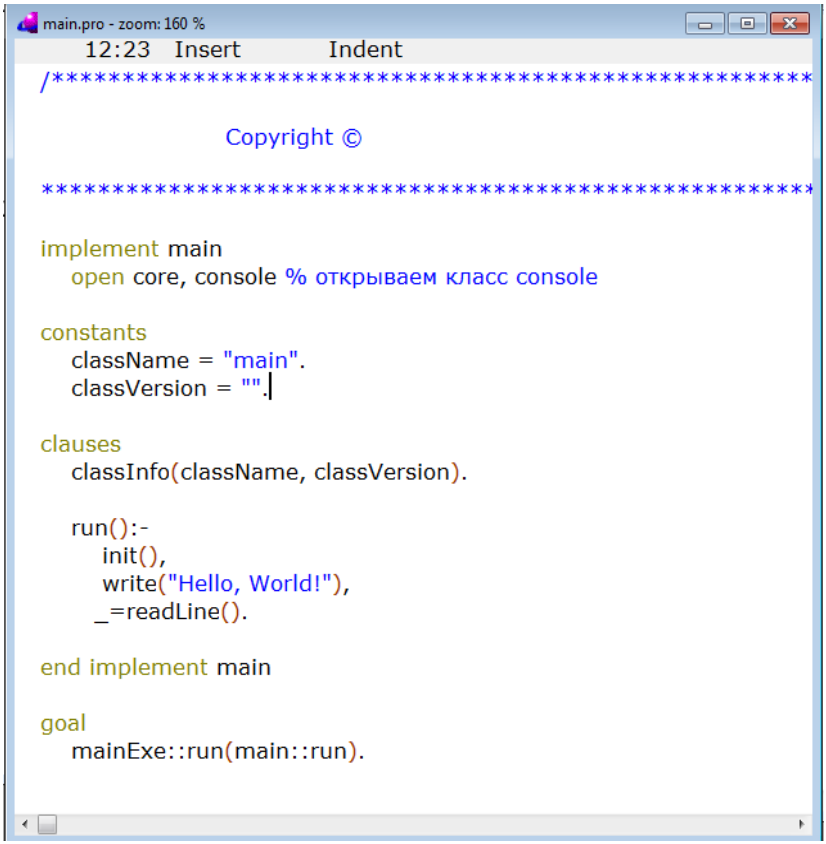
- Шаблон класса - main.cl.
- Содержание проекта - main.ph.
- Шаблон реализации приложения - main.pro.



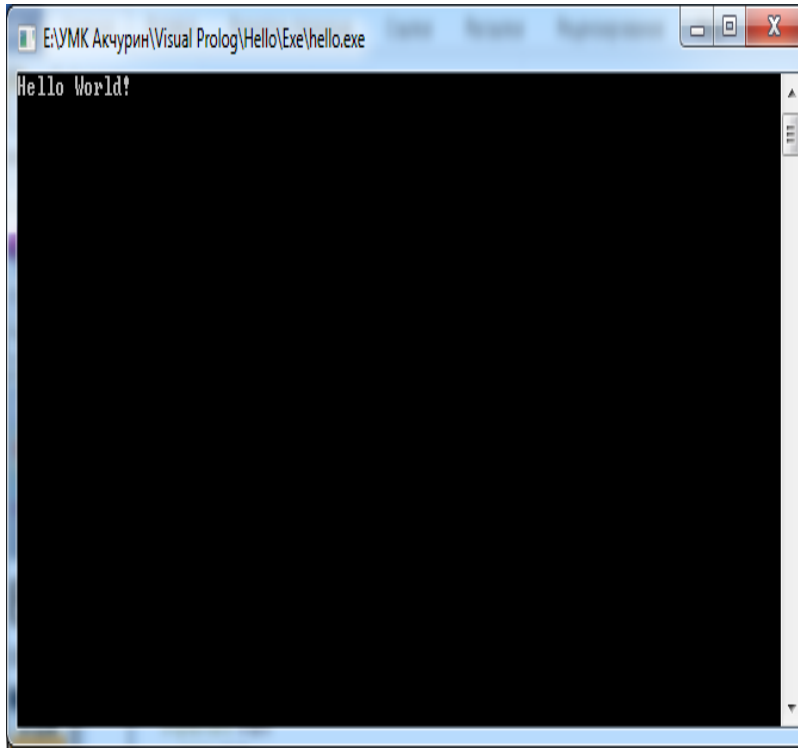
Для задания функциональности щелчком мыши выбираем main.pro, в шаблон main.pro нужно добавить свои операции. В листинге отмечено место, куда надо добавить коды.



Добавляем в файл main.pro функциональность. Вставляем в файл команду чтения текста ввода readLine(). Кроме того, во фрагменте заголовка открываем и класс консоли (**console**), что позволяет в предикате run() консоль не упоминать.



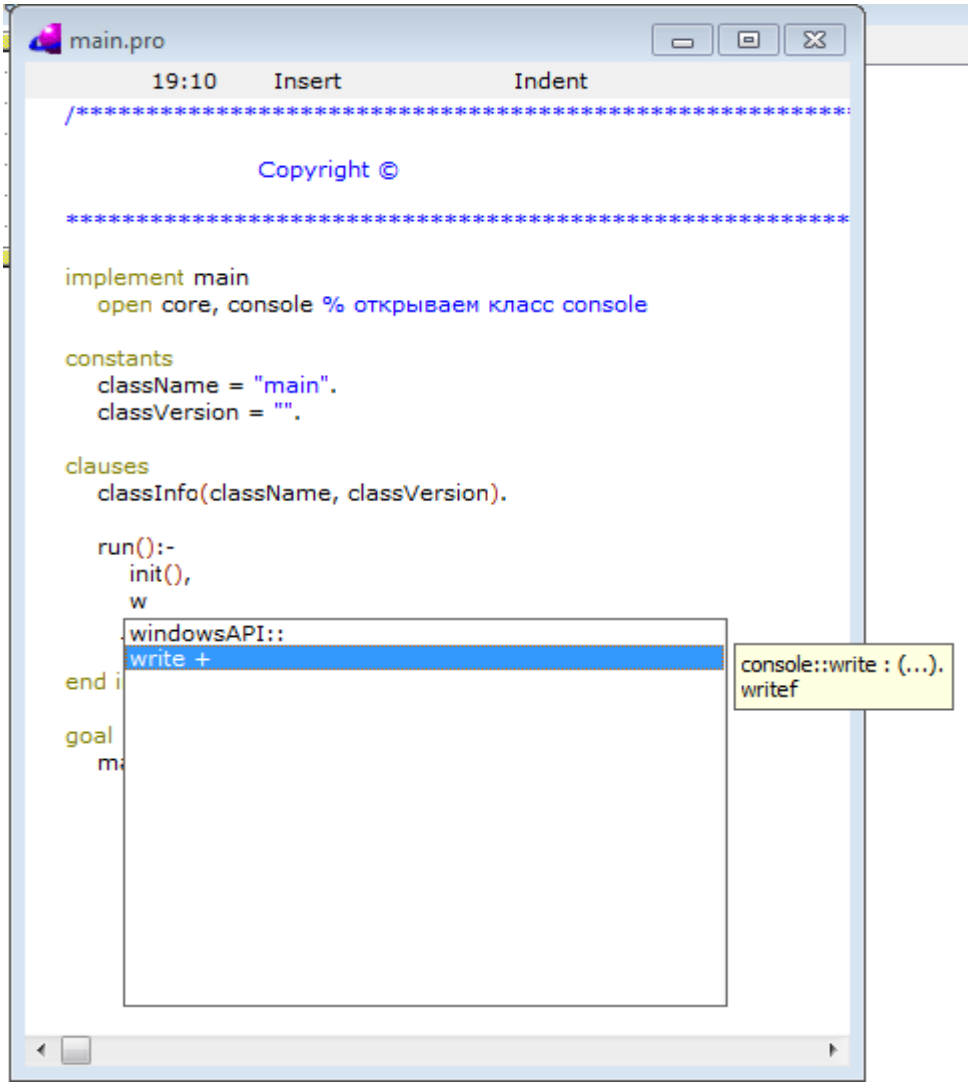
При запуске проекта командой Build/Execute появляется окно консоли с текстом Hello, World! Для завершения нажмите Enter.



При наличии ошибок выводится сообщения о них (номер ошибки, строка и символ в строке). В листинге делаются пометки ошибок подчеркиванием красным цветом. К сожалению, в листинге нет нумерации строк.

Для правильных кодов применятся цветовая семантическая подсветка. Например, строки показываются голубым цветом.

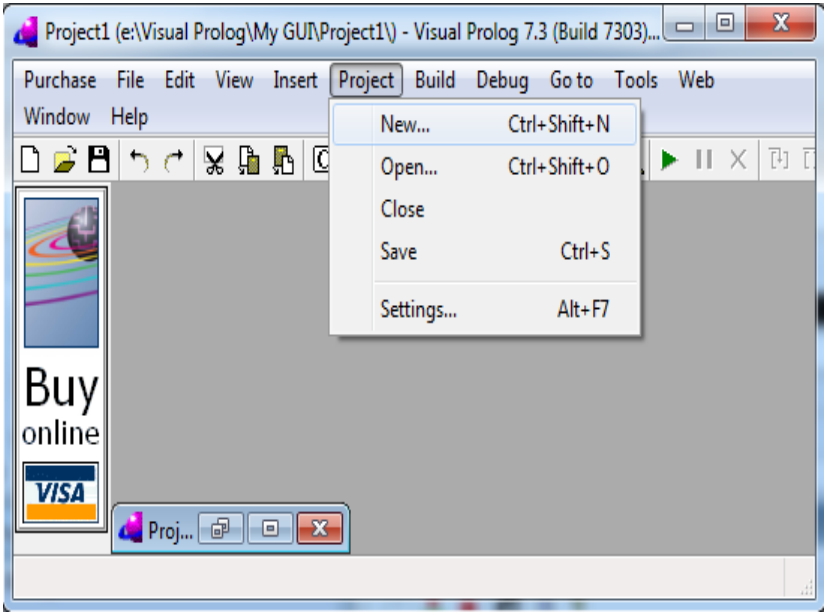
ИСР имеет **интеллектуальный подсказчик**. При наборе символа кода отображается список возможных продолжений для выбора, что избавляет программиста от запоминания и ручного набора предопределенных кодов. Если выбираемое продолжение кода имеет варианты, то они перечисляются во всплывающем окне справа.



Замечание. Подсказчику доступны идентификаторы, содержащиеся в классах проекта. По умолчанию выбирается только класс ядра **core**. Для работы с консолью следует добавить класс **console**, а для работы с математическими функциями класс **math**, Если это не сделано, то в коде перед идентификатором члена класса нужно добавлять имя класса и разделитель – двойное вертикальное двоеточие.

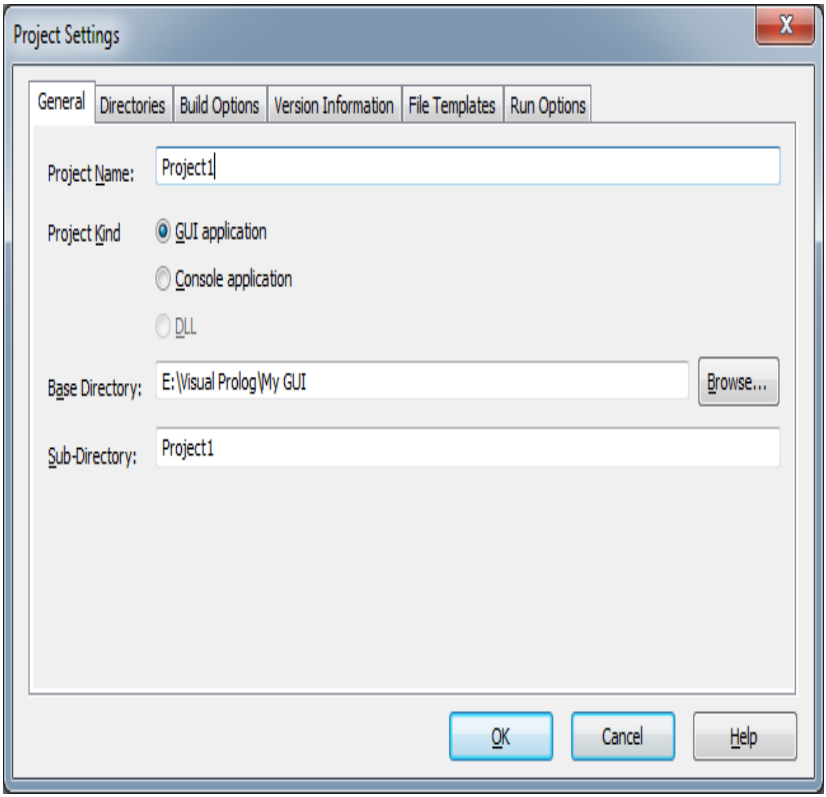
4.7 Создание проекта GUI в Visual Prolog

Когда вы заходите в ИСР системы Visual Prolog, то попадаете в среду. Мы будем ссылаться на меню ИСР как на «меню задач» (TaskMenu). Система окон и диалогов, которая создается для общения с пользователями программы, называется **Graphical User Interface** (сокращенно — GUI).

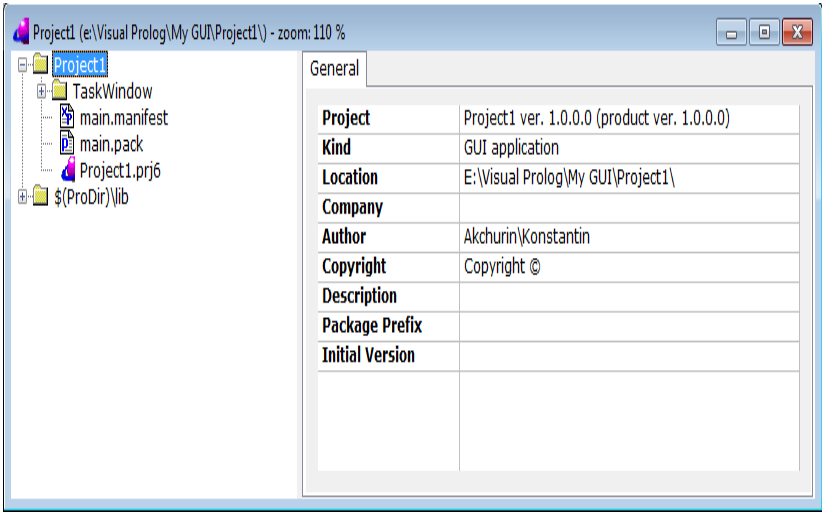


Выберите команду **Project/New** меню задач. Затем заполните диалоговое окно **Project Settings**. В нем нужно определить:

- Имя проекта – Project Name.
- Тип проекта – Project Kind.
- Базовый каталог - Base Directory. Выбор осуществляется с помощью браузера Browse, вызывающего диалоговое окно файлов.
- Подкаталог - Sub Directory. Задается автоматически.

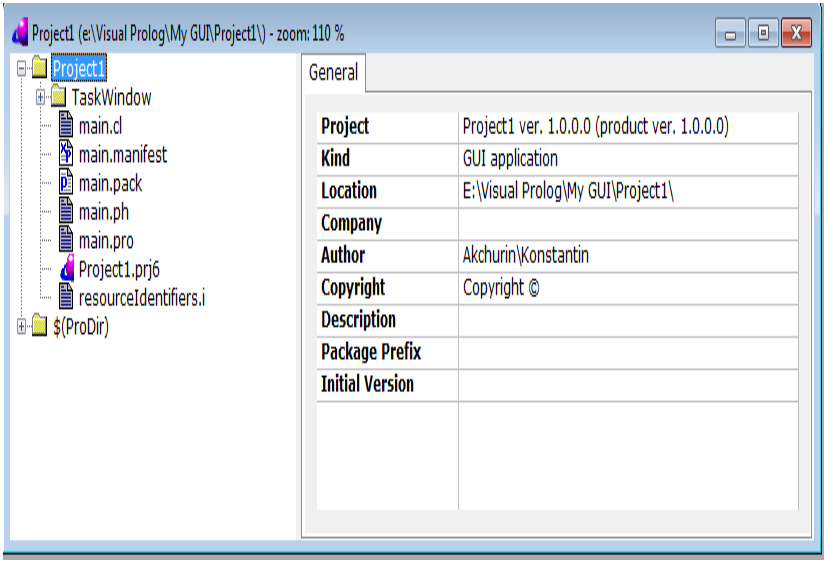


Нажмите кнопку **OK**, и перед вами появится окно шаблона дерева проекта.

