Эдуардо Коста

Visual Prolog 7.1 для начинающих

Перевод с английского

Eduardo Costa. *Visual Prolog* 7.1 *for Tyros*, 2007

Перевод: И. Алексеев, Е.А. Ефимова, 2008, М. Сафронов, 2007
Редактор перевода: Е.А. Ефимова, 2008

Оформление: М. Сафронов, И. Алексеев

Предисловие

Эта книга начиналась как личный проект. Моим намерением было просто написать руководство по логическому программированию для моего сына. Однако успех книги был огромным, и я получил множество предложений по улучшению текста или кода, а также пожеланий продолжать работу. Письма приходили из Саудовской Аравии, Китая, России, Испании, Франции, Бразилии, Канады и множества других мест. Я благодарен всем, кто проявил интерес к моей работе, особенно же мне хотелось бы поблагодарить следующих людей:

- Елену Ефимову (*Elena Efimova*), которая сделала много исправлений по тексту и по историческим сведениям о математике и логике.

- Марка Сафронова (*Mark Safronov*), который перевёл книгу на русский язык и сделал много исправлений по содержанию оригинала. Между прочим, русский перевод оформлен лучше оригинала.

- Томаса де Бура (*Thomas W. de Boer*), который подготовил параллельную редакцию книги, с более длинными объяснениями и текстом, подходящим для обычных начинающих.

- Стюарта Камминга (*Stuart Cumming*). Часто я привожу примеры, которые показывают то, чего следует избегать с точки зрения методики разработки программного обеспечения. Стюарт указал, что эти примеры могут только запутать читателя. Он предложил мне акцентировать внимание на слабых местах реализации, если та не слишком здравая. Хотя я согласен со Стюартом, я подумал, что будет лучше полностью *убрать* негативные примеры.

- Роуз Шапиро (*Rose Shapiro*), которая исправила мой латинский и описание многих исторических событий.

- Юрия Ильина (*Yuri Ilyin*), который помог мне своим опытом; без него эта книга не была бы написана.

- Рени Кари (*Reny Cury*), которая прошлась по рукописи, исправляя опечатки.

- Филиппа Аполинария (*Philippos Apolinarius*), который помог мне своим знанием ботанической латыни и китайского языка.

- Томаса Линдера Пулса (*Thomas Linder Puls*) и Елизавету Сафро (*Elizabeth Safro*) из PDC за поощрение и поддержку.

Содержание

Часть I
Savoir-faire[1]

[1] Навыки и умения

Глава 1: Введение

1.1. Создание проекта в Visual Prolog

Давайте создадим пустой проект, к которому вы позднее добавите функциональности. Среда, которую вы будете использовать для разработки программ, называется IDE, что является сокращением от ***I**ntegrated **D**evelopment **E**nvironment*[1]. Когда вы заходите в IDE системы Visual Prolog, то попадаете в среду, представленную на рисунке 1.1. Мы будем ссылаться на меню IDE как на «меню задач» (*task menu*). Система окон и диалогов, которую вы создадите для общения с потенциальными пользователями вашей программы, называется ***G**raphical **U**ser **I**nterface*[2], или сокращенно — *GUI*.

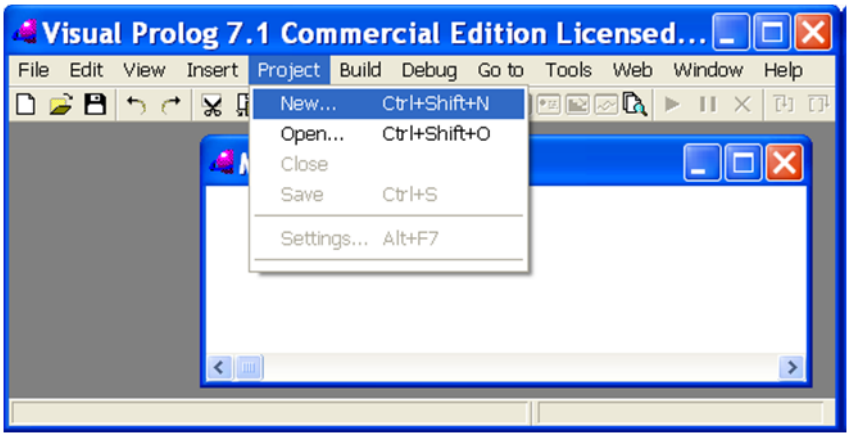


Рисунок 1.1 Создание нового проекта

1.1.1. Создание нового GUI-проекта: *name*

Этот шаг довольно прост. Выберите команду *Project/New* меню задач, как показано на рисунке 1.1. Затем заполните диалоговое окно *Project Settings* так, как показано на рисунке 1.2. Нажмите кнопку *Create*, и перед вами появится окно дерева проекта (см. рис. 1.3).

1.1.2. Компиляция и запуск программы

Для того чтобы скомпилировать программу, выберите команду *Build/Build* меню задач, как показано на рисунке 1.4. Для запуска программы выберите команду *Build/Execute*, и на экране появится окно, изображенное на рисунке 1.5. Всё, что от вас требуется для того, чтобы выйти из программы, — нажать кнопку в виде крестика, которая находится в верхнем правом углу окна. Если предпочитаете, выберите пункт *File/Exit* меню приложения.

[1] Интегрированная среда разработки.
[2] Графический интерфейс пользователя.

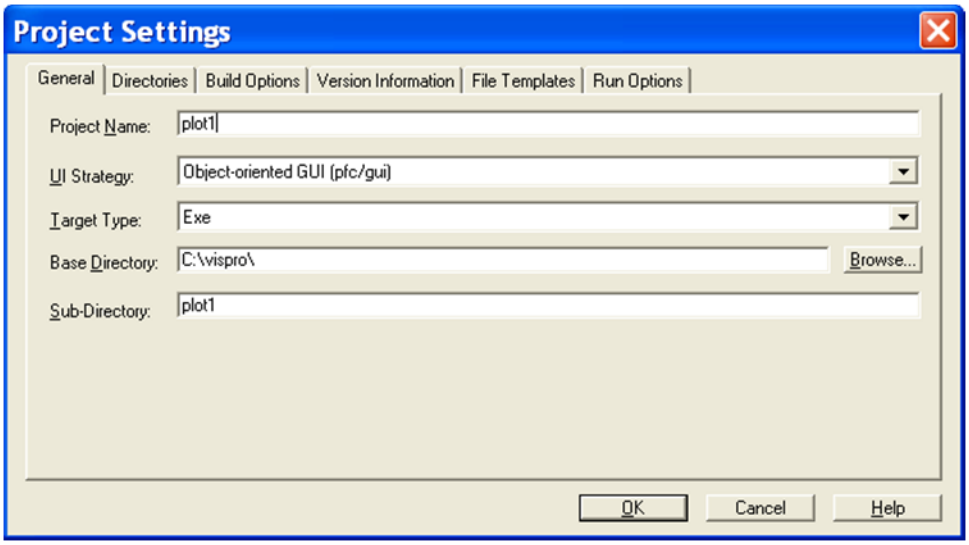


Рисунок 1.2 Настройки проекта

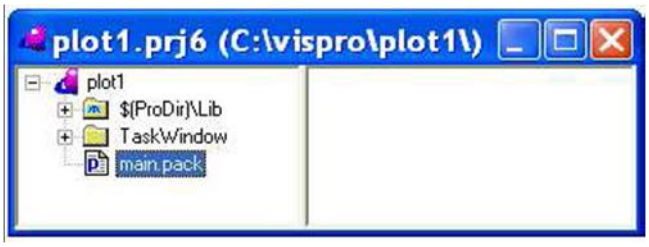


Рисунок 1.3 Дерево проекта

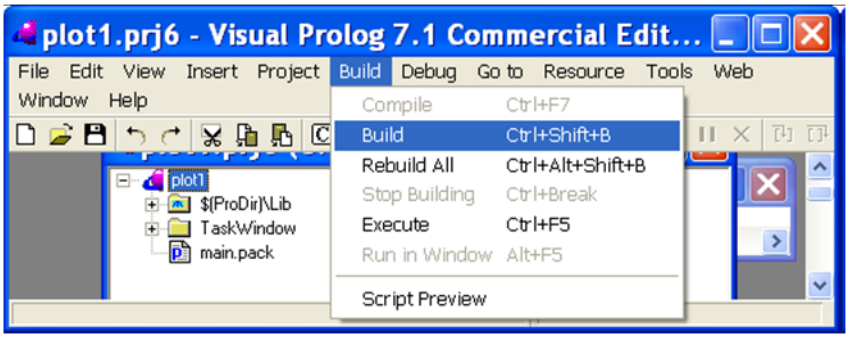


Рисунок 1.4 Компоновка проекта

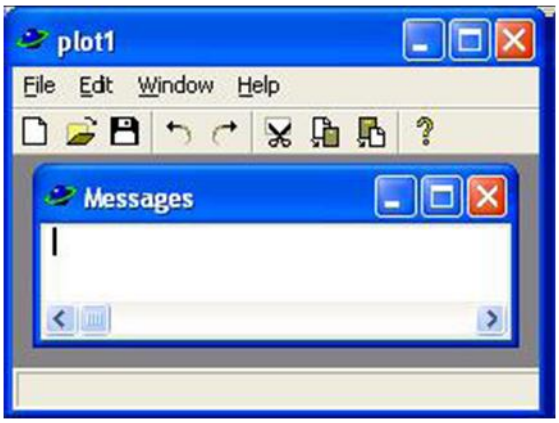


Рисунок 1.5 Пустое приложение

1.2. Примеры

Пример этой главы показывает, как запустить приложение. Он находится в папке *plot1*. В следующей главе вы узнаете, как добавить функциональность к проекту `plot1`. Это означает, что вы сделаете из него что-нибудь полезное.

1.3. Немного о логике: Древние греки

Древние греки изобрели особое общество, которое сильно отличалось от типов общественного устройства, существовавших до них. В других цивилизациях политические или законодательные решения проводились в жизнь правителем, небольшой группой аристократов или наместником. Эти люди могли письменно узаконить основания для своих решений. Делая это, они выдавали полученный свод правил за законы, полученные от Бога или мифологического героя. Что же касается греков, то их законы и политические решения вносились теми гражданами, которые добились места в суде или в законодательном собрании своими способностями, везением или путём выборов. Предложение выносилось на голосование и вводилось в действие, только если большинство голосовавших поддерживало его. Давайте посмотрим, как ситуацию описывал Перикл (*Pericles*).

> *Наша конституция не заимствует законы близлежащих стран, мы скорее являемся образцом для подражания, чем подражаем сами. Она предпочитает большинство меньшинству в управлении, вот почему это называется демократией. Если мы посмотрим на законы, то увидим, что они предоставляют равное правосудие для всех, независимо от частных различий. Для тех, кто не имеет достаточного социального положения, продвижение в общественной жизни подчиняется только праву, классовые различия не должны влиять на оценку достоинств. Никогда более бедность не заградит путь. Если человек способен служить государству, скромность его положения не является помехой ни для предложения им законов, ни для его политических действий.*

В подобном обществе тому, кто желал соблюсти свои интересы или отстоять свою точку зрения, необходимо было убеждать всех остальных в своей правоте, выигрывать дебаты и диспуты. Таким образом, необходимость заставила греков изобрести логику. Конечно, они разработали и иные способы убеждения, не только логику. Искусство лгать, психология толпы, наука введения в заблуждение, витиеватая речь и лесть — также присутствовали в арсенале греческих способов выигрывать дебаты. Однако эта книга сосредоточится на логике.

Глава 2: Формы

В этой главе вы добавите форму в пустой проект, созданный в главе 1. Форма — это окно, на котором можно разместить такие компоненты, как кнопки, запускающие действия, поля редактирования, которые можно использовать для ввода текста, и полотна, на которых можно рисовать.

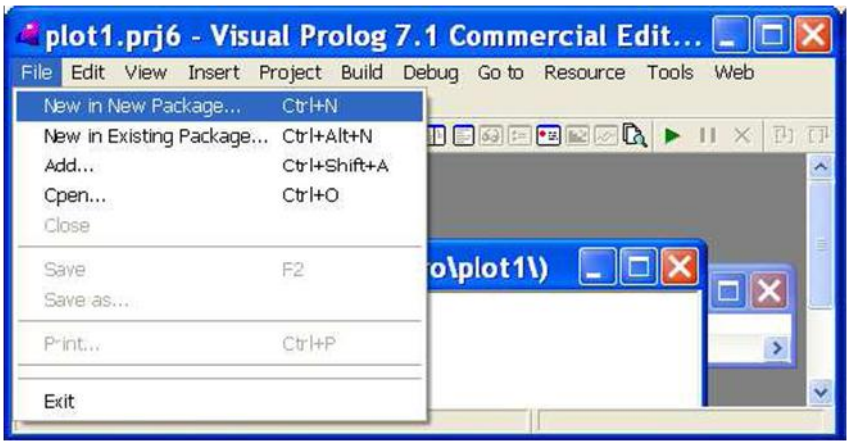


Рисунок 2.1 Добавление новой сущности к дереву проекта

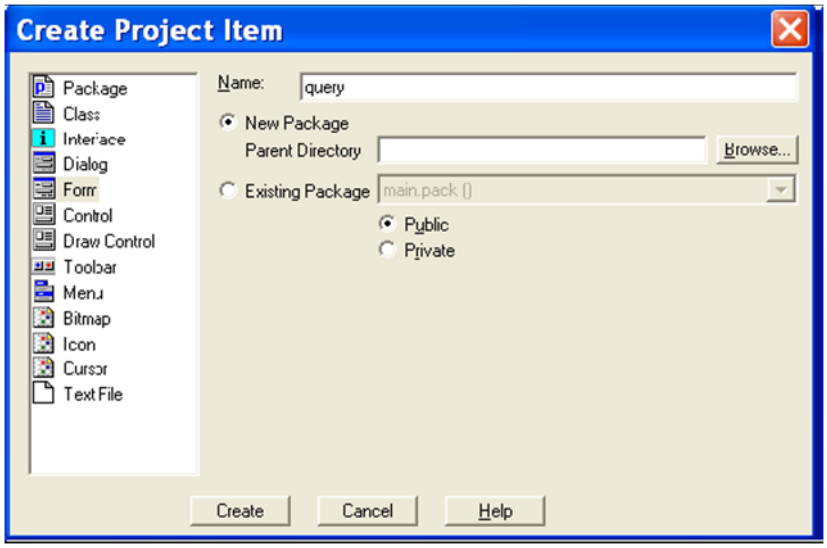


Рисунок 2.2 Создание новой формы

2.1. Создание формы: *folder/name*

Для того чтобы создать форму выберите команду меню *File/New in New Package* (см. рис. 2.1). Выделите пункт *Form* на левой панели диалогового окна *Create Project Item* и

заполните его так, как показано на рисунке 2.2. Новая форма называется *query*. Так как вы выбрали пункт *New in New Package*, Visual Prolog создаст форму в пакете с таким же названием. После нажатия на кнопку *Create* диалогового окна *Create Project Item*, IDE покажет вам прототип новой формы (см. рис. 2.3). Вы можете изменить размеры окна, сделав его немного больше прототипа. Для этого нужно щелкнуть мышью и, удерживая нижний правый угол, тянуть за него так же, как вы это делаете, когда изменяете размеры обычного окна.

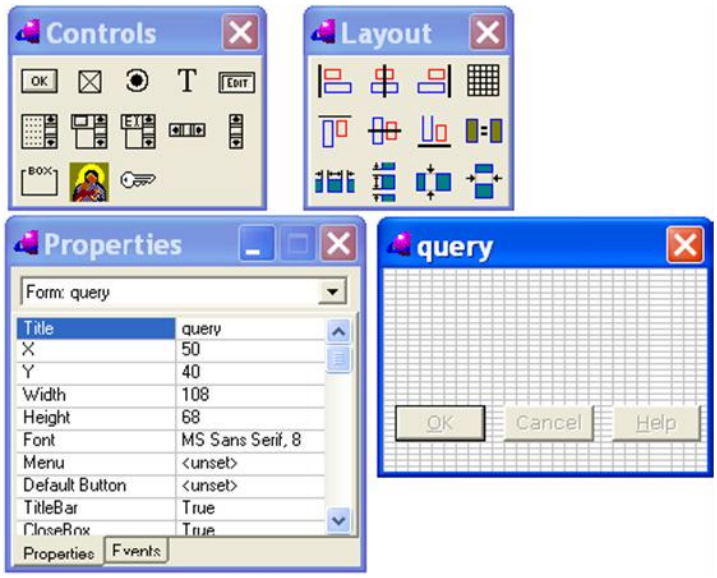


Рисунок 2.3 Изменение размеров формы

2.2. Включение пункта меню: *File/New*

Когда вы запускали пустое приложение, то, скорее всего, заметили, что пункт меню *File/New* отключен. Для того чтобы включить его, нажмите на элемент *TaskMenu.mnu* дерева проекта (см. рис. 2.4). Затем разверните дерево, которое появится в нижней части диалогового окна *TaskMenu,* и уберите галочку из поля *Disabled*, соответствующего пункту меню *&New/tF7*, как показано на рисунке 2.5.

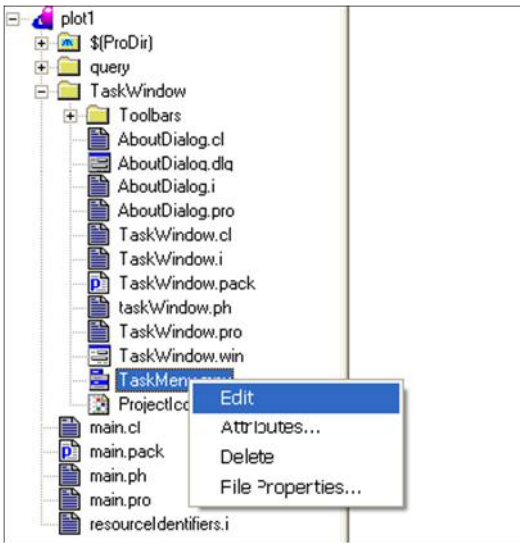


Рисунок 2.4 Дерево проекта/меню задач

2.3. Добавление кода к элементу дерева проекта. Эксперт кода

Для того чтобы добавить код к элементу *File/New* выделите элемент *TaskWindow.win* дерева проекта правой кнопкой мыши, и перед вами откроется контекстное меню. Выберите его пункт *Code Expert* (см. рис 2.6). Следуя рисунку 2.7, нажмите на элемент, показанный ниже:

Наконец, нажмите кнопку *Add* (ориентируйтесь по рис. 2.7) или дважды щелкните в области *id_file_new -> onFileNew*.

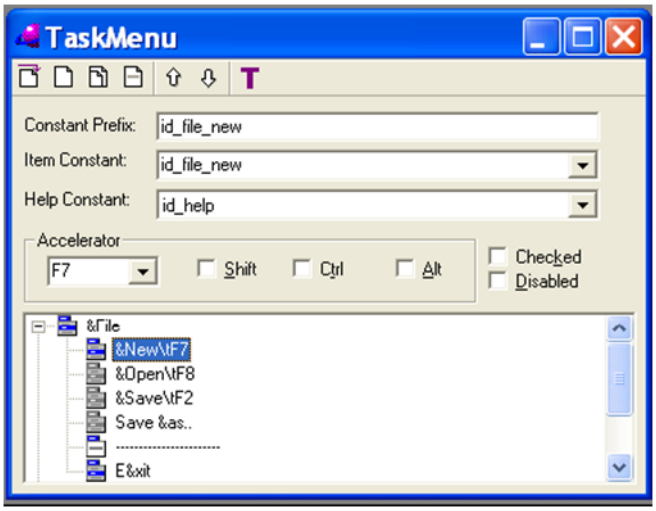


Рисунок 2.5 Включение элемента меню задач

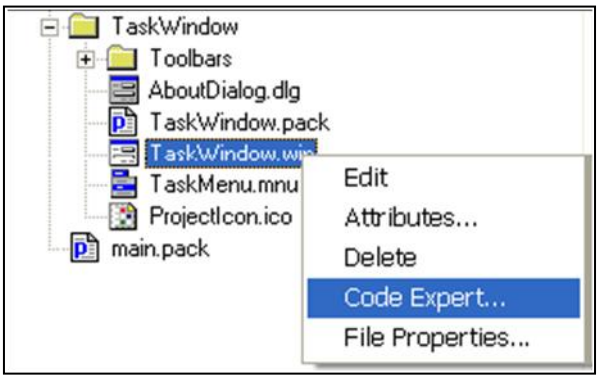


Рисунок 2.6 Переход в Code Expert

В результате откроется текстовый редактор, содержащий следующий фрагмент кода

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

Скомпилируйте программу, а затем измените этот фрагмент так[1]:

```
clauses
    onFileNew(W, _MenuTag) :-
        X = query::new(W),
        X:show().
```

Снова скомпилируйте программу, выбрав команду *Build/Build*. Запустив программу, вы увидите, что когда вы выбираете пункт меню *File/New*, создается новая форма.

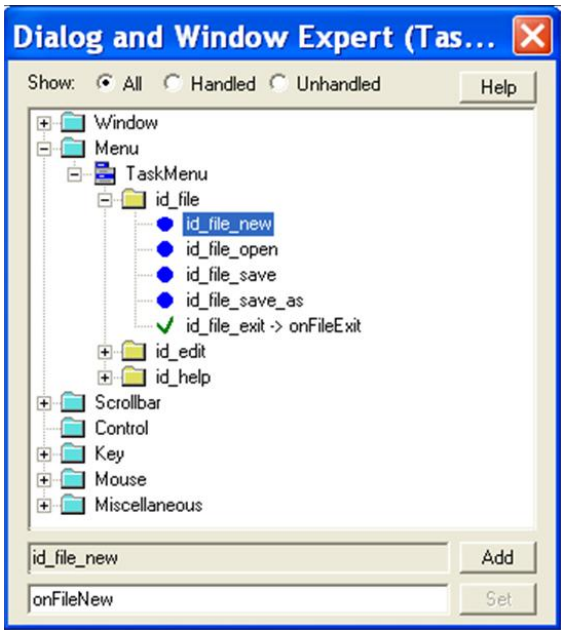


Рисунок 2.7 Dialog and Window Expert

2.4. Примеры

Пример этой главы показывает, как создать новую форму с помощью команды меню среды *File/New…* и последующего заполнения диалогового окна *Create Project Item*.

[1] Или проще (*здесь и далее примечания редактора перевода*):

```
clauses
    onFileNew(W, _MenuTag) :-
        _ = query::display(W).
```

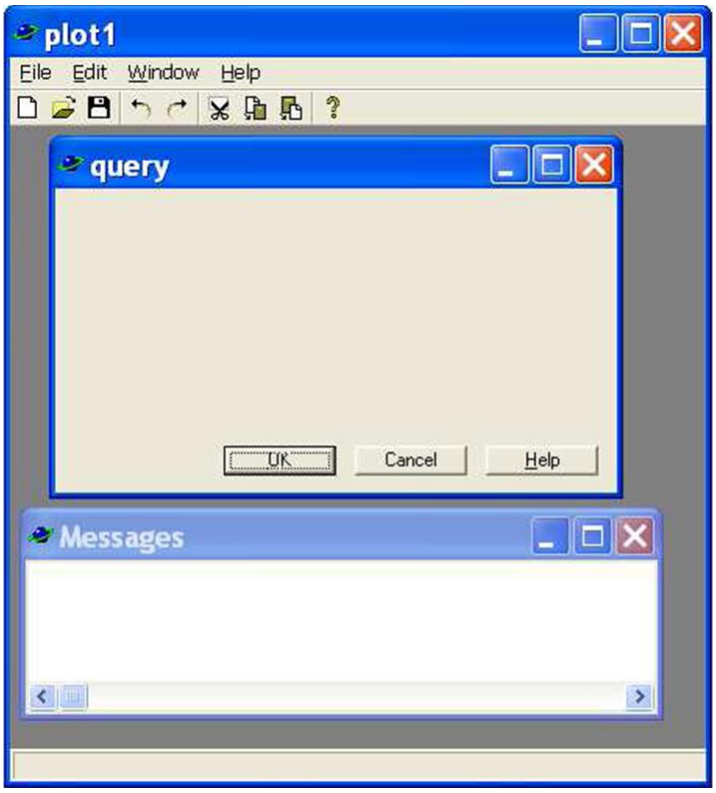


Рисунок 2.8 Появляющаяся форма

2.5. Немного о логике: Аристотелева силлогистика

Согласно Аристотелю, *суждение* состоит из двух *терминов*, субъекта и предиката, а также кванторов «каждый», «ни один», «некоторый», «не каждый». Слово *субъект* происходит от латинского переложения греческого выражения ὑποχείμενον, который обозначает то (или того), о чем или о ком это суждение (предмет суждения). *Предикат* выражает то, что именно говорит суждение об этом субъекте.

Португальский логик, врач и математик Педру Жулиан (*Pedro Julião*), также известный как Петр Испанский, внес значительный вклад во многие области знания, прежде чем был избран Папой Римским под именем Иоанна XXI. Например его считают отцом офтальмологии и профилактической медицины. Он также написал «Логический трактат» (*Summulae Logicales*), оказавший большое влияние. Ему приписывается изобретение следующих сокращений для логических кванторов:

Квантор	Тип суждения	Сокращение	Пример
a	Общеутвердительное	*P a Q*	Каждое *P* есть *Q*
e	Общеотрицательное	*P e Q*	Ни один *P* не есть *Q*
i	Частноутвердительное	*P i Q*	Некоторое *P* есть *Q*
o	Частноотрицательное	*P o Q*	Некоторое *P* не есть *Q*

Суждения делятся на контрарные, контрадикторные (противоречивые) и субконтрарные. Ниже приведены определения каждого из этих классов.

a. Суждения *P* и *Q* являются *противоречивыми*, если и только если:

   I. *P* и *Q* не могут быть истинными одновременно, а также

II. либо *P*, либо *Q* истинно.

Например, типы *a* и *o* противоречивы; то же верно для *e* и *i*.

b. Суждения *P* и *Q* являются *контрарными* тогда и только тогда, когда:

I. *P* и *Q* не могут быть истинными одновременно, а также
II. *P* и *Q* могут быть одновременно ложными.

Например, типы *a* и *e* контрарны.

c. Суждения *P* и *Q* *субконтрарны*, тогда и только тогда, когда:

I. *P* и *Q* не могут быть одновременно ложными, а также
II. *P* и *Q* могут быть одновременно истинными

Например, типы *i* и *o* субконтрарны.

Силлогизм — это рассуждение, состоящее из трех суждений (двух посылок и заключения), в котором присутствуют ровно три термина, каждый из которых используется точно дважды.

Пусть *P* — это предикат заключения (больший термин), а *S* — это субъект заключения (меньший термин). Аристотель называет эти термины крайними. Пусть также *M* обозначает термин, который является общим для посылок, но отсутствует в заключении; этот термин называется средним термином. Посылка, содержащая *P*, называется большей посылкой, а посылка, содержащая *S* — меньшей посылкой.

Пусть символ * обозначает любой из четырех кванторов '*a*', '*e*', '*i*', '*o*'. Тогда классификация силлогизмов может быть представлена с помощью следующих трех фигур[1]:

Первая фигура	*Вторая фигура*	*Третья фигура*
$\frac{M * P}{\underline{S * M}}$	$\frac{P * M}{\underline{S * M}}$	$\frac{M * P}{\underline{M * S}}$
$S * P$	$S * P$	$S * P$

В результате замены звездочек кванторами получается набор суждений — {большая посылка, меньшая посылка, заключение}. Согласно Петру Испанскому, каждый отдельный набор вида {*a*, *a*, *a*}, {*a*, *i*, *o*}, … образует правило, или модус. Для того чтобы вычислить, сколько существует различных модусов, вспомним уроки комбинаторики.

Размещение — это такой способ расположения элементов, при котором порядок их следования имеет значение (как в списках). С другой стороны, сочетания — это группы

[1]Классификация фигур силлогизмов осуществляется в зависимости от местоположения среднего термина в посылках. Петр Испанский объединял четвертую фигуру с первой. Четвертая фигура имеет вид:

$$\frac{P * M}{\underline{M * S}}$$
$$S * P$$

Порядок терминов в фигурах силлогизмов в переводе отличается от порядка, приведенного в оригинале данной книги. По мнению редактора перевода, первоначальный вид фигур вносит некоторую путаницу, и, скорее всего, появился вследствие опечаток.

элементов, в которых порядок следования этих элементов не является существенным (как в множествах — *ред*. *пер*.). Порядок кванторов в силлогизме имеет значение, поэтому мы имеем дело с размещениями. Мы должны разместить три элемента, выбрав их из четырех, при этом повторения не исключаются. Если мы выбираем *p* элементов из *n*-элементного множества, то общее количество размещений с повторениями находится следующим образом:

$$\underbrace{n \times n \times \ldots \times n}_{p} = n^p$$

В нашем случае нужно выбрать три квантора из множества {*a*, *e*, *i*, *o*}. Таким образом, существует 4^3=64 возможности для каждой фигуры, а так как мы имеем три фигуры, то это число возрастает до 192[1].

2.5.1. Истинные силлогизмы

Не все силлогизмы представимы с точки зрения логики. Аристотель признавал четыре допустимых вида (т. е. правильных модуса) для первой и второй фигур и шесть видов для третьей. С другой стороны, современная логика признаёт четыре вида для каждой из трех фигур[2].

Петр Испанский изобрел латинскую мнемонику для облегчения запоминания допустимых видов. В его схемах на вид указывают гласные. Например, имя BARBARA обозначает {*a*, *a*, *a*} — правильный модус для первой фигуры. Специалисты в области классической латыни заметят, что эти названия основаны на средневековом произношении, в котором такое слово, как CÆSARE записывалось как CESARE и произносилось соответственно. Так или иначе, вот полный список правильных модусов:

Первая фигура	**Вторая фигура**	**Третья фигура**
{a, a, a} – Barbara	{e, a, e} – Cesare	{o, a, o} – Bocardo
{e, a, e} – Celarent	{a, e, e} – Camestres	{e, i, o} – Ferison
{a, i, i} – Darii	{e, i, o} – Festino	{i, a, i} – Disamis
{e, i, o} – Ferio	{a, o, o} – Baroco	{a, i, i} – Datisi

Согласно Льюису Кэрроллу [Lewis Carroll], Венн предложил диаграмму, с помощью которой можно проверить, является ли силлогизм истинным. Рисунок 2.9 иллюстрирует метод Венна для силлогизма Barbara.

Первая посылка силлогизма Barbara говорит, что *все M есть P*. Таким образом, затемняется часть M, не содержащаяся в P (см. слева на рисунке). Вторая посылка говорит, что *все S есть M*. Поэтому нужно затемнить часть S, находящуюся вне M (см. справа). Часть S, оставшаяся незатемненной, полностью лежит внутри P. Следовательно, можно заключить, что *все S есть P*.

[1] В приведенной здесь первой фигуре существует две возможности для расположения термина *M*, поэтому всего возможностей — 256.

[2] Из четырех схем правильных модусов, приведенных для первой фигуры, первые три не являются правильными для четвертой фигуры.

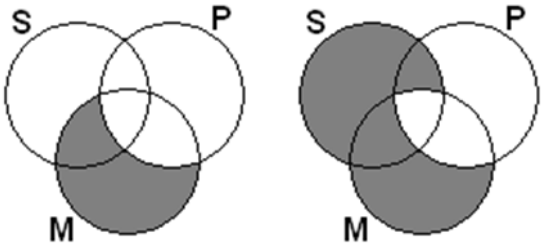


Рисунок 2.9 Силлогизм Barbara

$$\frac{M\ a\ P \qquad S\ a\ M}{S\ a\ P}$$

Теперь рассмотрим силлогизм *Darii*, который говорит[1]:

$$\frac{\forall\ M \text{ есть } P \qquad \exists\ S \text{ есть } M}{\exists\ S \text{ есть } P}$$

Символ ∀ здесь означает *все*, а символ ∃ — *некоторое*. Венн утверждает, что общая посылка должна быть отображена на диаграмме до частной посылки. Поэтому в левой части рисунка 2.10 мы затемнили часть M, которая находится вне P, как этого требует посылка ∀ *M есть P*.

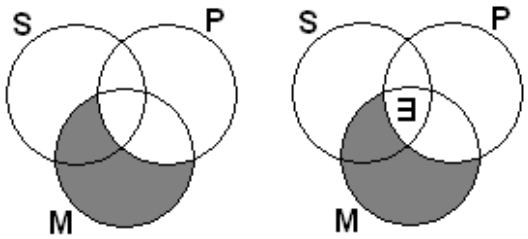
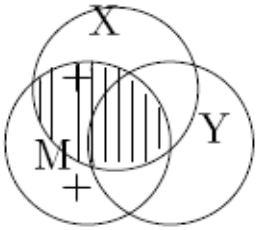


Рисунок 2.10 Силлогизм *Darii*

Следующий шаг состоит в том, чтобы поместить символ ∃ в область, соответствующую посылке ∃ *S есть M*. Как видно по рисунку 2.10, символ ∃ попадает в область SPM. Заключение состоит в том, что *некоторое S есть P*.

Отец Алисы Лидделл старался дать ей очень хорошее образование. Когда он заметил, что сколько-нибудь хороших греко-английских словарей не существует, он и его друг Скотт сели и подготовили очень хороший словарь, который с удовольствием используют почти все студенты, изучающие греческий. Он также пригласил известного писателя Льюиса Кэрролла в качестве преподавателя логики для двух своих дочерей. Для того чтобы сделать занятия более интересными, Льюис Кэрролл написал для детей несколько книг: «Алиса в Стране чудес» (*Alice in the Wonderland*), «Алиса в Зазеркалье» (*Through the Looking Glass*) и «Логическая игра» (*Game of Logics*). Ниже приведен пример из «Логической игры».

[1] Описание силлогизма *Darii* (включая рис. 2.10) в переводе отличается от описания, приведенного в оригинале. В переводе для этого силлогизма используется первая фигура, а автор использует для него четвертую фигуру, что нам представляется ошибочным.



Никакие философы не тщеславны.
Некоторые нетщеславные люди не играют в карты.

Поэтому некоторые не играющие в карты — не философы[1].

Следующее решение вышеприведённого силлогизма было любезно предоставлено мне лично г. Венном. Меньшая посылка говорит о том, что некоторые из элементов, составляющих ***my'*** *должны быть сохранены — отметим это крестиками. Большая посылка завляет, что все* ***xm*** *должны быть уничтожены, поэтому сотрём их. Тогда, так как некоторые из* ***my'*** *должны остаться, ясно, что это должны быть* ***my'x'****. Таким образом, должны существовать* ***my'x'****, или, отбрасывая* ***m****, должны существовать* ***y'x'****. Говоря обычным языком,* некоторые не играющие в карты — не философы*.*

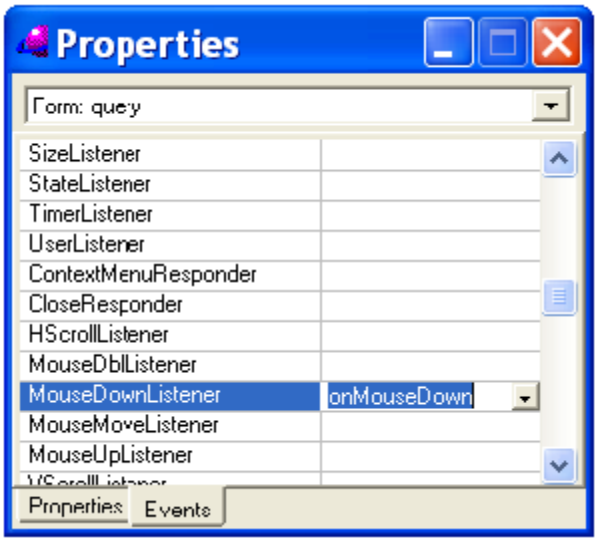
[1] Силлогизм относится к четвертой фигуре и принадлежит типу Fresison. На рисунке используются обозначения: X — философы, M — тщеславные люди, а Y — игроки в карты.

Глава 3: События мыши

В первых двух главах вы научились создавать приложение с формой, которая появляется, когда вы выбираете пункт меню *File/New* этого приложения.

3.1. Добавление кода к *MouseDownListener*

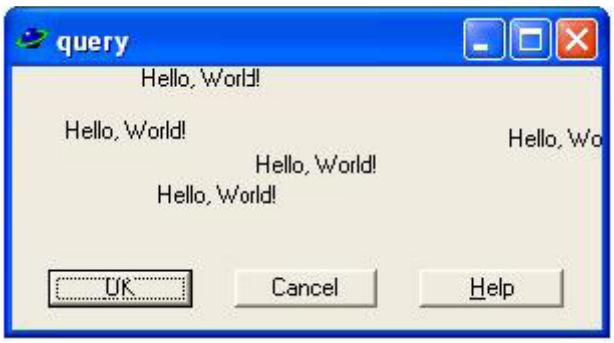
Нажмите на элемент *query.frm* в дереве проекта, для того чтобы опять открыть форму *query*, как показано на рисунке 2.3, и откройте вкладку *Events* диалогового окна *Properties*. Перед вами откроется приведенный ниже список событий.



Дважды щёлкните по *MouseDownListener* и замените процедуру для `onMouseDown/4` следующим кодом:

```
clauses
    onMouseDown(S, Point, _ShiftControlAlt, _Button) :-
        W= S:getVPIWindow(), Point= pnt(X, Y),
        vpi::drawText(W, X, Y, "Hello, World!").
```

Постройте приложение и запустите его. Выберите пункт *File/New* для создания новой формы. При каждом щелчке мышью в любой точке формы будет выводиться известное приветствие.



3.2. Обработчик onPaint

Для того чтобы нарисовать что-нибудь на форме, вы использовали обработчик события `onMouseDown/4`. Некоторые программисты не считают это хорошей идеей. Действительно, имеются языки, в которых этот подход трудно осуществим. Язык Java, кстати очень популярный, является одним из таких языков. На языке Java любое рисование следует производить внутри следующего метода:

```
public abstract void doPaint(java.awt.Graphics g)
```

Конечно, можно обойти это ограничение, даже на Java. В любом случае, давайте научимся рисовать с помощью обработчика событий `onPaint/3`.

- Создайте проект с именем `plot0`.
- Добавьте новую форму `query` в новый пакет `query`.
- Включите пункт меню *File/New*, и добавьте код

```
clauses
    onFileNew(S, _MenuTag) :-
        Q= query::new(S), Q:show().
```

для *ProjTree/TaskWindow.win/Menu/TaskMenu/id_file/id_file_new*.

- Скомпилируйте приложение.

Следующий шаг состоит в добавлении к обработчику события *MouseDownListener* следующего фрагмента кода:

```
class facts
    mousePoint:pnt := pnt(-1, -1).
predicates
    onMouseDown : drawWindow::mouseDownListener.
clauses
    onMouseDown(_S, Point, _ShiftControlAlt, _Button) :-
        mousePoint := Point,
        Point=pnt(X,Y),
        R=rct(X-8, Y-8, X+60, Y+8),
        invalidate(R).
```

Теперь вы готовы к тому, чтобы реализовать обработчик событий `onPaint` для формы `query`. Обработчик `onPaint` будет вызываться всякий раз, когда прямоугольная область формы будет перекрыта или станет недействительной. Когда изображение становится недействительным? Тогда, когда окно полностью или частично перекрывается другим окном, или когда вы обновляете его с помощью предиката `invalidate(R)`, где `R` — это прямоугольник. Прямоугольник описывается координатами левого верхнего и правого нижнего углов в виде:

```
R=rct(X-8, Y-8, X+60, Y+8)
```

Вы, вероятно, заметили, что обработчик `onMouseDown/4` делает недействительным прямоугольную область вокруг точки щелчка мыши. Поэтому обработчик `onPaint` будет воздействовать на обновляемую область, перерисовывая ее. Для того чтобы это реализовать, добавьте фрагмент кода, представленный на рисунке 3.2, в обработчик

PaintResponder. Для этого щелкните на элемент `query.frm` дерева проекта и откройте вкладку *Events* окна *Properties* (см. рис. 3.1, 2.3 и п. 3.1). Найдите *PaintResponder* в списке событий, который появится в диалоговом окне *Properties*.

Данный подход потребует от вас понимания массы всего нового, такого как конструкция `if-then-else` и идея *глобальной переменной*. Не беспокойтесь сейчас об этом. Вы все узнаете в подходящее время.

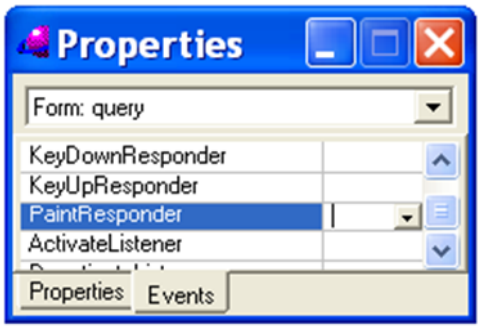


Рисунок 3.1 PaintResponder

```
predicates
    onPaint : drawWindow::paintResponder.
clauses
    onPaint(_Source, _Rectangle, GDIObject) :-
        if pnt(X, Y)= mousePoint, X>0
        then
            mousePoint := pnt(-1, -1),
            GDIObject:drawText(pnt(X, Y), "Hello, World!", -1)
        else succeed() end if.
```

Рисунок 3.2 Код для PaintResponder

3.3. Примеры

Примеры в этой главе показывают, как обрабатывать события мыши, такие как нажатие на левую кнопку.

3.4. Немного о логике: Булева алгебра

Исчисление высказываний — это формальная система, в которой формулы, представляющие высказывания, строятся из атомарных высказываний с помощью логических связок. Формулы в исчислении высказываний подчиняются следующим грамматическим правилам:

1. Пусть α-множество содержит атомарные формулы, обозначаемые в общем случае строчными буквами латинского алфавита. Любой элемент α-множества является правильно построенной формулой.
2. Если p — формула, то $\neg p$ — формула. Символ $\neg$ обозначает логическое отрицание. Таким образом, p — это *истина*, если $\neg p$ — *ложь*, и наоборот: p — *ложь*, если $\neg p$ — *истина*.
3. Если p и q — формулы, то $p \vee q$ (логическая дизъюнкция), $p \wedge q$ (логическая конъюнкция), $p \rightarrow q$, $p \equiv q$ — тоже формулы. Семантически формула $p \vee q$ означает «p или q» и является истинной, если истинна хотя бы одна из формул p и q. Формула $p \wedge q$ истинна, если только истинны обе формулы p и q. Формула $p \rightarrow q$ означает, что из p следует q, т. е. если p — истинна, то и q — истинна. Наконец, формула $p \equiv q$ означает, что p эквивалентна q: если p — истинна, то и q — истинна, а также, если q — истинна, то и p — истинна.

Таблицы истинности. Для того чтобы установить значение истинности формулы, часто обращаются к таблицам истинности. Рассмотрим примеры таблиц истинности. Пусть цифра 1 обозначает константу `true`, а цифра 0 — константу `false`. Тогда таблица истинности для отрицания ($\neg p$), логической конъюнкции ($p \wedge q$) и логической дизъюнкции ($p \vee q$) имеет вид:

p	q	$\neg p$	$p \wedge q$	$p \vee q$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Говоря неформальным языком, формула $p \rightarrow q$ означает следующее:

если p равно `true`, то q также равно `true`.

Ниже приведена соответствующая таблица истинности:

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Можно доказать, что формула $p \rightarrow q$ эквивалентна формуле $\neg p \vee q$. Проверим это:

p	q	$\neg p$	$\neg p \vee q$	$p \rightarrow q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

3.5. Способы аргументации

Логические программисты часто используют способ рассуждений под названием MODUS TOLLENDO PONENS. С помощью логических операций этот простой и очень полезный способ рассуждений записывается следующим образом:

$$\frac{p \vee q \quad \neg p}{\vdash q}$$

где знак $\vdash$ представляет логический вывод (заключение). Грубо говоря, сначала мы говорим, что p либо q истинно. Затем мы говорим, что p — это не *истина*. Отсюда мы выводим, что q — *истина*.

Очень важным способом рассуждений является MODUS PONENS. Важным потому, что он управляет механизмом логического вывода Пролога.

$$\frac{q \leftarrow p \quad p}{\vdash q}$$

Вы уже знаете, что $p \rightarrow q$ означает «если p, то q», т. е. p влечет q. Таким образом, формула $q \leftarrow p$ означает, что q следует из p, или «q, если p». Более подробно, первая строка правила говорит, что q — *истина*, если p — *истина*. Вторая строка говорит, что p — *истина*. Тогда человек или машина могут вывести, что q — *истина*. Рассмотрим конкретный пример.

счастливый(*Человек*) ← слушаетСократа(*Человек*)
слушаетСократа(*платон*)

$\vdash$ счастливый(*платон*)

Еще один правильный способ рассуждений называется MODUS TOLLENS:

$$\frac{p \rightarrow q \quad \neg q}{\vdash \neg p}$$

Глава 4: Меньше иллюстраций

Перед тем, как двигаться дальше, рассмотрим такие способы описания проектов, которые не требуют постоянного обращения к иллюстрациям.

4.1. Главное меню

Меню, показанное на рисунках 1.1 и 1.4, является главным меню интегрированной среды разработки VIP IDE. Обозначение *A/B* указывает на один из его пунктов. Например, используйте пункт меню *Project/New*, для того чтобы создать новый проект (см. рис. 1.1). В результате откроется диалоговое окно *Project Settings*, в котором имеется шесть следующих вкладок: *General*, *Directories*, *Build Options*, *Version Information*, *File Templates* и *Run Options*. В большинстве случаев требуется заполнить только вкладку *General*:

```
General
    Project Name: factorial
    UI Strategy: Object-oriented GUI (pfc/gui)
    Target Type: Exe
    Base Directory: C:\vispro
```

Когда в тексте будет написано:

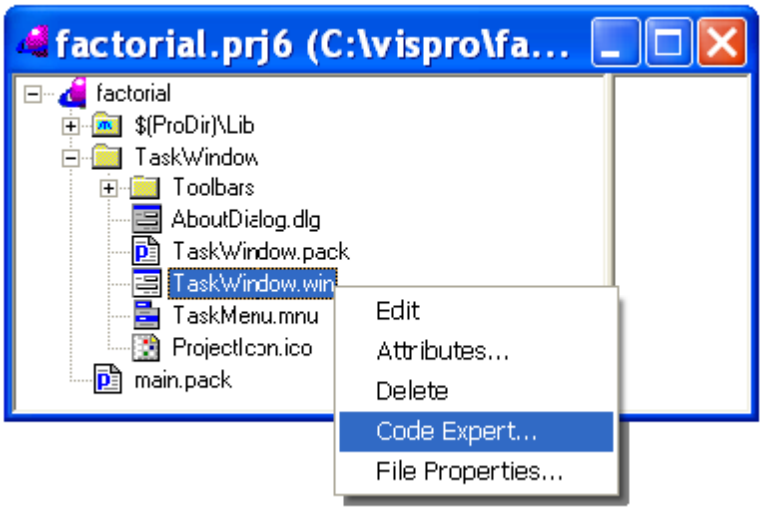
Создайте новый GUI-проект: `factorial` (*раздел* `1.1.1`)

вам следует войти в диалоговое окно *Project Settings* (выбрав команду меню *Project/New* среды разработки) и заполнить вкладку *General* так, как описано выше. Делайте это MUTATIS MUTANDIS[1], т. к. на вашем компьютере может не быть свободного места на диске C, или, быть может, вы захотите поместить свою программу в директорию, отличную от `C:\vispro`.

4.2. Дерево проекта

Простейший способ навигации по файлам и ресурсам — выбрать соответствующий элемент дерева проекта:

[1] С учетом различий, обстоятельств (*лат.*).



Если щелкнуть по папке левой кнопкой мыши, она раскроется, и появится ее содержание. А если щелкнуть по элементу дерева проекта правой кнопкой мыши, то откроется всплывающее меню, как показано выше на рисунке. Если я захочу, чтобы вы вошли в эксперт кода и добавили фрагмент кода к *TaskWindow.win*, то скажу:

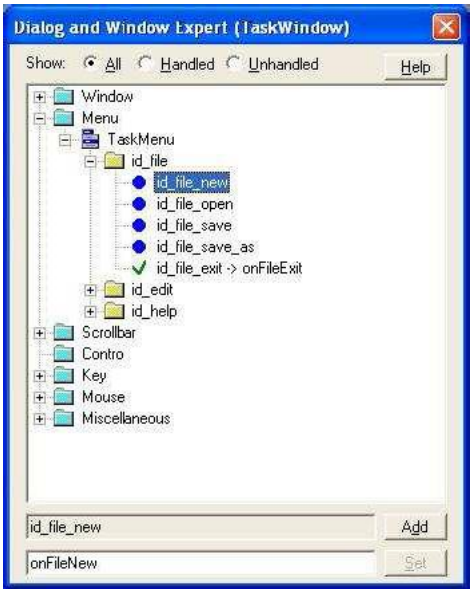
С помощью эксперта кода добавьте код к TaskWindow/TaskWindow.win (*раздел 2.3*)

Для того чтобы это выполнить, обратитесь к дереву проекта и сделайте следующее:

- Откройте папку *TaskWindow*, если она была закрыта.
- Правой кнопкой мыши щелкните по элементу *TaskWindow.win* дерева проекта, чтобы развернулось всплывающее меню вида:

```
Edit
Attribute
Delete
Code Expert
File Properties…
```

- В нем выберите пункт *Code Expert*.



4.2.1. Эксперт кода

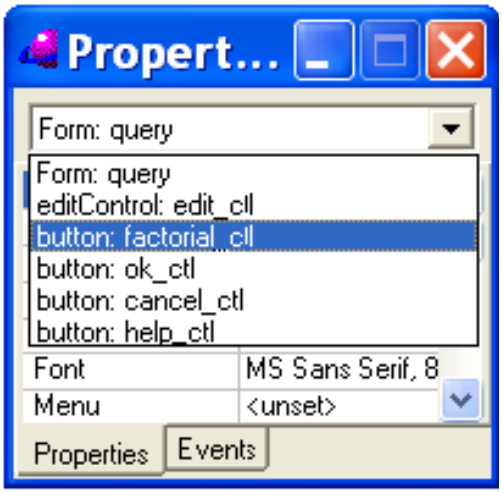
Эксперт кода также имеет форму дерева.

В соответствии со своим названием, эксперт кода используется для вставки кода или фрагментов кода в большое количество файлов проекта. Для того чтобы войти в него, нажмите правой кнопкой мыши на элемент дерева проекта, к которому вы хотите добавить код. Затем выберите пункт *Code Expert* контекстного меню.

Для форм существует более короткий путь в эксперт кода. Если вы щелкнете левой кнопкой мыши по элементу `.frm` в дереве проекта,

появится прототип формы, вместе с окном *Properties* и двумя панелями компонент, одна из которых используется для создания элементов управления, а другая — для схемы оформления. Если вы откроете вкладку *Events* окна *Properties*, то увидите список событий. В предыдущих главах вы имели возможность работать с двумя обработчиками событий — *MouseDownListener* и *PaintResponder*.

На формах присутствуют такие компоненты, как кнопки и поля редактирования. В верхней части диалогового окна *Properties* расположен раскрывающийся список для выбора этих компонентов. Если вы выберете один из этих компонентов и перейдёте на вкладку *Events*, то получите список событий, соответствующих этому компоненту. Подробнее об этом будет сказано позднее.



Для того чтобы пройти через дерево эксперта кода и достигнуть места, в которое вы хотите вставить код, щелкайте левой кнопкой мыши по соответствующим ветвям дерева. Если вы хотите, чтобы эксперт кода добавил прототип для листа дерева, щелкните по нему и нажмите кнопку *Add*, которая появится в нижней части диалогового окна. Затем снова щелкните на лист, чтобы добраться до кода.

Если я прошу вас: *с помощью эксперта кода добавьте фрагмент кода* (см. рис. 2.3)

```
clauses
    onFileNew(W, _MenuTag) :-
        S= query::new(W), S:show(W).
```

для *TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new*, вот шаги, которым вы должны следовать.

- ***Дерево проекта***. Откройте папку *TaskWindow* дерева проекта, щелкните правой кнопкой мыши по *TaskWindow.win*, чтобы открыть контекстное меню, и выберите пункт *Code Expert* (см. рис. 2.6).

- ***Эксперт кода***. Выберите *Menu* → *TaskMenu* → *id_file* → *id_file_new* и нажмите кнопку *Add* для создания прототипа кода. Наконец, дважды щелкните в области *id_file_new*→*onFileNew*. Ориентируйтесь по рисункам 2.6, 2.7 и разделу 2.3. Добавьте указанный выше код в файл `TaskWindow.pro`.

4.3. Создание элемента проекта

Для добавления нового элемента в дерево проекта выберите пункт *File/New in New Package* меню среды, если вы хотите поместить элемент внутри пакета, который будет иметь такое же название. В случае, если вы хотите поместить элемент в существующий пакет, используйте пункт меню *File/New in Existing Package*.

Внимательно проследите за тем, чтобы новый элемент или новый пакет попали в нужную папку. В приведенном ниже примере пакет, содержащий форму *query*, создается в корне проекта `factorial`. Всегда выбирайте имена, которые что-то значат. Например,

если бы пакет содержал элементы компьютерной графики, вы могли бы назвать его *canvasFolder*. Если бы он содержал формы для запросов, хорошим именем было бы *formContainer*, и так далее. В нашем примере:

- ***Создайте новый пакет в корне дерева проекта*** `factorial` (см. рис. 4.1). Пусть имя пакета будет `formcontainer`.

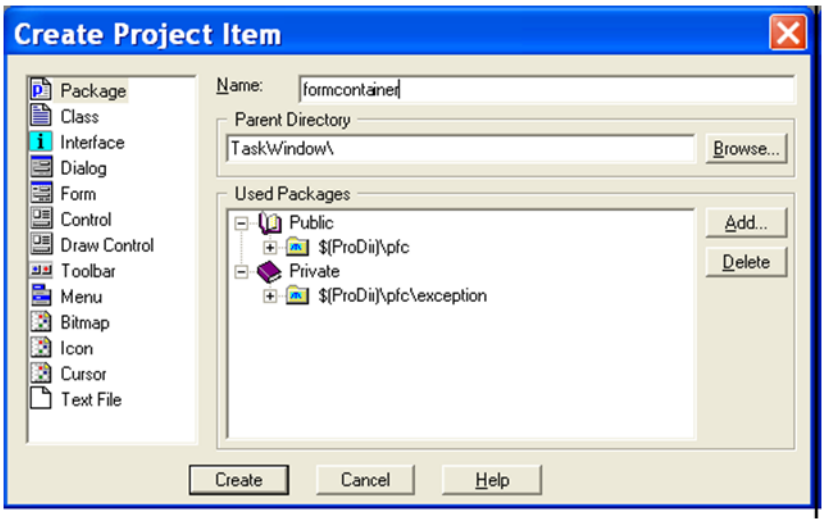


Рисунок 4.1 Создание нового пакета

- ***Создайте новую форму*** (`query`) внутри пакета `formcontainer`. Для этого выделите папку, соответствующую этому пакету в дереве проекта, и выберите пункт *File/New in Existing Package* меню задач (см. раздел 2.1). Для того чтобы при выборе *File/New in Existing Package* окно поместилось в этот пакет, вы должны убедиться, что папка пакета уже выделена.

Когда вы создаете форму, IDE показывает окно *Form Window* (см. рис. 4.2). Вы можете изменить размеры окна и добавить поле редактирования (`edit_ctl`) и кнопку (`factorial_ctl`), как показано на рисунке 4.2. Вы также можете добраться до диалогового окна *Form Window*, дважды щёлкнув левой кнопкой мыши по имени формы в дереве проекта.

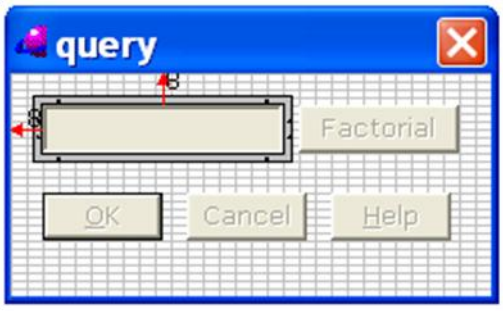


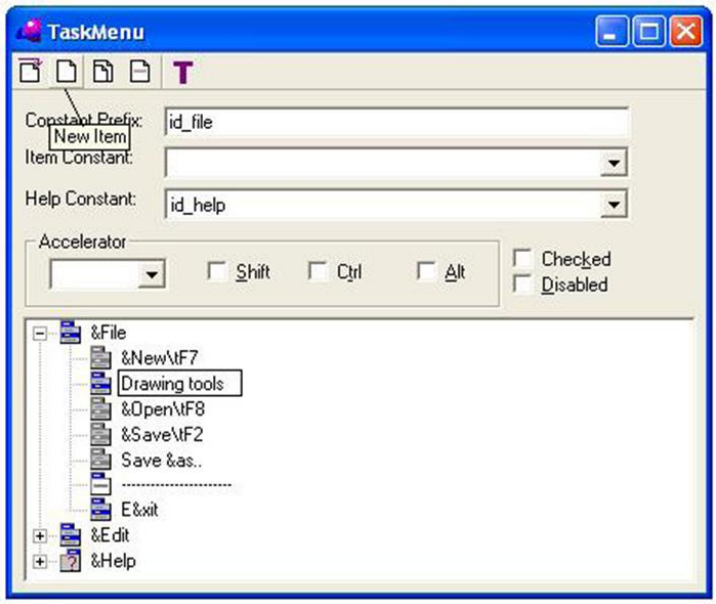
Рисунок 4.2 Форма с полем редактирования

Если вы дважды щёлкнете по элементу *ProjectTree/TaskMenu.mnu*, то получите диалоговое окно, которое было показано на рисунке 2.5. Вы можете развернуть дерево этой спецификации меню, щелкая по его ветвям, как говорилось ранее. Вам также может

понадобиться включить какой-либо пункт меню, как это было в разделе 2.2. Можно также создать новый пункт меню, как показано ниже на рисунке, где я нажал на кнопку (пиктограмму) под названием *New Item* и создал элемент *Drawing Tools*, заполнив появившееся поле. Как видно по рисунку, новый пункт меню создается изначально включенным. Символ & в записи *&File* обозначает, что клавиша F является акселератором.

Давайте начнем новый проект с нуля, чтобы посмотреть, что мы способны сделать при меньшем количестве иллюстраций и проверить, получилось ли у нас краткое описание Visual Prolog-проекта.

- ***Создайте новый GUI проект***: `factorial` (см. раздел 1.1.1).
- ***Добавьте новый пакет к дереву проекта***: `factorial/formcontainer`.
- ***Создайте новую форму***: `forms/query` (см. раздел 4.3). Добавьте в нее поле редактирования (с названием `edit_ctl`) и кнопку (с названием `factorial_ctl`), как показано на рисунке 4.2.
- ***Включите пункт меню*** *TaskMenu File/New* (см. раздел 2.2),



затем добавьте следующий код (см. раздел 2.3):

```
clauses
    onFileNew(W, _MenuTag) :-
        S= query::new(W), S:show().
```

для *TaskWindow.win/Menu/TaskMenu/id_file→id_file_new→onFileNew*.

- ***Постройте приложение***, для того чтобы вставить в проект новые классы.

4.4. Создание нового класса: *folder/name*

Для того чтобы создать новый класс и поместить его в пакет *formcontainer*, выделите папку *formcontainer* в дереве проекта и выберите пункт *File/New in Existing Package* меню

среды. Затем выберите элемент *Class* в диалоговом окне *Create Project Item* и введите `fn` в качестве имени класса в поле *Name*. Убедитесь, что галочка в поле *Create Objects* убрана.

Когда вы нажмёте кнопку *Create*, Visual Prolog покажет файлы `fn.pro` и `fn.cl`, которые содержат прототип класса `fn`. Наша задача — добавить функциональности к этим файлам, заменив их содержимое кодом, приведённым в разделе 4.6. Затем постройте приложение ещё раз, чтобы убедиться, что Visual Prolog добавил в проект класс `fn`.

4.5. Содержимое поля редактирования

Получение содержимого поля редактирования — дело хитроумное. Это верно для Visual C++, Delphi, Clean и др. Следует отдать должное Visual Prolog — по сравнению с другими языками он предлагает самый легкий доступ к элементам управления. Однако это все еще непросто. Для облегчения этого процесса IDE сохраняет идентификатор поля редактирования в факте-переменной. Для того чтобы убедиться, насколько это удобно, откройте форму `query.frm`, нажав на этот элемент дерева проекта.

В диалоговом окне *Properties* выберите из раскрывающегося списка элемент `button:factorial_ctl`, откройте вкладку *Event*, щёлкните по элементу *ClickResponder* списка событий и добавьте к коду для кнопки `factorial_ctl` следующий фрагмент:

```
clauses
    onFactorialClick(_S) = button::defaultAction() :-
        fn::calculate(edit_ctl:getText()).
```

Постройте и запустите приложение.