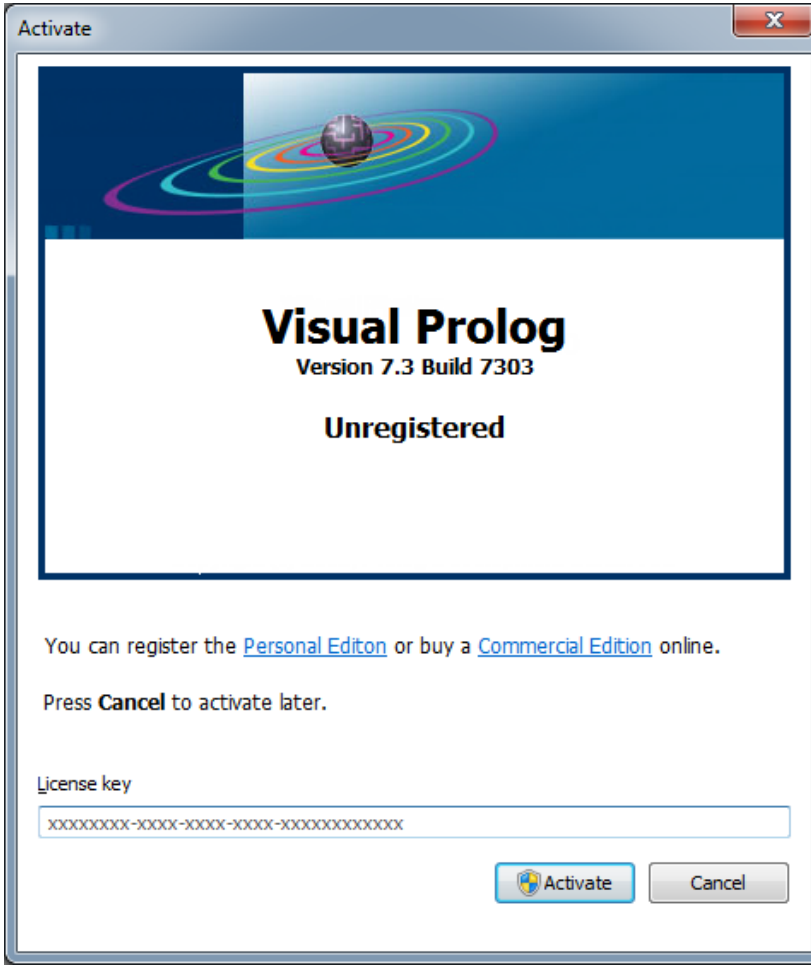


Компиляция и запуск программы

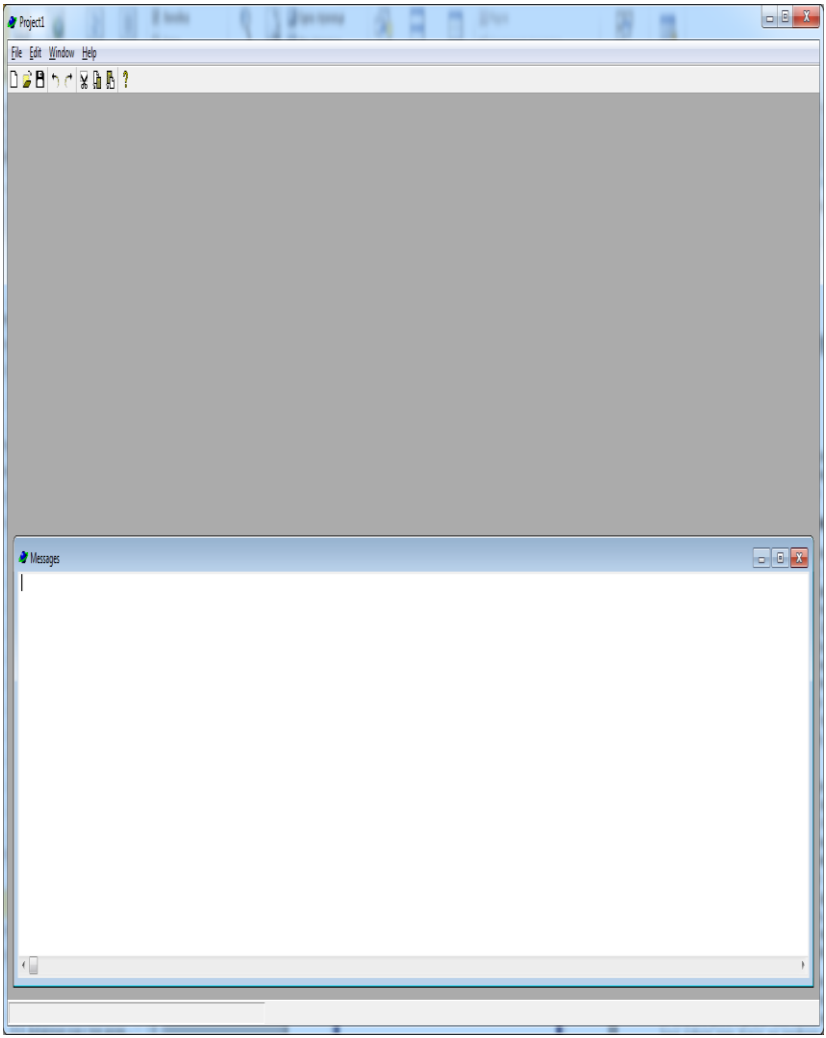
Для того чтобы скомпилировать программу, выберите команду **Build/Build** или **Build/Execute** меню задач. Осуществляется компиляция и запуск проекта. После компиляции получается дерево проекта. В нем к шаблону добавляются новые файлы.



Для запуска программы выберите команду **Build/Execute.** При этом появится окно с предложением активировать программу (купить платую версию).

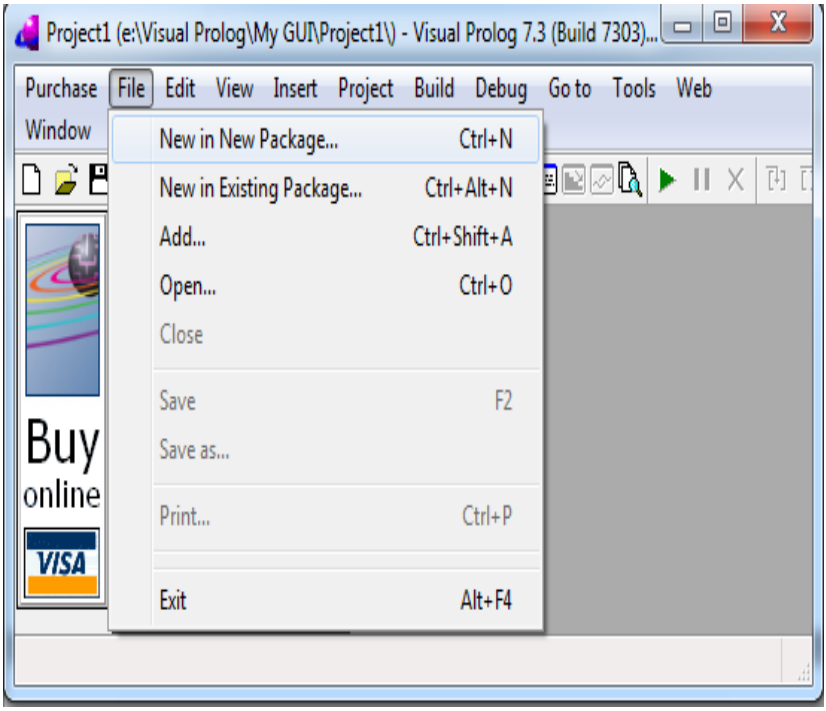


Нужно нажать кнопку Cancel и на экране появится окно.



Чтобы выйти из программы — нажать кнопку в виде крестика, которая находится в верхнем правом углу окна. Если предпочитаете, выберите пункт **File/Exit** меню приложения.

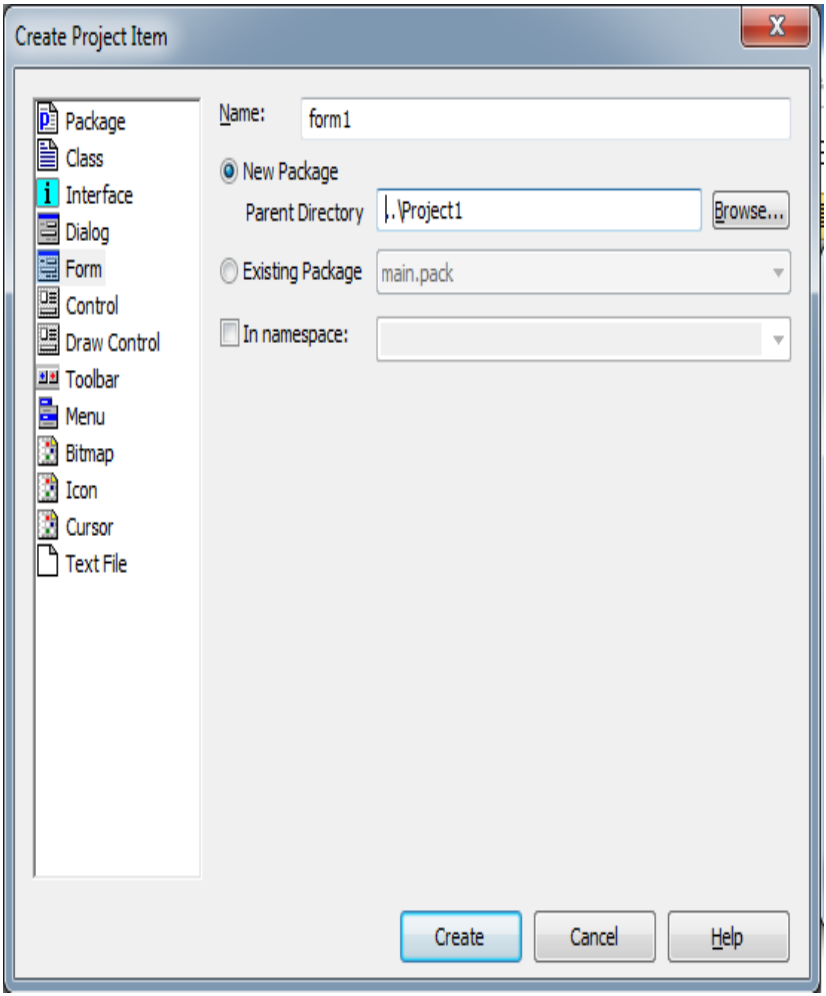
Для конкретных действий в проекте нужно создать тему. Для этого используем команду **File/New in New Package**.



Генерируется окно создания темы **Create Project Item**. Можно создавать темы:

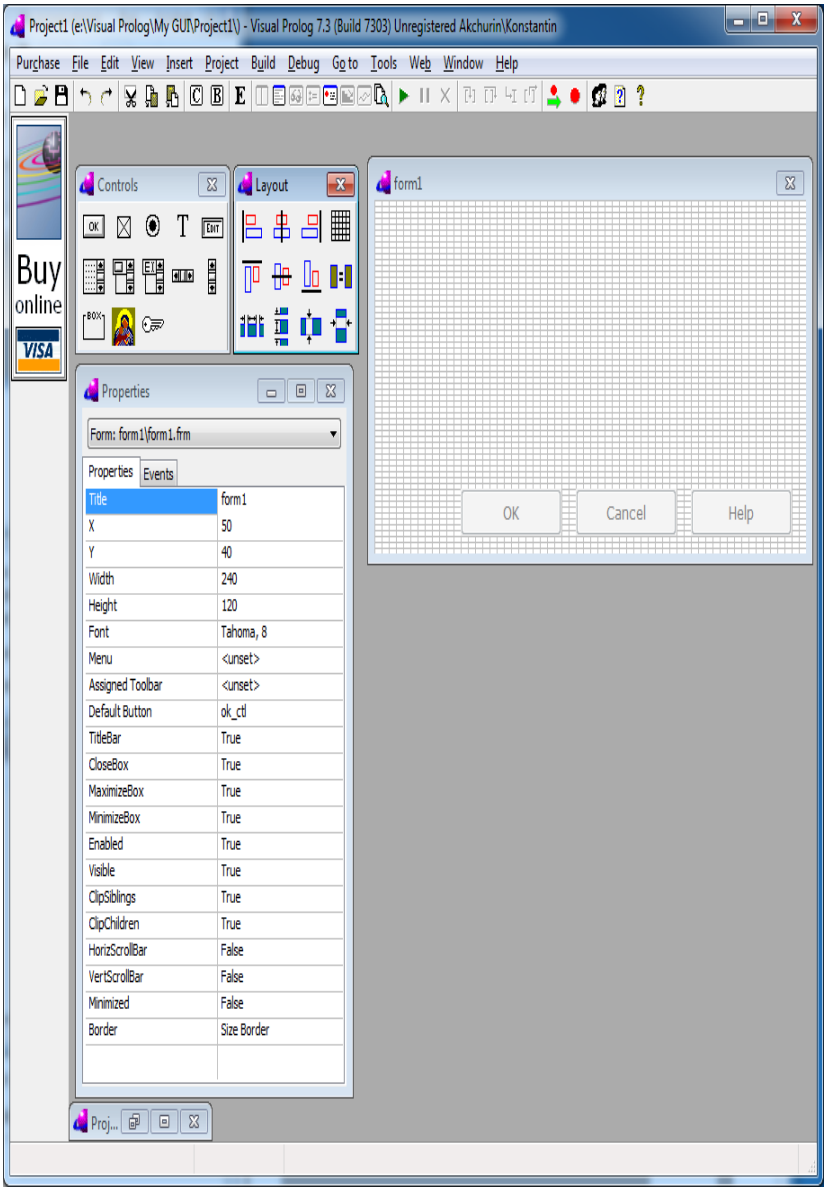
- Package – встроенный пакет,
- Class – класс,
- Interface – интерфейс,
- Dialog – диалог,
- Form – форма,
- Control – элемент управления,
- Toolbar – панель инструментов,
- Menu – меню,
- Icon – иконка,
- Cursor - курсор,
- TextFile – текстовый файл.

Например, выбираем форму.