Рисунок 4.3 Факториал

4.6. Примеры

В этой главе вы узнали, как создать форму, содержащую кнопку (`factorial_ctl`) и поле редактирования (`edit_ctl`). Вы также узнали, как получить доступ к содержимому поля редактирования. Ниже приведён класс `fn`, в котором реализуется функция вычисления факториала.

```
% Файл fn.cl
class fn
predicates
```

```
    classInfo : core::classInfo.
    calculate : (string) procedure.
end class fn

% Файл fn.pro
implement fn
    open core

class predicates
    fact : (integer, integer) procedure (i,o).
clauses
    classInfo("forms/fn", "1.0").

    fact(0, 1) :- !.
    fact(N, N*F) :- fact(N-1, F).

    calculate(X) :- N= toterm(X),
        fact(N, F), stdio::write(F, "\n").
end implement fn
```

4.7. Немного о логике: Исчисление предикатов

В исчислении высказываний нет переменных и кванторов. Положение стало другим, когда Фридрих Людвиг Готлоб Фреге (*Friedrich Ludwig Gottlob Frege*) представил человечеству исчисление предикатов. Однако обозначения Фреге были слишком трудны для использования. Современные обозначения ввел Джузеппе Пеано. Суждения Аристотеля в современных обозначениях исчисления предикатов выглядят следующим образом:

Все *a* есть *b*	$\forall X(a(X) \rightarrow b(X))$
Некоторое *a* есть *b*	$\exists X(a(X) \wedge b(X))$

Глава 5: Предложения Хорна

5.1. Функции

Я уверен, что вы знаете, что такое функция. Возможно, вы не знаете математического определения функции, но вы чувствуете, что это такое, исходя из опыта, полученного в ходе использования калькуляторов и компьютерных программ или после посещения одного из курсов элементарной алгебры. Функция имеет функтор, то есть имя, и аргументы. Например, sin(*X*) — это функция. Другим примером функции является функция mod(*X*, *Y*), которая возвращает остаток от деления *X* на *Y*. Когда вы хотите использовать функцию, то подставляете в переменные, или аргументы, конкретные значения. Например, если вы хотите найти остаток от деления 13 на 2, то можете напечатать mod(13, 2) в своем калькуляторе (если в нем, конечно, имеется эта функция). Если хотите найти sin(π/3), то можете набрать sin(3.1416/3).

Можно сказать, что функция — это отображение из множества всевозможных значений аргумента во множество результатов вычислений. Домен (т. е. область определения – *ред*. *пер*.) — это множество всевозможных значений аргумента. Образ — это множество результатов вычислений. Для функции синуса доменом является множество действительных чисел. Важно запомнить следующее. Математики настаивают, чтобы функция возвращала только одно значение для заданного аргумента. Поэтому если вычисления производят более одного значения, то это не функция. Например, значение выражения $\sqrt{4}$ может быть равно 2 или – 2[1]. Поэтому квадратный корень числа не является функцией. Однако вы можете сделать его соответствующим определению функции, взяв только неотрицательную часть образа[2].

Как быть с функциями нескольких аргументов? Например, функция max(*X, Y*) имеет два аргумента и возвращает значение наибольшего из них. В этом случае можно считать, что она имеет только один аргумент, который является парой элементов. Таким образом, аргументом max(5, 2) является пара (5, 2). Математики говорят, что областью определения такой функции является декартово произведение **R** $\times$ **R**.

Существуют функции, в которых функтор помещается между аргументами. Так обстоит дело в случае арифметических операций, где часто пишут 5 + 7 вместо + (5, 7).

5.2. Предикаты

Предикатами являются функции, домены которых отображаются в множество {`verum`, `falsum`}, или, если вам не нравятся латинские названия, используемые в логике, вы всегда можете полагаться на английский эквивалент: {`true`, `false`}. Существует несколько предикатов, известных любому, кто пробовал себя в программировании, или даже просто студенту, посещающему курс элементарной алгебры. 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`;

[1] Здесь автор не имеет в виду арифметический квадратный корень.

[2] То есть рассматривая знак радикала именно как арифметический квадратный корень.

X = Y есть `true`, если X равен Y, иначе `false`.

Предикат с одним аргументом говорит о свойстве или особенности этого аргумента. Можно сказать, что такой предикат работает как прилагательное. В языке C выражение `~X` возвращает `true`, если `X` есть `false`, в противном случае `~X` принимает значение `false`. Предикаты, эквивалентные этому, существуют и в других языках программирования. Приведем ещё несколько примеров одноместных предикатов:

`positive(X): true` , если `X` положительно, `false` иначе
`exists("text.txt"): true` , если файл *text.txt* существует, `false` иначе

Предикат, имеющий более одного аргумента, выражает отношение между ними. Например, для X = Y этим отношением является отношение равенства. Было бы интересно иметь язык программирования с предикатами, устанавливающими свойства и отношения, которые отличаются от тех немногих, что предлагают калькуляторы и основные языки программирования. Например, многие люди в течение всей своей жизни чувствовали непреодолимую потребность в предсказаниях одного из предикатов, приведенных на рисунке 5.1, особенно третьего предиката. Пролог — это язык программирования, который был изобретен для того, чтобы восполнить такую потребность.

`positive(X):` возвращает `true`, если X положителен, иначе `false`

`rain(Temperature, Pressure, Humidity)` возвращает `true`, если существует вероятность, что будет дождь при заданных температуре, давлении и влажности. Например

```
rain(100, 1.2, 90)
```

вернет `true`, т.е., вероятно, пойдет дождь, когда ваши измерительные приборы покажут 100°F, 1.2 атмосфер и 90% относительной влажности.

`invest(Rate, StandardDeviation, Risk).` Для заданной процентной ставки, стандартного отклонения и приемлемого риска этот предикат возвращает `true`, если вам стоит выбрать инвестирование.

Рисунок 5.1 Интересные предикаты

5.3. Решения

Предположим, что у вас есть предикат `city(Name, Point)`, который определяет координаты города на карте. Предикат `city/2` имеет домен[1]

```
city : (string Название, pnt Позиция).
```

[1] N.B. Предикатами являются функции, областью определения которых может быть любое декартово произведение, но образом является только множество {true, false}. – *прим. авт.*

и может быть определен как база фактов:

```
city("Salt Lake", pnt(30, 40)).
city("Logan", pnt(100, 120)).
city("Provo", pnt(100, 200)).
city("Yellowstone", pnt(200, 100)).
```

Этот предикат проверяет, является ли заданное положение заданного города верным, когда кто-либо в этом не уверен. Вот примеры запросов, которые можно задать с помощью предиката `city/2`:

```
city("Salt Lake", pnt(30, 40)) → true
city("Logan", pnt(100, 200)) → false
city("Provo", pnt(100, 200)) → true
```

Несомненно, вы могли бы найти применение для такого предиката. Однако предикат, который возвращает координаты города по его названию, был бы еще более полезным.

```
city("Salt Lake", P) → P= pnt(30, 40).
```

В этой новой разновидности предикатов слова, начинающиеся с заглавной буквы, называются переменными. Примеры переменных: *X, Y, Wh, Who, B, A, Xs, Temperature, Humidity, Rate*. Таким образом, если вы хотите узнать, является ли слово переменной, проверьте его первую букву. Если она заглавная или является знаком подчеркивания (_), значит, вы имеете дело с переменной.

Когда вы используете переменную, как например `P` в запросе `city("Salt Lake", P)`, вы хотите знать, что нужно подставить вместо `P`, чтобы значением предиката `city("Salt Lake", P)` было `true`. Ответом является `P=pnt(30, 40)`. Софокл сказал, что руки не должны быть быстрее ума, но и не должны отставать от него. Поэтому давайте определим предикат `city/2`.

- *Project Settings*. Войдите в диалоговое окно *Project Settings*, выбрав пункт *Project/New* меню задач среды, и заполните его.

```
General
    Project Name: mapDataBase
    UI Strategy: Object-oriented (pfc/gui)
    Target Type: Exe
    Base Directory: C:\Vispro
    Sub-Directory: mapDataBase\
```

- *Create Project Item: Form*. Выделите корень дерева проекта, затем выберите пункт меню *File/New*. В диалоговом окне *Create Project Item* выделите элемент *form* и заполните поле

```
Name: map
```

Добавьте в новую форму следующие кнопки: `Logan`, `SaltLake`, `Provo`.

- *Window Edit.* Измените размеры новой формы. Окно формы должно иметь достаточный размер, чтобы отобразить нашу «карту». Оставьте побольше пустого места в центре формы.
- *Build/Build.* Важный шаг: с помощью команды меню *Build/Build* постройте проект, иначе на следующем шаге система заявит об ошибке.
- *Project Tree/TaskMenu.mnu*. Включите пункт меню *File/New*.
- *Project Tree/TaskWindow.win/Code Expert*. Добавьте код

```
clauses
    onFileNew(S, _MenuTag) :-
        X = map::new(S), X:show().
```

для *Menu/TaskMenu/id_file/id_file_new/onFileNew*.

Постройте (*Build/Build*) проект снова (лучше перестраховаться, чем потом сожалеть).

- ***Создайте класс***. Создайте класс `draw` так, как это было объяснено в п. 4.4. Для того чтобы создать новый класс, выделите корень дерева проекта и выберите пункт меню *File / New in New Package*. Имя класса — *draw*, флажок в поле *Create Objects* снят. Постройте проект, для того чтобы вставить прототип нового класса в дерево проекта. Затем отредактируйте файлы `draw.cl` и `draw.pro` так, как показано на рисунках 5.2 и 5.3.

Для того чтобы вызывать предикат `drawThem` с помощью кнопок, обозначающих города, зайдите в дерево проекта и откройте форму `map.frm`, если она ещё не открыта. В диалоговом окне *Properties* выберите из списка компонентов кнопку `logan_ctl`, перейдите на вкладку *Event* и добавьте к обработчику *ClickResponder* следующий фрагмент кода:

```
clauses
    onLoganClick(S) = button::defaultAction() :-
        Parent = S:getParent(),
        P = Parent:getVPIWindow(),
        draw::drawThem(P, "Logan").
```

Повторите эти действия для городов “`Salt Lake`” и “`Provo`”. Не забудьте заменить название `logan_ctl` на названия `provo_ctl` и `saltlake_ctl`, соответственно. Замените также название города `Logan` в предикате `drawThem` на `Provo` или, соответственно, `Salt Lake`. Постройте проект и запустите программу. Если вы не помните, как строить и запускать программу, обратитесь к разделу 1.1.2. В новом приложении выберите пункт меню *File/New*. Появится новая форма. Когда вы нажмёте на какую-либо кнопку, программа отобразит соответствующий город.

```
% File: draw.cl
class draw
    open core, vpiDomains

predicates
    classInfo : core::classInfo.
    drawThem : (windowHandle, string) procedure.
end class draw
```

Рисунок 5.2 mapDataBase/draw.cl

```
% File:draw.pro
implement draw
    open core, vpiDomains, vpi

constants
    className = "draw".
    classVersion = "".

class facts
    city : (string, pnt).

clauses
    classInfo(className, classVersion).

    city("Salt Lake", pnt(30, 40)).
    city("Logan", pnt(100, 120)).
    city("Provo", pnt(100, 80)).
    city("Yellowstone", pnt(200, 100)).

    drawThem(Win, Name) :-
        B= brush(pat_solid, color_red),
        winSetBrush(Win, B),
        city(Name, P), !, P= pnt(X1, Y1),
        X2= X1+20, Y2= Y1+20,
        drawEllipse(Win, rct(X1, Y1, X2, Y2)).
    drawThem(_Win, _Name).
end implement draw
```

Рисунок 5.3 mapDataBase/draw.pro

5.4. Множественные решения

В предыдущем разделе вы видели примеры предикатов, которые возвращали решения через свои переменные, а не просто проверяли, истинно отношение или ложно. В приведенном выше примере предикат `city/2` использовался для получения только одного решения. Тем не менее, существуют ситуации, требующие большего количества решений. Пусть `conn/2` будет предикатом, устанавливающим связь между двумя городами.

```
conn(pnt(30, 40), pnt(100, 120)).
conn(pnt(100, 120), pnt(100, 200)).
conn(pnt(30, 40), pnt(200, 100)).
...
```

Вы можете использовать его для отыскания всех связей между городами, как показывает следующий пример.

```
conn(pnt(30, 40), W).      → W= pnt(100, 120)
                           → W= pnt(200, 100)
conn(X, Y).                → X= pnt(30, 40) / Y= pnt(100, 120)
                           → X= pnt(100, 120) / Y= pnt(100, 200)
                           → X= pnt(30, 40) / Y= pnt(200, 100)
```

Рассмотрим, например, запрос:

$$conn(pnt(30, 40), W)?$$

Ответом может быть как `W= pnt(100, 120)`, так и `W= pnt(200, 100)`.

5.4.1. Пример программы с множественными решениями

Давайте создадим программу, которая покажет, насколько замечательна возможность нахождения множественных решений в Прологе — возможность, отсутствующая в других языках.

- *Project Settings*. Зайдите в диалоговое окно *Project Settings*, выбрав команду *Project/New*, и заполните его следующим образом:

```
Project Name: drawMap
UI Strategy: Object-oriented GUI (pfc/gui)
```

- *Create Project Item: Package*. Выделите в дереве проекта элемент *drawMap*. Выберите пункт *File/New in New Package*. В диалоговом окне *Create Project Item* выберите элемент *Package* и заполните поля:

```
Name: plotter
Parent Directory:
```

- *Create Project Item: Form*. Выделите узел *plotter* дерева проекта. Выберите пункт *File/ New in New Package*. В диалоговом окне *Create Project Item* выберите пункт *Form*. Заполните поля:

```
Name: map
Package: plotter.pack (plotter\)
```

- ***Включите форму в проект***. Выберите *Build/Build* в меню задач.
- *Project Tree/TaskMenu.mnu*. Включите пункт меню *File/New*.
- *Project Tree/TaskWindow.win/Code Expert*. Добавьте код

```
clauses
    onFileNew(S, _MenuTag) :-
        F= map::new(S), F:show().
```

для *Menu/TaskMenu/id_file/id_file_new/onFileNew*.

- ***Создайте класс***. Создайте класс *draw*, как это было объяснено в разделе 4.4. Уберите галочку *Create Objects*. Поместите код, приведенный на рисунке 5.4, в файлы `draw.cl` и `draw.pro`. Постройте приложение.
- *Project Tree/map.frm*. Откройте `map.frm` и вставьте следующий код для *PaintResponder*:

```
clauses
    onPaint(S, _Rectangle, _GDIObject) :-
        W=S:getVPIWindow(),
        draw::drawThem(W).
```

Если вы построите и запустите программу, то при выборе команды *File/New* вы получите окно с картой, изображённое на рисунке 5.5.

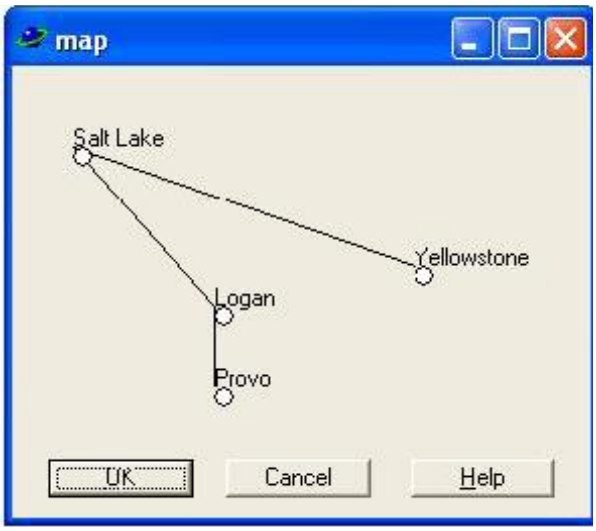


Рисунок 5.5 Города штата Юта

```
% Файл draw.cl
class draw
    open core, vpiDomains
predicates
    drawThem : (windowHandle) procedure.
end class draw

% Файл draw.pro
implement draw
    open core, vpiDomains, vpi
class facts
    city : (string Name, pnt Position).
    conn : (pnt, pnt).
class predicates
    connections : (windowHandle).
    drawCities : (windowHandle).
clauses
    city("Salt Lake", pnt(30, 40)).
    city("Logan", pnt(100, 120)).
    city("Provo", pnt(100, 160)).
    city("Yellowstone", pnt(200, 100)).

    conn(pnt(30, 40) , pnt(100, 120)).
    conn(pnt(100, 120), pnt(100, 160)).
    conn(pnt(30, 40), pnt(200, 100)).

    drawCities(W) :-
        city(N, P),
        P= pnt(X1, Y1),
        X2= X1+10, Y2= Y1+10,
        drawEllipse(W, rct(X1, Y1, X2, Y2)),
        drawText(W, X1, Y1, N), fail.
    drawCities(_Win).

    connections(Win) :- conn(P1, P2),
        drawLine(Win, P1, P2), fail.
    connections(_Win).

    drawThem(Win) :- connections(Win), drawCities(Win).
end implement draw
```

Рисунок 5.4 Файлы draw.cl и draw.pro

5.5. Логические связки

Я полагаю, что вы уже знакомы с логическим ***И***, которое имеется в таких языках, как C или Pascal:

```
if ((X>2) && (X<4)) { … }
```

Пусть P_1 и P_2 — предикаты. Тогда выражение «P_1 ***И*** P_2» истинно, если истинны оба выражения P_1 и P_2. Последовательность предикатов, соединённых логическим ***И***, называется конъюнкцией. В языке C выражение вида

```
(X>2) && (X<4)
```

является конъюнкцией. В Прологе предикаты конъюнкции разделяются запятыми. Поэтому выражение `(X>2) && (X<4)` принимает вид:

```
X>2, X<4
```

Логическое ***И*** называется связкой.

5.6. Импликация

Импликация — это связка, которая в Прологе изображается с помощью символа `:-`, что значит ***если***. Таким образом, правило

```
drawThem(Win) :- connections(Win), drawCities(Win).
```

означает, что вы выполните предикат `drawThem` на `Win`, если изобразите соединения `connections` на `Win` и города `drawCities` на `Win`.

5.7. Хорновские предложения

Существуют предложения Хорна, содержащие только один предикат. Например, ниже перечислены четыре однопредикатных предложения Хорна.

```
city("Salt Lake", pnt(30, 40)).
city("Logan", pnt(100, 120)).
city("Provo", pnt(100, 200)).
city("Yellowstone", pnt(200, 100)).
```

Однопредикатное предложение Хорна называется фактом. В нашем примере факты устанавливают отношение между городами и их координатами. Доменом предиката `city` является множество пар, состоящих из названия города и его координат. Предложение Хорна может также иметь вид:

H`:-`T_1`,` T_2`,` T_3`,` …

где T_i и H — предикаты. Так, запись

```
drawThem(Win) :- connections(Win), drawCities(Win).
```

является примером предложения Хорна. Часть предложения Хорна, расположенная до знака `:-`, называется *головой* (*head*), или *заголовком*. А часть предложения, находящаяся после знака `:-` называется *хвостом* (*tail*), или *телом*. В данном примере головой является `drawThem(Win)`, а хвостом — `connections(Win), drawCities(Win).`

Совокупность предложений Хорна с одинаковым заголовком определяет предикат. Например, четыре предложения Хорна о городах определяют предикат `city/2`, а предложение

```
drawThem(Win) :- connections(Win), drawCities(Win).
```

определяет предикат `drawThem/1`.

5.8. Объявления

Определение предиката не является полным без его объявления с указанием его типа и схемы входа-выхода параметров (*flow pattern*). Например, объявление предиката `drawThem/1` имеет вид:

```
predicates
    drawThem : (windowHandle) procedure (i).
```

Объявленный тип гласит, что аргумент `Win` предиката `drawThem(Win)` имеет тип `windowHandle`. Объявленная схема утверждает, что аргумент `drawThem/1` обязан быть входным, то есть, что он является константой, а не свободной переменной. В этом случае предикат принимает данные через свой аргумент. Вообще говоря, вы можете не указывать все схемы входа-выхода аргументов. Их самостоятельно выведет компилятор. Если он не сможет это сделать, то выдаст сообщение об ошибке, и вы сможете добавить необходимую схему. Если предполагается, что предикат будет использоваться только внутри его класса (и он не является объектным — *ред*. *пер*.), он объявляется как *class predicate* – предикат класса. Например:

```
class predicates
    connections : (windowHandle).
    drawCities : (windowHandle).
```

Однопредикатные предложения Хорна могут быть объявлены как факты. Например:

```
class facts
    city : (string Name, pnt Position).
    conn : (pnt, pnt).
```

Позднее вы увидите, что факты могут быть добавлены в базу данных или удалены из нее. Аргументами предиката `conn/2` является пара значений типа `pnt`. Тип `pnt` определен в классе `vpiDomains`. У меня есть два способа, как использовать его в классе `draw`. Я могу сделать подробное объявление и указать класс, которому принадлежит `pnt`:

```
class facts
    conn : (vpiDomains::pnt, vpiDomains::pnt).
```

Либо я могу открыть этот класс внутри класса `draw`. Я выбрал второй вариант:

```
open core, vpiDomains, vpi
```

5.8.1. Объявление режимов детерминизма

Объявление режимов детерминизма предиката, которые указывают на то, имеет ли он единственное решение или может иметь много решений, осуществляется с помощью следующих ключевых слов.

determ

Выполнение детерминированного предиката может завершиться либо неудачно (*fail*), либо успешно (*succeed*) и при этом иметь одно решение.

procedure

Предикаты этого вида всегда завершаются успешно и имеют одно решение. Предикаты `connections/1` и `drawCities/1`, определенные на рисунке 5.4, являются процедурами и могут быть объявлены так:

```
class predicates
    connections : (windowHandle) procedure (i).
    drawCities : (windowHandle) procedure (i).
```

multi

Такой предикат не может завершиться неудачно, при этом он имеет множество решений.

nondeterm

Недетерминированный предикат может завершиться либо неудачно, либо успешно, и при этом иметь множество решений. Предикаты `city/2` и `connection/2` имеют тип *nondeterm* и могли бы иметь следующее объявление:

```
class facts
    city : (string Name, pnt Position) nondeterm.
    conn : (pnt, pnt) nondeterm.
```

Если предикат имеет много решений и одно из его решений не в состоянии удовлетворить другой предикат в той же конъюнкции, то Пролог откатывается (*backtrack*) и предлагает другое решение в попытке удовлетворить эту конъюнкцию. Рассмотрим следующее предложение Хорна:

```
connections(Win) :-
    conn(P1, P2),
    drawLine(Win, P1, P2), fail.
connections(_Win).
```

Недетерминированный предикат `conn(P1, P2)` возвращает две точки, которые использует процедура `drawLine(Win, P1, P2)`, соединяющая эти точки прямой линией. Затем Пролог пытается удовлетворить предикат `fail`, который всегда неуспешен, как и показывает его имя. В результате происходит откат, и Пролог пытается найти другое решение, до тех пор, пока не исчерпываются все варианты. После этого он переходит ко второму предложению для предиката `connection/1`, которое всегда приводит к успеху.

5.9. Предикаты рисования

Мой шестнадцатилетний сын думает, что языки программирования используются только для создания игр, рисования, графики и тому подобного. Если вы придерживаетесь того же мнения, вам необходимо глубоко изучить предикаты рисования. Для них потребуется дескриптор (*handle*[1]) окна, в котором будут выполняться рисунки и чертежи. Вот как вы можете получить дескриптор:

```
clauses
    onPaint(S, _Rectangle, _GDIObject) :-
        W= S:getVPIWindow(), draw::drawThem(W).
```

Дескриптор `W` будет передаваться в класс `draw`. Например, в предложении

```
drawThem(Win) :- connections(Win), drawCities(Win).
```

он передается в `connections/1`, в котором является первым аргументом `drawLine/3`:

```
connections(Win) :- conn(P1, P2), drawLine(Win, P1, P2), fail.
```

Предикат `drawLine(Win, P1, P1)` проводит линию из `P1` в `P2` в окне `Win`. Как вы уже знаете, `P1` и `P2` — точки вида `pnt(10, 20)`. Предикат

```
drawEllipse(W, rct(X1, Y1, X2, Y2)),
```

с которым вы уже знакомы, проводит эллипс в окне `W`. Эллипс вписывается в прямоугольник `rct(X1, Y1, X2, Y2)`, где `X1, Y1` — координаты верхнего левого угла, а `X2, Y2` — координаты нижнего правого угла.

5.10. Объект GDI

В последнем примере рисование производилось в обработчике события `onPaint`. В этом случае хорошей идеей может быть использование методов так называемого *GDI-объекта*. Рассмотрим следующий пример.

- *Project Settings*. Создайте следующий проект:

```
Project Name: drawMapObj
UI Strategy: Object-oriented GUI (pfc/gui)
Target type: Exe
Base Directory: C:\vispro
```

- ***Создайте пакет***: `plotter`.
- ***Создайте форму внутри*** `plotter: map`.
- *Project Tree/TaskMenu.mnu*. Включите пункт меню *File/New*. Постройте (*Build/Build*) приложение, для того чтобы включить форму `map` в проект.
- *Project Tree/TaskWindow.win/Code Expert*. Добавьте код

[1] На данный момент вам не обязательно знать, что такое *handle* – *прим. авт.*

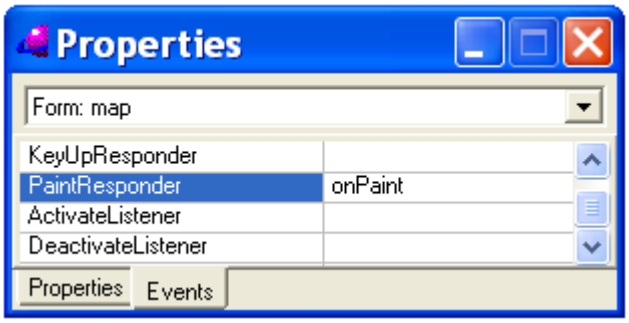
```
clauses
    onFileNew(S, _MenuTag) :-
        F= map::new(S), F:show().
```

для *Menu/TaskMenu/id_file/id_file_new/onFileNew*.

Создайте класс. Создайте класс `draw` внутри пакета `plotter`. Не забудьте отключить флажок *Create objects*. Вы найдете новую версию класса `draw` на рисунке 5.6. Постройте приложение.

Добавьте к обработчику *PaintResponder* формы `map.frm`. код

```
clauses
    onPaint(_S, _Rectangle, GDIObject) :-
        draw::drawThem(GDIObject).
```



5.11. Примеры

В этой главе имеются три примера: *mapDataBase* рисует положение трёх главных городов штата Юта, когда нажимается соответствующая городу кнопка; *drawMap* рисует карту направлений дорог Юты; *drawMapObj* является *GDI*-версией *drawMap*.

```
% Файл draw.cl
class draw
    open core
predicates
    drawThem : (windowGDI).
end class draw

% Файл draw.pro
implement draw
    open core, vpiDomains, vpi

class facts
    city : (string Name, pnt Position).
    conn : (pnt, pnt).

class predicates
    connections : (windowGDI).
    drawCities : (windowGDI).

clauses
    city("Salt Lake", pnt(30, 40)).
    city("Logan", pnt(100, 120)).
    city("Provo", pnt(100, 160)).

    conn(pnt(30, 40) , pnt(100, 120)).
    conn(pnt(100, 120), pnt(100, 160)).

    drawCities(W) :-
        city(N, P),
        P= pnt(X1, Y1),
        X2= X1+10, Y2= Y1+10,
        W:drawEllipse(rct(X1, Y1, X2, Y2)),
        W:drawText(pnt(X1, Y1), N), fail.
    drawCities(_Win).

    connections(W) :-
        conn(P1, P2),
        W:drawLine(P1, P2), fail.
    connections(_W).

    drawThem(Win) :- connections(Win),
        drawCities(Win).
end implement draw
```

Рисунок 5.6 Класс draw.

5.12. Немного о логике: Смысл предложений Хорна

Предложение Хорна имеет вид:

H :- T_1, T_2, T_3...

Смысл предложения состоит в следующем:

Если T_1 & T_2 & T_3 истинно,
то H также истинно.

Ранее вы видели, что это утверждение эквивалентно следующему:

H является истинным или конъюнкция T_1 & T_2 & T_3 не является истинной.

Последняя интерпретация проливает свет на значение символа, изображающего «шею» предложения: двоеточие заменяет связующее «или», а знак минус — логическое отрицание. Поэтому утверждение

H :- T

означает: *H* истинно или не-*T* истинно. Для того чтобы закончить наше изучение связок, используемых в Прологе, я напомню, что запятая заменяет ***И***, точка с запятой заменяет ***ИЛИ***, а знак **->** заменяет *if...then* – ***ЕСЛИ...ТО***.

Глава 6: Консольные приложения

Мы начали изучение Пролога с графических приложений, потому что большинство людей предпочитают пользоваться именно ими. Однако графические приложения добавляют детали, которые отвлекают ваше внимание от действительно важных вещей. Поэтому давайте рассмотрим несколько базовых схем программирования, не рассчитывая на графический интерфейс.

6.1. Отсечение

Что вам нужно сделать, если вы хотите, чтобы система не делала откат и не пыталась перейти к другому предложению после того, как найдёт решение? В этом случае вы ставите восклицательный знак в хвосте предложения Хорна. После того, как система найдет восклицательный знак, она прерывает поиск новых решений. Восклицательный знак называется *отсечением* (*cut*). Давайте проиллюстрируем использование отсечений очень популярным среди программистов на Прологе алгоритмом. Предположим, что вы хотите написать предикат, который находит факториал числа. Математики определяют факториал так:

$$\text{factorial}(0) \rightarrow 1$$
$$\text{factorial}(n) \rightarrow n \times \text{factorial}(n-1)$$

На языке предложений Хорна это определение становится следующим:

```
fact(N, 1) :- N<1, !.
fact(N, N*F1) :- fact(N-1, F1).
```

Отсечение предотвращает попытку Пролога применить второе предложение для `N = 0`. Другими словами, для запроса

```
fact(0, F)?
```

программа успешно использует первое предложение при `N = 0` и ответит, что `F = 1`. Без отсечения она могла бы предположить, что второе предложение определения

```
fact(N, 1) :- N<1.
fact(N, N*F1) :- fact(N-1, F1).
```

тоже приведет к решению. Тогда она попробовала бы использовать его и сбилась бы. Отсечение обеспечивает то, что факториал является функцией и отображает каждое значение домена в одно и только одно значение из образа. Для того чтобы реализовать вычисление факториала, следуйте следующим указаниям:

- ***Создание нового проекта***. Выберите пункт *Project/New* и заполните диалоговое окно *Project Settings* так:

```
General
```

```
Project Name: facfun
UI Strategy: console
```

Обратите внимание, что мы собираемся использовать консольную стратегию, а не GUI.

- ***Компиляция.*** Выберите пункт *Build/Build* главного меню, чтобы внести прототип класса `facfun` в дерево проекта. Отредактируйте файл `main.pro` так, как показано ниже.

```
% Файл main.pro
implement main
    open core

    class predicates
        fact : (integer N, integer Res) procedure (i,o).
    clauses
        classinfo("facfun", "1.0").

        fact(N, 1) :- N<1, !.
        fact(N, N*F) :- fact(N-1, F).

        run():- console::init(),
            fact(stdio::read(), F), stdio::write(F), stdio::nl.
end implement main

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

Снова постройте программу и запустите её, используя команду *Run in Window*. Напишите число в приглашении к вводу, и вы получите его факториал. **NB**: Чтобы протестировать консольную программу, используйте команду *Run in Window*, не *Execute*[1].

6.2. Списки

Существует книга о языке Lisp, в которой говорится: список — это упорядоченная последовательность элементов, где слово *упорядоченная* означает, что порядок перечисления элементов имеет значение. В Прологе список заключается в квадратные скобки и обычно имеет голову (*head*) и хвост (*tail*):

Список	Тип	Голова	Хвост
[3, 4, 5, 6, 7]	Integer	3	[4, 5, 6, 7]
[“wo3”, “ni3”, “ta1”]	String	“wo3”	[“ni3”, “ta1”]
[4]	Integer	4	[]
[3.4, 5.6, 2.3]	Real	3.4	[5.6, 2.3]

[1] Для того чтобы можно было использовать команду *Execute*, достаточно добавить в тело правила для предиката run/0 в качестве последней подцель _ = stdio::readLine(), которая будет ожидать от пользователя ввода любой строки.

Вы можете соотнести образец переменных со списком. Например, если вы соотнесете шаблон

```
[X|Xs]
```

со списком `[3.4, 5.6, 2.3]`, вы получите, что `X=3.4` и `Xs=[5.6, 2.3]`, то есть, что переменная `X` соответствует голове списка, а переменная `Xs` соответствует хвосту. Конечно, вы можете использовать другую пару переменных вместо `X` и `Xs` в шаблоне `[X|Xs]`. Так, шаблоны `[A|B]`, `[X|L]`, `[Head|Tail]`, `[First|Rest]` и `[P|Q]` являются эквивалентными шаблонами. Вот ещё несколько примеров соответствий для шаблона `[X|Xs]`:

Шаблон	Список	X	Xs
[X|Xs]	[3.4, 5.6, 2.3]	3.4	[5.6, 2.3]
[X|Xs]	[5.6, 2.3]	5.6	[2.3]
[X|Xs]	[2.3]	2.3	[]
[X|Xs]	[]	*Не соответствуют*	

Как вы можете видеть, шаблон `[X|Xs]` соответствует списку, содержащему как минимум один элемент. Вот шаблон, который соответствует спискам, содержащим по крайней мере два элемента:

Шаблон	Список	X1, X2	Xs
[X1, X2|Xs]	[3, 5, 2, 7]	X1=3, X2=5	[2, 7]
[X1, X2|Xs]	[2, 7]	X1=2, X2=7	[]
[X1, X2|Xs]	[7]	*Не соответствуют*	
[X1, X2|Xs]	[]	*Не соответствуют*	

Создадим маленький проект `avg`, вычисляющий длину списка действительных чисел. Как и в случае проекта `factorial` убедитесь, что `avg` — консольное приложение.

```
General
    Project Name: avg
    UI Strategy: console
```

Скомпилируйте проект, чтобы вставить класс `avg` в дерево проекта. Затем отредактируйте файл `avg.pro` так, как показано ниже. Запустите проект, используя команду *Run in Window*, как и ранее.

```
% Файл main.pro
implement main
    open core, console
domains
    rList= real*.
class predicates
    len:(rList, real) procedure (i, o).
clauses
    classInfo("avg", "1.0").

    len([], 0) :- !.
```

```
    len([_X|Xs], 1.0+S) :- len(Xs, S).

    run():-
        console::init(),
        List= read(),
        len(List, A),
        write(A), nl.
end implement main
goal
    mainExe::run(main::run).
```

Рассмотрим понятия, стоящие за этой маленькой программой. Первое, что вы сделали, это создали домен:

```
domains
    rList=real*.
```

Затем вы определили предикат `len/2`, который через первый аргумент получает список, а с помощью второго аргумента выводит его длину. Наконец, вы определили предикат `run()`, чтобы протестировать программу:

```
run() :-
    console::init(),            % Проинициализировали консоль
    List=read(),                % Прочитали список List с консоли
    len(List, A),               % Нашли длину списка
    write(A), nl.               % Вывели длину на экран
```

Объявление домена бывает необходимо, если вы имеете дело с составными алгебраическими типами данных. Однако этого не требуется для таких простых структур, как список действительных чисел. Ниже вы найдёте ту же программу без объявления домена.

```
% Файл main.pro
implement main
    open core, console
 class predicates
    len : (rList, real) procedure (i, o).
clauses
    classInfo("avg", "1.0").

    len([], 0) :- !.
    len([_X|Xs], 1.0+S) :- len(Xs, S).

    run():-
        console::init(),
        List= read(),
        len(List, A),
        write(A), nl.
end implement main
goal
    mainExe::run(main::run).
```

Эта программа имеет следующий недостаток: предикат `len/2` можно использовать только для подсчёта длины списка действительных чисел. Было бы неплохо, если бы предикат `len/2` мог принимать список любого типа. Это станет возможным, если вместо домена `real*` подставить переменную типа. Однако, прежде чем проверять, действительно ли эта схема работает, необходимо придумать для `L=read()` способ извещения того, *каким именно* будет конечный тип вводимого списка, и передать эту информацию в `len/2`. К счастью, существует предикат, который создаёт переменную любого заданного типа. Ниже вы узнаете, как его использовать.

```
implement main /* В файле main.pro */
    open core, console
class predicates
    len : (Element*, real) procedure (i, o).
clauses
    classInfo("avg", "1.0").

    len([], 0) :- !.
    len([_X|Xs], 1.0+S) :- len(Xs, S).

    run():- console::init(),
        hasDomain(real_list, L), L= read(),
        len(L, A), write(A), nl.
end implement main
goal mainExe::run(main::run).
```

Следующим шагом в наших рассуждениях является добавление предложений для предиката `sum/2` в файл `main.pro`:

```
    sum([], 0) :- !.
    sum([X|Xs], S+X) :- sum(Xs, S).
```

Давайте посмотрим, что произойдёт, если вызвать `sum([3.4, 5.6, 2.3], S)`.

1. Вызов `sum([3.4, 5.6, 2.3], S)`
   соответствует предложению `sum([X|Xs], S+X) :- sum(Xs, S)`, где `X=3.4`, `Xs=[5.6, 2.3]`,
   возвращая `sum([3.4, 5.6, 2.3], S[5.6, 2.3]+3.4) :- sum([5.6, 2.3], S[5.6, 2.3])`
2. Вызов `sum([5.6, 2.3], S[5.6, 2.3])`
   соответствует предложению `sum([X|Xs], S+X) :- sum(Xs, S)`, где `X=5.6`, `Xs=[2.3]`,
   возвращая `sum([5.6, 2.3], S[2.3]+5.6) :- sum([2.3], S[2.3])`
3. Вызов `sum([2.3], S[2.3])`
   соответствует предложению `sum([X|Xs], S+X) :- sum(Xs, S)`, где `X=2.3`, `Xs=[]`,
   возвращая `sum([2.3], S[]+2.3) :- sum([], S[])`
4. Вызов sum([], $S_{[]}$), соответствует предложению `sum([], 0.0) :- !`, возвращая `S[]=0`.

Достигнув конца списка, компьютер должен откатиться к началу вычислений. Что еще хуже, он должен сохранить каждое значение переменной `X`, которое он найдет по пути, чтобы произвести сложение `S+X` во время возвращения. Традиционный путь

предотвращения этого — использование накопителя. Вот определение, которое использует накопитель для суммирования элементов списка:

```
add([], A, A).
add([X|Xs], A, S) :- add(Xs, X+A, S).
```

Давайте посмотрим, как компьютер использует второй вариант суммирования элементов списка.

1. `add([3.4, 5.6, 2.3], 0.0, S)`
   соответствует предложению `add([X|Xs], A, S) :- add(Xs, X+A, S)`,
   возвращая `add([5.6, 2.3], 0+3.4, S)`
2. `add([5.6, 2.3], 0.0+3.4, S)`
   соответствует предложению `add([X|Xs], A, S) :- add(Xs, X+A, S)`
   возвращая `add([2.3], 0+3.4+5.6, S)`
3. `add([2.3], 0.0+3.4+5.6, S)`
   соответствует предложению `add([X|Xs], A, S) :- add(Xs, X+A, S)`
   возвращая `add([], 0+3.4+5.6+2.3, S)`,
   что соответствует предложению `add([], A, A)`, возвращая `S=11.3`.

Вы можете использовать `add/3` для вычисления среднего арифметического значений элементов списка чисел.

```
len([], 0) :- !.
len([_X|Xs], 1.0+S) :- len(Xs, S).

add([], A, A) :- !.
add([X|Xs], A, S) :- add(Xs, X+A, S).

sum(Xs, S) :- add(Xs, 0.0, S).

avg(Xs, A/L) :- sum(Xs, A), len(Xs, L).
```

Описанная выше программа проходит через список дважды: один раз, чтобы подсчитать сумму, и другой, чтобы найти длину. Используя два накопителя, вы можете вычислять длину и сумму одновременно.

```
% Файл main.pro
implement main
    open core, console
class predicates
    avg : (real*, real, real, real) procedure (i, i, i, o).
clauses
    classInfo("avg", "1.0").

    avg([], S, L, S/L).
    avg([X|Xs], S, L, A) :-
        avg(Xs,
            X+S,    % Добавление X к накопителю суммы
            L+1.0,  % Увеличение на единицу накопителя длины
            A).

    run():- console::init(),
```

```
        List= read(),
        avg(List, 0, 0, A),
        write(A), nl.
end implement main

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

Код, приведенный выше, демонстрирует программу для вычисления среднего арифметического элементов списка действительных чисел. При этом используются два накопителя — один для суммы, а другой для подсчета количества элементов.

6.3. Схемы обработки списков

Рассмотрим несколько важных схем программирования списков. Я считал вполне очевидным, что их изобрел не я. Однако, так как я получил массу писем с вопросами о происхождении этих схем, я добавил два пункта в список литературы: [John Hughes] и [Wadler & Bird].

Редукция

Схема, используемая для вычисления суммы элементов списка, называется *редукцией*, т. к. она сокращает размерность входных данных. Фактически списки можно считать одномерными данными, а сумму — величиной нулевой размерности. Имеются два способа выполнения редукции: рекурсивный и итеративный.

```
% Рекурсивная редукция
class predicates
    sum :(real*, real) procedure (i, o).
clauses
    sum([], 0) :- !.
    sum([X|Xs], S+X) :- sum(Xs, S).
```

Термин ***итеративный*** восходит к латинскому слову *iterum*, которое означает ***снова***. Вы также можете представлять, что он восходит к *iter/itineris* – ***путь, дорога***. Итеративная программа проходит через код снова и снова.

```
% Итеративная редукция
red([], A, A) :- !.
red ([X|Xs], A, S) :- red (Xs, X+A, S).
sumation(Xs, S) :- add(Xs, 0.0, S).
```

Предположим, что вместо сложения вы хотите произвести умножение. В этом случае вы можете написать следующую программу:

```
% Итеративная редукция
red([], A, A) :- !.
red ([X|Xs], A, S) :- red (Xs, X*A, S).
sumation(Xs, S) :- add(Xs, 0.0, S).
```

Два этих фрагмента кода идентичны, за исключением одной детали: в первом из них во втором аргументе итеративного вызова предиката `red/3` находится выражение `X+A`, в то время как в другом — `X*A`. [Wadler & Bird] и [John Hughes] предложили использовать подобный тип сходства в предикатах и функциях высших порядков. Еще до них [John Backus] сделал такое же предложение, когда он получил премию Тьюринга. Посмотрим, как можно использовать этот выгодный метод программирования на Visual Prolog.

В программе, приведённой на рисунке 6.1, я объявил домен двухместных функций. В разделе *class predicates*, я объявил некоторые функции, принадлежащие этому домену. Наконец, я определил предикат `red/4`, который захватывает структуру операции редукции.

```
implement main
    open core
domains
    pp2 = (real Argument1, real Argument2)
        -> real ReturnType.
clauses
    classInfo("main", "hi_order").

class predicates
    sum : pp2.
    prod : pp2.

    red : (pp2, real*, real, real)
        procedure (i, i, i, o).

clauses
    sum(X, Y)= Z :- Z=X+Y.
    prod(X, Y)= Z :- Z=X*Y.

    red(_P, [], A, A) :- !.
    red(P, [X|Xs], A, Ans) :- red(P, Xs, P(X, A), Ans).

    run():- console::init(),
        red(prod, [1,2,3,4,5], 1, X),
        stdio::write(X), stdio::nl,
        succeed().
end implement main

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

Рисунок 6.1. Редукция в Прологе

«Молния» (ZIP)

Это устройство, которой используется в сплошной застежке для одежды, изобретенной канадским инженером Гидеоном Сандбэком (*Gideon Sundbac*). Это устройство сводит с ума, когда не работает, и является причиной такого количества инцидентов с подростками, что врачам и медсёстрам потребовалось специальное обучение, чтобы разбираться с этими устройствами.

```
class predicates
    dotproduct : (real*, real*, real) procedure (i, i, o).
clauses
    dotproduct([], _, 0) :- !.
    dotproduct(_, [], 0) :- !.
    dotproduct([X|Xs], [Y|Ys], X*Y+R) :- dotproduct(Xs, Ys, R).
```

Пример показывает применение этой схемы (*zip scheme*) с последующим шагом редукции для вычисления скалярного произведения двух списков. В выражении `X*Y+R` умножение используется для закрытия одного зажима молнии.

Мы можем захватить эту схему в предикат более высокого порядка точно так же, как захватили схему редукции. В действительности, мы можем закрывать каждый зажим застёжки с помощью функций того же рода, что мы использовали для осуществления редукции. На рисунке 6.2 вы можете найти (не очень эффективную) реализацию `zip` в Прологе. Я добавил этот код, потому что не один человек написал мне, запрашивая более глубокий разбор этой важной темы. Я считаю, что не могу предоставить более глубокий разбор, так как это выходит за пределы этой книги. В конце концов, моя цель заключается в том, чтобы избегать чрезмерно крутых кривых обучения для начинающих. В то же время, примеров, приведенных на рисунках 6.1 и 6.2, должно быть достаточно для того чтобы показать, как писать предикаты высших порядков в Visual Prolog. Люди, которые хотят изучить возможности логики высшего порядка и функций высших порядков, могут обратиться к книгам таких авторов, как [Wadler & Bird] или статьям [John Hughes].

Предикат `zip` имеет два аргумента. Первый аргумент является двухместной функцией. В листинге 6.2 я представляю две из таких функций, но вы можете определить и другие функции. Второй и третий аргументы содержат списки, которые должны быть застёгнуты (соединены) друг с другом. Четвёртый аргумент содержит результат операции. Определение `zip/4` выглядит просто:

```
zip(_P, [], _, []) :- !.
zip(_P, _, [], []) :- !.
zip(P, [X|Xs], [Y|Ys], [Z|As]) :-
    Z= P(X, Y), zip(P, Xs, Ys, As).
```

Когда какой-либо из входных списков станет пустым, одно из первых двух предложений остановит рекурсию, положив пустой список в качестве результата. В противном случае третье предложение сцепит головы двух входных списков и продолжит обработку их оставшихся частей. Я полагаю, что для начала этого достаточно.

```
imlement main
    open core
domains
    pp2 = (real Argument1, real Argument2) -> real ReturnType.
clauses
    classInfo("main", "zip").
class predicates
    sum : pp2.
    prod : pp2.
    red : (pp2, real*, real, real) procedure (i, i, i, o).
    zip : (pp2, real*, real*, real*) procedure (i, i, i, o).
    dotprod : (real*, real*, real) procedure (i, i, o).
clauses
    sum(X, Y)= Z :- Z=X+Y.
    prod(X, Y)= Z :- Z=X*Y.

    zip(_P, [], _, []) :- !.
    zip(_P, _, [], []) :- !.
    zip(P, [X|Xs], [Y|Ys], [Z|As]) :-
        Z= P(X, Y), zip(P, Xs, Ys, As).

    red(_P, [], A, A) :- !.
    red(P, [X|Xs], A, Ans) :- red(P, Xs, P(X, A), Ans).

    dotprod(Xs, Ys, D) :-
        zip(prod, Xs, Ys, Z), red(sum, Z, 0, D).

    run():- console::init(),
        V1= [1,2,3,4,5], V2= [2,2,2,2,2],
        dotprod( V1, V2, X),
        stdio::write(X), stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

Рисунок 6.2 Реализация Zip в Прологе

6.4. Обработка строк

Ниже вы найдете несколько примеров операций со строками. Этих примеров должно быть достаточно для того, чтобы показать вам, как разбираться с подобными структурами данных. Вы можете найти больше операций с помощью справочной системы Visual Prolog.

```
implement main
    open core, string
class predicates
    tokenize:(string, string_list) procedure (i, o).
clauses
    classInfo("main", "string_example").

    tokenize(S, [T|Ts]) :-
        frontToken(S, T, R), !, tokenize(R, Ts).
    tokenize(_, []).

    run():- console::init(),
        L= ["it ", "was ", "in ", "the ", "bleak ", "December!"],
        S= concatList(L), UCase= toUpperCase(S),
        RR= string::concat("case: ", Ucase),
        R1= string::concat("It is ", "upper ", RR),
        stdio::write(R1), stdio::nl, tokenize(R1, Us),
        stdio::write(Us), stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

Попытайтесь понять, как работает `frontToken`, потому что это весьма полезный предикат. Он используется для разбиения строки на *токены* (*token*) в процессе, называемом лексическим анализом. Например, для вызова

```
frontToken("break December", T, R)
```

вы получите `T="break"` и `R=" December"`.

Часто требуется преобразовать строковое представление терма в другое, необходимое для работы. В этом случае можно использовать функцию `toTerm`. Ниже в примере `Sx=stdio::readLine()` считывает строку из командной строки в `Sx`. Затем `toTerm` преобразует строку в действительное число. Вызов `hasdomain(real, IX)` обеспечивает то, что `toTerm` вернёт действительное число.

```
implement main
clauses
    classinfo("main", "toTerm_example").

    run():- console::init(),
        Sx= stdio::readLine(),
        hasdomain(real, IX),
        IX= toTerm(Sx),
        stdio::write(IX^2).
```

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

Ниже вы сможете увидеть, что эта схема работает также для списков и других составных структур данных. К сожалению, `hasdomain` не принимает знак звездочки '*' в имени домена. Таким образом, вы должны объявить домен списка самостоятельно или использовать заранее объявленный домен, такой как `core::integer_list`.

```
implement main
clauses
    classInfo("main", "hasdomain_example").

    run():- console::init(),
        hasdomain(core::integer_list, Xs),
        Xs= toTerm(stdio::readLine()),
        stdio::write(Xs), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

Вы научились преобразовывать строковое представление типов данных в термы Пролога. Теперь посмотрим, как выполнить обратную операцию, т. е. преобразовать терм в его строковое представление. Если вам нужно стандартное представление терма, то вы можете использовать предикат `toString`.

```
implement main
clauses
    classInfo("main", "toString_example").

    run():- console::init(),
        Xs= [3.4, 2, 5, 8],
        Str= toString(Xs),
        Msg= string::concat("String representation: ", Str),
        stdio::write(Msg), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

Если вы хотите правильно отформатировать числа или другие простые термы при преобразовании их в строку, вы можете использовать функцию форматирования `string::format`. Пример приведён ниже.

```
implement main
clauses
    classInfo("main", "formatting").

    run():- console::init(),
        Str1= string::format("%8.3f\n %10.1e\n", 3.45678, 35678.0),
        Str2= string::format("%d\n %10d\n", 456, 18),
        Str3= string::format("%-10d\n %010d\n", 18, 18),
        stdio::write(Str1, Str2, Str3).
end implement main
goal
    mainExe::run(main::run).
```

В примере, приведенном выше, формат `"%8.3f\n"` означает, что я хочу отобразить действительное число и выполнить возврат каретки; ширина поля, отведённого для числа, равна 8 знакам, число должно быть отображено с тремя знаками после запятой. Формат `"%010d\n"` выражает требование целого числа, выровненного по правому краю поля шириной 10; пустые места поля должны быть заполнены нулями. Формат `"%-10d\n"` определяет представление целого числа, выровненного по левому краю поля шириной 10; знак минус указывает на левое выравнивание, правое выравнивание установлено по умолчанию. Формат `"%10.1e\n"` определяет научную запись для действительных чисел. Ниже вы можете увидеть вывод на экран:

```
   3.457
   3.6e+004
456
         18
18
 0000000018
```

Еще ниже вы найдёте список типов данных, принимаемых форматируемой строкой. Заметим, что при конвертации действительных чисел в текстовое представление они обрезаются и округляются до 17 цифр, если не была указана другая точность.

`f`- Форматировать как действительное число с фиксированной запятой (как 123.4).
`e`-Форматировать действительное число в экспоненциальной записи (как 1.234e+002).
`g`- Форматировать в либо формате `f`, либо в формате `e` — как запись будет короче.
`d`- Форматировать как знаковое целое.
`u`- Форматировать как беззнаковое целое.
`x`- Форматировать как шестнадцатеричное число.
`o`- Форматировать как восьмеричное число.
`c`- Форматировать как символ.
`B`- Форматировать как двоичный тип Visual Prolog.
`R`- Форматировать как номер ссылки базы данных.
`P`- Форматировать как параметр процедуры.
`s`- Форматировать как строку.

6.4.1. Полезные предикаты для работы со строками

Ниже вы найдёте список других предикатов, которые могуть быть полезны при разработке ваших приложений. В них параметры `adjustBehaviour`, `adjustSide` и `caseSensivity` имеют следующие определения:

```
domains
    adjustBehaviour =
        expand();
        cutRear();
        cutOpposite().
    adjustSide = left(); right().
    caseSensitivity =
        caseSensitive();
```

```
        caseInsensitive();
        casePreserve().

adjust : (string Src, charCount FieldSize, adjustSide Side)
        -> string AdjustedString.
adjust : (string Src, charCount FieldSize, string Padding, adjustSide
    Side)
        -> string AdjustedString.
adjustLeft : (string Src, charCount FieldSize)
        -> string AdjustedString.
adjustLeft : (string Src, charCount FieldSize, string Padding)
        -> string AdjustedString.
adjustRight : (string Src, charCount FieldSize)
        -> string AdjustedString.
adjustRight : (string Src, charCount FieldSize, string Padding)
        -> string AdjustedString.
adjustRight : (string Src, charCount FieldSize, string Padding,
            adjustBehaviour Behaviour) -> string AdjustedString.

/*********************************
Project Name: adjust_example
UI Strategy: console
*********************************/
implement main
clauses
    classInfo("main", "adjust_example").
    run():- console::init(),
       FstLn= "Rage --"
           " Goddess, sing the rage of Peleus' son Achilles,",
        Str= string::adjust(FstLn, 60, "*", string::right),
        stdio::write(Str), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

```
****Rage -- Goddess, sing the rage of Peleus' son Achilles,

F:\Project\Visual Prolog Projects\Container\Exe>pause
Для продолжения нажмите любую клавишу . . .
```

```
concat : (string First, string Last)
        -> string Output procedure (i,i).
concatList : (core::string_list Source)
        -> string Output procedure (i).
concatWithDelimiter : (core::string_list Source, string Delimiter)
        -> string Output procedure (i,i).
```

```
create : (charCount Length) -> string Output procedure (i).
create : (charCount Length, string Fill)
        -> string Output procedure (i,i).
createFromCharList : (core::char_list CharList)
        -> string String.
/*********************************
Project Name: stringops
UI Strategy: console
*********************************/
implement main
clauses
    classInfo("main", "stringops").
    run():-
        console::init(),
        SndLn= [ "murderous", "doomed",
            "that cost the Achaeans countless losses"],
        Str= string::concatWithDelimiter(SndLn, ", "),
        Rule= string::create(20, "-"),
        stdio::write(Str), stdio::nl,
        stdio::write(Rule), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

```
murderous, doomed, that cost the Achaeans countless losses
--------------------

F:\Project\Visual Prolog Projects\Container\Exe>pause
Для продолжения нажмите любую клавишу . . .
```

```
equalIgnoreCase : (string First, string Second) determ (i,i).
front : (string Source, charCount Position, string First, string Last)
        procedure (i,i,o,o).
frontChar : (string Source, char First, string Last) determ (i,o,o).
frontToken : (string Source, string Token, string Remainder) determ (i,
o,o).
getCharFromValue : (core::unsigned16 Value) -> char Char.
getCharValue : (char Char) -> core::unsigned16 Value.
hasAlpha : (string Source) determ (i).
hasDecimalDigits : (string Source) determ (i).
hasPrefix : (string Source, string Prefix, string Rest) determ (i,i,o).
hasSuffix : (string Source, string Suffix, string Rest) determ (i,i,o).
isLowerCase : (string Source) determ (i).
isUpperCase : (string Source) determ (i).
isWhiteSpace : (string Source) determ.
lastChar : (string Source, string First, char Last) determ (i,o,o).
length : (string Source) -> charCount Length procedure (i).
replace : (string Source, string ReplaceWhat,
            string ReplaceWith, caseSensitivity Case)
            -> string Output procedure
replaceAll : (  string Source, string ReplaceWhat,
            string ReplaceWith) -> string Output.
replaceAll : (  string Source, string ReplaceWhat,
```

```
            string ReplaceWith, caseSensitivity Case)
            -> string Output.
caseSensitivity = caseSensitive; caseInsensitive; casePreserve.
search : (string Source, string LookFor)
        -> charCount Position determ (i,i).
search : (string Source, string LookFor, caseSensitivity Case)
        -> charCount Position determ (i,i,i).
split : (string Input, string Separators) -> string_list.
split_delimiter : (string Source, string Delimiter)
            -> core::string_list List procedure (i,i).
subString : (string Source, charCount Position, charCount HowLong)
            -> string Output procedure (i,i,i).
toLowerCase : (string Source) -> string Output procedure (i).
    Конвертирует буквы в строке в строчные
toUpperCase : (string Source) -> string Output procedure (i).
    Конвертирует буквы в строке в прописные
trim : (string Source) -> string Output procedure (i).
    Удаляет предшествующие и последующие строке пробелы.
trimFront : (string Source) -> string Output procedure (i).
    Удаляет предшествующие строке пробелы.
trimInner : (string Source) -> string Output procedure (i).
    Удаляет группы пробелов из строки Source.
trimRear : (string Source) -> string Output procedure (i).
    Удаляет следующие за строкой пробелы.
/*********************************
Project Name: trim_example
UI Strategy: console
*********************************/
implement main
clauses
    classInfo("main", "trim_example").
    run():-
        console::init(),
        Str= " murderous, doomed ",
        T= string::trim(Str),
        stdio::write(T), stdio::nl,
        T1= string::trimInner(T),
        stdio::write(T1), stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

```
murderous,        doomed
murderous, doomed

F:\Project\Visual Prolog Projects\Container\Exe>pause
Для продолжения нажмите любую клавишу . . .
```

6.5. Немного о логике: Грамматика предикатов

Определим неформальные правила грамматики для исчисления предикатов.

- Выражение $p(t_1, t_2, ..., t_{n-1}, t_n)$ называется литералом. В литерале символ p/n является n-местным символом, отличающимся от всех остальных символов языка, а t_i являются термами. Термами могут быть:
    - Функциональные выражения, например, выражения вида $f(x_1, x_2, ..., x_{n-1}, x_n)$. Функциональное выражение может иметь нулевую арность. Это означает, что слова *a*, *x*, *b* и *rosa* могут быть функциональными выражениями.
    - Числительные, строки и символы также могут быть термами.

  Формула, состоящая из одного литерала, называется атомарной формулой.

- Пусть t_1 и t_2 — термы. Тогда выражения $t_1 = t_2$, $t_1 > t_2$, $t_1 < t_2$, $t_1 \geq t_2$ и $t_1 \leq t_2$ являются литералами и их можно использовать для создания формул.
- Если P и Q формулы, то $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$ и $P \equiv Q$ — также формулы. Кроме этого, если P – формула, а X – переменная, то $\forall X (P)$ (P истинно при любом X) и $\exists X (P)$ (существует такой X, который обращает P в истину) также являются формулами.
- Переменная, связанная квантором существования $\exists$, называется экзистенциональной переменной. Таким образом, переменная X в формуле

  $$\exists X (\sin(X) = 0)$$

  является экзистенциональной. Все такие переменные можно удалить из формулы, подставив вместо них константы. В выражении, приведенном ниже, вместо X можно подставить π:

  $$\sin(X) = 0$$

Клаузальные формы

Любое предложение для предиката может быть переписано в клаузальной форме, т.е. в виде конъюнкции дизъюнкций, без использования квантора существования и квантора всеобщности внутри скобок. Запись $A \vee B$ обозначает дизъюнкцию, а запись $A \wedge B$ — конъюнкцию, поэтому любая формула может быть переписана следующим образом:

$$\forall X \forall Y \forall Z ... \begin{pmatrix} (L_{11} \vee L_{12} \vee L_{13} ...) \wedge \\ (L_{21} \vee L_{22} \vee L_{23} ...) \wedge \\ (L_{31} \vee L_{32} \vee L_{33} ...) \wedge \\ ... \end{pmatrix}$$

где с помощью L_{ij} обозначены литералы или отрицания литералов. Мы уже видели, как удалить из формулы кванторы существования. Рассмотрим оставшиеся необходимые шаги для получения клаузальной формулы.

1. Удалите импликации и эквиваленции:

$\alpha \rightarrow \beta$	$\neg\alpha \vee \beta$
$\alpha \equiv \beta$	$(\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$

2. Распределите отрицания. Используйте таблицы истинности, чтобы доказать, что

$$\neg (\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

и что

$$\neg (\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$

Затем используйте эти результаты, для того чтобы получить следующие подстановки:

$\neg (\alpha \vee \beta)$	$\neg\alpha \wedge \neg\beta$
$\neg (\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$

3. Норвежский математик Тораф Альберт Сколем (*Thoraf Albert Skolem*; 1887 – 1963) изобрёл метод удаления экзистенциальных переменных, названный в его честь. Сколемизация — это подстановка констант (и сколемовских функций — *ред*. *пер*.) вместо экзистенциальных переменных. Например:

$\exists X\,(p(X))$	$p(s_0)$
$\forall X\,(\text{man}(X) \rightarrow \exists W(\text{loves}(W, X)))$	$\forall X\,(\text{man}(X) \rightarrow \text{loves}(w(X), X))$

4. Преобразуйте конъюнкции, используя следующие правила:

$(\alpha \wedge \beta) \vee \gamma$	$(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
$\neg (\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$

Глава 7: Грамматика

Философ Витгенштейн в «Логико-философском трактате» (*Tractatus Logicus Philosophicus)* выразил мнение, что *Мир есть положение вещей*. Предметы и объекты имеют свойства — такие как местоположение, целостность[1], цвет и т.д. Множество значений, которые принимают эти свойства, называется *состоянием*. Например согласно Евклиду, точка не имеет никаких свойств, кроме местоположения. Поэтому состояние точки может быть задано её координатами.