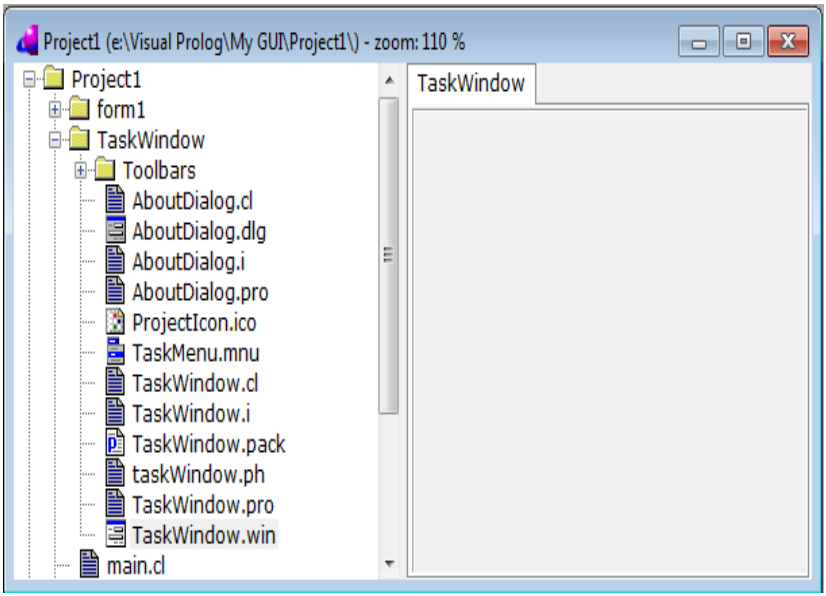


Нажимаем кнопку **create**, генерируется окно конструктора формы, которое содержит встроенные окна:

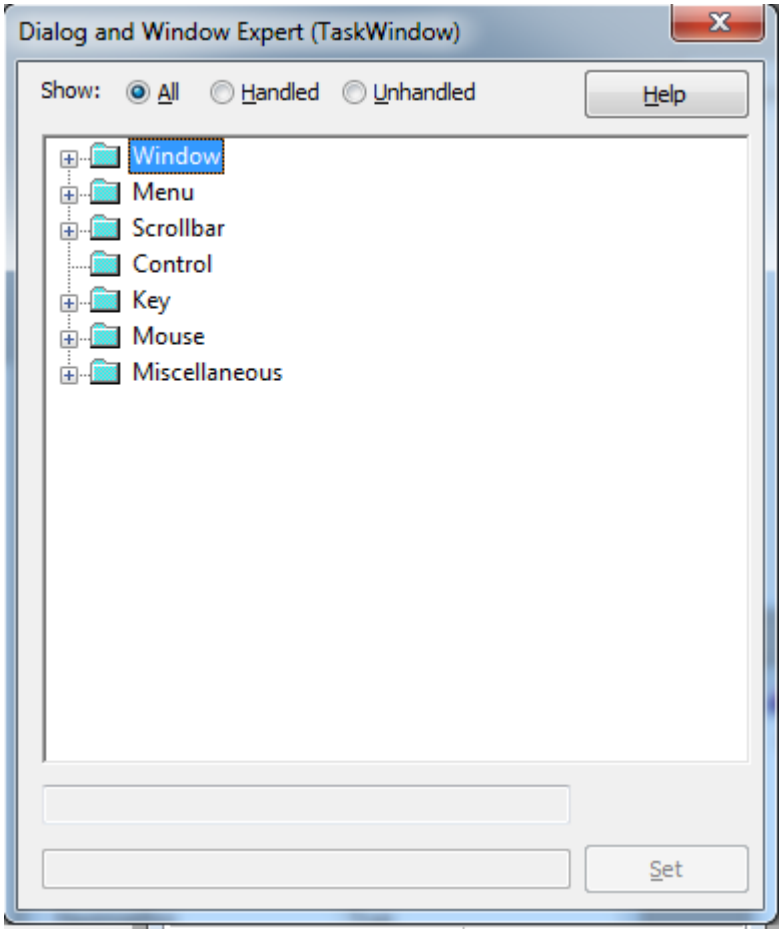
- Form1 -форма,
- Properties – свойства,
- Controls - элементы управления, которые можно использовать в форме (путем переноса),
- Layout - средства управления положением компонент в форме.



Для регулирования поведения окна проекта в дереве проекта нужно вызвать поле TaskWindow.win.



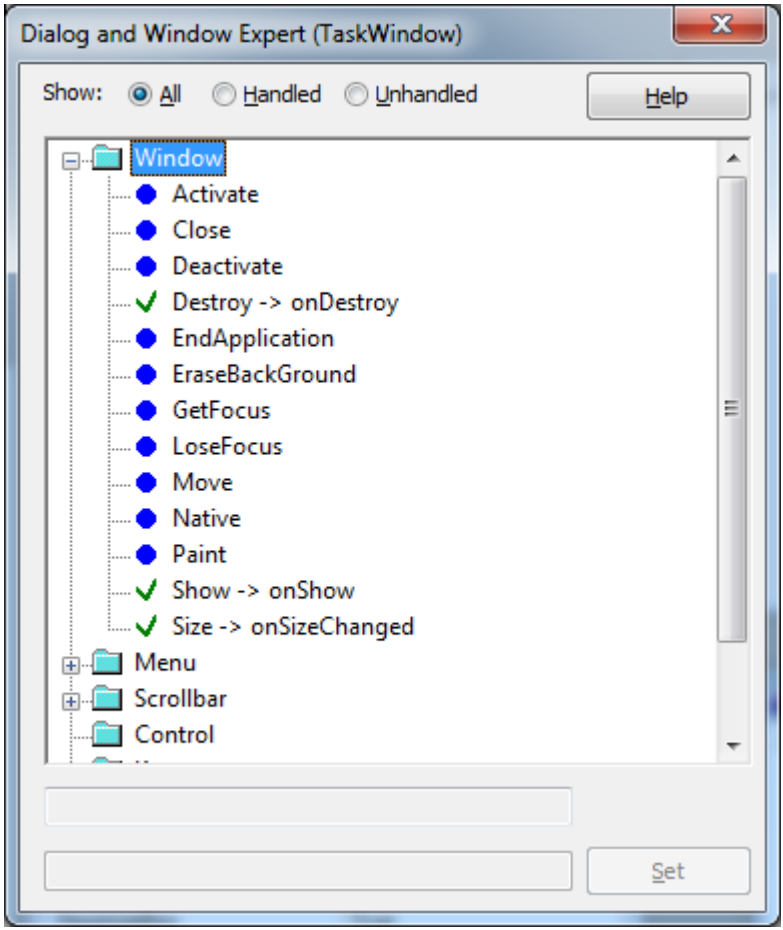
Открывается окно эксперта диалога и окон Dialog and Window Expert (TaskWindow):



В нем нужно изменить установки, которые по умолчанию запрещают некоторые действия.

Для поля Windows нужно сделать доступными:

- Destroy – уничтожение,
- Show – показать,
- Size – размер.

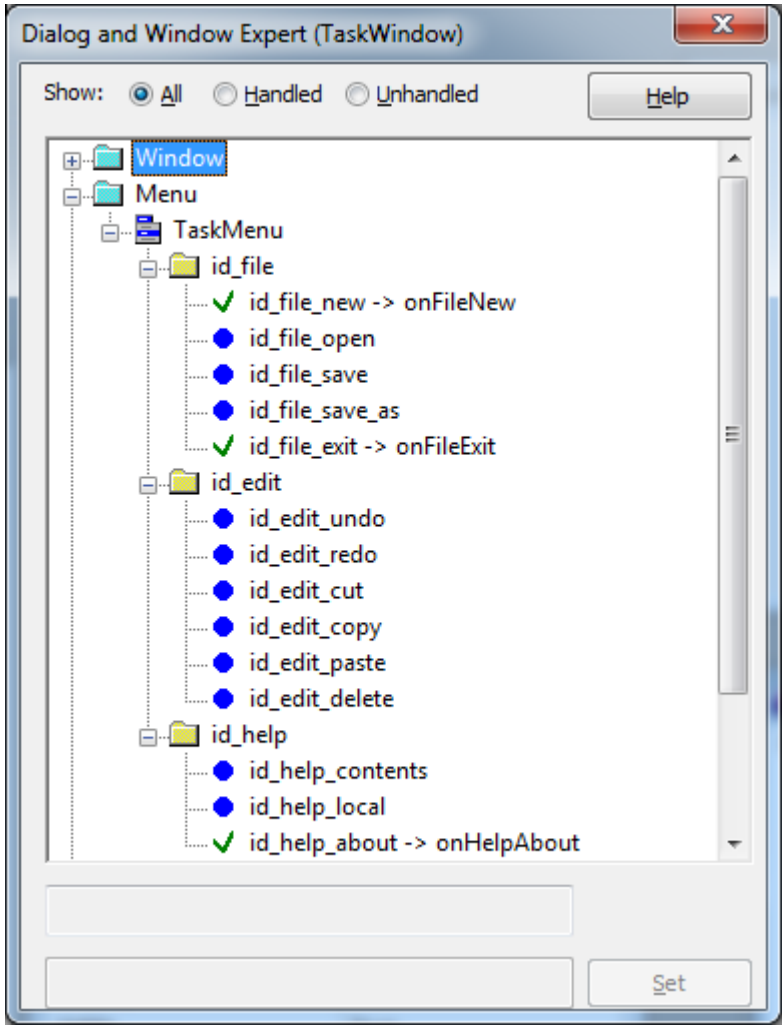


Для поля Menu нужно сделать доступными:

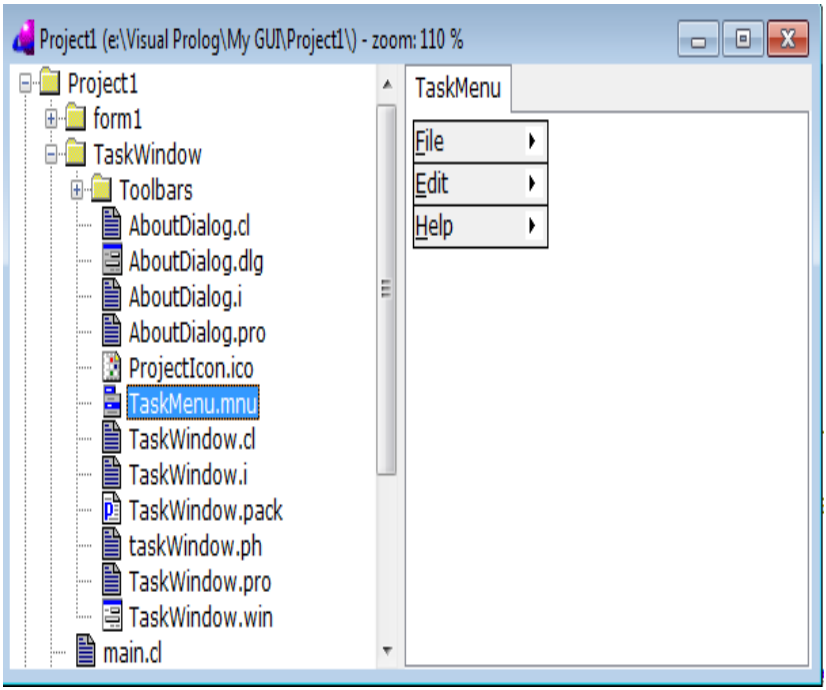
- in_file_new – новый файл открыть,
- in_file_exit – файл закрыть,
- id_help_about – открыть справку.

Остальные поля менять не надо.

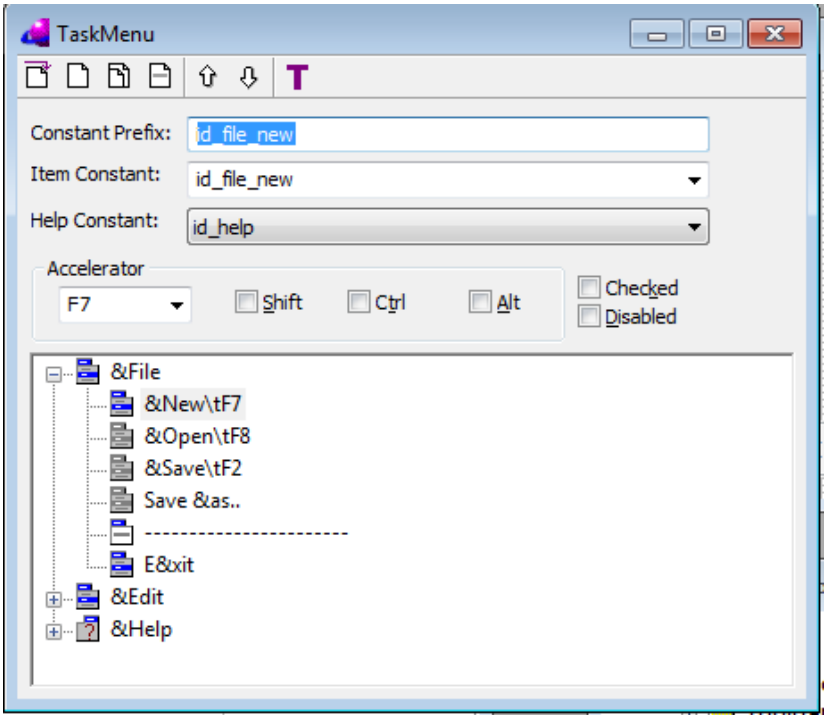
Поля без галочки выключены. Для включения пункта двойной щелчок по нему.



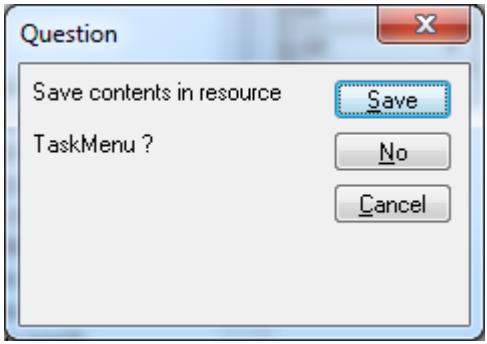
Чтобы открывалось окно формы, нужно разрешить открытие файла формы (по умолчанию это запрещено). Открываем файл TaskMenu двойным щелчком по TaskMenu.mnu.



Открывается окно TaskMenu, в котором для &File\>&New\F7 нужно убрать галочку в Disable

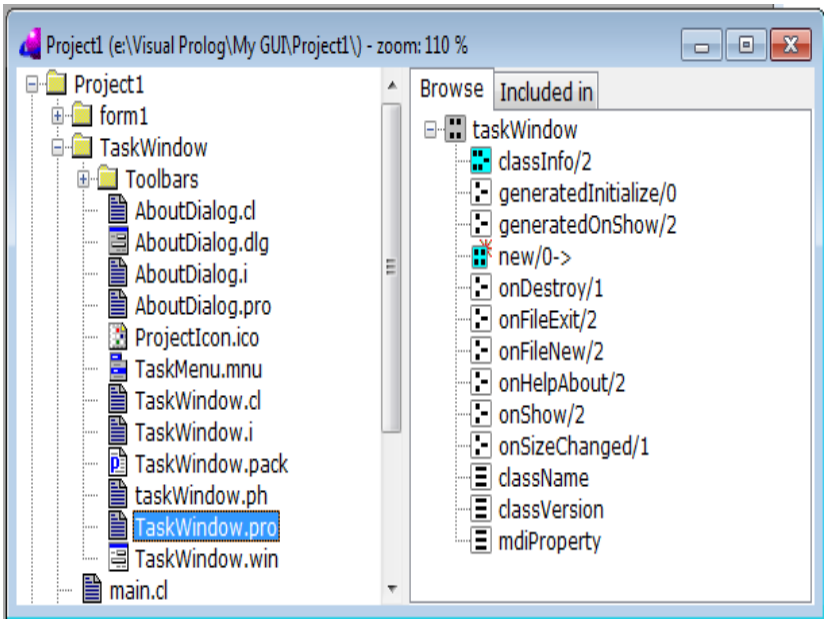


При закрытии этого окна возникает запрос на сохранение сделанных изменений.



Выбираем Save – сохранить.

Осталось прописать ссылку на открываемый файл. Открываем файл TaskWindow,pro двойным щелчком по TaskWindow,pro.



Открывается файл TaskWindow,pro. Его содержимое создано автоматически при компиляции. Только не прописано правило реагирования на открытие нового файла –

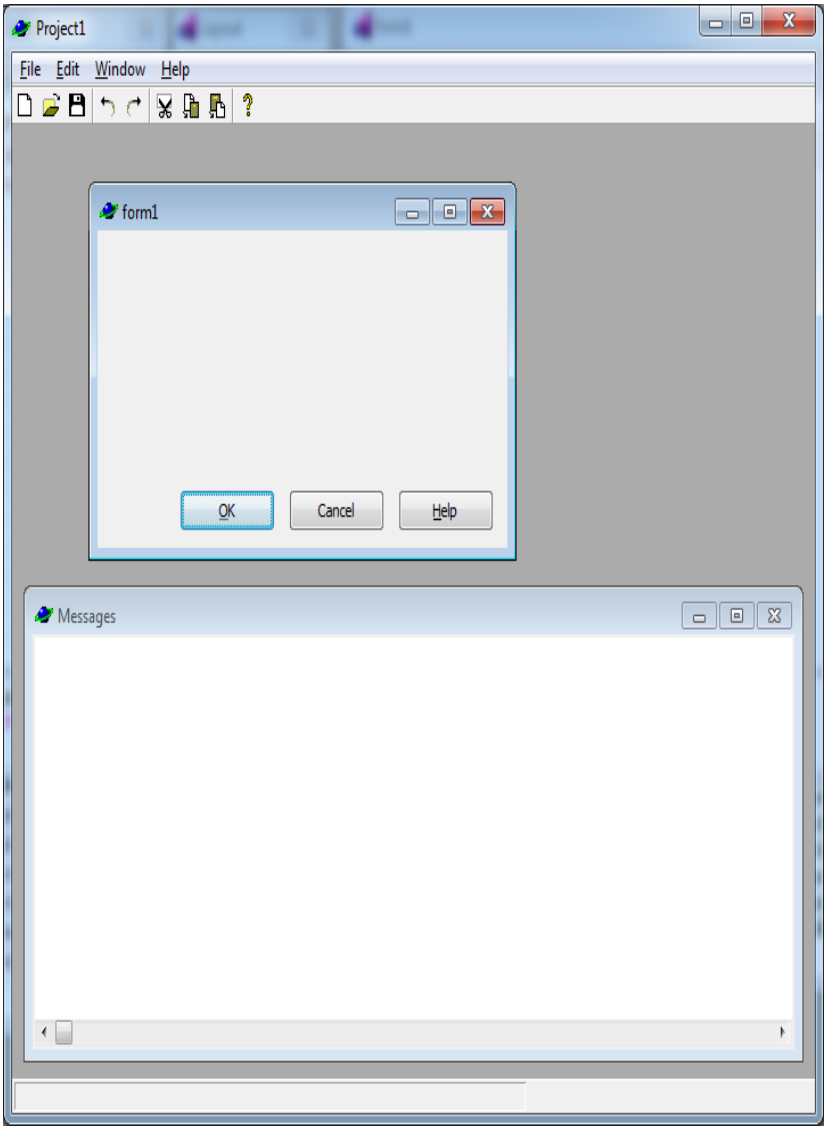
```
clauses
    onFileNew(_Source, MenuTag).
```


TaskWindow.pro (TaskWindow\) - zoom: 110 %
58:1 Insert Indent
predicates
onSizeChanged : window::sizeListener.
clauses
onSizeChanged(_):-
vpiToolbar::resize(getVPIWindow()).
predicates
onFileNew : window::menuItemListener.
clauses
onFileNew(_Source,_MenuTag).
% This code is maintained automatically, do not update it manually. 13:05:16-10.11.2012
predicates
generatedInitialize : ().
clauses
generatedInitialize():-
setText("Project1"),
setDecoration(titlebar([closebutton(),maximizebutton(),minimizebutton()])),
setBorder(sizeBorder()),
setState([wsf_ClipSiblings]),
setMdiProperty(mdiProperty),
menuSet(resMenu(resourceIdentifiers::id_TaskMenu)),
addShowListener(generatedOnShow),
addShowListener(onShow),
addSizeListener(onSizeChanged),
addDestroyListener(onDestroy),
addMenuItemListener(resourceIdentifiers::id_help_about, onHelpAbout),
addMenuItemListener(resourceIdentifiers::id_file_exit, onFileExit),
addMenuItemListener(resourceIdentifiers::id_file_new, onFileNew).
predicates
generatedOnShow: window::showListener.
clauses
generatedOnShow(_,_):-
projectToolbar::create(getVPIWindow()),
statusLine::create(getVPIWindow()),
succeed.
% end of automatic code
Здесь пусто

Вносим туда правило отображения формы form1 с помощью конструктора формы

```
clauses
    onFileNew(S, MenuTag) :-
        X=form1::new(S),X:show().
```

Делаем повторную компиляцию и запуск командой **Build/Execute**. В результате отображается окно проекта без формы. Команда **File/New** в окне проекта приводит к отображению формы в окне проекта.



4.8 Создание DLL в Visual Prolog

Только в коммерческой версии

4.9 Персональная и коммерческая версии

Feature	Personal Edition	Commercial Edition
Commercial Usage		X
Development Environment (IDE)	X	X
Text Editor	X	X
Dialog Editor	X	X
Menu Editor	X	X
Toolbar Editor	X	X
Bitmap/Icon/Cursor Editor	X	X
PFC GUI Support	X	X
Standard GUI Controls, Windows, Dialogs, Forms	X	X
Additional GUI Controls		X
Running External Applications		X
COM Support		X
ActiveX Support		X
Multithread Support	X	X
ODBC Support		X
PIPE Support		X
CGI Support		X
ISAPI Support		X

Time Measuring of Programs for Optimization		X
Basic VPI Library	X	X
Extended VPI Library		X
Regular Expressions Support	X	X
SMAPI Support		X
Help Generating Tool		X
Send Email Support		X
Handle Templates Support	X	X
Collections support	X	X
Mutable variables support	X	X
Cryptography support		X
GDI+ support		X
Common strategy for events processing support		X
Coverage analysis tool		X
Memory profiling analysis tool		X
Source Gear Vault integration support		X
Migration Tool (for migrating Visual Prolog 5.2 projects)	Limited support Download	Download
Examples	X Limited number of exam-ples	X

5 Синтаксис языка Visual Prolog

5.1 Литералы

Алфавит (или множество литер) языка программирования Visual Prolog составляют символы таблицы кодов **unicod**. Литерал представляет собой постоянное значение, у которого нет имени. Например, 5 и "Hello World" являются литералами.

Литералы по назначению делятся на следующие категории:

- IntegerLiteral - для целых чисел,
- CharacterLiteral - символ,
- RealLiteral - для чисел с плавающей запятой,
- StringLiteral - строка,
- BinaryLiteral - двоичные,
- ListLiteral – списки,
- CompoundDomainLiteral - составной литерал:

Литералы для целых чисел

Категория	Литералы
8-ричный префикс	0 или o
8-ричная цифра	одна из 0 1 2 3 4 5 6 7.
Десятичная цифра	одна из 0 1 2 3 4 5 6 7 8 9
16-ичный префикс	0x
16-ричная цифра	одна из 0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f
Унарный плюс:	+
Унарный минус:	-

Целочисленный литерал может принадлежать к **integer** или неподписанным доменам, и он не должен превышать максимального и минимального целого или значения без знака.

Литералы для вещественных чисел

Формат числа:

<Знак><Целая часть>.<Дробная часть><E**><Знак порядка><Порядок>**

Категория	Литералы
Знак числа	один из: + или -. Плюс можно опускать
Целая часть мантиссы	набор из 0 1 2 3 4 5 6 7 8 9
Разделитель частей в мантиссе	Точка
Дробная часть мантиссы	набор из 0 1 2 3 4 5 6 7 8 9

Разделитель	один из e или E
Знак порядка	один из + или -. Плюс можно опускать
Порядок	одна из 0 1 2 3 4 5 6 7 8 9

Литерал не должен превышать максимального и минимального значений.

Символьные Литералы

Это любой печатаемый символ или escape-последовательность (это символ \ и символ управления принтером):

Категория	Литералы
Символ \	\\
Символ табуляции Tab	**\t**
Символ новой строки NewLine	**\n**
Символ возврата каретки return	**\r**
Символ одиночной кавычки	\'
Символ двойной кавычки	\"
XXXX точно 4 символа Unicode	**\u*XXXX***

Строковые Литералы

Категория	Литералы
Строка	”любые символы”
Строка без escape	@”любые символы кроме escape”

Двоичные Литералы

Используется формат:

$**[** Значение **]**

Значение должно быть любым арифметическим выражениеем, возвращающим целое число в диапазоне от до 255.

Списочные Литералы

Список содержит в квадратных скобках компоненты, разделенные запятыми.

Все литералы для компонент списка должны принадлежать к одному домену (или совместимым доменам). Этот домен может быть любого пользователя или встроенным, например, это может integral, character, binary, compound domain и т.д.

Для поддержки иерархического дерева списков определены 2 формата:

- [ОдиночныйЛитерал, СписочныйЛитерал]. Все элементы списка перечислены с разделением запятыми.

- [ОдиночныйЛитерал | СписочныйЛитерал]. ОдиночныйЛитерал – голова списка, СписочныйЛитерал – хвост списка.

Примеры допустимых литералов списков:

```
[]                          % пустой список,
[1,2,3]                     % список целых чисел,
[“abc”, “выбираются”]  % список строк
```

Пример с ошибкой, потому что все элементы в списке должны быть одного и того же типа:

```
[1, “abc”]      % это недопустимый список, разные типы.
```

5.2 Лексические элементы

Алфавит Visual Prolog служит для построения слов, которые называются лексемами.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Используются для задания имен объектов. Могут включать символы, знак подчеркивания и цифры.

Замечание. Контролируется только первый символ.

Категории имен:

- Начинаются с **маленькой** буквы. Это:
  - Константы.
  - Домены (типы).
  - Интерфейсы.
  - Факты.
  - Предикаты.
- Начинаются с **заглавной** буквы. Это имена **переменных**.
- Символ подчеркивания (_). Это имя анонимной переменной.

Для придания именам смысла применяется многословная форма с выделением начала слова заглавной буквой. Например, моя Программа.

В типах Visual Prolog используется иерархическая схема именования с разделителем двойное вертикальное двоеточие (::). При таком подходе связанные типы группируются в пространства имен, что упрощает их поиск и создание ссылок.

Первая часть полного имени — это имя класса. Последняя часть имени — это имя члена класса. Например,

```
console::read()
```

представляет собой инструкцию чтения с консоли read(), которая принадлежит классу console.

Различают типы лексем:

- идентификаторы;
- ключевые слова;
- знаки (символы) операций;
- литералы;
- знаки пунктуации
- разделители.

5.2.1 Идентификаторы

Определены типы:

- Последовательность букв, цифр и символов подчеркивания, которая **начинается с маленькой буквы**.
- Последовательность из букв, цифр и символов подчеркивания, которая **начинается с большой буквы или с символа подчеркивания**.
- **Анонимный**, это символ подчеркивания (_).
- **Ellipsis** - это тройное многоточие (,,,).

5.2.2 Комментарии

Это фрагменты, с помощью которых в файл исходника вводятся пояснения. Компилятор их игнорирует. В Visual Prolog определены комментарии:

- % - комментарий до конца строки.
- /* - символы начала многострочного комментария.
- */ - символы конца многострочного комментария.

5.2.3 Ключевые слова

Это зарезервированные слова, их не может применять пользователь.

Разделены на 2 группы:

Главные:

- class clauses constants constructors
- delegate domains

- end
- facts
- goal guard
- implement inherits interface
- monitor
- namespace
- open
- predicates
- properties
- resolve
- supports

Вспомогательные:

- align and anyflow as
- bitsize
- catch
- determ digits div do
- else elseif erroneous externally
- failure finally foreach from
- if
- language
- mod multi
- nondeterm
- or
- procedure
- quot
- rem
- single
- then to try

Все ключевые слова за исключением **as** и **language** являются зарезервированными словами.

Обратите внимание, что элемент **div** и **mod** также являются зарезервированными словами, но они классифицируются как операторы.

guards и **monitor** не используется в языке, но зарезервированы для будущего использования.

5.2.4 Знаки пунктуации

Они для компилятора имеют синтаксический и семантический смысл. Некоторые могут использоваться как операторы.

Знак	Название	Назначение
;	Точка с запятой	ИЛИ
!	Восклицательный знак	Удаление точек возврата.
,	Запятая	В предложениях означает И. В списках разделяет члены.
.	Точка	Завершает факт или предложение
#		Для директив компилятора.
[ ]	Квадратные скобки	А них список членов, разделение запятыми
|	Вертикальная черта	Символ унификации. В списке заменяет запятую.
( )	Круглые скобки	В них список аргументов, разделенных запятыми.
:-	Вертикальное двоеточие и тире	В предложениях для разделения имени и перечня действий
:	Вертикальное двоеточие	В определениях предикатов.
::	Двойное вертикальное двоеточие	Разделитель имени класса и поля класса

5.3 Классы

Декларация класса определяет внешний вид класса в окружениях: окружения могут увидеть и использовать те сущности, которые упомянуты в Декларации класса. Мы говорим, что Декларация класса описывает открытую часть класса. Декларация класса может содержать определения констант, доменов и Декларации предикатов. Если класс типа конструктора (**ConstructionType**), то он создает объекты этого типа. Классы этого типа имеют по крайней мере один конструктор, но их можно объявить и больше. Классы, в которых явно не объявлен конструктор, автоматически оснащаются конструктором по умолчанию **new/0**.

Объекты строятся путем вызова одного из конструкторов класса. Конструкторы используются также при инициализации унаследованных классов. Все упомянутое в Декларации класса принадлежит классу, но не объектам, которые он создает. Все, что относится к объектам, должно быть объявлено в типе объектов, построенных с помощью класса.

Любая Декларация класса должна иметь сопровождающую реализацию класса. Декларация/реализация предикатов, объявленных в классе, осуществляется в реализации класса. Аналогичным образом определение предикатов, поддерживаемых объектами, построенными классом, осуществляется в реализации класса. Оба вида предикатов могут быть реализованы с помощью предложений (**clauses**), но объекты предикатов также могут наследоваться от других классов.

Важно заметить, что в Декларации класса не указывается что-нибудь о наследуемом коде. Наследуемый код является полностью частным, что может быть только в реализации класса. Это отличие от многих других объектно-ориентированных языков программирования, оно позволяет скрыть все детали реализации.

5.4 Реализации

Реализация класса используется для предоставления определений предикатов и конструкторов, объявленных в декларации класса, а также определения любого предиката, поддерживаемого построенными объектами.

Класс можно объявить частным (то есть внутри реализации) и определить больше сущностей, чем те, которые упомянуты в Декларации. Важно, что реализация может объявить базу данных фактов, которые могут использоваться для перенесения класса и состояния объекта. Реализация является смешанной сферой, в том смысле, что она содержит реализацию и класса, и объектов, производимых в классе. Часть класса в классе совместно используется во всех объектах данного класса, в отличие от части объекта, которая является индивидуальной для каждого объекта. Как часть класса, так и часть объекта могут содержать факты и предикаты, тогда как домены, функторы и константы всегда принадлежат части класса, то есть они не принадлежат отдельным объектам.

По умолчанию все предикаты и факты, объявленные в реализации, являются объектами. Чтобы объявить членов класса в разделе (например, предикаты и факты), нужно добавить префикс с ключевым словом **class**. Все члены, объявленные в таких разделах, являются членами класса. Члены класса могут ссылаться на часть класса в классе, но не на часть объекта. Члены объекта, с другой стороны, могут получить доступ как к части класса в классе, так и к части объекта в классе. В коде реализации объект владельца поддерживается всеми объектными предикатами. Доступ к объекту владельца можно также получить непосредственно через специальную переменную «**This**».

5.5 Конструирование объекта

Раздел описывает конструирование объекта, как такового, рассмотрены только классы, которые производят объекты. Объекты строятся путем вызова конструктора. Конструкторы объявляются явно в секциях **constructors** (конструкторы) классов объявления и реализации (см. также конструктор по умолчанию). Конструктор на самом деле имеет два связанных предиката:

- Предикат класса - функция, которая возвращает новый сконструированный объект.
- Предикат объекта, который используется при инициализации унаследованных объектов.

Связанный предикат объекта используется для инициализации объекта. Этот предикат может вызываться только из конструктора в самом классе и от конструктора в классе, который наследуется от класса (т.е. Инициализация базового класса). Функция связанного класса определяется неявно, то есть не существует предложений для нее явно. Функция класса выделяет память для хранения объекта, выполнения внутренней инициализации объекта и затем вызывает конструктор объекта на созданный объект. Наконец, построенный объект возвращается, как результат работы конструктора.

Итак, перед предложениями конструктора должно быть сделано:

- все переменные объекта facts, которые имеют выражение инициализации, инициализируются,
- все объекты facts, которые имеют предложения, инициализируются из этих предложений.

Эта инициализация осуществляется также на всех (транзитивно) унаследованных дочерних объектах перед вызовом предложений конструктора. Конструктор предложений должен:

- инициализировать все одиночные объекты facts и переменные объектов facts, которые не инициализированы перед входом,
- инициализировать все подобъекты.

Конструктор предложений может делать также другие вещи, но он должен выполнить инициализацию, упомянутую здесь, чтобы обеспечить, что объект правилен после создания.

Примечание. В ходе создания объекты могут оказаться неправильными, программисту нужно позаботиться о том, чтобы не было доступа к неинициализированным частям объекта (смотрите правила для создания объектов).