Первое предложение трактата: *The World is all, that is the case* — «*Мир есть всё, что имеет место*». Последнее предложение: *Whereof one cannot speak, thereof one must be silent* — «*О чем невозможно говорить, о том следует молчать*». Практически невозможно создать структуру данных для того, чтобы представить идеи Витгенштейна в компьютере. Однако, если бы ее задумал лингвистический гений, то эта структура данных несомненно должна была бы иметь три конструктора: узел *world* (мир), узел *case* (случай) и узел *silent* (молчание). Вот семантический класс, единственное применение которого состоит в определениии семантического домена трактата:

```
class semantic
    open core

domains
    category = art; nom; cop; rel.
    tree = case(category, string); world(tree, tree); silence.

predicates
    classInfo : core::classInfo.
end class semantic
```

В данном коде появилось новшество: использование конструкторов структур данных. Конструктор аналогичен функции, то есть он имеет функтор и аргументы. Например, конструктор `case/2` имеет два аргумента, грамматическую категорию и строку, представляющую токен.

7.1. Грамматический разбор

Действовать — значит изменять свойства объектов. Таким образом, результатом действия является изменение состояния. Например, когда предикат разбирает список слов, мир будет претерпевать изменение состояния. К конечному состоянию парсер израсходует часть списка. Для того чтобы понять этот момент, создадим класс *german* для разбора первого предложения трактата. Используя его в качестве модели, вы можете написать более мощный класс, для того чтобы разобрать всю книгу. Это дело верное, так как трактат очень короткий.

```
% Файл german.cl
class german
```

[1] Оно целое или разбитое? – *прим. авт.*

```
    open core, semantic
predicates
    classInfo : core::classInfo.
    article : (tree, string*, string*) nondeterm (o, i, o).
    noun : (tree, string*, string*) nondeterm (o, i, o).
    nounphr : (tree, string*, string*) nondeterm (o, i, o).
    copula : (tree, string*, string*) nondeterm (o, i, o).
    phr : (tree, string*, string*) nondeterm (o, i, o).
end class german

% Файл german.pro
implement german
    open core, semantic
clauses
    classInfo("german", "1.0").

    article(case(art, ""), ["die"|Rest], Rest).
    article(case(art, ""), ["der"|Rest], Rest).

    noun(case(nom, "Welt"), ["Welt"|R], R).
    noun(case(nom, "Fall"), ["Fall"|R], R).

    copula(world(case(cop, "ist"), T), ["ist"|R1], R) :-
        nounphr(T, R1, R).

    nounphr(world(T1, T2), Start, End) :-
        article(T1, Start, S1), noun(T2, S1, End).
    nounphr(case(nom, "alles"), ["alles"|R], R).

    phr(world(T1, T2), Start, End) :-
        nounphr(T1, Start, S1), copula(T2, S1, End).
end implement german
```

Если вы обратите внимание на объявление предиката `article`, то заметите, что он имеет схему `(o, i, o)`, то есть он получает список слов и выводит дерево разбора[1] и частично израсходованный входной список. Определение предиката `noun` работает точно так же, как и определение предиката `article`. Пример лучше тысячи слов: подставим список `["die", "Welt", "ist", "alles"]` в предикат `article/3`.

```
article(T1, ["die", "Welt", "ist", "alles"], S1).
```

Этот вызов будет соответствовать первому предложению `article/3`, при этом

```
Rest=["Welt", "ist", "alles"].
```

В результате будет выведено `T1=case(art, "")`, и `S1= ["Welt", "ist", "alles"]`. Далее, подставим `S1= ["Welt", "ist", alles"]` в предикат `noun/3`.

```
noun(T2, ["Welt", "ist", "alles"], End)
```

[1] Если вы не знаете эти лингвистические термины, посмотрите книги по обработке естественных языков или по построению компиляторов, что вам больше нравится – *прим. авт.*

Этот вызов будет соответствовать первому предложению `noun/3`, при этом `R=["ist", "alles"]`. Результатом будет `T2=case(nom, "Welt")` и `End=["ist", "alles"]`. Определение предиката `nounphr` делает эти два вызова последовательно при разборе группы существительного `["die", "Welt"]`. Предикаты `copula/3` и `phr/3` ведут себя так же, как и предикат `nounphr`.

7.2. Порождающие грамматики

В то время как парсер немецкого принимает на вход предложение и возвращает узел (терм), парсер английского, приведенный ниже, получает терм и создает из него предложение. Можно сказать, что терм представляет собой значение (смысл) предложения. Парсер немецкого принимает предложение на немецком и выводит его значение. Парсер английского получает значение и строит английский перевод.

```
% Файл english.cl
class english
    open core, semantic
predicates
    classInfo : core::classInfo.
    article : (tree, string*) nondeterm (i,o)(o,o).
    noun : (tree, string*) nondeterm (i,o)(o,o).
    nounphr : (tree, string*) nondeterm (i,o)(o,o).
    copula : (tree, string*) nondeterm (o,o)(i, o).
    phr : (tree, string*) nondeterm (i,o).
end class english

% Файл english.pro
implement english
    open core, semantic

clauses
    classInfo("english", "1.0").

    article(case(art, ""), ["the"]).

    noun(case(nom, "Welt"), ["world"]).
    noun(case(nom, "Fall"), ["case"]).
    noun(case(nom, "alles"), ["all"]).

    nounphr( world(T1, T2), list::append(A,N)) :-
        article(T1, A), noun(T2, N).
    nounphr( case(nom, "alles"), ["all"]).

    copula(world(case(cop, "ist"), T), list::append(["is"], R1)) :-
        nounphr(T, R1).

    phr(world(T1, T2), list::append(Start, S1)) :-
        nounphr(T1, Start), copula(T2, S1).
end implement english
```

Предложения Хорна, которые выводят фразу в соответствии с грамматикой заданного языка, называются *продукциями* — правилами вывода. В классе *english* предложения для предикатов `article/2`, `noun/2` и `nounphr` являются примерами правил вывода.

Каждое *английское* правило вывода возвращает список слов. Имея это в виду, рассмотрим правило вывода для предиката `phr/2`.

```
phr(world(T1, T2), list::append(Start, S1)) :-
    nounphr(T1, Start), copula(T2, S1).
```

Второй аргумент заголовка правила использует функцию `list::append(Start, S1)`, для того чтобы соединить группу существительного с глаголом-связкой и тем самым составить фразу.

Для реализации и тестирования программы создайте консольный проект `tratactus`. Вставьте классы `german`, `english` и `semantic` в дерево проекта. Не забудьте снять галочку в поле *Create Objects*. Добавьте код в соответствующие файлы классов. Программа для тестирования приведена на рисунке 7.1. После компиляции программы, если вы хотите проверить её внутри IDE, используйте пункт *Build/Run in Window*. Кстати, если вы не хотите увидеть множество ошибок, добавьте первым в проект класс `semantic`, скомпилируйте приложение. Затем добавьте класс `german`, снова скомпилируйте программу. Следующим шагом будет добавление класса `english` и ещё одна компиляция приложения. Наконец, вы можете вставить код для класса `main` и скомпилировать приложение в последний раз.

```
/**********************************
Project Name: tratactus
UI Strategy: console
**********************************/
implement main
    open core, console, semantic
class predicates
    test:().
clauses
    classInfo("main", "tratactus").

    test() :-
        german::phr(T, ["die", "Welt", "ist", "alles"], _X),
        write(T), nl, english::phr(T, Translation), !,
        write(Translation), nl.
    test().

    run():- console::init(), test().
end implement main
goal mainExe::run(main::run).
```

Рисунок 7.1 Die Welt ist alles, was der Fall ist

7.3. Почему Пролог?

Одной из целей языков программирования высокого уровня является увеличение возможностей для программистов *рассуждать*, для того чтобы прийти к более высокой производительности труда и к искусственному интеллекту. Несмотря на единодушное мнение о том, что Пролог соответствует этим целям более, чем любой другой язык, вокруг этого тезиса существуют некоторые споры. Так что давайте посмотрим, что мы понимаем под высокой производительностью и искусственным интеллектом.

Высокая производительность. Продуктивный программист совершает больше действий за меньшее время. В программном проекте существуют элементы, которые требуют наибольшего количества усилий. Например, такие структуры данных, как списки и деревья необходимы везде, но в языках, подобных C или Java, они сложны для кодирования и работы с ними. В Прологе нет ничего легче, чем работать со списками или деревьями. Грамматический разбор — это ещё один сложный предмет, без которого невозможно жить; между прочим, разбор — это то, что вы делали для того, чтобы реализовать маленькую *английскую* грамматику, и что вы изучили в этой главе. Построения прикладной логики также очень громоздки, но мне нет необходимости развивать эту тему, потому что Пролог имеет *логику* даже в своем названии.

Искусственный интеллект делает ваши программы легко приспосабливаемыми к изменениям и более дружественными к пользователю. Пролог, вместе с Lisp и Scheme, очень популярен среди людей, которые работают в области искусственного интеллекта. Это означает, что вы найдете библиотеки Пролога почти для любого алгоритма из этой области.

7.4. Примеры

Пример этой главы (*tratactus*) показывает, как можно написать программу, которая переводит философский немецкий на английский.

7.5. Немного о логике: Натуральная дедукция

Как Грис [Gries], так и Дейкстра [Dijkstra] предлагали использовать логику для построения корректных программ и для доказательства того, что данная программа корректна. Многие говорят, что эти авторы устарели, потому что устарели языки, используемые в этих книгах (паскалеподобные языки). Глупо так говорить, потому что методы, которые они предлагают, основываются на исчислении предикатов, которое не похоже, чтобы устаревало. Кроме этого, можно использовать систему Дейкстры/Гриса (*Dijkstra/Gries*) для программирования на современных языках, таких как Visual Prolog, Clean или Mercury. На самом деле, так как алгоритм Дейкстры основан на исчислении предикатов, гораздо проще применить его на диалекте Пролога, чем Паскаля. Кон [Cohn] и Ямаки [Yamaki] изобрели для этого очень простую процедуру:

- Напишите спецификацию вашего алгоритма в клаузальной форме.
- Преобразуйте спецификацию в код Пролога, введя механизмы контроля, такие, как отсечение и конструкция `if-then-else`.

- Докажите, что введение механизмов контроля не изменило семантику алгоритма.

Для доказательства корректности программы Грис предлагает специфицировать ее с помощью исчисления предикатов, а затем доказывать её корректность, используя метод, известный как натуральная дедукция. Натуральная дедукция в логике — это подход к теории доказательств, который пытается предоставить более естественную модель логических рассуждений, чем модель, предложенная Фреге и Расселом.

Эпистемология говорит, что суждение — это нечто, что можно знать. Объект очевиден, если о нём фактически известно. Что-либо очевидно, если это можно наблюдать. Например, глобальное потепление очевидно, потому что учёные имеют сведения о том, что оно происходит. Но часто очевидность не наблюдается напрямую, а выводится из базовых суждений. Шаги вывода и составляют доказательство. Существует много разновидностей суждений:

Аристотелевы суждения: *A истинно*, *A ложно* и *A является высказыванием*.
Темпоральная логика: *A истинно в момент времени t*.
Модальная логика: *A обязательно истинно* или *A возможно истинно.*

Ранее мы видели, как доказывать суждения вида «A является высказыванием». Грамматические правила, приведённые в разделах 3.5 и 6.5, предоставляют правила вывода для того чтобы решить, является ли строка символов правильной формулой исчисления высказываний или исчисления предикатов. Однако данная грамматика для исчисления высказываний является неформальной. В формальном изложении грамматические правила называются правилами построения (*formation rules*) и представляются с помощью знака /\ F. Поэтому правила вывода для распознавания правильно построенной формулы исчисления предикатов могут быть записаны следующим образом:

$$\frac{A\ prop}{\neg A\ prop}\bigwedge F \quad (7.1)$$

$$\frac{A\ prop\ B\ prop}{A \wedge B\ prop,\ A \vee B\ prop,\ A \rightarrow B\ prop,\ A \equiv B\ prop}\bigwedge F \quad (7.2)$$

В системе натуральной дедукции существуют правила построения для соответствующего исчисления — как для вставки, так и для удаления — для каждой логической связки, имеющейся в рассматриваемом исчислении. Что касается исчисления предикатов, то для него существуют правила удаления и вставки для отрицания ($\neg P$), конъюнкции ($P \wedge Q$), дизъюнкции ($P \vee Q$), эквиваленции ($P \equiv Q$), импликации ($P \rightarrow Q$), квантора всеобщности ($\forall X (P)$) и квантора существования ($\exists X (P)$).

В правиле вывода выражение, расположенное выше черты, называется посылкой, а выражение, расположенное ниже черты, заключением. Символ ├ не является частью правильно построенной формулы — это метасимвол.

Запись $P \vdash Q$ понимается как *P влечет Q*, а запись $\vdash Q$ означает, что *Q — теорема*. Ниже приведены несколько правил вывода системы натуральной дедукции, которые можно использовать для предложений исчисления предикатов, представленных в клаузальной форме.

вставка И	$\wedge/I: \frac{E_1, E_2, \dots E_n}{E_1 \wedge E_2, \dots E_n}$
удаление И	$\wedge/E: \frac{E_1 \wedge E_2 \dots E_n}{E_i}$
вставка ИЛИ	$\vee/I: \frac{E_i}{E_1 \vee E_2 \dots E_{n-1} \vee E_n}$
удаление ИЛИ	$\vee/E: \frac{E_1 \vee E_2, E_1 \to E, E_2 \to E}{E}$
вставка импликации	$\to/I: \frac{E_1, E_2 \vdash E}{E_1 \wedge E_2 \to E}$
MODUS PONENS	$\to/E: \frac{E_1 \to E_2, E_1}{E_2}$
вставка НЕ	$\neg/I: \frac{E \vdash E_1 \wedge \neg E_1}{\neg E}$
MODUS PONENS	$\neg/E: \frac{\neg E \vdash E_1 \wedge \neg E_1}{E}$

Мы намереваемся работать с клаузальной формой, поэтому вам не потребуются правила, соответствующие кванторам. Давайте немного поиграем с нашей новой игрушкой. Докажем, что $p \wedge q \vdash p \wedge (r \vee q)$:

	$p \wedge q \vdash p \wedge (r \vee q)$	
1	$p \wedge q$	посылка 1
2	p	исключение $\wedge$, 1
3	q	исключение $\wedge$, 1
4	$r \vee q$	вставка $\vee$, 3
5	$p \wedge (r \vee q)$	вставка $\wedge$, 2, 4

В приведённом выше доказательстве каждый шаг всегда содержит правило вывода и строки, для которых оно было использовано. Например, строка $p \wedge (r \vee q)$ получается по правилу вставки $\wedge$, примененному к строкам 2 и 4.

Глава 8: Рисование

В этой главе вы научитесь рисовать с помощью обработчика события `onPaint`. Начните с создания проекта и формы, на которой вы будете рисовать.

- ***Создайте проект***:

```
Project Name: painting
Object-oriented GUI (pfc/gui)
```

- ***Создайте пакет*** `paintPack` в корне дерева проекта.
- ***Вложите*** `canvas.frm` внутрь `paintPack`. Постройте (*Build/Build*) приложение.
- ***Включите пункт*** *File/New* меню приложения и добавьте код

```
clauses
    onFileNew(S, _MenuTag) :-
        X= canvas::new(S), X:show().
```

для *TaskWindow/Code Expert/Menu/TaskMenu/id_file/id_file_new*.

Теперь, если вы скомпилируете и запустите программу, то при выборе пункта *File/New* меню вашего приложения будет выполнен предикат, приведенный выше. Команда `X=canvas::new(S)` создаёт новый объект `X` класса *window*. Это окно будет дочерним главному окну `S`. Команда `X:show()` посылает сообщение объекту `X` показать это окно.

Что вы будете делать дальше? На вашем месте я бы нарисовал что-нибудь на форме *canvas*, чтобы создать интересный фон.

8.1. Процедура onPainting

Когда окну требуется прорисовка, оно вызывает обработчик события *onPainting*. Поэтому если вы добавите инструкции к *onPainting*, то они будут выполнены.

- ***Создайте класс*** `dopaint` внутри `paintPack`. Отключите *Creates Objects*.
- ***Добавьте*** приведённый ниже код в файлы `dopaint.cl` и `dopaint.pro`.

```
% Файл dopaint.cl
class dopaint
    open core
predicates
    bkg:(windowGDI).
end class dopaint

% Файл dopaint.pro
implement dopaint
    open core, vpiDomains
clauses
```

```
    bkg(W) :-
        P= [pnt(0,0), pnt(10,80), pnt(150, 100)],
        W:drawPolygon(P).
end implement dopaint
```

- Постройте (*Build/Build*) приложение, чтобы добавить класс `dopaint` к проекту.
- Откройте окно *Properties* формы `canvas.frm`, перейдите на вкладку *Events* и добавьте следующий код к *PaintResponder*:

```
onPaint(_Source, _Rectangle, GDIObj) :-
    dopaint::bkg(GDIObj).
```

В случае если файл `canvas.frm` закрыт, откройте его с помощью дерева проекта.

Как я сказал ранее, когда окну требуется прорисовка, оно вызывает обработчик `onPaint`, который в данном случае вызовет предикат `dopaint::bkg(GDIObj)`. В классе `dopaint` имеется метод `bkg(GDIObj)`, который рисует в окне, на которое указывает аргумент `GDIObj`.

Давайте поймём, что делает метод `bkg(W)`. Как можно видеть выше, переменная `P` хранит список точек. Каждая точка определяется своими координатами. Например, `pnt(0, 0)` — это точка с координатами (`0, 0`). Когда `bkg(W)` вызывает `W:drawPolygon(P)`, он посылает сообщение объекту `W` и просит его нарисовать многоугольник, вершины которого заданы списком `P`. Скомпилируйте программу и проверьте, будет ли она работать точно так, как сказано.

Давайте улучшим метод `bkg(W)`. Он рисует белый треугольник на поверхности формы `canvas`. Для того чтобы сделать треугольник красным, нужно изменить кисть рисования. Кисть представляет собой структуру с двумя аргументами вида:

```
brush=brush(patStyle PatStyle, color Color).
```

Цвета представляются числами, которые определяют количество в них красного, зеленого и синего цвета. Вы, возможно, знаете, что можете получить множество цветов, комбинируя красную, зеленую и синюю палитры. Давайте представим числа в шестнадцатеричной системе счисления. Шестнадцатеричные числа имеют в записи шестнадцать цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Красный цвет задаётся числом `0x000000FF`, где `0x` показывает, что мы имеем дело с шестнадцатеричной системой счисления. Представлять цвета числами неплохо, но вы также можете использовать идентификаторы. Например, `color_Red` представляет красный цвет, как вы могли догадаться. Вот также названия для других шаблонов:

- `pat_Solid`: сплошная кисть.
- `pat_Horz`: горизонтальная штриховка.
- `pat_Vert`: вертикальная штриховка.
- `pat_FDiag`: штриховка с наклоном под 45°.

Ниже приведена модификация `bkg(W)`, рисующая красный треугольник:

```
bkg(W) :-
    Brush= brush(pat_Solid, color_Red),
    W:setBrush(Brush),
    P= [pnt(0,0), pnt(10,80), pnt(150, 100)],
    W:drawPolygon(P).
```

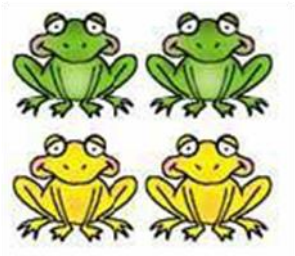
Наконец, следующая версия `bkg(W)` рисует эллипс и очищает в нём зеленый прямоугольник.

```
bkg(W) :-
    R= rct(40, 60, 150, 200),
    W:drawEllipse(R),
    R1= rct(60, 90, 140, 130),
    W:clear(R1, color_Green).
```

Последнее, чему вы научитесь в этой главе — как добавлять в ваше окно `.bmp` изображение. Для этого надо модифицировать метод `bkg(W)` следующим образом:

```
bkg(W) :-
    P= vpi::pictLoad("frogs.bmp"),
    W:pictDraw(P, pnt(10, 10), rop_SrcCopy).
```

Первым шагом является загрузка изображения из файла. Это делает предикат `pictLoad/1`. Следующий шаг — нарисовать изображение. В данном примере изображение `P` помещено в точку `pnt(10, 10)`.



frogs.bmp

A3mask.bmp

A3.bmp

Часто бывает необходимо нарисовать маленькое изображение поверх другого изображения и убрать при этом задний фон с маленького изображения. Например, поместим орла среди лягушек из предыдущего примера.

Первый шаг — нарисовать маску орла, используя `rop_SrcAnd`. Маска изображает черную тень орла на белом фоне. Следующий шаг — нарисовать самого орла, используя `rop_SrcInvert`. Орёл должен совершенно точно соответствовать своей маске.

Программа, которая размещает орла среди лягушек, приведена ниже и реализована в папке *paintEagle* (см. примеры).

Проект *paintEagle* в точности соответствует проекту *painting*, за исключением определения `bkw(W)`.

```
% Файл dopaint.cl
class dopaint
    open core
predicates
    bkg:(windowGDI).
end class dopaint

% Файл dopaint.pro
implement dopaint
    open core, vpiDomains
clauses
```

```
    bkg(W) :-
        P= vpi::pictLoad("frogs.bmp"),
        W:pictDraw(P, pnt(10, 10), rop_SrcCopy),
        Mask= vpi::pictLoad("a3Mask.bmp"),
        Eagle= vpi::pictLoad("a3.bmp"),
        W:pictDraw(Mask,pnt(50, 50),rop_SrcAnd),
        W:pictDraw(Eagle,pnt(50, 50),rop_SrcInvert).
end implement dopaint
```

8.2. Пользовательский элемент управления

Мы рисовали прямо на полотне формы. Однако хорошей идеей также является создание пользовательского элемента управления и его использование для рисования.

- ***Создайте новый проект***

```
Project Name: customcontrol
Object-oriented GUI (pfc/gui)
```

- ***Выберите пункт*** *File/New in New Package* меню IDE. В диалоговом окне *Create Project Item* выберите элемент *DrawControl* на левой панели. Названием нового элемента управления будет `canvas`.

```
Name: canvas
New Package
Parent Directory [     ]
```

Оставьте поле *Parent Directory* пустым. Нажмите кнопку *Create*.

- На экране появится прототип полотна. Вы можете закрыть его, чтобы избежать загромождения IDE. Постройте приложение, чтобы включить в него новый элемент управления.
- ***Добавьте новую форму к дереву проекта***. Выберите команду *File/New in New Package* в меню задач и выберите пункт *Form* на панели в левой части диалогового окна *Create Project Items*. Пусть названием новой формы будет `greetings`. В результате нажатия кнопки *Create* вы получите прототип формы `greetings`.
- ***Выберите символ с изображением ключа*** в окне *Controls*. Щёлкните на поверхности формы `greetings`. Система покажет вам меню, содержащее список нестандартных элементов управления, среди них будет элемент `canvas`.

- Удалите кнопку *Ok* с формы `greetings` и замените её кнопкой `hi_ctl`, как показано на рисунке 8.1.

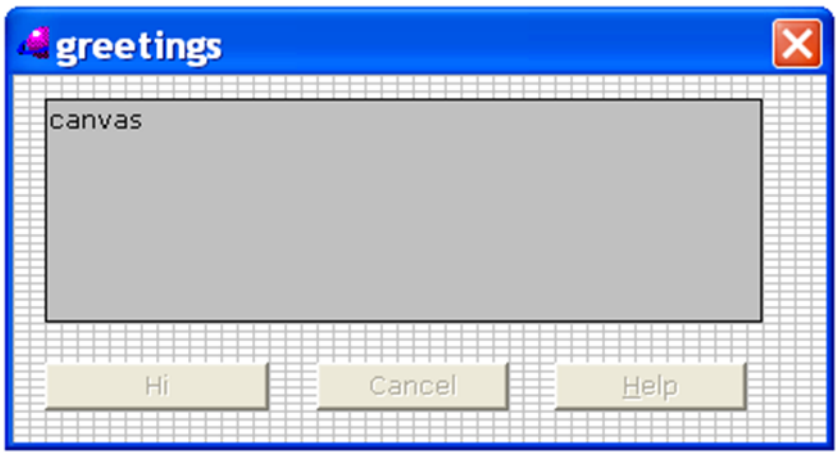


Рисунок 8.1 Форма с пользовательским элементом управления

- ***Постройте приложение***.
- Отправляйтесь в диалоговое окно *Properties* для кнопки `hi_ctl` и добавьте следующий фрагмент к *ClickResponder*:

```
clauses
    onHiClick(_Source) = button::defaultAction :-
        W= custom_ctl:getVPIWindow(),
        X= convert(integer, math::random(100)),
        Y= convert(integer, math::random(100)),
        vpi::drawText(W, X , Y, "Hello!").
```

- ***Снова постройте приложение***. Включите пункт *TaskMenu.mnu→File/New*. Добавьте код

```
clauses
    onFileNew(S, _MenuTag) :-
        F= greetings::new(S), F:show().
```

для *TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new.*

Постройте и запустите приложение. Выберите пункт *File/New*, в результате появится новая форма. Когда вы щёлкнете мышью в поле элемента *canvas* этой формы, будет напечатано теплое приветствие.

8.3. Немного о логике: Принцип резолюций

Наиболее важным правилом вывода для программистов на Прологе является принцип резолюций, который предложил Робинсон [Robinson] в 1963 году:

$$P \vee Q,\ \neg P \vee R \vdash Q \vee R$$

Для доказательства этого принципа нам понадобятся два правила. Одним из них является первый закон дистрибутивности:

$$(a \vee b) \wedge (a \vee c) \equiv a \vee (b \wedge c)$$

Существует также второй закон дистрибутивности, который мы не будем использовать при доказательстве принципа резолюций, он имеет вид:

$$(a \wedge b) \vee (a \wedge c) \equiv a \wedge (b \vee c)$$

Рассмотрим конкретный случай применения этих законов. Лисий (*Lysias*) был афинским юристом, жившим во времена Сократа (*Socrates*). Он был сыном того самого Кефала (*Kephalos*), который развлекал Сократа в «Государстве» (*Republic*) Платона (*Plato*). Лисий был метеком[1], следовательно, не имел гражданских прав. Несмотря на это, его семья была очень богатой, как Платон пишет об этом в «Государстве». Его выступления как в защиту, так и в обвинение были очень интересны. Например, во времена диктатуры тридцати тиранов правители страны решили взять восемь богатых метеков и двух бедных, убить их и ограбить. Лисий и его брат Полемарх (*Polemarchus*) были среди этих обречённых на смерть. Однако Лисию удалось сбежать, а затем вернуться в свою страну, чтобы восстановить демократию и наказать одного из тиранов, убивших его брата.

В другой раз Лисий защищал человека, убившего любовника своей жены. Для того чтобы понять доводы его защиты, вам необходимо знать несколько фактов о греческом обществе. Мужчины и женщины жили раздельно: мужчины жили в андроне[2] (*androecium*), а женщины — в гинекее[3] (*gynoecium*). Вы, возможно, помните, что у покрытосеменных (*angiosperms*) также существуют андроцеи (*androecia* — помещения для самцов) и гинецеи (*gynoecia* — помещения для самок). Причина состоит в том, что Карл фон Линней (*Carl von Linné*) назвал эти части растительной физиологии в соответствии с разделением помещений у древних греков. Другим греческим обычаем было приданое. Когда женщина покидала отчий дом, она могла забрать изрядную сумму денег в дом мужа. Так что никто не был склонен разводиться с женой, потому что за этим последовал бы возврат денег в её семью.

Когда Эратосфен (*Eratosthenes*) обнаружил, что жена ему изменяет, он решил убить её любовника, так как развод мог означать потерю её богатого приданого[4]. Он притворился, что отправился в своё хозяйство на окраине города, но вернулся как раз вовремя, чтобы убить негодяя на кровати неверной жены. Однако обвинитель заявил, что алчный муж убил соперника на домашнем алтаре, куда тот убежал в поиске защиты у Зевса. Если бы Эратосфен убил человека на алтаре, суд мог бы обречь его на смерть. Но

[1] Метек — иностранец, постоянно проживающий в стране. Лисий родился в Афинах, но был метеком по отцу.

[2] Мужская половина греческого дома.

[3] Женская половина греческого дома.

[4] В древнегреческих источниках приводится совсем другая история. Существует знаменитая «Оправдательная речь по делу об убийстве Эратосфена», написанная Лисием и произнесенная афинянином Евфилетом. Земледелец Евфилет убил любовника своей жены Эратосфена. В своей оправдательной речи он, в частности, упоминает то, что женская и мужская половины его дома устроены абсолютно одинаково, поэтому после рождения ребенка он переселился наверх, а его жена — вниз, что обеспечило легкий доступ к ней коварному любовнику. О приходе любовника ночью ему сообщила служанка, после этого он потихоньку выбрался на улицу, созвал друзей и знакомых и вместе с толпой внезапно ворвался в спальню жены, так что все увидели Эратосфена в постели жены Евфилета. Тот хотел было откупиться, но Евфилет не согласился, считая, что закон важнее. В своей речи Евфилет, с помощью Лисия, опровергает все выдвинутые против него обвинения.

если бы он убил его на кровати, он был бы полностью оправдан. Лисий построил защиту своего клиента на основе следующего аргумента:

> *Он был на алтаре или он был на кровати, и он был на алтаре или он был убит.*

Обвинитель принял этот аргумент. Затем Лисий упростил его, используя первый закон дистрибутивности:

> *Он был на алтаре или он был на кровати и он был убит. Так как вы знаете, что он был убит, то он был на кровати.*

Эратосфен был оправдан[1]. После этого долгого отступления давайте докажем принцип резолюций.

	$p \wedge q, \neg p \vee r \vdash q \vee r$	
1	$p \vee q \vee r$	вставка $\vee$, посылка 1
2	$\neg p \vee q \vee r$	вставка $\vee$, посылка 2
3	$(p \vee q \vee r) \wedge (\neg p \vee q \vee r)$	вставка $\wedge$, 1, 2
4	$(q \vee r) \vee (p \wedge \neg p)$	первый закон дистрибутивности, 3
5	$q \vee r$	MODUS TOLLENDO PONENS

[1] В упомянутой в предыдущем примечании истории про Евфилета и Эратосфена Евфилет был оправдан.

Глава 9: Типы данных

Большинство современных языков программирования работают со строго типизированными данными. Это означает, что компилятор проверяет, принадлежат ли данные, подаваемые на вход предикату или процедуре, правильному типу. Например, арифметические операции ($x + y$, $x \times y$, $a - b$, $p \div q$) работают с целыми, действительными или комплексными числами. Поэтому компилятор удостоверяется, что ваша программа передает этим операциям числа, а не что-нибудь другое. Если в вашем коде имеется логическая ошибка, и вы пытаетесь поделить строку на число, то компилятор упирается. Я уверен, что вы и сами скажете, что это единственное разумное, что должно быть. Я согласен. Однако не все разработчики языков имеют намерение включить в свои компиляторы проверку типов. Например, Стандартный Пролог не проверяет тип до тех пор, пока ошибочный код не будет выполняться. Программа выходит из строя, когда она уже находится в компьютере конечного пользователя.