Конструктор по умолчанию — 0-арный конструктор с именем **new/0**. Если класс, который создает объекты не объявляет конструктор в объявлении класса, конструктор по умолчанию (то есть **new/0**) неявно объявляется (в объявлении класса). Это означает, что каждый класс имеет по крайней мере один конструктор.

Итак запись:

```
class aaa
end class aaa
```

эквивалентна записи

```
class aaa
    constructors
        new : ().
end class aaa
```

Легально явно повторно объявлять конструктор по умолчанию. Не обязательно определять (т.е. реализовывать) конструктор по умолчанию. Если он не определен, то неявное определение подразумевается.

Итак запись:

```
implement aaa
end implement aaa
```

эквивалентна записи

```
implement aaa
    clauses
        new().
end implement aaa
```

учитывая, что **aaa** имеет конструктор по умолчанию.

Обратите внимание, что класс имеет конструктор по умолчанию, если и только если:

- явно не объявлен конструктор вообще;
- или объявлен **new/0** как конструктор.

Пример**.** Легально для реализации использовать неявно объявленный в классе aa_class конструктор по умолчанию:

```
clauses
    new() :-
    ...
```

```
end implement
```

Пример**.** Класс **bb_class** явно объявляет конструктор, который не является конструктором по умолчанию. Впоследствии класс не имеет конструктора по умолчанию.

```
class bb_class : aa
    constructors
        newFromFile : (file File).
end class
```

Пример**.** Класс **cc_class** объявляет конструктор **newFromFile/1**, но он также объявляет **new/0** конструктор по умолчанию. Поэтому очевидно он имеет конструктор по умолчанию **new/0.**

```
class cc_class : aa
    constructors
        new : ().                   % конструктор по умолчанию
        newFromFile : (file File).  % объявление конструктора
end class
```

5.5.1 Частные конструкторы

Также можно в реализации классов объявить «частные» конструкторы. Это может быть разумным, например, в следующих ситуациях:

- Когда некоторый предикат возвращает объект типа конструктора, затем в классе реализации можно объявить, реализовать и вызвать «частный» конструктор для создания таких объектов.
- Когда класс объявляет несколько «public» конструкторов, имеющих одинаковую «большую часть», то разумно определить в реализации класса «частный» конструктор, который реализует эту «большую часть». Затем предложения всех этих «public» конструкторов просто вызывают этот «частный» конструктор для реализации этой «большей части».

Обратите внимание, что если класс, который может строить объекты, не объявляет конструкторы в объявлении класса, то конструктор по умолчанию **new/0** будет объявлен неявно независимо от того объявляет ли реализации класса «частные» конструкторы.

5.5.2 Создание подобъектов

Все конструкторы отвечают за инициализацию построенных объектов правильными состояниями. Для получения правильных состояний и все подобъекты (то есть унаследованные классы) должны быть инициализированы правильно. Под-объекты могут быть инициализированы одним из двух способов:

- программист вызывает конструктор унаследованного класса,
- автоматически вызывается конструктор по умолчанию.

В последнем случае требуется, чтобы унаследованный класс фактически имел конструктор по умолчанию, но нет разницы, как объявлен этот конструктор - явно или неявно, он будет вызываться в обоих случаях!

Если унаследованный класс не имеет конструктора по умолчанию, то другой конструктор должен вызываться явно. По умолчанию вызов конструкторов унаследованных классов происходит сразу же после инициализации переменных и предложений фактов перед входом к предложениям конструктора.

Конструкторы унаследованных классов вызываются версией, которая не возвращает значений. Если вы вызываете версию, которая возвращает значение, то вы на самом деле создаете новый объект, а не вызываете конструктор на «**This**» (см. пример ниже).

Пример 1 Данная реализация класса bb_class наследует от класса aa_class и конструктор по умолчанию из bb_class вызывает конструктор aa_class с вновь созданным объектом cc_class.

```
implement bb_class inherits aa_class
clauses
    new() :-
        C = cc_class::new(),    % создан объект cc_class
        aa_class::newC(C).      % вызов конструктора подобъекта
        ...
end implement
```

Пример 2**.** Если базовый класс не создан явно, то он неявно построен с использованием конструктора по умолчанию.

Запись:

```
implement bbb
    inherits aaa
clauses
    myNew() :-
        doSomething().
end implement bbb
```

Это то же самое:

```
implement bbb
    inherits aaa
clauses
```

```
    myNew() :-
        aaa::new(),
        doSomething().
end implement bbb
```

Если **aaa** не имеет конструктора по умолчанию, то это даст сообщение об ошибке.

5.5.3 Инициализация одноместного объекта

Все конструкторы для инициализации/создания подобъектов должны инициализировать все факты одноместного объекта, прежде чем они впервые упоминаются.

Обратите внимание, что один класс фактов может быть инициализирован только с предложениями, поскольку они не связаны с объектом. Факт класса должен быть доступен до создания первого объекта.

Пример**.** Этот пример показывает:

- как инициализировать переменную факта с помощью выражения;
- как инициализировать один факт (точка) посредством предложения (clause);
- как инициализировать один факт в конструкторе,
- где вызывается конструктор по умолчанию унаследованного класса:

```
implement bb_class inherits aa_class
    facts
        counter : integer := 0.
        point : (integer X, integer Y) single.
        c : (cc C) single.
    clauses
        point(0, 1).
% Объект создан, факты counter и point инициализированы до входа
% Конструктор по умолчанию aa_class::new/0 также вызывается до входа
        new() :-
        C = cc_class::new(),
        assert(c(C)).
    ...
end implement
```

5.5.4 Конструирование делегированием

В качестве альтернативы для создания объекта непосредственно в конструкторе можно делегировать работу другого конструктора того же класса. Это делается просто путем вызова другого конструктора (который не возвращает

значения). При конструировании делегированием мы должны быть уверены, что объект на самом деле создан и что он не «перестроенный». Одноместным фактам может быть присвоено значение столько раз, как вы любите, поэтому они не могут быть «перестроенными». Наследуемые классы, с другой стороны, могут быть инициализированы только один раз во время создания объекта.

Пример**.** Показывает типовое использование конструирования делегированием. Конструктор **new/0** вызывает другой конструктор **newFromC/1** со значением по умолчанию.

```
implement aa_class
facts
    c : (cc C) single.
clauses
    new() :-
        C = cc_class::new(),
    newFromC(C).
    newFromC(C) :-
    assert(c(C)).
    ...
end implement
```

5.5.5 Правила конструирования объектов

Программист должен обеспечить следующее:

- все подобъекты инициализированы и построены ровно один раз каждый.
- все одноместные факты инициализированы (по крайней мере один раз).
- нет ссылки на подобъекты, пока они не инициализированы и построены.
- нет использования одноместных фактов, пока они не инициализированы.

Нет никаких гарантий, что умный компилятор выявит такие проблемы во время компиляции. Компилятор может предложить создать программу проверки, он может также **не безопасно** предложить пропустить такую проверку.

5.5.6 Объект This

Предикат объекта всегда ссылается на объект. Этот объект использует факты объекта и включен в реализацию объекта предикатов. Предикат объекта имеет доступ к этому неявному объекту. Мы будем называть этот объект **This**. Существует два вида доступа к This, явный и неявный:

- Явный This. В каждом предложении каждого объекта предикат переменной This неявно определен и привязан к This, то есть объекту, предикат которого выполняется.

- Неявный This. В предложении предиката объекта другие предикаты объекта могут быть вызван непосредственно, потому что This неявно подразумевается для операции. Члены super-classes также могут вызываться напрямую, поскольку однозначно, какой метод вызывается. Аналогичным образом можно получить доступ к фактам объекта (которые хранятся в This).

This и наследование. This всегда ссылается на объект, принадлежащему к классу, в котором This используется, также если этот класс наследуется другим классом.

5.6 Типы

В Visual Prolog типы делятся на типы объектов и типы значений.

- Объекты имеют изменяемое состояние.
- Значения являются неизменяемыми.

Типы объектов определяются определениями интерфейса.

Типы значений включают числовые типы, строки, типы символов и составные домены. Составные домены известны как типы алгебраических данных. Простыми формами составного домена являются структура и перечисление, тогда как более сложные формы представляют древовидные структуры.

Подтипы

Типы организованы в иерархии подтипов. Подтипы используются для внедрения полиморфизма: любой контекст, который ожидает значение некоторого типа, одинаково хорошо будет принимать значение любого подтипа. Или для обратного направления, мы можем сказать, что значения определенного типа автоматически преобразуются в любой супертип там, где это необходимо, и таким образом может использоваться как имеющий супертипа без явного преобразования типов.

Подтипы могут порождаться из любого типа значений, за исключением типов алгебраических данных. Типы, производные от алгебраических данных типов это скорее синонимы, а не подтипы, т.е. они того же типа. Понятие подтипы тесно связано с понятием подмножеств. Однако важно заметить, что несмотря на то, что тип «математически» является подмножеством другого типа, ему не обязательно быть подтипом. Тип является подтипом другого типа, только если он объявляется именно так.

Пример.

```
domains
```

```
t1 = [1..17].
t2 = [5..13].
t3 = t1 [5..13].
```

t1 составной тип, его значения целые числа от 1 до 17. Аналогично **t2** содержит значения от 5 до 13. Так что **t2** является подмножеством **t1**, но **t2** не является подтипом типа **t1**. С другой стороны **t3** (который содержит те же значения, что и **t2**) является подтипом типа **t1**, потому что он объявлен таковым.

Язык содержит несколько неявных отношений подтипов, но явные отношения оговариваются в определении подтипа. Типы объектов организованы в иерархии подтипов, корень которой в предопределенном объекте, т.е. любой тип объекта является подтипом типа объекта. Объектные подтипы определяются с помощью определений, что один интерфейс поддерживает другой. Если объект тип объекта интерфейса, который поддерживает некоторые другие интерфейс, то объект также имеет этот тип и можно без дальнейшего вмешательства использовать его как такой объект.

5.7 Объектная система

5.7.1 Внешний вид

Концепция класса в Visual Prolog основывается на следующих трех семантических сущностях:

- объекты,
- интерфейсы,
- классы.

Объект. Это набор именованных объектов предикатов и набор поддерживаемых интерфейсов.

Объекты на самом деле также имеют состояние, но это состояние можно только изменить и наблюдать через члена предикатов. Мы говорим, что состояние инкапсулируется в объекте. Инкапсуляция означает способность объекта скрыть свои внутренние данные и методы, делая программно доступными только части объекта. Хорошо известна важность инкапсуляции и модульности. Инкапсуляция помогает построению более структурированных и читаемых программ, потому что объекты рассматриваются как черные ящики. Посмотрите на сложную проблему, найдите часть, которую можно объявить и описать. Инкапсулируйте ее в объект, постройте интерфейс и продолжайте это, пока вы не объявите все подзадачи. Когда вы имеете инкапсулированные объекты этой проблемы и обеспечили их правильную работу, можно использовать их.

Интерфейс. Интерфейс является объектным типом. Он имеет имя и определяет набор именованных объектных предикатов. Интерфейсы структурированы в иерархии средств поддержки (структура подобна решетке. ее корень в объекте интерфейса). Если объект имеет тип, объявленный как интерфейс, он также имеет тип любых поддерживаемых интерфейсов. Таким образом, поддерживает иерархия средств поддержки — это также иерархия типов. Интерфейс является подтипом всех поддерживаемых интерфейсов.

Можно также сказать, что объект поддерживает интерфейс. Если интерфейс называется X, то мы говорим, что объект - X, или X объект.

Класс. Класс — именованный родитель объектов. Он может создавать объекты, соответствующие определенным интерфейсам. Любой объект создается с помощью класса. Если объект был создан классом, который использует интерфейс C для создания объектов, то мы называем это «C объект». Все объекты, которые построены по определению класса, имеют то же определение предикатов члена объекта, но каждый объект имеет свое собственное состояние. Таким образом, предикаты объекта фактически является частью класса, тогда как состояние объекта является частью самого объекта.

Класс также содержит другой набор именованных предикатов и инкапсулированных состояний, известных как члены класса и состояния класса, соответственно. Члены и состояния класса существуют на основе класса, тогда как члены и состояния объекта существуют на основе объекта. Состояния класса доступны как членам класса, так и членам объекта.

Обратите внимание! Набор предикатов объекта, которые определяет класс, является объединением предикатов, объявленных (транзитивно) и в интерфейсах этого класса. Более конкретно это означает, что если предикат объявлен в двух различных интерфейсах, то класс будет предоставлять только одно определение этого предиката. Таким образом, предполагаемые семантики этих двух унаследованных предикатов совпадают. Обратите внимание, что поддержка интерфейса должен быть указана явно. Тот факт, что некоторые классы предоставляет предикаты соответствующего интерфейса, не означает, что класс поддерживает интерфейс.

Модуль. Это класс, которому не нужно быть способным производить объекты. Такой класс может иметь только члены и состояние класса.

Идентичность. Каждый объект уникален: объекты имеют сменные состояния и поскольку состояние объектов можно наблюдать с помощью своего предиката, объект идентичен только себе. Т.е. даже если состояния два объектов идентичны, сами объекты не идентичны, потому что мы можем изменить состояние одного объекта, не изменяя состояние другого объекта.

Мы никогда не имеем прямого доступа к состоянию объекта, но мы всегда имеем доступ к состоянию объекта с помощью ссылки на объект и хотя объект идентичен только себе, мы можем иметь много ссылок на тот же объект. Таким образом, один и тот же объект доступен через много различные ссылки. Классы и интерфейсы также являются уникальными; они идентифицируются по их именам. Два интерфейса или класса не могут иметь то же имя в одной и той же программе. Класс и интерфейс могут иметь одинаковые имена, если класс конструирует объекты этого интерфейса. Суть заключается в том, что структурное равенство не предполагает идентичности объектов, классов, интерфейсов.

5.7.2 Внутренний вид

В предыдущем разделе были описаны объекты, классы и интерфейсы, с точки зрения их внешнего поведения. Этот раздел будет расширять это описание внутренними вопросами. Внутренние вопросы имеют более программный характер; они связаны с разделением классов на части declaration (Декларации) и implementation (Реализации). С программной точки зрения классы являются центральной темой: код содержится в классах.

Интерфейсы имеют главным образом статическое значение. В самом деле, интерфейсы существуют только в текстовом представлении программы, не существует программного представления (прямого) интерфейса.

Объекты, с другой стороны, имеют главным образом динамическое значение. Объекты не видны непосредственно в программе; они не существуют до тех пор, пока программа действительно работает. Класс состоит из декларации и реализации. В Декларации провозглашается общественности доступные части класса и объектов, которые она порождает. С другой стороны, реализация определяет сущности, объявленные в объявлении класса. Базовая реализация предикатов находится в предложениях, но предикаты могут также быть определены с помощью наследования или с использованием внешних библиотек.

Объявление класса в Visual Prolog является чисто декларативным. Он только определяет, к каким сущностям вы можете получить доступ, но не как или где они осуществляются. Реализация класса можно объявить и определить дальнейшие сущностей (например, домены, предикаты, и т.д.), которые видны только внутри самого класса. Т.е. они являются частными.

Состояние объекта хранится в объекте, как его факты. Эти факты объявляются как нормальный раздел фактов (база данных) в реализации класса. Факты являются локальными для каждого объекта (как и других объектов сущностей), тогда как факты класса являются общими для всех объектов данного класса. Факты могут быть объявлены только в реализации класса и, таким образом, не

могут быть доступны (непосредственно) вне класса. Реализация класса может также объявить, что она поддерживает большее интерфейсов, чем упомянуто в Декларации. Эта информация является, однако, видимой только в самой реализации и, следовательно, частная.

Наследование кода. В Visual Prolog наследование кода происходит только в реализации класса. Visual Prolog имеет **множественное** наследование. Наследовать от класса можно, упомянув класс в специальной секции наследования **inherits**.

Классы, которые наследуют от родительских классов, называют дочерними. Дочернему классу доступен его родительский класс только через общий интерфейс, то есть он не получает каких-либо дополнительных привилегий, чем кто-нибудь еще, которые используют этот родительский класс.