9.1. Примитивные типы данных

Visual Prolog имеет множество примитивных типов данных:

integer: `3`, `45`, `65`, `34`, `0x0000FA1B`, `845`. Кстати говоря, `0x0000FA1B` — это шестнадцатеричное число, то есть число, выраженное в системе счисления с основанием 16, которая использует следующие цифры: 0, 1, 2,3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

real: `3.45`, `3.1416`, `2.18E-18`, `1.6E-19` и пр.

string: `"pencil"`, `"John McKnight"`, `"Cornell University"` и пр.

symbol : `"Na"`, `"Natrium"`, `"K"`, `"Kalium"` и пр.

Вы заметили, что элементы типа *symbol* выглядят так же, как строки: и те, и другие представляют собой последовательности Unicode-символов. Однако хранятся они по-разному. Элементы типа *symbol* хранятся в таблице идентификаторов, и Пролог для их внутреннего представления использует адреса в этой таблице. Таким образом, если элемент типа *symbol* встречается в программе много раз, он будет занимать меньше места, чем строка. Ниже приведен очень простой пример использования действительных чисел, элементов типа *symbol* и строк. Этот пример взят из химии.

```
/*******************************
Project Name: primtype1
UI Strategy: console
*******************************/
% Файл main.pro
implement main
    open core
```

```
constants
    className = "main".
    classVersion = "primtype1".
clauses
    classInfo(className, classVersion).

class predicates
    energy : (real, real, real) procedure (i, i, o).
    family : (symbol, string) procedure (i, o).

clauses
    energy(N, Z, E) :- E= -2.18E-19*Z*Z/(N*N).

    family("Na", "alkaline metal") :- !.
    family(_, "unknown").

    run():- console::init(),
        energy(4, 1, E1),
        energy(2, 1, E2),
        DeltaE= E1-E2,
        stdio::write(DeltaE), stdio::nl,
        family("Na", F),
        stdio::write("Na", " is an ", F), stdio::nl,
        succeed().
end implement main
goal
    mainExe::run(main::run).
```

9.2. Множества

Множество — это коллекция предметов. Вы, разумеется, знаете, что в коллекциях нет повторяющихся предметов. Я имею в виду, что если парень или девушка имеет коллекцию наклеек, то они не захотят иметь две одинаковые наклейки в своей коллекции. Если у кого-нибудь из них есть одинаковые, то он обменяет такую наклейку на другую, которая в его коллекции отсутствует.

Возможно, если бы ваш отец имел коллекцию греческих монет, он с удовольствием принял бы ещё одну драхму в свою коллекцию. Однако две драхмы не являются совершенно одинаковыми, одна из них может быть старше другой.

9.3. Множества чисел

Математики собирают другие предметы, а не монеты, марки или логарифмические линейки. Они собирают числа, например. Поэтому вам положено изучить многое о множествах чисел.

N — это множество натуральных чисел[1]. Вот как математики записывают элементы множества N: {0, 1, 2, 3, …} .

Z — это множеством целых чисел, т. е. **Z** = {…, – 3, – 2, – 1, 0, 1, 2, 3, … }.

Почему множество целых чисел изображается буквой **Z**? Я не знаю, но я могу сделать обоснованное предположение. Теория множеств была открыта Георгом Фердинандом Людвигом Филиппом Кантором (*Georg Ferdinand Ludwig Philipp Cantor*), русским (по матери – *ред. пер.*), отцом которого был датчанин. Но он написал свою «Теорию множеств» (*Mengenlehre*) на немецком! На этом языке целые числа могут иметь такое странное имя как *Zahlen*.

Вы можете подумать что теория множеств скучна, однако, многие люди считают ее довольно интересной. Например, есть аргентинец, которого ученые рассматривают как величайшего писателя, жившего после падения греческой цивилизации. Конечно, только Греция могла выдвинуть автора лучше. Вы, вероятно, слышали, что чилийцы говорят, что аргентинцы слишком воображают о себе. *Вы знаете, что является наилучшей сделкой? Заплатить настоящую цену за аргентинцев, а потом перепродать их за то, что они считают своей ценой.* Однако вопреки мнению чилийцев Хорхе Луис Борхес (*Jorge Luiz Borges*) является величайшим писателем, который писал на языке, отличном от греческого. Знаете ли вы, каков был его любимый предмет? Это была теория множеств, или *Der Mengenlehre*, как ему нравилось ее называть.

Пожалуйста, мои аргентинские друзья, не держите зла. В конце концов, ваша страна — четвёртая в списке моих любимых стран, после Греции, Франции и Парагвая. Парагвай находится в моём списке так высоко только потому, что Хосе Асунсьон Флорес (*José Asunción Flores*) был парагвайцем. Для тех, кто не знает Хосе Асунсьон Флореса — послушайте его песню «Индеанка» (*India*) и, я уверен, вы отметите Парагвай на своей карте. Что касается Аргентины, она является родиной таких людей, как Бернардо Усай (*Bernardo Houssay*), Луис Федерико Лелуар (*Luis Federico Leloir*), Сезар Мильштейн (*César Milstein*), Адольфо Перез Эскивел (*Adolfo Pérez Esquivel*), Карлос Сааведра Ламас (*Carlos Saavedra Lamas*), Хорхе Луис Борхес, Альберто Кальдерон (*Alberto Calderon*), Альберто Гинастера (*Alberto Ginastera*), Хосе Кура (*José Cura*), Дарио Волонте (*Dario Volonté*), Вероника Даль (*Verónica Dahl*), Каролина Монард (*Carolina Monard*), Че Гевара (*Che Guevara*), Диего Марадона (*Diego Maradona*), Хуан Мануэль Фангио (*Juan Manuel Fangio*), Освальдо Гольов (*Osvaldo Golijov*), Ева Перон (*Eva Peron*), Гектор Панидза (*Hector Panizza*) и др. Пока я не забыл, Хосе Асунсьон Флорес жил в Буэнос-Айресе. Но вернемся к множествам.

Когда математик хочет сказать, что элемент принадлежит множеству, он записывает это следующим образом:

$$3 \in \mathbf{Z}$$

Если он хочет сказать, что нечто не является элементом множества, например, он хочет сказать, что – 3 не является элементом **N**, он пишет:

$$-3 \notin \mathbf{N}$$

Давайте сведем воедино обозначения, которые используют преподаватели алгебры, когда они объясняют теорию множеств своим студентам.

[1] Автор включает 0 в множество натуральных чисел.

Двойная вертикальная черта[1]. Таинственное обозначение { x^2 || $x \in$ **N** } представляет множество квадратов x^2, таких что x является элементом **N**, т. е. множество {0, 1, 4, 9, 16, 25, …}.

Ограничения[2]. Если вы хотите сказать, что x является элементом **N**, при условии, что $x > 10$, то можете написать { x^2 || $x \in$ **N** $\wedge x > 10$ }.

Коньюнкция. В математике вы можете использовать символ $\wedge$, чтобы сказать *и*; таким образом, $x > 2 \wedge x < 5$ означает, что $x > 2$ *и* $x < 5$.

Дизъюнкция. Выражение

$$(x < 2) \vee (x > 5)$$

означает $x < 2$ *или* $x > 5$.

Используя вышеприведенные обозначения, вы можете определить множество рациональных чисел в виде:

$$Q = \left\{ \frac{p}{q} \middle\| p \in Z \wedge q \in N \wedge q \neq 0 \right\}.$$

На неформальном языке это означает, что рациональное число — это дробь вида

$$\frac{p}{q},$$

где p — элемент **Z**, а q — элемент **N**, и при этом q не равно 0.

Visual Prolog не имеет специального обозначения для множеств, но вы можете использовать для их представления списки. Например, выберите пункт *Project/New* главного меню и заполните диалоговое окно *Project Settings* следующим образом:

```
General
Project Name: zermelo
UI Strategy: console
```

Обратите внимание, что мы собираемся использовать консольную стратегию, а не GUI. Выберите пункт меню *Build/Build*, чтобы добавить прототип класса `zermelo` в дерево проекта. Отредактируйте файл `zermelo.pro` так, как показано ниже. Снова постройте проект и запустите его, используя команду *Build/Run In Window*.

```
implement main
    open core

clauses
    classInfo("main", "zermelo").

    run():-
```

[1] В математике обычно используется одинарная вертикальная черта, а не двойная.
[2] Автор употребляет слово guard — ограждение.

```
        console::init(),
        Q= [tuple(X, Y) || X= std::fromTo(1, 4),
                           Y= std::fromTo(1, 5)],
        stdio::nl,
        stdio::write(Q), stdio::nl, stdio::nl,
        foreach tuple(Num, Den)= list::getMember_nd(Q) do
            stdio::write(Num, "/", Den,", ")
        end foreach,
        stdio::nl.
end implement main

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

Вам может быть интересно, почему я назвал программу именем немецкого математика Эрнста Фридриха Фердинанда Цермело (*Ernst Friedrich Ferdinand Zermelo*). Одной из причин является то, что он был назначен на почётную должность во Фрайбурге в Брайсграу (*Freiburg im Breisgau*) в 1926 г., которую он покинул в 1935 г., потому что осуждал режим Гитлера. Другая причина состоит в том, что он изобрёл обозначение для множеств, которое мы использовали.

```
        Q= [tuple(X, Y) || X= std::fromTo(1, 4),
                           Y= std::fromTo(1, 5)]
```

В обозначениях Цермело это выражение записывается следующим образом:

$$\mathbf{Q} = \{(X, Y) \mid\mid X \in [1 \ldots 4] \wedge X \in [1 \ldots 5]\}$$

Говоря обычном языком, **Q** — это список пар (*X*, *Y*), таких, что *X* принадлежит [1, 2, 3, 4], а *Y* принадлежит [1, 2, 3, 4, 5]. Фрагмент

```
        foreach tuple(Num, Den)= list::getMember_nd(Q) do
            stdio::write(Num, "/", Den,", ")
        end foreach
```

говорит компьютеру выполнить `write(Num, "/", Den,", ")` для каждой пары `(Num, Den)`, которая является членом списка **Q**.

Множество рациональных чисел называется так потому, что его элементы могут быть представлены как дроби вида

$$\frac{p}{q},$$

где $p \in \mathbf{Z}$ и $q \in \mathbf{N}$, $q \neq 0$.

Следующий раздел несколько сложен для усвоения. Поэтому если вы хотите его пропустить, вы свободно можете это сделать. В разделе о действительных числах я приведу все важные результаты, поэтому вы не пропустите чего-либо стоящего и можете не затрачивать усилия на понимание целой страницы трудной математики.

9.4. Иррациональные числа

Во времена Пифагора[1] древние греки полагали, что любая пара линейных отрезков соизмерима, то есть, что вы всегда можете найти такую меру, что длины любых двух отрезков будут даны целыми числами относительно этой меры (т. е. единицы измерения — *ред*. *пер*.). Следующий пример поможет вам понять греческую теорию соизмеримых величин в работе. Рассмотрим квадрат, изображённый на рисунке 9.1.

Если бы греки были правы, я бы смог найти меру, возможно очень малую, которая производит одно целое измерение диагонали квадрата и другое целое измерение его стороны. Предположим, что p — это результат измерения стороны квадрата, а q — результат измерения диагонали. Теорема Пифагора утверждает, что $\overline{AC}^2 + \overline{CB}^2 = \overline{AB}^2$, поэтому

$$p^2 + p^2 = q^2 \;\therefore\; 2p^2 = q^2 \qquad (9.1)$$

Вы также можете выбрать меру так, чтобы p и q не имели общих делителей (отличных от единицы). Например, если как p, так и q делится на 2, то вы могли бы удваивать длину меры до тех пор, пока вы не получите значение, которое больше не делится на 2. Например, если $p = 20$ и $q = 18$, то удвоив длину меры, вы получите $p = 10$ и $q = 9$. Таким образом, можно предположить, что выбрана такая мера, что p и q не являются чётными одновременно. Но из уравнения (9.1) следует, что q^2 — чётное. А если q^2 чётное, то q также чётное. Вы можете проверить, что квадрат нечётного числа всегда является нечётным числом. Число q чётное, поэтому вы можете подставить вместо него $2 \times n$ в уравнение (9.1) .

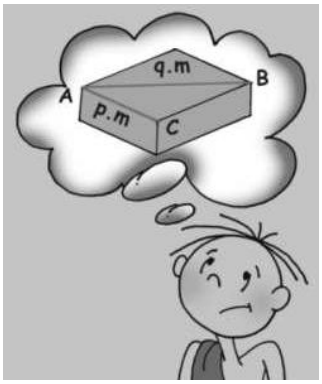


Рисунок 9.1 Думающий грек

$$2p^2 = q^2 = (2 \times n)^2 = 4 \times n^2 \;\therefore\; 2 \times p^2 = 4 \times n^2 \;\therefore\; p^2 = 2 \times n^2 \qquad (9.2)$$

Из уравнения (9.1) видно, что q чётное; уравнение (9.2) доказывает, что p также чётное. Но это противоречит нашему допущению, что p и q не являются одновременно чётными. Следовательно, p и q не могут быть одновременно целыми числами в уравнении (9.1), которое можно переписать в виде

$$\frac{p}{q} = \sqrt{2}\,.$$

Число $\sqrt{2}$, равное отношению диагонали квадрата к его стороне, не является элементом **Q**, т. е. $\sqrt{2} \notin \mathbf{Q}$. Это доказал греческий философ Гиппас (*Hippasus*) из Метапонта (*Metapontum*).

Греки были народом мудрых мужчин и женщин. Несмотря на это, они имели странную привычку советоваться с необразованной крестьянской девушкой в Дельфах, перед тем, как сделать что-то полезное. Следуя этой традиции, Гиппас спросил

[1] Речь идет о ранних пифагорейцах.

дельфийскую жрицу — необразованную девушку — что ему нужно сделать, чтобы задобрить Аполлона. Она велела ему измерить сторону и диагональ квадратного алтаря бога, используя одну и ту же меру. Доказав, что проблема неразрешима, Гиппас открыл тип чисел, которые не могли быть представлены в виде дроби. Такие числа называются иррациональными.

9.5. Действительные числа

Множество всех чисел, (целых,) иррациональных и рациональных, обозначается **R** и называется множеством действительных чисел. В Visual Prolog множество **Z** представлено множеством `integer`, хотя множество `integer` не покрывает все целые числа, но его достаточно для удовлетворения ваших нужд. Действительные числа Visual Prolog принадлежат множеству `real` — подмножеству **R**.

Если x ∈ `integer`, то программисты на Visual Prolog говорят, что x имеет тип `integer`. Они также говорят, что `r` имеет тип `real`, если `r` ∈ `real`. Есть также другие типы, кроме `integer` и `real`. Вот список примитивных типов.

integer — Целые числа между –2147483648 and 2147483647.
real — Действительные числа должны быть записаны с десятичной точкой: `3.4`, `3.1416`, ….
string — Заключённая в кавычки строка символов: `"3.12"`, `"Hippasus"`, `"pen"`, ….
char — Символы в одинарных кавычках: `'A'`, `'b'`, `'3'`, `' '`, `'.'`, ….

9.6. Математика

Я всегда думал, что удовлетворить запросам математиков очень тяжело, так как они такие требовательные, и мне ужасно трудно даже делать вид, что я следую формальным объяснениям. В частности, моё определение различных множеств чисел не такое строгое, как ожидало бы большинство математиков. Однако я не имею намерений писать для журнала AMS[1] или делать сколько-нибудь значительный вклад в теорию чисел. Я буду доволен, если окажусь способен дать нематематикам рабочие знания о затрагиваемых понятиях. Под рабочими знаниями я имею в виду то, что я надеюсь, что мои читатели смогут понять, что такое тип и как типы связаны с теорией множеств. В качестве вознаграждения я надеюсь, что читающие литературу студенты найдут книги Борхеса более лёгкими для усвоения после прочтения моего текста. Следовательно, мои аргентинские друзья смогут простить меня за шутки о них.

9.7. Форматирование

Метод *format* создаёт правильные представления примитивных типов данных для вывода. Например, он хорош для представления цветов в виде шестнадцатеричных чисел. В этом представлении основные цвета выглядят так: `0x00FF0000` (красный), `0x0000FF00` (зелёный) и `0x000000FF` (синий). Однако, если вы попытаетесь вывести эти значения, используя `stdio::write/n`, вы получите их в десятеричном представлении. Используя `string::format/n` так, как показано ниже, вы получите печать основных цветов в шестнадцатеричном представлении.

[1] American Mathematical Society (Американское математическое общество).

```
/*****************************
Project Name: primtype2
UI Strategy: console
*****************************/
% Файл main.pro
implement main
    open core

constants
    className = "main".
    classVersion = "primtype2".
clauses
    classInfo(className, classVersion).

class predicates
    color : (symbol, integer) procedure (i, o).
clauses
    color("red", 0x00FF0000) :- !.
    color("blue", 0x000000FF) :- !.
    color("green", 0x0000FF00) :- !.
    color(_, 0x0).

    run():- console::init(),
        color("red", C),
        S= string::format("%x\n", C),
        stdio::write(S), stdio::nl,
        succeed().
end implement main
goal
    mainExe::run(main::run).
```

Первый аргумент функции `format` определяет, каким вы хотите видеть вывод на печать. В данном примере аргументом является `"%x\n"`: вы хотите шестнадцатеричное представление числа, завершающееся возвратом каретки (`"\n"`). Аргументы, следующие за первым, показывают данные, которые вы хотите напечатать. Вот несколько примеров форматирования:

```
S= string::format("Pi=%4.3f\n", 3.14159)          Pi=3.142
S= string::format("%4.3e\n", 33578.3)             3.358e+004
S= string::format("Fac(%d)=%d", 5, 120)           Fac(5)=120
S= string::format("%s(%d)=%d", "f", 5, 120)       f(5)=120
```

Спецификация поля форматирования имеет вид:

%[-][0][width][.precision][type],

где дефис (-) означает, что поле выравнивается по левому краю, по умолчанию оно выравнивается по правому краю. Ноль перед *width* заполняет форматированную строку нулями до тех пор, пока не будет достигнута минимальная ширина; *width* определяет минимальный размер поля; *precision* определяет точность числа с плавающей точкой. Наконец, *type* может быть вида **f** (число с фиксированной точкой), **e** (научная запись действительных чисел), **d** (десятичное целое), **x** (шестнадцатеричное целое), **o**

(восьмеричное целое). Строки представляются в виде `"%s"`. Вы можете добавить любое сообщение в спецификацию формата; вы также можете добавить возврат каретки (`"\n"`).

9.8. Домены

Вам могут потребоваться другие типы данных, кроме примитивов, предоставленных Visual Prolog. В этом случае вам необходимо определить новые типы данных, которые вы поместите в ваш код. В предыдущих главах вы уже изучили, как объявлять домены. Вы также изучили, как создавать составные структуры данных, такие как списки. Следовательно, всё, что нам следует сделать в этом разделе, это вспомнить уже усвоенные понятия и собрать воедино то, что мы изучили.

9.8.1. Списки

Вы знаете, что списки — это упорядоченные последовательности элементов. Вы также знаете, как создавать домены списков и объявлять предикаты, использующие списки. Например, вот несколько доменов списков:

```
domains
    reals= real*. % Список действительных чисел
    integers= integer*. % Список целых чисел
    strings= string*. % Список строк
    rr= reals*. % Список списков действительных чисел
```

Раз вы имеете объявление домена, вы можете использовать его как любой другой примитивный тип. В случае списков объявление домена не обязательно, так как можно использовать списочные типы, такие как `real*`, `string*` или `integer*` прямо в объявлении предиката, как показано в программе, приведенной на рисунке 9.2.

9.8.2. Функторы

Исчисление предикатов, область логики, имеет тип данных, который называется функциональным символом. Функциональный символ имеет имя, или функтор, за которым следуют аргументы. Аргументы записываются между круглыми скобками и отделяются друг от друга запятыми. Короче говоря, они имеют точно такую же форму записи, что и функции. Вот несколько примеров функциональных символов:

```
author("Wellesley", "Barros", 1970)
book(author("Peter", "Novig", 1960),
          "Artificial Intelligence")
date("July", 15, 1875)
entry("Prolog", "A functional language")
index("Functors", 165)
sunday
```

Последний пример показывает, что функциональный символ может вообще не иметь аргументов. Сказав об этом, давайте посмотрим, как объявлять типы данных для новых функторов.

```
/*****************************
Project Name: newdomains
UI Strategy: console
*****************************/
% Файл main.pro
implement main
    open core

class predicates
    sum : (real*, real) procedure (i, o).

clauses
    classInfo("main", "newdomains").

    sum([], 0) :- !.
    sum([X|Xs], X+S) :- sum(Xs, S).

    run():- console::init(),
        sum([1, 2, 3, 4, 5, 6], A),
        stdio::writef("The sum is %-4.0f", A).
end implement main
goal
    mainExe::run(main::run).
```

Рисунок 9.2 Объявления списков

Следующий пример вычисляет день недели для любой даты между 2000-ым и 2099-ым годами. Он имеет следующие объявления доменов:

ДОМЕН	Комментарии
year= integer.	*year* является синонимом *integer*. Использование синонимов делает вещи яснее.
day= sun; mon; tue; wed; thu; fri; sat; err.	Этот домен имеет множество функциональных символов без аргументов. Каждый функциональный символ отделен от следующего точкой с запятой и представляет день недели.
month= jan; feb; mar; apr; may; jun; jul; aug; sep; oct; nov; dec.	Ещё один домен (тип) более, чем с одним функциональным символом.
month code= m(month, integer).	Здесь у нас функциональный символ с двумя аргументами.
numberOfDays= nd(integer);february.	Другой тип с двумя функциональными символами: первый функциональный символ имеет один аргумент; второй не имеет ни одного.

```
/*****************************
Project Name: dayOfWeek
UI Strategy: console
*****************************/
% Файл main.pro
% Эта программа вычисляет день недели
% для любого года между 2000 and 2099.
implement main
    open core

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

domains
    year= integer.
    day= sun; mon; tue; wed; thu; fri; sat; err.
    month= jan; feb; mar; apr; may; jun; jul; aug; sep; oct; nov; dec.
    month_code= m(month, integer).
    numberOfDays= nd(integer); february.

class predicates
    dayOfWeek : (integer, day) procedure (i, o).
    calculateDay : (string Month, integer Day_of_the_month,
                    year Y, day D)
        procedure (i, i, i, o).

class facts
    monthCode : (string, month_code, numberOfDays).

clauses
    classInfo(className, classVersion).

    monthCode("January", m(jan, 6), nd(31)).
    monthCode("February", m(feb, 2), february ).
    monthCode("March", m(mar, 2), nd(31)).
    monthCode("April", m(apr, 5), nd(30)).
    monthCode("May", m(may, 0), nd(31)).
    monthCode("June", m(jun, 3), nd(30)).
    monthCode("July", m(jul, 5), nd(31)).
    monthCode("August", m(aug, 1), nd(31)).
    monthCode("September", m(sep, 4), nd(30)).
    monthCode("October", m(oct, 6), nd(31)).
    monthCode("November", m(nov, 2), nd(30)).
    monthCode("December", m(dec, 4), nd(31)).

    dayOfWeek(0, sun) :- !.
    dayOfWeek(1, mon) :- !.
    dayOfWeek(2, tue) :- !.
    dayOfWeek(3, wed) :- !.
    dayOfWeek(4, thu) :- !.
```

```
        dayOfWeek(5, fri) :- !.
        dayOfWeek(6, sat) :- !.

        dayOfWeek(_, err).

        calculateDay(Month, MD, Year, D) :-
            Y= Year-2000, % Последние цифры года
            monthCode(Month, m(_M, C), _), % Код месяца
            !,
            S= Y+Y div 4+C+MD,
            R= S mod 7,     /* День недели как целое между 0 (sunday)
                и 6 (saturday) */
            dayOfWeek(R, D). % Функциональный символ дня
            calculateDay(_, _, _, err).

        run():-
            console::init(),
            calculateDay("May", 3, 2005, R),
            stdio::write("Day of the week: ", R), stdio::nl,
            succeed(). % place your own code here
    end implement main
    goal
        mainExe::run(main::run).
```

9.9. Немного о логике: Предложения Хорна

Предложение Хорна имеет самое большее один положительный литерал. Для математиков предложения Хорна обладают интересными свойствами. Однако для ученых, занимающихся компьютерными науками, они ещё более привлекательны, потому что могут быть представлены в следующем виде:

$H_1 \leftarrow + L_{11} \wedge + L_{12} \wedge + L_{13} \ldots$
$H_2 \leftarrow + L_{21} \wedge + L_{22} \wedge + L_{23} \ldots$
$H_3 \leftarrow + L_{31} \wedge + L_{32} \wedge + L_{33} \ldots$
...

где знак $\leftarrow$ является символом импликации, а $+ L_{ij}$ — отрицанием отрицательного литерала. Доказательство того, что клаузальное предложение $H \vee \neg T$ может быть представлено в виде $H \leftarrow T$, было приведено ранее. Запись вида $H \leftarrow T_1, T_2, T_3, \ldots$ привлекательна потому, что она имеет сходство с неформальными правилами, которые люди используют для выражения своих знаний:

Пойдёт дождь, если ласточки летают низко над землёй.

Поэтому неудивительно, что первые попытки использовать логику как язык программирования были сфокусированы на предложениях Хорна. Однако разработчики Пролога столкнулись с тем, что одних только предложений Хорна не достаточно для большинства приложений. Эта трудность была решена введением в предложения Хорна механизмов управления, таких как отсечение.

Глава 10: Как решать это в Прологе

Название этой главы — дань уважению Гельдеру Коэльо (*Helder Coelho*), первому автору лучшей книги по Прологу из когда-либо написанных «Как решать это в Прологе» [How To Solve It With Prolog]. Эта книга была напечатана в Португальской национальной лаборатории гражданского строительства (*Laboratorio Nacional de Engenharia Civil*), в которой работал Коэльо. Позднее Коэльо опубликовал «Пролог в примерах» (*Prolog by Example*) [Coelho/Cotta], которая тоже хороша, но не так хороша, как его первая книга.

Книга Коэльо — это собрание коротких задач, предложенных и решённых великими математиками и учёными в областики информатики. Всё это организовано в виде FAQ[1]. Задачи интересны, а их решения показательны. Что не менее важно, эта книга ещё и очень занимательная. Я надеюсь, что скоро можно будет увидеть ее новое издание, так как она раскрывает историю создания логического программирования, открытия и исследования его создателей.

Все программы в этой главе являются консольными приложениями, при этом я привожу только имплементацию класса `main`. Вы должны завершить все остальное. Например, если я приведу следующий код:

```
implement main
    open core
clauses
    classInfo("main", "hello").

clauses
    run():- console::init(), stdio::write("Hello, world!\n").
end implement main
goal
    mainExe::run(main::run).
```

вы должны создать консольный проект под названием `hello`.

10.1. Полезные примеры[2]

Пример[3]: найти все элементы списка L. Логическая программа:

```
implement main
    open core
class predicates
    member : (integer, integer*) nondeterm anyflow.
    member : (string, string*) nondeterm anyflow.
    test : (string*) procedure (i).
```

[1] Frequently Asked Questions — часто задаваемые вопросы.

[2] В оригинале Utilities.

[3] В оригинале используется выражение verbal statement — словесная формулировка, постановка задачи.

```
    test : (integer*) procedure (i).

clauses
    classInfo("main", "utilities").

    member(H, [H|_]).
    member(H, [_|T]) :- member(H, T).

    test(L) :- member(H, L),
        stdio::write(H), stdio::nl, fail or succeed().

    run():- console::init(), L= [2,3,4,5], test(L),
        S= ["a", "b", "c"], test(S).
end implement main
goal
    mainExe::run(main::run).
```

Пример показывает, что вы можете определить один предикат для разных доменов. Например, существует определение `member/2` для `string*` и другое определение для `integer*`. Это же относится к `test/1`.

Определение предиката `test/1` использует предикат `fail` для отката и вывода всех решений недетерминированного предиката `member`. В результате программа печатает все элементы списка.

Пример: проверить, является ли список U пересечением списков L1 и L2; найти пересечение списков L1 и L2.

```
implement main
    open core, stdio
class predicates
    isec:(integer*, integer*, integer*) nondeterm anyFlow.
    memb:(integer, integer*) nondeterm anyFlow.
    tst:(integer*, integer*, integer*, string) procedure(i, i, i, o).
    findsec:(integer*, integer*, integer*) procedure (i, i, o).
    length:(integer*, integer) procedure (i, o).

clauses
    classInfo("main", "intersection").

    memb(H, [H1|T]) :- H=H1; memb(H, T).

    isec(L1, L2, [H|U]) :- memb(H, L1), memb(H, L2), !,
    isec(L1, L2, U).
    isec(_, _, []).

    length([], 0) :- !.
    length([_|R], L+1) :- length(R, L).

    tst(L1, L2, U, R) :-findsec(L1, L2, S),
        length(U, LU), length(S, LS), LU= LS,
        isec(L1, L2, U), !, R= "yes"; R= "no".
```

```
    findsec([H|T], L, [H|U]) :- memb(H, L), !, findsec(T, L, U).
    findsec([_|T], L, U) :- findsec(T, L, U), !.
    findsec(_L1, _L2, []).

    run():- console::init(), L1= [3, 6, 4, 5],
        L2= [4, 5, 6], U1= [4, 6, 5], U2= [4, 3],
        tst(L1, L2, U2, Resp2), tst(L1, L2, U1, Resp1),
        write(Resp1, ", ", Resp2), nl,
        findsec(L1, L2, I), write(I), nl.
end implement main
goal
    mainExe::run(main::run).
```

Пример: определить отношение `append/3` между тремя списками, такое, что последний список является результатом соединения первых двух списков.

```
implement main
    open core
class predicates
    app : (Elem* L1, Elem* L2, Elem* L3) nondeterm anyFlow.
    test1 : (string*) procedure (i).
    test2 : (integer*, integer*) procedure (i, i).
clauses
    classInfo("main", "append").

    app([], L, L).
    app([H|T], L, [H|U]) :- app(T, L, U).

    test1(U) :- app(L1, L2, U),
        stdio::write(L1, " ++ ", L2, "= ", U), stdio::nl, fail.
    test1(_).

    test2(L1, L2) :- app(L1, L2, U),
        stdio::write(L1, " ++ ", L2, ": ", U), stdio::nl, fail.
    test2(_, _).

clauses
    run():- console::init(),
        test1(["a", "b", "c", "d"]), stdio::nl,
        test2([1, 2, 3], [4, 5, 6]).
end implement main
goal
    mainExe::run(main::run).
```

Эта задача является фаворитом всех времён. Если вы запустите программу, то предикат `test1/1` найдёт и выведет все способы разбиения списка `["a", "b", "c", "d"]`. С другой стороны, предикат `test2/2` соединяет два списка. Оба предиката используют предикат `app/3`, который осуществляет две операции: соединение списка и его разбиение.

Пример: написать программу обращения списка.