5.7.3 Обзор и видимость

Категории имен. Все имена (идентификаторы) в Visual Prolog синтаксически делятся на две основные группы:

- имена констант (начинаются с буквы в **нижнем** регистре),
- имена переменных (начинаются с буквы в **верхнем** регистре или знака подчеркивания).

Имена констант (идентификаторы) делятся на следующие категории:

- Имена типов (то есть домены и интерфейсы).
- Имена доменов переносчиков (т.е. классы и интерфейсы).
- Имена без скобок (например, константы и переменные фактов).
- Имена арности N, возвращающие значение (т.е. функции, функторы и переменные фактов функционального типа).
- Имена арности N, не возвращающие значение (то есть, предикаты, факты и переменные фактов типа предиката).

Visual Prolog требует, чтобы имена не конфликтовали при объявлении, потому что будет невозможно решить конфликт в точке использования. Объявления могут вызвать конфликт, только если они находятся в одной и той же области видимости, поскольку квалификация с областью может быть использован для решения конфликта.

Пакеты. Базовый модуль организации кода в Visual Prolog — пакет. Мы используем пакеты в Visual Prolog для организации и структурирования. Использование пакетов обеспечивает однородность в принципах структурирования среди различных проектов. Пакеты определяют стандарт для инструментов структурирования и легкость совместного использование исходного кода среди

проектов. Пакет представляет собой совокупность нескольких сгруппированных интерфейсов и классов. Пакет предоставляет некоторое общее имя для всех этих интерфейсов и классов. Каждая декларация или реадизация каждого интерфейса или класса из пакета помещается в отдельный файл. Каждое имя файла (из этих файлов) совпадает с именем класса или интерфейса, которое объявлено или реализовано в этом файле. Все файлы пакета хранятся в той же директории отдельный пакет. Если пакет содержит sub-packages, то они помещаются в подкаталогах каталога пакета. Концепция пакетов используется для объединения нескольких связанных интерфейсов и классов.

Пакеты могут играть роль некоторых библиотеки классов. Пакеты можно использовать в вашей программе вместо прямого помещения всех используемые интерфейсов и классов в вашу программу. Принятые в Visual Prolog структура пакетов и как пакеты должны быть включены в проекты, описаны в VDE части этой помощи.

Видимость, скрытие и квалификации. Большинство правил обзора уже упоминалось выше. Этот раздел будет завершать картину. Определение интерфейса, объявление класса и реализации класса являются областями обзора (они не могут быть вложенными). Реализация (часто) расширяет области обзора для объявления класса. Видимость - везде в области. Это означает, что независимо от того, где в области что-то объявлено, оно видно во всей области обзора.

Общедоступные имена из поддерживаемых интерфейсов и super-classes доступны внутри области обзора прямо (т.е. без квалификации), если однозначно, откуда они берутся. Нельзя использовать имя, происхождение которого является неоднозначным. Все неясности в вызове предиката могут быть удалены, если квалифицировать имя предиката с именем класса (например, cc::p). Квалификация также используется для вызова объекта предиката super-classes для текущего объекта.

Visual Prolog имеет следующие скрытые иерархии: локальные, суперкласса и открытые области обзора.

Иерархия означает, что локальное объявление будет скрывать декларацию суперкласса. Но нет никаких сокрытий между суперклассами (все имеют одинаковые предпочтения). Если два или более суперкласса содержат противоречивые декларации, то к ним можно получить доступ только посредством квалификации.

Пример. Предположим интерфейс aa и класс aa_class:

```
interface aa
    predicates
```

```
        p1 : () procedure ().
        p2 : () procedure ().
        p3 : () procedure ().
end interface

class aa_class : aa
end class
```

Также предположим класс **bb_class**:

```
class bb_class
    predicates
        p3 : () procedure ().
        p4 : () procedure ().
    end class bb_class
```

В контексте этих классов рассмотрим реализацию класса cc_class

```
implement cc_class inherits aa_class
    open bb_class
        predicates
            p2 : () procedure ().
            p5 : () procedure ().
    clauses
        new() :-
            p1(), % aa_class::p1 не виден
            p2(), % cc::p2 (скрывает aa_class::p2)
            aa_class::p2(), % aa_class::p2 виден
            p3(), % неправильный вызов: aa_class::p3 or bb_class::p3
            aa_class::p3(), % aa_class::p3 виден
            bb_class::p3(), % bb_class::p3
            p4(), % bb_class::p4
            p5(), % cc::p5
end implement cc_class
```

5.8 Компиляция

В Visual Prolog программа исполняется путем запуска ***.exe** файла, полученного путем компиляции построенного проекта.

5.8.1 Единицы компиляции

Программа состоит из целого ряда единиц компиляции. Компилятор компилирует каждую из этих единиц компиляции отдельно. Результатом компиляции является объектный файл единицы

Эти объектные файлы (и возможно другие файлы) компонуются вместе, чтобы произвести цель проекта. Программа должна содержать 1 раздел **goal**, который является точкой входа в программу.

Единица компиляции должна быть автономной в том смысле, что все ссылочные имена должны быть объявлены или определены в ней.

Определения интерфейса и декларации классов могут быть включены в несколько единиц компиляции (они должны быть идентичными во всех единицах компиляции, где они включены), тогда как реализация класса (определения) определена только в одной единице компиляции.

Каждая объявленная тема должна быть также определена в проекте, но некоторые темы могут быть определены в библиотеке. Это означает, что они не нуждаются в текстовом определении. Единица компиляции (которая возможно производится с помощью директивы компилятора **#include**) представляет собой последовательность из тем компиляции.

Тема компиляции – это:

- директива компилятора - Directive,
- пространство состояния - NamespaceEntrance,
- интерфейс - InterfaceDefinition,
- объявление класса - ClassDeclaration,
- реализация класса - ClassImplementation ,
- цель - Goal,
- элемент условной компиляции - ConditionalItem.

5.8.2 Директивы компилятора

Программа состоит из целого ряда единиц компиляции. Компилятор компилирует каждую из этих единиц отдельно. Результатом компиляции является объектный файл. Эти объектные файлы (и возможно другие файлы) связаны вместе, чтобы произвести цель проекта. Программа должна содержать ровно один раздел goal, который является точкой входа в программу. Единица компиляции должна быть автономной в том смысле, что все ссылки на имена должны быть объявлены или определены в единице компиляции.

Определения интерфейса и объявления класса могут быть включены в несколько единиц компиляции (объявления определения должны быть идентичными во всех местах, где они включены), тогда как реализация класса (определения) могут быть определены только в одной единице. Каждый объявленный элемент должен быть также определен в проекте, но некоторые элементы могут быть определены в библиотеках. Это означает, что они не нуждаются в текстовом определении.

Единица компиляции (которая возможно состоит из нескольких элементов, объединяемых с помощью директивы #include) представляет собой последовательность тем компиляции.

Тема компиляции - это интерфейс, объявление класса, реализацию класса, цель секции, элемент условной компиляции, которые описаны в условной компиляции.

Директивы:

- NamespaceEntrance.
- ConditionalItem.
- InterfaceDefinition.
- ClassDeclaration.
- ClassImplementation.
- GoalSection.

5.9 Интерфейсы

Определение интерфейса определяет тип именованного объекта. Интерфейсы могут поддерживать другие интерфейсы. Все предикаты, объявленные в интерфейсе, являются членами объекта типа интерфейс. Интерфейс также является глобальной областью, в которой могут быть определены константы и домены. Таким образом, константы и домены, определенные в интерфейсе, не являются частью типа, который обозначает интерфейс (или объектов, имеющих этот тип). На такие домены и константы можно ссылаться из других областей квалификации с именем интерфейса: interface::constant, или с использованием открытых квалификаций. (См. открытые квалификации).

Объявление интерфейса:

```
interface Имя интерфейса (малыми буквами)
    область обзора
    разделы
конец интерфейса
```

Имя_интерфейса в конце разработки должно (если присутствует) быть идентичным в начале разработки.

Интерфейс object

Если интерфейс явным образом не поддерживает интерфейсы, то он неявно поддерживает интерфейсы, встроенные в объекты. Объект - пустой интерфейс, то есть он не содержит предикаты и т.д. Назначение объекта –это универсальный базовый тип всех объектов.

5.10 Квалификации

Задают область обзора. На предикаты из заданной области обзора не нужно явно в коде прописывать место порождения .

5.10.1 Квалификация open

Квалификация open (открыть) используется для более удобной ссылки на уровне класса сущности. Раздел open приносит имена одной области в другую область, так что на них можно ссылаться без квалификации.

Раздел **open** не влияет на имена членов объектов, они могут быть доступны только с помощью объекта в любом случае. Но имена членов класса, домены, функторы и константы могут быть доступны без квалификации. Когда имена вводятся в область обзора таким образом, может случиться, что некоторые имена становятся неоднозначными. Раздел open имеет эффект только в области, в которой они использован. Особенно это означает, что раздел open в объявлении класса не влияет на реализацию класса.

5.10.2 Квалификация supports

Может использоваться только в interfaceDefinition и classImplementation. Используются две вещи:

- указание что один интерфейс A расширяет другой интерфейс B и, таким образом, что объект типа A является подтипом объекта типа B
- объявление, что объекты определенные как «приватные» имеют больше типов объектов, чем те, который указан типа конструируемого

Поддерживает эти транзитивные отношения: Если интерфейс A поддерживает интерфейс B и B в свою очередь поддерживает C, то A также поддерживает C. Если интерфейс явным образом не поддерживает интерфейсы, то он неявно поддерживает объект предопределенного интерфейса.

5.10.3 Наследуемая квалификация

Наследуемая квалификация используются, чтобы заявить, что реализация наследует от одного или нескольких классов. Наследование влияет только на объектную часть класса.

Наследование от других классов предназначено для наследования поведения от этих классов. Когда класс **cc** наследует от класса **aa**, это означает, что реализация **cc** класса автоматически неявно (частно) поддерживает тип конструирования (интерфейс) из класса **aa**. (Если реализация класса **aa** уже поддерживает тип конструирования **cc** явно, то разницы нет.

Поэтому, обратите внимание, что одинаковые предикаты, например **p**, не могут быть объявлены в конструкторах типа интерфейса класса **cc** и унаследованного класса **aa**. Компилятор обнаружит это и сгенерирует ошибку объявления предиката в двух местах.

Давайте обсудим это в деталях. Предположим, что класс **cc** имеет конструктор типа интерфейса **cci** и некоторый другой класс **aa** имеет конструктор типа интерфейса **aai**. Пусть оба **aai** и **cci** интерфейса объявляют один и тот же предикат **p**. Пока **aa** и **cc** классы являются независимыми, компилятор не может обнаружить каких-либо проблем. Но как только мы заявляем, что **cc** класс наследует от класса **aa**, то класс **cc** начинает также поддерживать **aai** интерфейс. Таким образом, класс **cc** начинает видеть в обеих декларациях предикат **p**, будет сообщаться об ошибке времени компиляции. Единственная возможность избежать такой двусмысленности в предикате **p** - использование предикатов из раздела интерфейса в объявлении интерфейса **cci**. Например, как:

```
interface cci
    predicates from aai
    p(), ...
end interface cci
```

Предикат объекта может быть унаследован: Если класс не реализует определения предикатов объектов, но один из классов, которые он наследует, реализует этот предикат, то этот предикат будет использоваться для текущего класса. Класс, который наследует от другого класса, не имеет каких-либо особых привилегий к унаследованному классу: он может получить доступ к внедренному объекту только через конструктора интерфейса.

Наследования должны быть однозначными. Если класс определяет предикат сам, то нет никакой двусмысленности, потому что это тот предикат, который наблюдается. Если наследуется только один класс, который поддерживает предикат, то это также однозначно. Но если два или более классов поддерживают предикат, то это неоднозначно для класса, который предоставляет определение. В этом случае двусмысленность должна разрешаться с помощью квалификации разрешимости **resolve**.

Объектные предикаты от унаследованных классов могут вызываться напрямую из объектного предиката в текущем классе, поскольку внедренный подобъект неявно используется в качестве владельца предиката. Класс квалификации может использоваться для разрешения неясности вызова для объектных предикатов от унаследованных классов.

Формат:

inherits ClassName список наследуемого

5.10.4 Квалификация резольвент

Квалификация **resolve** используется для разрешения выполнения предиката из указанного источника.

Как уже отмечалось, все неясности, связанные с вызовом предиката, можно избежать с помощью полных имен. Но когда речь заходит о наследовании, это не так. Рассмотрим следующий пример:

```
interface aa
predicates
    p : () procedure ().
    ...
end interface

class bb_class : aa
end class

class cc_class : aa
end class

class dd_class : aa
end class

implement dd_class
    inherits bb_class, cc_class
end implement
```

В этом случае неоднозначно, какой из классов **bb_class** и **cc_class** будет реализовывать **aa** для **dd_class**. Обратите внимание, что когда мы говорим, что класс реализует интерфейс, это означает, что он предоставляет определения для предикатов, объявленных в интерфейсе. Конечно, можно было добавить предложения в реализации dd_class, которые бы эффективно решали задания. Рассмотрим, например, следующее предложение, которое будет «экспортировать» предикат p из bb_class:

```
clauses
    p() :- bb_class::p().
```

Но с этим кодом мы реально не унаследовали поведение от bb, мы на самом деле делегировали работу в часть bb_class нашего класса. Поэтому для решения такого рода неоднозначности, чтобы использовать реальные наследования, вместо делегирования мы используем секцию **resolve** (разрешимость). Формат секции:

resolve Resolution, список резольвент

Секция Resolution содержит ряд резольвент :

- InterfaceResolution – резольвента интерфейса.
- PredicateFromClassResolution – резольвента предиката класса.
- PredicateRenameResolution – резольвента переименования предиката.
- PredicateExternallyResolution – резольвента внешнего предиката.

Резольвента предиката.

Форма предиката

PredicateFromClassResolution:
PredicateNameWithArity из ClassName

Предикат из класса резолюции говорит, что предикат реализуется с помощью указанного класса ClassName. Для разрешения предиката в классе:

- класс должен реализовывать предикат, который будет из класса резолюции, это означает, что предикат должен создаваться в том же интерфейсе.
- класс должен упоминаться в секции **inherits** (наследует).

Резольвента переименования предиката.

Предикат переименовать резольвенту говорит, что она реализована как предикат с другим именем. Предикат должен исходить от унаследованного класса и его тип, режим и поток должны точно совпадать.

PredicateRenameResolution :
PredicateNameWithArity from ClassName :: PredicateName

Interface Resolution. Резольвента интерфейса используется для разрешения полного интерфейса от одного из унаследованных классов. Таким образом, интерфейс резольвент - короткий путь заявления, что все предикаты в интерфейсе должны быть резольвентами из того же класса.

Класс должен публично поддерживать резольвируемый интерфейс. Если резольвенты предикатов и резольвенты интерфейса охватывают некоторое имя предиката, используется резольвент предиката. То есть конкретные резольвенты переопределяют менее специфические. Это действительно для предиката, который покрывается несколькими резольвентами интерфейса, до тех пор, пока это резольвенты предикатов одного и того же класса. Если с другой стороны предикат резольвируется к разным классам резольвент интерфейса, то результат неоднозначен и должен разрешаться резольвентой предиката.

Примечание: Синтаксис резольвент не способен решать различные перегрузки предикатов в различные классы.

Пример**.** Мы можем преодолеть двусмысленность из примера выше, используя резольвенту интерфейса. В этом случае мы решили наследовать реализацию **aa_class** от **cc_class**, за исключением того, что мы будет наследовать **p** из **bb_class**.

```
implement dd_class
    inherits bb_class, cc_class
resolve
    interface aa from cc_class
    p from bb_class
end implement
```

Резольвента внешнего предиката.

Предикат внешней резольвации говорит, что предикат вообще не реализован в самом классе, но он находится во внешней библиотеке. Внешние резольвации можно использовать только для класса предикатов. Т.е. предикаты объектов не могут резольвироваться извне. Важно, что вызовы, имя ссылки и аргумент типов соответствуют их осуществлению в библиотеке. Частные и общедоступные предикаты могут резольвироваться извне.

```
PredicateExternallyResolution : PredicateNameWithArity externally
```

Dynamic External Resolution. Предикат внешней резольвации также предоставляет синтаксис для динамической загрузки предикатов частных и общедоступных классов из библиотек DLL. Используется следующий синтаксис:

```
PredicateExternallyResolutionFromDLL :
PredicateNameWithArity externally from DllNameWithPath
DllNameWithPath :
    StringLiteral % Строка пути к  DLL
```

Если предикат predicateNameWithArity пропущен в пути к DLL DllNameWithPath, то динамическая загрузка обеспечивает возможность запуска программы только тогда, когда она на самом деле вызовет пропущенный предикат. При таком вызове произойдет ошибка времени выполнения. DllNameWithPath - это путь к библиотеке DLL на компьютере, где программа должна запускаться. Он может быть абсолютным или относительным. Например, если требуемая библиотека dll находится на один уровень вверх из каталога, в который приложение загружено, то DllNameWithPath должен быть прописан как «../ DllName». Смотрите также порядок поиска динамической библиотеки в MSDN.

5.10.5 Квалификация делегата

Формат секции

delegate Delegation список делегирования.

Квалификация **DelegateQualification** используется для делегирования реализаций предикатов объектов в указанный источник. Секция **DelegateQualification** содержит ряд делегирований:

- PredicateDelegation - Делегирование предиката.
- InterfaceDelegation - Делегирование интерфейса.

Делегирование интерфейса используется для делегирования реализаций полного набора предикатов объекта, объявленного в интерфейсе, в реализацию другого объекта, хранящегося, как переменная факта. Таким образом, делегация интерфейса является коротким путем объявления, что реализации всех предикатов в интерфейсе следует делегировать в реализацию объекта, который хранится в переменной. Секция **DelegateQualification** выглядит, как секция **resolve** (предикат/интерфейс), за исключением того, что вы делегируете к переменным фактов, сохраняя построенные объекты классов, а не наследуемые классы.

Делегирование предиката.

Объект Predicate Delegation заявляет, что предикат функциональность предиката делегируется в предикат объекта с переменной факта **FactVariable_of_InterfaceType**. Чтобы делегировать предикат для объекта с переменной факта:

- Переменная факта FactVariable_of_InterfaceType должна иметь тип интерфейса (или его подтипом), который объявляет предикат predicateNameWithArity.
- Объект, поддерживающий интерфейс, должен быть построен и присвоен переменной факта FactVariable_of_InterfaceType.

Рассмотрим следующий пример:

```
interface a
predicates
    p1 : ().
    p2 : ().
end interface

interface aa
    supports a
```

```
end interface

class bb_class : a
end class

class cc_class : a
end class

class dd_class : aa
constructors
    new : (a First, a Second).
    end class
implement dd_class

delegate p1/0 to fv1, p2/0 to fv2
facts
    fv1 : a.
    fv2 : a.
clauses
    new(I,J):-
    fv1 := I,
    fv2 := J.
end implement
```

Позже будет возможно построить объекты типа **a** и назначать их переменным фактов **fv1** и **fv2**, чтобы определить, к объектам какого класса мы действительно делегируем определения функциональности **p1** и **p2**.

На самом деле в Visual Prolog делегирование имеет тот же эффект, как если бы вы добавили предложения в реализацию **dd_class**, явно указав, от объекта какого класса функциональность предиката «экспортируется». Например, как если бы следующее положение определялось в реализации **dd_class**:

```
clauses
    p1() :- fv1:p1().
```

Делегирование интерфейса.

Когда вам нужно указать, что функциональность всех предикатов, объявленных в интерфейсе InterfaceName делегируются предикатам объектов того же наследуемого класса, можно использовать InterfaceDelegation.

Таким образом, **InterfaceDelegation** - это короткий путь заявить, что функциональность всех предикатов, объявленных в интерфейсе **InterfaceName,** долж-

на быть делегирована объектам, хранящимся в переменной факта **FactVariable_of_InterfaceType**. Объекты должны быть связаны с переменной факта **FactVariable_of_InterfaceType**, которая должна быть типа **InterfaceName** (или его подтипом). Чтобы делегировать интерфейс к объекту с переменной факта:

- тип переменной факта **FactVariable_of_InterfaceType** должен иметь тип интерфейса **InterfaceName** или его подтипа.
- объект, поддерживающий интерфейс, должен быть построен и присвоен переменной факта **FactVariable_of_InterfaceType**.

Предикат делегирования имеет более высокий приоритет, чем делегирование интерфейса, если предикату указаны оба делегирования. Предикат делегирования задается для предиката и он объявлен в интерфейсе, который имеет интерфейс делегирования. Затем будет осуществляться делегирование предиката с высшим приоритетом.

5.11 Универсальные интерфейсы и классы

Интерфейсы и классы можно параметризировать с параметрами типа, так что они могут использоваться в различных реализациях в различных контекстах. Этот раздел должен рассматриваться как расширение отдельных разделов о:

- Interfaces
- Classes
- Implementations

Прагматическая причина использовать универсальные классы и интерфейсы - объявить параметризированные факты объекта и реализовать операции по этим фактам. Как показано в приведенном ниже примере очереди.

Пример. Очередь (Queue), рассмотрим этот интерфейс

```
interface queue_integer
predicates
    insert : (integer Value).
    tryGet : () -> integer Value.
end interface queue_integer
```

Объект этого типа — это очередь целых чисел. Если вы замените «integer» на «string», то будет тип, описывающий очередь строк.

Универсальный интерфейс может использоваться для описания всех таких интерфейсов в определении единого интерфейса:

```
interface queue{@Elem}
```

```
predicates
    insert : (@Elem Value).
    tryGet : () -> @Elem Value determ.
end interface queue
```

@Elem является переменной типа область (отличается от локального типа переменных префиксом @). Очередь {integer} представляет очередь целых чисел, очередь {string} представляет очередь строк и так далее. Мы можем объявить класс универсальной очереди как:

```
class queueClass{@Elem} : queue{@Elem}
end class queueClass
```

queueClass{@Elem} конструирует объектный тип queue{@Elem} для любого появления @Elem.

Реализация может быть такой::

```
implement queueClass{@Elem}
facts
    queue_fact : (@Elem Value).
clauses
    insert(Value) :-
        assert(queue_fact(Value)).
clauses
    tryGet() = Value :-
        retract(queue_fact(Value)),
    !.
end implement queueClass
```

Этот фрагмент кода показывает, как создавать очередь целых чисел и вставить элемент в нее:

```
...,
Q = queueClass{integer}::new(),
Q:insert(17)
,...
```

Не нужно применять тип явно, вместо этого компилятор может вывести его из контекста:

```
...,
Q = queueClass::new(),
Q:insert(17),
...
```

Компилятор видит, что Q должно быть очередью целых чисел, потому что мы вставляем 17 в нее.

5.11.1 Универсальные интерфейсы

Синтаксис. Универсальный интерфейс имеет список параметров типа:

InterfaceDeclaration :
    interface InterfaceName { ScopeTypeVariable-**comma-sep-list-opt** }
ScopeQualifications Sections
    ...
    ScopeTypeParameter :
    @ UppercaseName

Параметры типа области могут использоваться в любом декларации или определении в интерфейсе.

Семантика. Универсальный интерфейс определяет все интерфейсы, которые могут быть получены путем создания экземпляра параметра фактического типа. Параметры типа области и открытия связаны в интерфейсе.

Ограничения. Имя закрытия не должно иметь параметров:

```
interface xxx{@A}
    ...
end interface xxx % здесь нет параметров
```

Незаконно использовать одинаковое имя интерфейса для интерфейсов с различными арностями (в одном и том же пространстве имен):

```
interface xxx % xxx/0
    ...
end interface xxx

interface xxx{@A, @B} % ошибка: несколько классав, интерфейсов и/или пространств имен имеют одинаковое имя  'xxx'
    ...
end interface xxx
```

Параметры могут использоваться в поддерживаемых интерфейсах:

```
interface xxx{@P} supports yyy{@P} % легально, @P связан
    ...
end interface xxx
interface xxx supports yyy{@P} % нелегально, @P не связан
    ...
end interface xxx
```

Поддерживаемые интерфейсы могут создаваться с помощью любого типа выражений (покуда связаны параметры):

interface xxx{@A} supports yyy{integer, @A*} ...

5.11.2 Универсальные классы

Синтаксис. Универсальные классы имеют список параметров типа и конструируют объекты типа интерфейса, которые используют эти параметры.

```
ClassDeclaration :
class ClassName { ScopeTypeVariable список} % список не обязателен
    ConstructionType
    ScopeQualifications
    ...
    ScopeTypeParameter :
    @ UppercaseName
    ConstructionType :
        : TypeExpression
```

Тип конструктора должен быть универсальным (то есть это универсальный интерфейс).

Семантика. Универсальный класс объявляет класс с универсальным конструктором. Тип построенного объекта будет выведен из использованного конструктора.

Ограничения. Имя закрытия имя не должно иметь параметров.

```
class xxx{@A} : xxx{@AAA}
    ...
end class xxx % здесь нет параметров
```

Незаконно использовать одинаковое имя интерфейса для интерфейсов с различными арностями (в одном и том же пространстве имен):

```
class xxx % xxx/0
    ...
end class xxx
class xxx{@A} : yyy{@A} % ошибка: несколько классов, интерфейсов и/или пространств имен имеют одинаковое имя 'xxx'
    ...
end class xxx
```

Если класс и интерфейс имеют одно и то же имя, то класс должен построить объекты этого интерфейса.

```
interface xxx{@A}
    ...
end interface xxx
class xxx{@Q, @P} : object % ошибка: несколько классов, интерфейсов и/или пространств имен имеют имя 'xxx'
    ...
end class xxx
```

Параметры в конструкторе типа и т.д. должны быть связаны:

```
class bbb : xxx{@Q} % ошибка: несколько параметров в '@Q' использованы в расширении типа
```

Все параметры класса должны быть использованы в типе конструктора:

```
class xxx{@P} : object % ошибка: не использован параметр '@P'
...
```

В объявления класса параметр обзора может использоваться только в конструкторах, доменах и константах.

```
class xxx{@P} : xxx{@P}
domains
    list = @P*.             % легально
constants
    empty : @P* = [].       % легально
constructors
    new : (@P Init).        % легально
predicates
    p : (@P X).       % ошибка: параметр '@P' использован в появлении класса
end class xxx
```

5.11.3 Универсальные реализации

Синтаксис.

Универсальные реализации имеют список параметров типа.

```
ClassImplementation :
    implement ClassName { ScopeTypeVariable-comma-sep-list-opt } ScopeQualifications Sections
    ...
    ScopeTypeParameter :
    @ UppercaseName
```

Семантика. Универсальный класс объявляет класс с универсальным конструктором. Тип построенного объекта будет выведен из использованного конструктора.

Ограничения. Имя закрытия имя не должно иметь параметров:

```
implement xxx{@A}
    ...
end implement xxx % здесь нет параметров
```

Параметры должны быть так же, как в соответствующем классе и декларация и иметь тот же порядок.

```
class xxx{@A} : aaa{@A}
    ...
end class xxx

implement xxx{@B} % ошибка: параметр '@B' не такой же как в декларации '@A'
    ...
end implement xxx
```

В реализации классов параметр scope может использоваться в конструкторах, доменах, константы и в сущностях объекта (факты, предикаты и свойства объекта).

```
implement xxx{@P}
domains
    list = @P*.             % легально
constants
    empty : @P* = [].   % I легально
constructors
    new : (@P Init).        % I легально
predicates
    op : (@P X).            % легально
class predicates
    cp : (@P X). % ошибка: параметр '@P' использован в сущности класса
facts
    ofct : (@P). % легально
class facts
    cfct : (@P). % ошибка: Scope параметр '@P' использован в сущности класа
    ...
end implement xxx
```

5.12 Завершение

Как только объект не может быть достигнут программой, он может быть завершен. Семантика языка не указывает точно, когда работа объекта будет завершена. Единственно гарантируется, что он не завершается до тех пор, пока это не указано в коде программы. На практике объект завершается, когда он удаляется сборщиком мусора. Завершение объекта противоположно созданию и будет удалять объект из памяти. Классы также могут реализовывать метод завершения, который является предикатом, который вызывается, когда объект завершается (прежде чем он удаляется из памяти).

Финализатор — процедура без аргументов и не возвращающая значение, имеет имя **finalize**. Предикат неявно объявляется и не может использоваться непосредственно из программы. Основная цель финализатора - возможность освобождать внешние ресурсы, но нет никаких ограничений на то, что она может сделать. Финализаторы однако следует использовать с осторожностью, надо помнить о том, что время их вызова точно не известно и, следовательно, трудно предсказать общее состояние программы, когда они вызываются. Обратите внимание, что у финализатора нет никаких причин для отзыва фактов из объекта, потому что это делается автоматически в процессе работы финализатора. Все объекты финализируются, прежде чем программа может завершится (если не помешает ненормальная ситуация, как сбой питания).

Пример**.** Пример использует финализатор, чтобы правильно закрыть подключение к базе данных.

```
implement aa_class
facts
    connection : databaseConnection.
clauses
    finalize() :-
        connection:close().
    ...
end implement aa_class
```

5.13 Мониторы

Монитор - это конструкция языка для синхронизации двух или более потоков, которые используют общий ресурс, обычно это аппаратные устройства или набор переменных.

Один поток помещает запись данных а монитор, другой берет ее оттуда.

Компилятор прозрачно вставляет код блокирования и разблокирования монитор надлежащим образом для назначенных процедур, вместо того, чтобы программисту вводить коды доступа к параллельным примитивам явным образом.

Входы в монитор Visual Prolog могут контролироваться охранными предикатами (условий).

Синтаксис. Монитор интерфейсов и классов – области обзора:

```
Scope : one of
    ...
    MonitorInterface
    MonitorClass
    ...
```

Интерфейс монитора определяется путем написания ключевого слова **monitor** перед определением регулярного интерфейса:

```
MonitorInterface :
    monitor IntertfaceDefinition
MonitorClass :
    monitor ClassDeclaration
```

Монитор классов и интерфейсов не может объявить **multi** и **nondeterm** предикаты.

Ограничения.

- Регулярный интерфейс не может поддержать интерфейс монитора.
- Монитор класса не может построить объекты.
- Не законно наследовать от монитора (т.е. от класса, реализующего интерфейс монитора).

Семантика. Предикаты и свойства, объявленные в мониторе, являются входами в монитор. Поток входит в монитор через вход и остается там до нового входного потока. Только один поток может находиться в мониторе в заданное время, каждая запись охраняется как критический регион. Семантика упрощена, чтобы понять преобразование программы. Рассмотрим этот учебный пример:

```
monitor class mmmm
predicates
    e1 : (a1 A1).
    e2 : (a2 A2).
    ...
    en : (an An).
```

```
end class mmmm
%----------------------------------------
implement mmmm
clauses
    e1(A1) :- <B1>.
clauses
    e2(A2) :- <B2>.
    ...
clauses
    en(An) :- <Bn>.
end implement mmmm
```

Где <B1>, <B2>,..., <Bn> - тела предложений.

Приведенный код соответствует следующему «нормальному» коду:

```
class mmmm
predicates
    e1 : (a1 A1).
    e2 : (a2 A2).
    ...
    en : (an An).
end class mmmm

implement mmmm
class facts
    monitorRegion : mutex := mutex::create(false).
clauses
    e1(A1) :-
        _W = monitorRegion:wait(),
    try
        <B1>
        finally
        monitorRegion:release()
    end try.
clauses
    e2(A2) :-
        _W = monitorRegion:wait(),
    try
        <B2>
        finally
        monitorRegion:release()
        end try.
```

```
    ...
clauses
    en(An) :-
        _W = monitorRegion:wait(),
    try
        <Bn>
        finally
        monitorRegion:release()
    end try.
end implement mmmm
```

Каждый класс **monitor** продлевается с мьютекса, который используется для создания критической области вокруг каждой записи тела предложения.

Код для монитора объектов подобен, за исключением того, что объект mutex принадлежит объекту.

Охранники. Рассмотрим монитор защищенной очереди: некоторые потоки (производители) вставляют элементы в очередь для потребителей (pick-out). Однако, вы не может забрать элемент из очереди, если она пуста.