```
/*****************************
Project Name: reverse
UI Strategy: console
*****************************/
implement main
    open core

class predicates
    rev : (integer*, integer*) procedure (i, o).
    rev1 : (integer*, integer*, integer*) procedure (i, i, o).

clauses
    classInfo("main", "reverse").

    rev1([], L, L) :- !.
    rev1([H|T], L1, L) :- rev1(T, [H|L1], L).

    rev(L, R) :- rev1(L, [], R).

    run():- console::init(), L= [1, 2, 3, 4, 5],
        rev(L, R), stdio::write("Reverse of", L, "= ", R),
        stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

Эта программа является полезной и показательной. Она *показательна*, потому что показывает, как использовать накопители очень эффективным образом. Предположим, что вы зададите цель

```
rev([1, 2, 3], R).
```

Тогда машина вывода попытается приравнять `rev([1, 2, 3], R)` заголовку одного из находящихся в программе предложений Хорна. В данном случае успех будет достигнут, если положить `L=[1, 2, 3]` в предложении

```
rev(L, R) :- rev1(L, [], R)
```

Но если `L=[1, 2, 3]`, то это предложение принимает вид:

```
rev([1, 2, 3], R) :- rev1([1, 2, 3], [], R)
```

Этот процесс сопоставления цели и головы правила называется унификацией, при этом говорят, что `rev([1, 2, 3], R)` унифицируется с

```
rev(L, R) :- rev1(L, [], R)
```
, при `L=[1, 2, 3]`

и приводит к `rev1(L, [], R)`, что равно `rev1([1, 2, 3], [], R)`, так как `L=[1, 2, 3]`. Результат унификации приводит к хвосту предложения Хорна, потому что вы должны

доказать хвост, для того чтобы доказать заголовок. Например, рассмотрим следующее предложение Хорна на естественном языке:

X — дедушка Y, если X — отец Z и Z — отец Y.

Если вы хотите доказать, что X — дедушка Y, то вы должны доказать, что Y — отец Z и Z — отец Y. Короче говоря, нужно доказать хвост, для того чтобы доказать голову предложения Хорна. Все это долгое обсуждение приводит к следующему:

- `rev([1,2,3], R)` унифицируется с

  `rev(L, R) :- rev1(L, [], R)`, при `L=[1,2,3]`

  и возвращает `rev1([1,2,3], [], R)`.
- `rev1([1,2,3], [], R)` унифицируется с

  ```
  rev1([H|T],L1,L) :- rev1(T,[H|L1],L) / H=1, L1=[], T=[2,3]
  ```

  и приводит к `rev1([2, 3], [1], L)`.
- `rev1([2, 3], [1], L)` унифицируется с

  ```
  rev1([H|T],L1,L) :- rev1(T,[H|L1],L) / H=2, L1=[1], T=[3]
  ```

  и возвращает `rev1([3], [2,1], L)`.
- `rev1([3], [2,1], L)` унифицируется с

  ```
  rev1([H|T],L1,L) :- rev1(T,[H|L1],L) / H=3, L1=[2,1], T=[]
  ```

  и возвращает `rev1([], [3,2,1], L)`.
- `rev1([], [3,2,1], L)` унифицируется с

  `rev1([], L, L) :- !.` для `L=[3,2,1]`

  и приводит к результату `L=[3,2,1]`.

Лаплас (*Laplace*) говаривал, что Ньютон был не только величайшим учёным всех времен и народов, но еще и очень удачливым человеком, потому, что родился до Лапласа. Конечно, *Le Marquis*[1] мог бы открыть Небесную Механику до Ньютона, если бы имел возможность это сделать. Я полагаю, что вы бы написали алгоритм быстрой сортировки до Ван Эмдена (*Van Emden*), если бы он не был достаточно удачлив и не приложил руку к созданию компилятора Пролога до вас. Алгоритм быстрой сортировки был изобретён Тони Хоаром (*Tony Hoare*), которому был нужен быстрый метод сортировки списков и векторов. В то время B+деревьев еще не существовало, поэтому единственным способом осуществить быстрый поиск в списке клиентов был способ хранить его в отсортированном виде. Давайте посмотрим, как Ван Эмден объяснял этот алгоритм своему другу Коэльо. Прежде всего он реализовал предикат `split/4`, который делит список `L` на два подсписка: *S*(*mall*) и *B*(*ig*). Подсписок `S` содержит все элементы списка `L`, которые меньше или равны *P*(*ivot*). Подсписок `B` содержит элементы, которые больше, чем `P`. Вот как Ван Эмден реализовал предикат `split/4`:

[1] Маркиз (*фр.*).

```
implement main
    open core
clauses
    classInfo("main", "quicksort").

class predicates
    split:(E, E*, E*, E*) procedure (i, i, o, o).
clauses
    split(P, [H|T], S, [H|B]) :- H>P, !, split(P, T, S, B).
    split(P, [H|T], [H|S], B) :- !, split(P, T, S, B).
    split(_, [], [], []).

    run():- console::init(),
        split(5, [3, 7, 4, 5, 8, 2], Small, Big),
        stdio::write("Small=", Small, ", Big= ", Big), stdio::nl,
        split("c", ["b", "a", "d", "c", "f"], S1, B1),
        stdio::write("Small=", S1, ", Big= ", B1), stdio::nl.

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

Если вы запустите эту программу, то компьютер выдаст

```
Small=[3,4,5,2], Big=[7,8]
```

что он и должен делать. Теперь давайте поймём идею, стоящую за алгоритмом быстрой сортировки. Предположим, что у вас есть список `[5, 3, 7, 4, 5, 8, 2]`, и вы хотите упорядочить его.

- Берём первый элемент `L=[5, 3, 7, 4, 5, 8, 2]` как ведущий и делим список `L` на `Small=[3,4,5,2]` и `Big=[7,8]`.
- Сортируем `Small`, используя тот же алгоритм; в результате получим
  `Sorted_small=[2, 3, 4, 5]`
- Сортируем `Big` и получаем `Sorted_Big=[7, 8]`.
- Присоединяем `Sorted_small` к `[Pivot|Sorted_Big]`. Вы получите сортированный список: `[2, 3, 4, 5, 5, 7, 8]`.

Этому алгоритму нужен предикат, который присоединяет один список к другому. Вы можете использовать предикат `app/3`, который вы изучили ранее. Однако тот алгоритм является недетерминированным, в чем нет необходимости, так как нам не нужна возможность разбиения списка на два. Всё, что нам нужно — это обычная конкатенация. Таким образом, вы можете вставить отсечение после первого предложения для `app/3`.

```
app([], L, L) :- !.
app([H|T], L, [H|U]) :- app(T, L, U).
```

Объявите app как процедуру с двумя входными аргументами и одним выходным.

```
app : ( integer_list, integer_list, integer_list)
    procedure (i, i, o).
```

Для того чтобы протестировать алгоритм, мы собираемся использовать список целых чисел. Конечно, было бы более полезным использовать список строк.

```
implement main
    open core
clauses
    classInfo("main", "quicksort").

class predicates
    split : (E, E*, E*, E*) procedure (i, i, o, o).
    app : (E*, E*, E*) procedure (i, i, o).
    sort : (E*, E*) procedure (i, o).

clauses
    split(P, [H|T], S, [H|B]) :- H>P, !, split(P, T, S, B).
    split(P, [H|T], [H|S], B) :- !, split(P, T, S, B).
    split(_, [], [], []).

    app([], L, L) :- !.
    app([H|T], L, [H|U]) :- app(T, L, U).

    sort([P|L], S) :- !, split(P, L, Small, Big),
        sort(Small, QSmall), sort(Big, QBig),
        app(QSmall, [P|QBig], S).
    sort([], []).

    run():- console::init(), L= [5, 3, 7, 4, 5, 8, 2],
        sort(L, S),
        stdio::write("L=", L , ", S= ", S), stdio::nl,
        Strs= ["b", "a", "c"], sort(Strs, Qs),
        stdio::write("Qs= ", Qs), stdio::nl.
end implement main

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

Пролог был изобретен Аланом Колмероэ (*Alain Colmerauer*), французским лингвистом. Его студент Филипп Руссель (*Philippe Roussel*) реализовал интерпретатор Пролога. Возможно, что название «Пролог» языку программирования дала жена Русселя. Уоррен (*Warren*) первым реализовал компилятор Пролога. Приведённая ниже задача была предложена Уорреном.

Пример: количество дней в году равно 366 каждые четыре года; в остальных случаях оно равно 365. Сколько дней в 1975-ом, 1976-ом и 2000-ом годах?

```
implement main
    open core

class predicates
    no_of_days_in:(integer, integer) procedure (i, o).
    member:(integer, integer_list) nondeterm (o, i).
```

```
    test:(integer_list) procedure (i).
clauses
    classInfo("numdays", "1.0").

    member(H, [H|_T]).
    member(H, [_|T]) :- member(H, T).

    no_of_days_in(Y, 366) :-
        0= Y mod 400, !.
    no_of_days_in(Y, 366) :-
        0= Y mod 4,
        not(0= Y mod 100), !.
    no_of_days_in(_Y, 365).

    test(L) :- member(X, L),
        no_of_days_in(X, N),
        stdio::write("Year ", X, " has ", N, " days."),
        stdio::nl, fail.
    test(_L).

    run():- console::init(),
        test([1975, 1976, 2000]).
end implement main
goal
    mainExe::run(main::run).
```

Луис Мониш Перейра (*Luis Moniz Pereira*) — это португалец, который работает в области искусственного интеллекта. Он был одним из первых, кто использовал Пролог для искусственного интеллекта. Он также является соавтором книги «Как решать это в Прологе».

Пример: написать программу для раскрашивания произвольной плоской карты не более чем четырьмя цветами так, чтобы никакие два соседних региона не были окрашены в один и тот же цвет.

Эту задачу решает классическая программа порождения и проверки. В предикате `test(L)` вызовы

```
generateColor(A), generateColor(B),
generateColor(C), generateColor(D),
generateColor(E), generateColor(F),
```

порождают цвета для шести регионов карты. Затем `test(L)` строит карту в виде списка пар соседних стран:

```
L= [nb(A, B), nb(A, C), nb(A, E), nb(A, F),
    nb(B, C), nb(B, D), nb(B, E), nb(B, F),
    nb(C, D), nb(C, F), nb(C, F)]
```

Наконец предикат `aMap(L)` проверяет, является ли `L` допустимой картой. Она будет допустимой, если никакие из двух соседних стран не будут окрашены в один цвет. Если

предикат `aMap(L)` является ложным, то предложенные цвета не являются правильными. Поэтому программа откатывается к вызову `generateColor(X)` для получения нового набора цветов.

Задача четырёх красок интересна по двум причинам.

1. Схема порождения и проверки — это техника, которая одной из первых применялась в искусственном интеллекте.

2. Существует очень известная теорема, которая утверждает:

   Любая карта может быть раскрашена четырьмя цветами так, что никакие две соседние страны не будут окрашены в один цвет.

Эта теорема была доказана в 1976 году Кеннетом Аппелем (*Kenneth Appel*) и Вольфгангом Хакеном (*Wolfgang Haken*), двумя математиками из университета Иллинойса.

```
implement main
    open core
domains
    colors= blue; yellow; red; green.
    neighbors= nb(colors, colors).
    map= neighbors*.

class predicates
    aMap : (map) nondeterm anyFlow.
    test : (map) procedure anyFlow.
    generateColor : (colors) multi (o).

clauses
    classInfo("main", "fourcolors").

    generateColor(R) :-
        R= blue; R= yellow;
        R= green; R= red.

    aMap([]).
    aMap([X|Xs]) :-
        X= nb(C1, C2), not(C1 = C2),
        aMap(Xs).

    test(L) :-
        generateColor(A), generateColor(B),
        generateColor(C), generateColor(D),
        generateColor(E), generateColor(F),
        L= [ nb(A, B), nb(A, C), nb(A, E), nb(A, F),
             nb(B, C), nb(B, D), nb(B, E), nb(B, F),
             nb(C, D), nb(C, F), nb(C, F)],
        aMap(L), !; L= [].

    run():- console::init(), test(L),
```

```
        stdio::write(L), stdio::nl.
end implement main

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

Пример: написать программу для игры в Ним. Состояние в игре Ним может быть описано в виде множества кучек спичек. Мы будем представлять каждую кучку спичек в виде списка целых чисел.

```
[1, 1, 1, 1, 1]
```

Следовательно множество кучек является списком списков. Например

```
[ [1,1,1,1,1],
  [1,1],
  [1,1,1,1] ]
```

Два игрока, вы и компьютер, по очереди делаете ходы. Как только игрок не сможет сделать ход, игра заканчивается, и этот игрок проигрывает. Ход состоит в вытаскивании из ровно одной кучки как минимум одной спички. Если вы возьмёте в нашем примере три спички из третьей кучки, то вы сделаете допустимый ход, и состояние станет выглядеть следующим образом:

```
[ [1,1,1,1,1],
  [1,1],
  [1] ]
```

Для того чтобы реализовать этот проект, мы будем использовать технику программирования под названием инкрементальная разработка систем (*incremental development of systems*). Сначала вы реализуете и тестируете программу, которая соединяет два списка. Так как вы собираетесь использовать эту программу как для разбиения, так и для конкатенации списков, вам нужно протестировать обе эти возможности. Действительно, если вы запустите первую программу, приведённую ниже, то получите следующее:

```
[[1],[1],[1],[1],[1],[1],[1],[1]]
[][[1],[1],[1,1]]
[[1]][[1],[1,1]]
[[1],[1]][[1,1]]
[[1],[1],[1,1]][]
```

Первая строка является результатом конкатенации двух множеств кучек. Последующие строки показывают все возможности разбиения списка `[[1],[1],[1,1]]`. Поэтому первая программа, по-видимому, работает верно.

```
implement main
    open core
domains
    ms= integer*.
    hs= ms*.
```

```
class predicates
    append : (E*, E*, E*) nondeterm anyFlow.

clauses
    classInfo("main", "nimgame").

    append([], Y, Y).
    append([U|X], Y, [U|Z]) :- append(X, Y, Z).

clauses
    run():-
        console::init(),
        append([[1],[1],[1],[1],[1]], [[1],[1],[1]], L), !,
        stdio::write(L), stdio::nl; succeed().
end implement main

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

Следующий шаг состоит в том, чтобы написать предикат, который берет по крайней мере одну спичку из кучки; конечно, он может взять более одной спички. После удаления одной или более спичек из кучки предикат `takesome/3` вставляет измененную кучку во множество кучек. Если вы протестируете программу, она напечатает следующий результат:

```
[[1,1,1,1],[1,1]]
[[1,1,1],[1,1]]
[[1,1],[1,1]]
[[1],[1,1]]
[[1,1]]
```

NB: первое предложение для предиката `takesome/3` обеспечивает то, что предикат не вставит пустую кучку во множество кучек.

```
implement main
    open core

domains
    ms= integer*.
    hs= ms*.

class predicates
    append : (E*, E*, E*) nondeterm anyFlow.
    test : () procedure.
    takesome : (ms, hs, hs) nondeterm anyFlow.

clauses
    classInfo("main", "nimgame").

    append([], Y, Y).
    append([U|X], Y, [U|Z]) :- append(X, Y, Z).
```

```
    takesome([_X], V, V).
    takesome([_X, X1|Y], V, [[X1|Y]|V]).
    takesome([_X|T], V, Y) :- takesome(T, V, Y).

    test() :- L= [1, 1, 1, 1, 1],
        V= [[1, 1]],
        takesome(L, V, Resp),
        stdio::write(Resp), stdio::nl,
        fail; succeed().

clauses
    run():- console::init(),
        test().
end implement main

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

Теперь вы готовы сгенерировать все возможные ходы из заданной позиции. Если вы запустите тестовую программу, она выдаст следующее:

```
Possible moves from [[1,1,1],[1,1]]: % Возможные ходы
[[1,1],[1,1]]
[[1],[1,1]]
[[1,1]]
[[1,1,1],[1]]
[[1,1,1]]
```

```
implement main
    open core
domains
    ms= integer*.
    hs= ms*.

class predicates
    append : (E*, E*, E*) nondeterm anyFlow.
    takesome : (ms, hs, hs) nondeterm anyFlow.
    move : (hs, hs) nondeterm anyFlow.

clauses
    classInfo("main", "nimgame").

    append([], Y, Y).
    append([U|X], Y, [U|Z]) :- append(X, Y, Z).

    takesome([_X], V, V).
    takesome([_X, X1|Y], V, [[X1|Y]|V]).
    takesome([_X|T], V, Y) :- takesome(T, V, Y).

    move(X, Y) :- append(U, [X1|V], X),
        takesome(X1, V, R), append(U, R, Y).
```

```
    run():- console::init(), L= [[1, 1, 1], [1, 1]],
        stdio::write("Possible moves from ", L, ": "),
        stdio::nl, move(L, Resp),
        stdio::write(Resp), stdio::nl, fail; succeed().
end implement main
goal mainExe::run(main::run).
```

Сердцем программы является предикат

```
winningMove(X, Y) :- move(X, Y), not(winningMove(Y, _)).
```

Если выигрышная стратегия существует, то этот потрясающий предикат в одну строчку найдёт её!

```
implement main
    open core
domains
    ms= integer*.
    hs= ms*.

class predicates
    append : (E*, E*, E*) nondeterm anyFlow.
    takesome : (ms, hs, hs) nondeterm anyFlow.
    move : (hs, hs) nondeterm anyFlow.
    winningMove : (hs, hs) nondeterm (i, o).

clauses
    classInfo("main", "nimgame").

    append([], Y, Y).
    append([U|X], Y, [U|Z]) :- append(X, Y, Z).

    takesome([_X], V, V).
    takesome([_X, X1|Y], V, [[X1|Y]|V]).
    takesome([_X|T], V, Y) :- takesome(T, V, Y).

    move(X, Y) :- append(U, [X1|V], X),
        takesome(X1, V, R), append(U, R, Y).

    winningMove(X, Y) :- move(X, Y), not(winningMove(Y, _)).

    run():- console::init(), L= [[1, 1, 1], [1, 1]],
        winningMove(L, S),
        stdio::write("Winning move: ", S),
        stdio::nl, fail; succeed().
end implement main
goal mainExe::run(main::run).
```

Ниже приведен пример игры против компьютера. Как видите, если существует возможность выиграть, то машина это сделает.

```
[[1,1,1],[1,1]]
```

```
Your move: [[1], [1,1]]    % Ваш ход
My move: [[1],[1]]         % Мой ход
Your move: [[1]]
My move: []
I win                      % Я победил
```

Для того чтобы реализовать программу, вам осталось создать класс для самой игры. Пусть этот класс называется `nim`. Основная программа приведена ниже.

```
% Файл main.pro
implement main
    open core, stdio

clauses
    classInfo("main", "nimgame-1.0").

    run():- console::init(), L= [[1, 1, 1], [1, 1]],
        write(L), nl, nim::game(L), !; succeed().
end implement main
goal mainExe::run(main::run).
```

Декларация следующего класса содержит метод `game/1` и домены, которые вам необходимы для описания состояния игры. Этот класс определяется ниже.

```
% Файл nim.cl
class nim
    open core
domains
    ms= integer*.
    hs= ms*.
predicates
    classInfo : core::classInfo.
    game : (hs) determ (i).
end class nim

% Файл nim.pro
implement nim
    open core, stdio

class predicates
    append : (hs, hs, hs) nondeterm anyFlow.
    takesome : (ms, hs, hs) nondeterm anyFlow.
    move : (hs, hs) nondeterm anyFlow.
    winningMove : (hs, hs) nondeterm (i, o).
    iwin : (hs) determ (i).
    youwin : (hs, hs) determ (i, o).

clauses
    classInfo("nim", "1.0").

    append([], Y, Y).
    append([U|X], Y, [U|Z]) :- append(X, Y, Z).
```

```
        takesome([_X], V, V).
        takesome([_X, X1|Y], V, [[X1|Y]|V]).
        takesome([_X|T], V, Y) :- takesome(T, V, Y).

        move(X, Y) :- append(U, [X1|V], X),
            takesome(X1, V, R), append(U, R, Y).

        winningMove(X, Y) :- move(X, Y), not(winningMove(Y, _)).

        iwin([]) :- !, write("I win"), nl.

        youwin(L, L2) :- winningMove(L, L2), !.
        youwin(_L, _L2) :- write("You win"), nl, fail.

        game(L1) :- write("Your move: "),
            L= console::read(), move(L1, LTest),
            L= LTest, youwin(L, L2), !,
            write("My move: "), write(L2), nl,
            not(iwin(L2)), game(L2).
    end implement nim
```

Стратегия, которая используется в этой очень простой программе, называется алгоритмом минимакса. Этот алгоритм может применяться в любой игре с двумя соперниками, где оба имеют полное знание о состоянии игры. Примеры таких игр: шахматы, Го, шашки, крестики-нолики и Ним. Он также может быть использован для управления роботами. На самом деле, существует разновидность фильтров, основанных на алгоритме минимакса, которая аналогична фильтру Калмана.

В стратегии минимакса состояния игры образуют дерево, которое называется деревом поиска. Когда наступает ход компьютера, он генерирует состояние с минимальным значением для оппонента. С другой стороны, он пытается следовать ветвям дерева, которые ведут к выигрышной позиции. В простой игре, такой как Ним, компьютер способен найти путь, обеспечивающий победу. В более сложных играх, таких как шахматы, это не всегда возможно — необходимо прерывать поиск до того, как будет достигнута уверенность в победе. В этом случае игровые стратеги придумывают функции, которые присваивают значение позиции, даже если неизвестно, выигрышная это позиция или проигрышная.

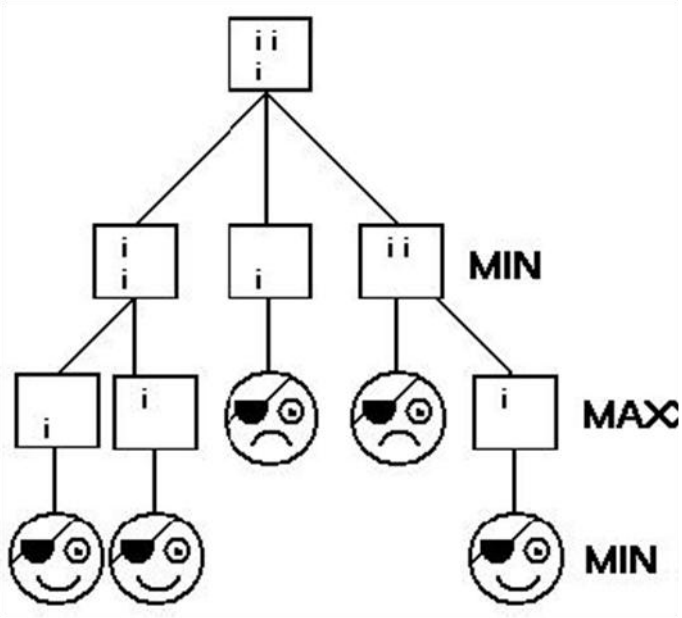


Рисунок 10.1 Дерево минимакса

Глава 11: Факты

Факт — это предложение Хорна, в котором нет тела. Факты могут быть добавлены, изменены или удалены динамически, в ходе исполнения программы. Следующий пример поможет вам понять, почему факты необходимы в Прологе. Предположим, что вы хотите создать маленькую базу данных публикаций. Когда у вас есть новая книга, вы хотите добавить её в базу данных.

```
addItem(journal("AI in focus", "MIT Press"))
```

Предикат `addItem/1` можно определить следующим образом:

```
class predicates
    addItem : (volume).
clauses
    addItem(V) :-
        num := num+1,
        assert(item(V, num)).
```

В результате выполнения `addItem(journal("AI", "AldinePress"))` в базу данных добавляется предложение

```
item(journal("AI", "AldinePress")).
```

и увеличивается на единицу значение переменной `num`. Следующее объявление

```
class facts - bib
    num : integer := 0.
    item : (volume, integer) nondeterm.
```

создаёт базу фактов `bib` с помощью переменной `num` и предиката `item/2`. Предикат имеет тип *nondeterm*, а переменная — тип *single*.

Домен `volume`, который обеспечивает типизацию записей в этой маленькой базе данных, определяется в виде

```
domains
    name= string.
    author= n1(name); n2(name, name); etal(name).
    publisher= string.
    title= string.
    volume= journal(title, publisher); book(title, author).
```

Программа, приведенная ниже, показывает, как определять факты и как сохранять факты базы данных в файле.

```
% Файл main.pro
implement main
    open core
```

```
domains
    name= string.
    author= n1(name); n2(name, name); etal(name).
    publisher= string.
    title= string.
    volume= journal(title, publisher); book(title, author).

class facts - bib
    num:integer := 0.
    item:(volume, integer) nondeterm.

class predicates
    addItem:(volume).
    prtDataBase:().

clauses
    classInfo("main", "facttest").

    addItem(V) :- num := num+1,
        assert(item(V, num)).

    prtDataBase() :- item(V, I),
        stdio::write(I, "=", V), stdio::nl,
        fail.
    prtDataBase().

clauses
    run():-
        console::init(),
        addItem(journal("AI in focus", "MIT Press")),
        addItem(book( "Databases in Prolog",
                n1("Wellesley Barros"))),
        file::save("bibliography.fac", bib),
        prtDataBase().
end implement main
goal
    mainExe::run(main::run).
```

После создания базы данных и сохранения её в файле вы можете использовать её в другой программе. Для того чтобы увидеть это, создайте консольный проект `factread` и добавьте в него следующий код:

```
% Файл main.pro
implement main
    open core
domains
    name= string.
    author= n1(name); n2(name, name); etal(name).
    publisher= string.
    title= string.
    volume= journal(title, publisher); book(title, author).
```

```
class facts - bib
    num:integer := 0.
    item:(volume, integer) nondeterm.

class predicates
    prtDataBase:().

clauses
    classInfo("main", "factread").

    prtDataBase() :- item(V, I),
        stdio::write(I, "=", V), stdio::nl,
        fail.
    prtDataBase().

clauses
    run():- console::init(),
        file::consult("bibliography.fac", bib),
        prtDataBase().
end implement main
goal
    mainExe::run(main::run).
```

Вы должны переместить файл

```
bibliography.fac,
```

созданный приложением `facttest`, в папку *factread/exe*. Затем запустите программу

```
factread.exe.
```

Вы увидите, что программа прочитает базу данных и напечатает её.

11.1. Класс file

Вы использовали класс *file* для сохранения фактов в файле. В классе *file* имеется много других полезных предикатов, которые вы изучите в данном разделе.

11.1.1. Чтение и запись строки

Часто вам требуется прочитать текст из файла, сделать с ним что-нибудь, а затем записать его обратно. В прошлом, когда у компьютеров было не так много памяти, существовала необходимость читать потоки символов. Сегодня память даже персональных компьютеров исчисляется гигабайтами. Поэтому наилучший подход к решению проблемы — прочитать весь файл в строку, а затем использовать мощный класс *string* для её обработки. Пролог имеет два предиката для работы такого типа:

```
readString : (string FileName, boolean IsUnicodeFile)
    -> string String procedure (i,o).
```

```
writeString : (string Filename, string Source, boolean IsUnicodeFile)
    procedure (i,i,i).
```

Оба предиката имеют версии, не требующие указания, имеет ли файл формат Unicode.

```
readString : (string FileName)
    -> string String procedure (i).
writeString : (string Filename, string Source) procedure (i,i).
```

На рисунке 11.1 показан пример использования этих предикатов. В данном примере из файла считывается строка, затем ее символы преобразуются в символы верхнего регистра и новая строка записывается в другой файл.

Я полагаю, что вы способны понять функциональность класса *file* с помощью справочного руководства. Поэтому я не буду задерживаться на этой теме.

```
% Файл main.pro
implement main
    open core

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

clauses
    classInfo(className, classVersion).

clauses
    run():-
        console::init(),
        Str= file::readString("test.txt", Unicode),
        stdio::write(Str),
        S_Upper= string::toUpperCase(Str),
        file::writeString("upper.txt", S_Upper, Unicode),
        succeed(). % place your own code here

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

Рисунок 11.1 Чтение строки из файла

11.2. Константы

Для того чтобы научиться использовать константы построим форму, к которой подключим главное меню. Visual Prolog, как и большинство других языков, представляет пункты меню с помощью целых чисел. Это представление хорошо для компьютера, но не так хорошо для человека, который будет анализировать программу. Поэтому Visual

Prolog предоставляет дескрипторы пунктов меню в виде констант с осмысленными именами.

- Создайте новый проект.

```
Project Name: janua
UI Strategy: Object-oriented GUI (pfc/gui)
```

Кстати говоря, JANUA — это латинское слово, означающее дверь. *Окно* — это маленькая дверь, или JANELLA. Входная дверь года — это MENSIS JANUARIUS, или *January* — январь.

- Создайте новую форму в новом пакете, пусть форма называется `txtWin`. С помощью диалогового окна *Properties* добавьте `TaskMenu` к форме `txtWin`.

- Постройте проект. Используйте вкладку *Events* диалогового окна *Properties*, для того чтобы вставить в *MenuItemListener* следующий фрагмент кода:

```
clauses
    onMenuItem(_Source, M) :-
        Item= resourceIdentifiers::id_help_contents,
        M= Item, !,
        stdio::write("Hello").
    onMenuItem(_Source, _M) .
```

- Включите пункт *File/New* в TaskMenu и добавьте фрагмент

```
clauses
    onFileNew(S, _MenuTag) :-
        X= txtWin::new(S), X:show().
```

для *TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.*

Глава 12: Классы и объекты

Создадим опытный консольный проект, для того чтобы достичь лучшего понимания того, как работают классы и объекты.

- Создайте новый консольный проект.

```
Project Name: classexample
UI Strategy: console
```

Постройте (*Build/Build*) приложение. Создайте новый класс: `account`. Не убирайте галочку *Creates Objects*.

- Измените файл `account.i` следующим образом:

```
% Файл account.i
interface account
    open core
predicates
    ssN : (string SocialSecurity) procedure (o).
    setSocialSecurity : (string SSN) procedure(i).
    deposit : (real).
end interface account
```

Класс `account` будет создавать новый объект — счет (*account*) всякий раз, когда вы вызовете метод

```
A=account::new().
```

Вы можете посылать сообщения этому объекту. Например, вы можете сохранить в нем номер социального страхования[1] клиента:

```
A:setSocialSecurity("AA345")
```

Впоследствии вы сможете извлечь этот самый номер социального страхования:

```
A:ssN(SocialSecurity)
```

Как `ssN/1`, так и `setSocialSecurity/1` должны быть объявлены в интерфейсе, который находится в файле `account.i`. Для работы со счётом требуются и другие методы. Например, вам могут понадобиться методы для депозитов, паролей и т.д. Я оставлю создание этих методов вашему воображению. Интересным дополнением могли бы быть методы криптографии.