Если мы реализуем очередь с использованием монитора, то формируется запись «pick-out», но вывод тормозится, если очередь пуста. Потребителям придется делать «опрос» очереди до тех пор, пока запись не будет получена. Такие опросы используют системные ресурсы, и обычно желательно их избежать. Эта проблема может быть решена путем предикатов охранников.

Каждая запись может иметь охранника, связанного с реализацией. Охранник добавляется как специальное предложение перед другими предложениями входа.

```
Clause :                    % один из
    ...
    GuardClause.
    GuardClause : one of
    LowerCaseIdentifier guard LowerCaseIdentifier .
    LowerCaseIdentifier guard AnonymousPredicate .
```

Пример 1. Охранник может быть именем предиката.

```
clauses
    remove guard remove_guard.
    remove() = ...
```

Пример 2. Охранник может быть анонимным предикатом.

```
clauses
    remove guard { :- element_fact(_), ! }.
    remove() = ...
```

Предикаты охранники строятся при создании монитора. Для классов монитора это означает, что при запуске программы объектные предикаты создаются сразу же после создания объекта. Предикаты охранники строятся также всякий раз, когда предикат покидает монитор. Но они не создаются в любое другое время.

Если для определенного охранника соответствующий вход открыт, то он закрывается при ошибке ввода. Можно ввести только открытые записи.

Пример. Класс очереди, который решает проблему pick-out, с помощью предиката охранника на операцию удаления (**remove**).

```
monitor class queue
predicates
    insert : (integer Element).
predicates
    remove : () -> integer Element.
end class queue

implement queue
class facts
    element_fact : (integer Element) nondeterm.
clauses
    insert(Element) :-
        assert(element_fact(Element)).
clauses
    remove guard remove_guard.
    remove() = Element :-
        retract(element_fact(Element)),
    !;
    common_exception::raise_error(common_exception::classInfo,
            predicate_name(), " Охранник создается, когда очередь не пустая ")
predicates
    remove_guard : () determ.
clauses
    remove_guard() :-
        element_fact(_),
    !.
end implement queue
```

Обратите внимание, что **remove** (удалить) является процедурой, поскольку потоки, которые вызывают remove будут ждать, пока есть элемент для них. Предикат охранник **remove_guard** запускается, если в очереди есть элемент. Таким образом, **remove_guard** оценивается каждый раз, когда поток покидает монитор, и база данных фактов **element_fact** может быть изменена только потоком, который находится внутри монитора. Следовательно, роль охранник остается разумной все время (т.е. когда нет потоков в мониторе).

Предикаты охранники обрабатываются в процессе, описанном выше. Пример очереди фактически такой же, как код **«monitor-free»:**

```
class queue
predicates
    insert : (integer Element).
predicates
    remove : () -> integer Element.
end class queue

implement queue
class facts
    monitorRegion : mutex := mutex::create(false).
    remove_guard_event : event := event::create(true, toBoolean(remove_guard())).
    element_fact : (integer Element) nondeterm.
clauses
    insert(Element) :-
        _W = monitorRegion:wait(),
    try
        assert(element_fact(Element))
        finally
        setGuardEvents(),
        monitorRegion:release()
    end try.
clauses
    remove() = Element :-
        _W = syncObject::waitAll([monitorRegion, remove_guard_event]),
    try
        retract(element_fact(Element)),
        !;
        common_exception::raise_error(common_exception::classInfo,
              predicate_name(), " Охранник создается, когда очередь не пустая ")
        finally
            setGuardEvents(),
```

```
            monitorRegion:release()
    end try.
class predicates
    remove_guard : () determ.
clauses
    remove_guard() :-
        element_fact(_),
    !.
class predicates
    setGuardEvents : ().
clauses
    setGuardEvents() :-
        remove_guard_event:setSignaled(toBoolean(remove_guard())).
end implement queue
```

Событие создается для каждого предиката охранника. Это событие имеет значение **signaled** *(*сигнальное), если предикат остается работающим. Как уже упоминалось, он устанавливается во время создания монитора и каждый раз, когда предикат покидает монитор (прежде чем он покидает критическую область).

При вводе записи потоков ожидает как для **monitorRegion**, так и для события охранника, чтобы быть в состоянии сигнальном вообще.

В коде выше инициализации класса самого и охранника события сделаны в неопределенном (**undetermined**) порядке. Этим обеспечивается то, что охранник события инициализируются после выполнения всех других инициализаций объектов класса.

Примеры практического использования этого раздел показывают несколько случаев, когда мониторы удобны.

Запись в файл журнала (Writing **to a log file). Для н**ескольких потоков необходимо записывать информацию в один файл журнала.

```
monitor class log
properties
    logStream : outputStream.
predicates
    write : (...).
end class log

implement log
class facts
```

```
    logStream : outputStream := erroneous.
clauses
    write(...) :-
        logStream:write(time::new():formatShortDate(), ": "),
        logStream:write(...),
        logStream:nl().
end implement log
```

Монитор гарантирует, что записи журнала не смешиваются друг с другом, и что поток изменяет только место между записями журнала.

5.13.1 Общие выходные потоки

Монитор может быть использован для защиты от операций потока вывода:

```
monitor interface outputStream_sync
    supports outputStream
end interface outputStream_sync

class outputStream_sync : outputStream_sync
    constructors
        new : (outputStream Stream).
end class outputStream_sync

implement outputStream_sync
    delegate interface outputStream to stream
    facts
        stream : outputStream.
    clauses
        new(Stream) :- stream := Stream.
end implement outputStream_sync
```

Попробуйте один из этих вариантов или ознакомьтесь с советами по поиску, содержащимися в справке.

```
clauses
    write(...) :-
        logStream:write(time::new():formatShortDate(), ": "),
        logStream:write(...),
        logStream:nl().
```

состоит из трех **отдельных** операций, поэтому она может еще быть **case(fx)** для двух потоков**.** Первый пишет время, второй многоточие «...», и т.д.

5.13.2 Очередь

Очередь выше сделана хорошо, но на самом деле можно еще лучше создать объект очередей. С помощью универсальных интерфейсов мы можем создать весьма общие очереди:

```
monitor interface queue{@Elem}
predicates
    enqueue : (@Elem Value).
predicates
    dequeue : () -> @Elem Value.
end interface queue

class queue{@Elem} : queue{@Elem}
end class queue

implement queue{@Elem}
facts
    value_fact : (@Elem Value).
clauses
    enqueue(V) :-
    assert(value_fact(V)).
clauses
    dequeue guard { value_fact(_), ! }.
    dequeue() = V :-
    value_fact(V),
    !.
    dequeue() = V :-
    common_exception::raise_error(....).
end implement queue
```

Обратите внимание, что PFC уже содержит аналогичный класс monitorQueue.

5.14 Операторы

Синтаксис операторов в Visual Prolog сходен с синтаксисом других языков программирования. Операторы используются для выполнения вычислений, назначения значений, проверки на равенство и неравенство и т. д.

Язык Visual Prolog предоставляет большой набор операторов, которые представляют собой символы, определяющие операции, которые необходимо выполнить с выражением.

Операторы в выражениях исполняются с приоритетами. Высший приоритет имеют основные операторы, далее мультипликативные, затем аддитивные.

5.14.1 Операторы арифметики

Все бинарные, операторы (–) и (+) могут быть унарными. Перечень операторов:

Оператор	Формат	Назначение
+	Z=X+Y	Сложение
-	Z=X-Y	Вычитание
/	Z=X/Y	Деление X на Y
*	Z=X*Y	Умножение X на Y
^	Z=X^Y	X в степени Y
=	Z=X	Присвоение
Операции с целыми числами. Округление в сторону минус бесконечность.		
div	Z=X div Y	Целое при делении. X и Y –целые числа. • -9 / 5 = -1.8 • Целая часть: -1.8=> -2
mod	Z=X mod Y	Остаток при делении. X и Y –целые числа. • -9 / 5 = -1.8 • Целая часть: -1.8=> -2 • Остаток 2*5 – 9 =1
Операции с целыми числами. Округление в сторону нуля.		
quot	Z=X quot Y	Целое при делении. X и Y –целые числа. • -9 / 5 = -1.8 • Целая часть: -1.8 => -1
rem	Z=X rem Y	Остаток при делении. X и Y –целые числа. • -9 / 5 = -1.8 • Целая часть: -1.8=> -1 • Остаток 1*5 – 9 = -4

5.14.2 Логические операторы

Оператор	Пример	Назначение
; or	clauses ppp() :- qqq(V), write(V), fail.	В правиле ppp() проверяется истинность qqq(), rrr().
, and	clauses ppp() :-	В правиле ppp() проверяется истинность qqq(), rrr().

	qqq(), rrr().	Если ppp()=Faulse, то ppp() = Faulse, rrr() не проаеряется. Если ppp()=True, то проверяется rrr(). Если оно True, то ppp()=True, иначе ppp() = False.
not		Нет
!	!	Удаление точек возврата

5.14.3 Операторы отношений

Оператор	Пример	Назначение
=	X=Y ...	X равно Y
<	X<Y	X меньше Y
>	X<Y	X больше Y
<>	Y<X>Z	X больше Y и больше Z
><	Y>X<Z	X меньше Y и больше Z
<=	X<=Y	X меньше или равно Y
>=	X>=Y	X больше или равно Y

5.15 Императивные инструкции

В Visual Prolog кроме декларативных инструкций Пролога добавлены императивные инструкции:

- try catch.
- if then else.
- foreach do.

5.15.1 Инструкция try catch

Ключевые слова **try** и **catch** используются вместе. Если предполагается, что блок кода может вызвать исключение, воспользуйтесь ключевым словом **try**, и используйте **catch**, чтобы сохранить код, который будет выполнен при возникновении исключения. В этом примере в результате деления на ноль создается исключение, которое затем перехватывается. При отсутствии блоков try и catch произойдет сбой программы.

5.15.2 Инструкция if then else

Инструкция **if then else** (если то иначе) предназначена для обработки ветвлений. Она выполняет группы инструкций, вложенных в секции **then** и **else**. Если в группах несколько инструкций, то они разделяются запятыми. Секция **else** может отсутствовать.

Синтаксис:

```
If (Условие)
    then
        Инструкции
    else
        Инструкции
end if.
```

5.15.3 Инструкция foreach

Инструкция **foreach** (для каждого) предназначена для обработки массивов. Она повторяет группу вложенных в нее инструкций для каждого элемента массива.

Синтаксис:

```
foreach Имя списка()
    do
        Инструкции тела цикла
end foreach.
```

Если в теле цикла несколько инструкций, то они разделяются запятыми.

Пример. Вывод в консоль содержимого массива целых чисел.

```
foreach Massiv(Z)
    do
        M= Massiv(Z),
        write(M)
end foreach.
```

5.16 Методы рисования

В Visual Prolog определены методы рисования линий и фигур. Все методы перегружаемые, то есть выполняются по-разному с разными аргументами.

5.16.1 drawPixel

Точка с заданным цветом.

```
drawPixel (Window, pnt, color) language c.
```

Здесь

- Window – где рисуем,
- pnt – точка,
- color – цвет.

5.16.2 drawLine

Прямая линия между двумя точками.

```
drawLine(Window, pnt[1], pnt[2]) language c.
```

Здесь

- Window – где рисуем,
- pnt[1], pnt[2] – точки границы линии, отмечены точками.

5.16.3 drawPolyline

Рисует список прямых линий.

drawPolyline (Window, pnt*[1]) language c.

Здесь

- Window – где рисуем,
- pnt*[1] - список точек.

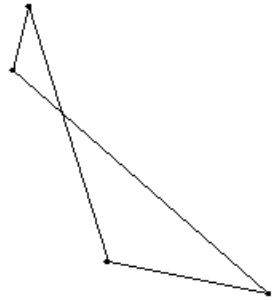
5.16.4 drawPoligon

Полигон. Это многоугольник, формируемый соединением прямыми линиями точек списка. Крайние точки замыкаются. Образуется замкнутая фигура с возможными пересечениями линий.

drawPoligon (Window, pnt*[1]) language c.

Здесь

- Window – где рисуем,
- pnt*[1] - список точек.



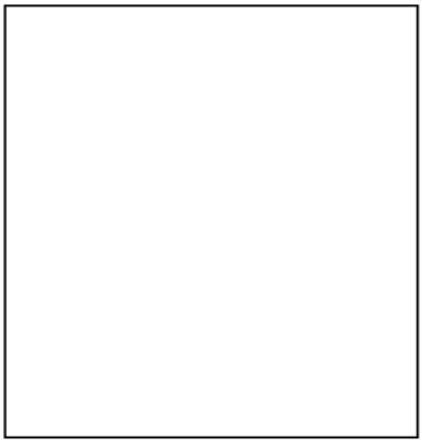
5.16.5 drawRect

Прямоугольник. Синтаксис метода.

drawRect(Window, rect) language c.

Здесь

- Window – где рисуем,
- rect – прямоугольник, свойства которого задаются целыми числами.



5.16.6 drawRoundRrect

Прямоугольник со скругленными улами.

drawRoundRrect(Window, rect, EllipseWidth, EllipseHeight) language c.

Здесь

- Window – где рисуем,
- rect – прямоугольник, свойства которого задаются целыми числами,
- EllipseWidth, EllipseHeight – ширина и высота эллипса, дугами которого делается скругление.

5.16.7 drawFocusRect

Прямоугольник с фокусом Рисует прямоугольник с указанными координатами в стиле,придающем прямоугольнику фокус.

drawFocusRect : (rct `Rectangle`).

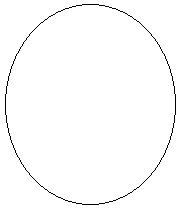
5.16.8 drawEllipse

Эллипс.

drawEllipse : (window, rect) language c.

Здесь

- window – где рисуем,
- rect – прямоугольная область, в которую вписывается эллипс.



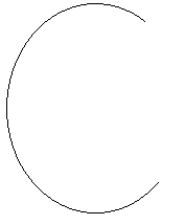
5.16.9 drawArc

Дуга, часть эллипса.

drawArc : (window, rect, StartX, StartY, StopX, StopY) language c.

Здесь

- window – где рисуем,
- rect – прямоугольная область, в которую вписывается эллипс,
- StartX, StartY – приближенная начальная точка, точные значения вычисляются как координаты точки пересечения линии из центра rect к приближенной начальной точке
- StopX, StopY – приближенная конечная точка, точные значения вычисляется как координаты точки пересечения линии из центра rect к приближенной конечной точке



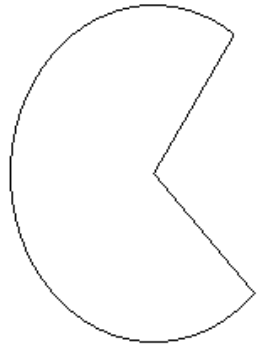
5.16.10 drawPie

Сектор эллипса. От дуги отличается тем, что концы дуги соединяются с центром эллипса радиусами.

```
drawPie : (window, rect, StartX, StartY,  StopX, StopY)  language c.
```

Здесь

- window – где рисуем,
- rect – прямоугольная область, в которую вписывается эллипс,
- StartX, StartY – приближенная начальная точка, точные значения вычисляются как координаты точки пересечения линии из центра rect к приближенной начальной точке
- StopX, StopY – приближенная конечная точка, точные значения вычисляется как координаты точки пересечения линии из центра rect к приближенной конечной точке



5.16.11 drawFloodFill

В Visual Prolog определен метод заливки фигур. Фигура – это замкнутая область, ограниченная линией заданного цвета. Заливка разных фигур осуществляется текущей кистью. Определены разные стиляи заливки.

```
drawFloodFill (Window, pnt, color) language c.
```

Здесь

- Window – где рисуем,
- pnt – стартовая точка,
- color – цвет линии границы области.

5.16.12 drawText

Рисует графическое представление текста в GUI окне, начиная со стартовой точки.

drawText : (pnt, string).

Здесь:

- pnt - стартовая точка в текущем окне.
- string - текст.

5.16.13 drawTextInRect

Рисует графическое представление текста внутри прямоугольника в текущем окне, разбивая текст на строки, длина которых задается в спиике длин.

drawTextInRect : (rct, string,integer*).

Здесь:

- rct – прямоугольник, в котором рисуем текст,
- string,
- integer* - список длин строк.