- Измените файл `account.pro` так, как показано ниже.

[1] Social Security Number — SSN.

```
% Файл account.pro
implement account
    open core
facts - customer
    funds : real := 3.0.
    personalData : (string Name, string SocialSecurity) single.
clauses
    classInfo("account", "1.0").

    personalData("", "").

    ssN(SS) :- personalData(_, SS).

    setSocialSecurity(SSN) :- personalData(N, _),
        assert(personalData(N, SSN)).

    deposit(Value) :- funds := funds+Value.
end implement account
```

- Измените предикат `run/0` в файле `main.pro`, для того чтобы протестировать новый класс.

```
run():- console::init(),
    A= account::new(), A:setSocialSecurity("AA345"),
    A:ssN(SocialSecurity),
    stdio::write(SocialSecurity), stdio::nl.
```

12.1. Факты объектов

Объекты обладают фактами, которые могут быть использованы для хранения их состояния. Философ Витгенштейн очень любил факты и состояния вещей. Давайте посмотрим на первые несколько предложений его «Логико-философского трактата»[1] (*Tractatus Logico-Philosophicus*).

1. *Мир есть всё, что имеет место.*
   The World is everything that is the case.
   В немецком оригинале философ использовал слово *Fall*: *Die Welt ist alles, was der Fall ist.* Латинское слово *casus* означает *падение*, которое объясняет перевод. Однако, что падает в мире? Падают кости Фортуны. Имеется масса возможных событий. Фортуна выбирает то, что случается, с помощью игральных костей[2]. По крайней мере, это то, во что верили римляне. Вы

[1] См. **Витгенштейн Л.** Логико-философский трактат / Пер. с нем.: И.С. Добронравов; пер. с англ.: Д.Г. Лахути. Общ. ред. и предисл. В.Ф. Асмуса. М.: Наука, 1958. — 133 с.

[2] *Д.Г. Лахути* сделал следующее примечание к данному фрагменту оригинала: Здесь весьма тонкий момент, вызывающий много споров среди переводчиков «Трактата». Мы перевели это так, как обычно переводится немецкая идиома “*der Fall ist*” и английская ”*is the case*”, которым перевели его Рамсей и Огден в 1922 г. — "имеет место". Но дело в том, что немецкое *Fall* само по себе может означать "случай, происшествие, падение"; английское *case* (происходящее от латинского casus – случай, казус) тоже означает «случай», поэтому некоторые переводчики переводят это предложение как «Мир есть все, что случается (или: происходит)», хотя мы (и

помните, что Юлий Цезарь сказал *Alea jacta est*[1]? Я знаю! Он сказал это по-гречески, а Светоний (*Suetonius*) перевёл это на латынь, но смысл остается.

- *Мир есть совокупность фактов, а не вещей.*
  The world is the totality of facts, not of things.
  = *Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge.* Вы заметили, как Витгенштейн любил факты? Позднее вы увидите, что он думал, что факты должны иметь состояния.

- *Мир распадается на факты.*
  The world divides into facts.
  С этого момента я избавлю вас от немецкого оригинала.

- *Мир определён фактами и тем, что это **все** факты.*
  The world is determined by the facts, and by these begin all the facts.

- *Потому что совокупность всех фактов определяет как всё то, что имеет место, так и всё то, что не имеет места.*
  For the totality of facts determines both what is the case, and also all that is not the case.
  Обратите внимание, что Витгенштейн рассматривает как факты, которые случаются (то, что имеет место), так и факты, которые являются лишь возможностями (то, что не имеет место).

2. *То, что имеет место, что является фактом, — это существование атомарных фактов.*
   What is the case, the fact, is the existence of atomic facts.

   - *Атомарный факт есть соединение объектов.*
     An atomic fact is a combination of objects.

   - *Объект прост.*
     The object is simple.

   - *Объекты содержат возможность всех положений вещей.*
     Objects contain the possibility of all states of affairs.
     В Visual Prolog объектные факты и переменные используются для представления состояния объекта.

   - *Возможность вхождения объекта в атомарные факты есть его форма.*
     The possibility of its occurrence in atomic facts is the form of the object.

Как можно видеть, идея классов и объектов — это старая идея в западной философии. То же верно и для состояний. В нашем примере состояниями объектов являются индивидуальные счета. Состояние счёта (нашего объекта) определяется фондом, идентификатором владельца счёта, его именем, номером социального

другие) с этим не согласны. А английское *fall* не означает «случай», но означает «падение, падать». Автор играет разными смыслами немецкого и английского *Fall/fall*: «что падает в мире? Падают кости Фортуны» и т.д..

[1] Жребий брошен (*лат.*).

страхования, паролем и т.д. Это состояние представляется с помощью факта и переменной:

```
facts - customer
    funds : real := 3.0.
    personalData : (string Name,string SocialSecurity) single.
```

Предикаты интерфейса используются для того, чтобы получить доступ к фактам, которые выдают состояние объекта.

```
% Файл account.i
interface account
    open core
predicates
    ssN : (string SocialSecurity) procedure (o).
    setSocialSecurity : (string SSN) procedure(i).
    deposit : (real).
end interface account
```

Примерно восемь человек написали мне о том, что названием книги Витгенштейна должно быть “*Tratactus Logicus-Philosophicus*”. Доктор Мари Роуз Шапиро (*Mary Rose Shapiro*) вступила в долгую дискуссию о различиях между классической и философской латынью. Она заявляет, что название в таком виде, в каком я его написал, является ошибкой, совершённой Огденом (*Ogden*), когда тот переводил книгу на английский язык. Мои знания в этой теме не простираются настолько глубоко, но я решил сохранить название книги таким, под которым она есть на каждой книжной полке.

Глава 13: Джузеппе Пеано

Джузеппе Пеано (*Giuseppe Peano*) был одним из величайших математиков всех времён. Он написал небольшую книгу под названием

Arithmetices Principia, Nova Methodo Exposita

Из этого вы можете понять, насколько он был велик. Не многие люди пишут на латыни в наше время и находят читателей. Гаусс (*Gauss*) был одним из немногих математиков, которые были в состоянии писать на латыни. Пеано был ещё одним. Конечно, Гаусс преимущественно писал на латыни. Пеано писал, в основном, на французском, языке науки начала прошлого века, и на итальянском, языке его матери. Единственной книгой, написанной им на латинском языке, была «Начала Арифметики» (*Arithmetices Principia*), очень короткая книга на 29 страниц. В ней он предложил миру современные обозначения формальной логики — области математики, которая дала начало Прологу. Он говорил и о других вещах, таких как теория чисел и рекурсия. Я настоятельно советую вам выучить латынь и прочитать книгу Пеано.

13.1. Черепашья графика

Перед тем, как погрузиться во вклад Пеано в науку (а он велик), давайте реализуем в Прологе *черепашью графику* (*turtle graphics*). Создайте объектно-ориентированный проект GUI со следующими настройками:

```
General
    Project name: peano
    UI Strategy: Object-oriented GUI (pfc/gui)
    Target type: Exe
```

- ***Создайте новую форму в новом пакете***: `canvas` (раздел 2.1) Название формы — `canvas`, она должна находиться в корне дерева проекта `peano`.
- ***Создайте класс под названием*** `curve`: откройте окно проекта, выделите папку *canvas*, выберите пункт меню *File/New in Existing Package* и создайте класс. Уберите галочку *Create Objects*. Постройте приложение.
- ***Создайте пакет под названием*** `turtleGraphics` в каталоге `C:\vispro` и вставьте в него класс под названием `turtle`. Для того чтобы сделать это, выберите пункт *File/New in New Package*. Затем в диалоговом окне *Create Project Item* нажмите на кнопку *Browse* для указания *Parent Directory* и выберите каталог `C:\vispro\` в окне *Set New Directory*. Не забудьте отключить *Create Objects*. Проверьте, что вы положили пакет `turtleGraphics` вне проекта `peano`. Так будет легче многократно его использовать. Постройте приложение.
- Включите пункт меню *File/New* в *TaskMenu.mnu*.
- Добавьте следующий код

```
clauses
    onFileNew(S, _MenuTag) :-
```

```
        X= canvas::new(S),
        X:show().
```

для *TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new*.

- Отредактируйте файлы `curve.cl` и `curve.pro` так, как показано на рисунке 13.1. Отредактируйте также файлы `turtle.cl` и `turtle.pro` так, как показано на рисунках 13.2 и 13.3. Постройте программу.
- Добавьте к *PaintResponder* формы `canvas.frm` следующий код:

```
clauses
    onPaint(S, _Rectangle, _GDIObject) :-
        W= S:getVPIWindow(), curve::drawCurve(W).
```

Скомпилируйте и запустите проект. Когда вы выберете команду *File/New* меню приложения, на экране появится окно с нарисованной на нём звездой.

```
% Файл curve.cl
class curve
    open core, vpiDomains
predicates
    classInfo : core::classInfo.
    drawCurve:(windowHandle).
end class curve

% Файл curve.pro
implement curve
    open core, vpi, vpiDomains, math
class predicates
    star:(windowHandle, integer, real, integer)
        procedure (i, i, i, i).
clauses
    classInfo("plotter/curve", "1.0").
    star(_Win, 0, _A, _L) :- !.
    star(Win, N, A, L) :- turtle::right(A),
        turtle::forward(Win, L),
        star(Win, N-1, A, L).
    drawCurve(Win) :- star(Win, 10, 3*pi/5, 40).
end implement curve
```

Рисунок 13.1 Файлы curve.cl и curve.pro

```
% Файл turtle.cl
class turtle
    open core, vpiDomains

predicates
    classInfo : core::classInfo.
    forward:(windowHandle, integer) procedure.
```

```
    move:(integer) procedure.
    right:(real) procedure.
    left:(real) procedure.
end class turtle
```

Рисунок 13.2 Файл turtle.cl

```
% Файл turtle.pro
implement turtle
    open core, vpiDomains, vpi, math
class facts
    turtle:(pnt, real) single.

clauses
    classInfo("turtleGraphics/turtle", "1.0").

    turtle(pnt(80, 80), -pi/2).

    forward(Win, L) :- turtle(P1, Facing), P1= pnt(X1, Y1),
        X2= math::round( X1+ L*cos(Facing)),
        Y2= math::round(Y1+L*sin(Facing)),
        P2= pnt(X2, Y2), drawline(Win, P1, P2),
        assert(turtle(P2, Facing)).

    move(L) :- turtle(P1, Facing), P1= pnt(X1, Y1),
        X2= round( X1+ L*cos(Facing)),
        Y2= round(Y1+L*sin(Facing)),
        P2= pnt(X2, Y2), assert(turtle(P2, Facing)).

    right(A) :- turtle(P1, Facing), NewAngle= Facing+A,
        assert(turtle(P1, NewAngle)).

    left(A) :- turtle(P1, Facing), NewAngle= Facing-A,
        assert(turtle(P1, NewAngle)).
end implement turtle
```

Рисунок 13.3 Файл turtle.pro

13.2. Состояния черепахи

Класс `turtle` реализует рисующую черепаху аналогично тому, как это делается в Лого (Logo) — языке, созданном Папертом (*Papert*) в развитие идей Жана Пиаже (*Jean Piaget*). Черепаха — это объект, который движется по окну, оставляя за собой след. Состояние черепахи описывается с помощью двух переменных — её положения и направления её движения. Эти переменные задаются фактом:

```
class facts
    turtle:(pnt, real) single.
clauses
    turtle(pnt(200, 200), -pi/2).
```

Рассмотрим следующий предикат:

```
forward(Win, L) :-
    turtle(P1, Facing), P1= pnt(X1, Y1),
    X2= math::round(X1 + L*cos(Facing)),
    Y2= math::round(Y1 + L*sin(Facing)),
    P2= pnt(X2, Y2), drawline(Win, P1, P2),
    assert(turtle(P2, Facing)).
```

Он реализует движение черепахи вперёд. Если черепаха находится в точке `P1=pnt(X1, Y1)` и движется на расстояние `L` в направлении `Facing`, то можно найти её новое положение, вычислив проекции `L` на оси X и Y:

$$X_2 = X_1 + L \times \cos(F) \qquad (13.1)$$
$$Y_2 = Y_1 + L \times \sin(F) \qquad (13.2)$$

После того, как новое положение найдено, используется вызов

```
drawline(Win, P1, P2)
```

для соединения точки `P1` с точкой `P2`. Наконец, информация о новом положении черепахи добавляется в базу данных:

```
assert(turtle(P2, Facing))
```

Предикат `move/1` аналогичен предикату `forward/2`, но он не соединяет новое положение черепахи со старым. Предикаты `right/1` и `left/1` поворачивают черепаху направо или налево, соответственно. Их реализации просты.

13.3. Рекурсия

Постулат рекурсии Пеано может быть сформулирован следующим образом. *Для того чтобы доказать, что предикат истинен для любого натурального числа*:

- Докажите, что он истинен для 0.
- Докажите, что он истинен для *N*, если он истинен для *N* – 1.

Конечно, данная версия постулата пробегает до 0, и написана она на английском. Оригинал пробегает до ∝, и он был на латыни. Неплохо, не правда ли? В любом случае, давайте рассмотрим предикат `star/4` (см. рис. 13.1) в свете постулата Пеано. Первое предложение

```
star(_Win, 0, _A, _L) :- !.
```

говорит: вы добавляете ноль вершин к звезде, если ничего не делаете. Второе предложение

```
star(Win, N, A, L) :- turtle::right(A),
    turtle::forward(Win, L),
    star(Win, N-1, A, L).
```

говорит: вы дорисовываете N сторон к звезде, если вы рисуете одну вершину (поворачиваетесь на A радиан направо и идёте L пикселей прямо) и переходите к рисованию остальных N – 1 вершин.

13.4. Кривая Пеано

Пеано придумал рекурсивную (непрерывную — *ред*. *пер*.) кривую, которая заполняет плоскость, и математики рассматривают это как очень интересное свойство. Когда я создавал проект `peano`, в действительности я хотел нарисовать именно кривую Пеано, но тогда я решил начать с более лёгкого примера. В любом случае, для того чтобы получить кривую Пеано, всё, что вам нужно сделать, — это заменить файл `curve.pro`, приведенный на рисунке 13.1 файлом `curve.pro`, показанным на рисунке 13.4.

```
implement curve
    open core, vpi, vpiDomains, math

class predicates
    peano:(windowHandle, integer, real, integer)
        procedure (i, i, i, i).
clauses
    classInfo("plotter/curve", "1.0").

    peano(_Win, N, _A, _H) :- N<1, !.
    peano(Win, N, A, H) :- turtle::right(A),
        peano(Win, N-1, -A, H),
        turtle::forward(Win, H),
        peano(Win, N-1, A, H),
        turtle::forward(Win, H),
        peano(Win, N-1, -A, H),
        turtle::left(A).

    drawCurve(Win) :-
        turtle::move(-60), peano(Win, 6, pi/2, 5).
end implement curve
```

Рисунок 13.4 Кривая Пеано

13.5. Latino Sine Flexione[1]

Как вы видели, Пеано создал современные обозначения для логики. Он также выдвинул идею рекурсии. Рекурсия является основной схемой программирования на Прологе, поэтому можно сказать, что Пеано сделал два важных вклада в логическое программирование. Однако он на этом не остановился. Пролог был изобретён Колмероэ для обработки текстов на естественном языке.

> *Язык программирования Пролог родился из проекта, целью которого было не создание языка программирования, а обработка текстов на естественном языков; в данном случае французского.*
>
> ***Колмероэ***

Относительно естественных языков Пеано сделал интересное предложение. Большинство европейских языков заимствовали огромное число латинских слов, так почему бы не использовать латынь для научного общения? Всё, что вам нужно — это выбрать латинские слова, общие для основных европейских языков, и вы получите словарь достаточно большой, для того чтобы выразить почти любую идею. Проблема состоит в том, что латынь — это трудный язык, с его склонениями и формами глаголов. Пеано позаботился и об этом: в своем варианте латыни он исключил все флексии. Ознакомимся с несколькими цитатами из «Ключа к интерлингве» (*Key to Interlingua*). Слово «интерлингва[2]» является официальным названием латыни без флексий.

13.6. Цитаты из «Ключа к интерлингве»

1. *Интерлингва заимствует каждое слово, общее для английского, немецкого, французского, испанского, итальянского, португальского, русского языков и каждое англо-латинское слово*.
2. *Каждое слово имеет форму латинской основы или корня*.

Словарь интерлингвы составлен из великого множества латинских слов, которые вошли в английский язык и, следовательно, они легко осмысливаются англоговорящим человеком. Он включает в себя все термины, используемые в научной номенклатуре ботаники, химии, медицины, зоологии и других наук. Многие из этих терминов являются либо греческими, либо греко-латинскими. Несколько необходимых классических латинских слов, не имеющих международных эквивалентов, являются частью словаря, а именно:

Latino	Английский	Русский	Пример
que	that	что, который	Interlingua adopta vocabulo *que* existe in Anglo, Germano, Franco et Russo.
de	of	из	Latino habe praecisione *de* expressione.
et	and	и	et cetera, et al.
sed	But	но	Vocabulario de Latino non es formato ad arbitrio

[1] Латынь без флексий (без словоизменений).

[2] Интерлингва Пеано была создана в 1903 г. и доработана в 1908 г. В 1950 г. в Нью-Йорке была создана интерлингва IALA (международной ассоциации вспомогательного языка).

			sed consiste de vocabulos in usu in Anglo, Russo et Germano.
isto	this, that	это, что	pro isto («потому что»)
illo	he, it	он, оно	Illo es Americano.
illa	she	она	Illa es Americana
me	I, me	я	Me programma in Prolog.
te	you	ты	Te programma in Prolog.
nos	we	мы	Nos programma in Prolog.
vos	you guys	вы	Vos programma in Prolog.
id	it	это	i.e., id est
qui?	who?	кто?	Qui programma in Prolog?
que?	what, which	что, который	Te programma in que?
omne	all	всё	Illo es omnepotente= Он всемогущ.

Язык *Latino* не должен был иметь никаких флексий. Множественное число образуется с помощью слова *plure*, как показано ниже в примере:

> *Plure vocabulo in Latino es in usu in Anglo*
> *Латинские слова используются в английском языке*.

Однако Пеано был демократом, а большинство людей хотело образовывать множественное число с помощью *s*, поэтому он одобрил частичное *s*-множественное.

Прилагательные создаются из существительных с помощью предлога *de*:

- de fratre = *fraternal* (братский)
- de amico = *friendly, amicable* (дружеский)

Конечно, имеется много слов, которые являются прилагательными по природе: internacionale, breve (*short* — короткий), magno (*large, big* — большой), commune (*common* – общий) и т.д. Вы можете использовать эти прилагательные как наречия, без изменений. Вы можете также образовывать наречия с помощью слова modo: in modo de fratre (*fraternally* — по-братски), in mode de patre (*paternally* — по-отечески), in modo rapido (*rapidly* — быстро). Я полагаю, что этого достаточно. Вот пример текста на *Latino sine flexione*:

> *Interlingua es lingua universale que persona in modo facile scribe et intellige sine usu de speciale studio. Interlingua adopta vocabulo que existe in Anglo, Germano, Franco et Russo. Usu de Interlingua es indicato pro scientifco communicatione et pro commerciale correspondentia. Usu de Interlingua in internationale congressu facilita intelligentia et economiza tempore. In additione Interlingua es de valore educationale nam (because) illo es de logico et naturale formatione.*

Зачем нам нужен международный язык, такой как интерлингва Пеано, если у нас уже есть английский? Я думаю, что он нам не нужен. Однако люди, поддерживающие интерлингву, приводят несколько интересных аргументов в защиту своего проекта. Например, человечество не может позволить международному языку изменяться всякий раз, когда экономическое превосходство и культурный престиж перейдет от одной нации к другой. В XIX веке французский был языком дипломатии. Потом американская экономика превзошла французскую, и английский язык стал международным языком. В этом столетии Китай будет доминирующей суперсилой и будет навязывать письменный китайский другим нациям. На эту тему существует интересная история. Люди всё ещё

используют латынь для описания растений. Так что, если вы откроете новое растение, вы должны описать его на латинском. Это было верно также для зоологов и анатомов, но не так давно они перешли на английский язык. Поэтому некоторые биологи предложили последовать их примеру и перейти на современный язык и в ботанике. Проблема состоит в том, что язык, который они хотят использовать — китайский, а не английский!

Существует и другой аргумент в пользу *Latino*. Такие языки, как французский, английский и китайский сложны для изучения даже для людей, которые говорят на них, как на языках своих матерей. Не многие французы могут писать на приемлемом французском. Очень немногие американцы могут приблизиться к общению на письменном английском. Мой сын учится в восьмом классе средней школы в Маунт Логане (Mount Logan), и я часто хожу туда проверить его успехи. Большинство его одноклассников не могут читать, не говоря уже о том, чтобы писать по-английски. Если это происходит с американцами, что я могу сказать про японцев, которые пытаются выучить английский? Итак: десяти лет недостаточно для того, чтобы научить американца читать и писать прозу по-английски, но через десять дней мы сможем писать на более, чем приемлемом *Latino*. Но давайте оставим эту дискуссию мудрым и построим проект, который берёт текст на `LsF` (*Latino sine Flexione*) и объясняет трудные слова и выражения.

- ***Создайте новый GUI проект***: `LsF`.
- ***Добавьте новый пакет к дереву проекта***: `LsF/editor`. Добавьте пакет

`editorControl`

к корню дерева проекта. Для этого выберите команду меню *File/Add*, найдите каталог Visual Prolog и выберите файл

`\pfc\gui\controls\editorControl\editorcontrol.pack`

Постройте приложение.

- ***Создайте новую форму***: `editor/txtWin`. Измените размер формы, потому что `editorControl` должен быть довольно большим. Добавьте пользовательский элемент управления на форму. На пиктограмме такого элемента изображен ключ. Система предложит вам список названий элементов управления, выберите в нем `editorControl`. Измените размеры панели редактора.

- ***Постройте*** (*Build/Build*) приложение.
- ***Создайте класс*** `editor/grammar`, отключите *Creates Objects*. Если вы не помните, как это делать, посмотрите в раздел 4.4. Постройте приложение ещё раз.

Измените файлы `grammar.cl` и `grammar.pro` так, как показано ниже.

```
% Файл grammar.cl
class grammar
    open core
predicates
    classInfo : core::classInfo.
    dic:(string).
end class grammar

% Файл grammar.pro
implement grammar
    open core
clauses
    classInfo( "editor/grammar", "1.0").
    dic(S) :- stdio::write(S), stdio::nl.
end implement grammar
```

- ***Включите пункт*** *File/New* (см. раздел 2.2) меню приложения. Добавьте фрагмент

```
clauses
    onFileNew(S, _MenuTag) :- X= txtWin::new(S), X:show().
```

для *ProjWin/TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new*.

- Добавьте фрагмент

```
onHelpClick(_Source) = button::defaultAction :-
    editor_ctl:getSelection(Start, End),
    Txt=editor_ctl:getText(Start, End),
    grammar::dic(Txt).
```

для обработчика *ClickResponder* кнопки `help_ctl` формы `txtWin.frm`. В том же файле, в который вы добавляли последний код для *onHelpGrammar* (`txtWin.pro`), измените определение предиката `new` так, как показано на рисунке 13.5. Наконец, создайте файл `LsF.txt` в каталоге *\exe* и поместите в него следующий текст.

```
Interlingua es lingua universale que persona in modo facile
scribe et intellige sine usu de speciale studio. Interlingua es
indicato pro scientifico communicatione et pro commerciale
correspondentia. Usu de Interlingua in internationale congressu
economiza tempore.
```

Если вы откроете новое окно *txtWin* из приложения, в нем появится данный текст. Если вы выделите часть текста и выберете пункт *Help* меню приложения, то вы отобразите выделенный текст в окне сообщений (*message window*). Следующим шагом является добавление словарных сущностей в выбранный текст перед тем, как писать его в окне сообщений (см. рис. 13.6). Мы начнём с разбиения выделенного текста на список слов. На рисунке 13.6 это делает следующий предикат:

```
strTok:(string, string_list) procedure (i, o).
```

Обработка текстов на естественном языке — это наиболее сложная область *computer science*. Этот вопрос выходит за рамки данного документа. Если вы хотите постигнуть это искусство, вы можете начать с добавления сущностей в программу, приведенную на рисунке 13.6.

13.7. Примеры

Примеры этой главы находятся в папках *peano* и *LsF*.

```
class predicates
    tryGetFileContent:(
        string FileName,
        string InputStr,
        string UpdatedFileName) procedure (i, o, o).
clauses
    tryGetFileContent( "", "", "LsF.txt") :- !.
    tryGetFileContent(FileName, InputStr, FileName) :-
        trap(file::existFile(FileName), _TraceId, fail), !,
        InputStr= file::readString(FileName, _).
    tryGetFileContent(FileName, "", FileName).

    new(Parent):- formWindow::new(Parent),
        generatedInitialize(),
        PossibleName= "LsF.txt",
        tryGetFileContent(PossibleName, InputStr, _FileName), !,
        XX=string::concat("Here: ", InputStr),
        editorControl_ctl:pasteStr(XX).
```

Рисунок 13.5 Вставка файла в редактор

```
% Файл grammar.pro
implement grammar
    open core, string
class predicates
    strTok:(string, string_list) procedure (i, o).
    addExplanation:(string, string) procedure (i, o).
class facts
    entry:(string, string).
clauses
    classInfo( "editor/grammar", "1.0").

    entry("que", "that").
    entry("sine", "without").
    entry("es", "am, are, is, was, were, to be").
    entry("de", "of").
    entry("et", "and").

    addExplanation(T, R) :- entry(T, Meaning), !,
        R= format("%s= %s\n", T, Meaning).
        addExplanation(_T, "").
```

```
    strTok(S, [T1|L]) :- frontToken(S, T, Rest), !,
        addExplanation(T, T1), strTok(Rest, L).
    strTok(_S, []).

    dic(S) :- strTok(S, SList),
        Resp= concatList(SList),
        stdio::write(Resp), stdio::nl.
end implement grammar
```

Рисунок 13.6: Словарь «трудных» слов LsF

Глава 14: *L*-системы

Аристид Линденмайер (*Aristid Lindenmayer*) — это датский биолог, придумавший умный способ описания форм растений. Как вы знаете, биологи тратят большую долю своего времени на описание форм растений. Эти описания обычно делаются на упрощённой версии латыни, которую изобрел Карл Линней (*Carl Linaeus*).

Метод Линденмайера описания форм растений основывается на рисующей черепахе, которую вы изучили в разделе 13.1. В интернете вы найдёте огромное количество материала по этой теме. Поэтому я ограничу себя примером, который вы можете использовать как стартовую точку для ваших собственных проектов.

- ***Создайте GUI проект***: `Lsys`.
- ***Добавьте пакет*** в корень дерева проекта: `aristid`.
- ***Добавьте форму*** в пакет `aristid`: `canvas`.
- ***Постройте приложение.***
- ***Добавьте класс*** к `aristid`: `draw`. Снимите галочку *Create Objects*. Постройте проект, чтобы вставить класс в дерево проекта.
- ***Включите пункт*** *File/New* панели задач. Добавьте код:

```
clauses
    onFileNew(S, _MenuTag) :-
        X= canvas::new(S), X:show().
```

для *TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.*

14.1. Класс draw

Теперь отредактируйте класс `draw` так, как показано ниже. Постройте приложение.

```
% Файл draw.cl
class draw
    open core, vpiDomains
predicates
    classInfo : core::classInfo.
    tree:(windowHandle).
end class draw

% Файл draw.pro
implement draw
    open core, vpiDomains, vpi, math
clauses
    classInfo("plotter/draw", "1.0").
domains
    command = t(commandList); f(integer); r(integer); m.
    commandList= command*.
```

```
class facts
    pos:(real Delta, real Turn) single.
    grammar:(commandList) single.

class predicates
    move:(windowHandle, pnt, real, commandList)
        procedure (i, i, i, i).
    derive:(commandList, commandList)
        procedure (i, o).
    mv:(windowHandle, pnt, real, command, pnt, real)
        procedure (i, i, i, i, o, o).
    iter:(integer, commandList, commandList) procedure (i, i, o).

clauses
    pos(4.0, 0.1745329).
        grammar([f(1)]).

    iter(0, S, S) :- !.
    iter(I, InTree, OutTree) :-
        derive(InTree, S),
        iter(I-1, S, OutTree).

    derive([], []) :- !.
    derive([f(0)|Rest], [f(0),f(0)|S]) :- !,
        derive(Rest, S).
    derive([f(_)|Rest],
        [f(0),  t([r(1),f(1)]), f(0),
            t([r(-1),f(1)]), r(1), f(1)|S]) :- !,
        derive(Rest, S).
    derive([t(Branches)|Rest], [t(B)|S]) :- !,
        derive(Branches, B),
        derive(Rest, S).
    derive([X|Rest], [X|S]) :-
        derive(Rest, S).

    mv(Win, P1, Facing, f(_), P2, Facing) :- !,
        pos(Displacement, _Turn),
        P1= pnt(X1, Y1),
        X2= round(X1+ Displacement*cos(Facing)),
        Y2= round(Y1+Displacement*sin(Facing)),
        P2= pnt(X2, Y2),
        drawline(Win, P1, P2).
    mv(_Win, P1, Facing, m, P2, Facing) :- !,
        pos(Displacement, _Turn),
        P1= pnt(X1, Y1),
        X2= round(X1+ Displacement*cos(Facing)),
        Y2= round(Y1+Displacement*sin(Facing)),
        P2= pnt(X2, Y2).
    mv(_Win, P1, Facing, r(Direction), P1, F) :- !,
        pos(_Displacement, Turn),
        F= Facing+ Direction*Turn.
    mv(Win, P1, Facing, t(B), P1, Facing) :-
```

```
            move(Win, P1, Facing, B).

        move(_Win, _P1, _Facing, []) :- !.
        move(Win, P1, Facing, [X|Rest]) :-
            mv(Win, P1, Facing, X, P, F),
            move(Win, P, F, Rest).

        tree(Win) :-
            grammar(Commands),
            iter(5, Commands, C),
            Facing= -pi/2,
            Point= pnt(100, 250),
            move(Win, Point, Facing, C).
    end implement draw
```

Наконец, добавьте следующий фрагмент кода

```
    clauses
        onPaint(S, _Rectangle, _GDIObject) :-
            draw::tree(S:getVPIWindow()).
```

для обработчика события *PaintResponder*. Постройте и запустите приложение.

Откройте форму с помощью пункта *File/New* меню приложения. Она покажет дерево, изображенное на рисунке.

14.2. Примеры

Пример этой главы находится в папке *Lsys*.

Глава 15: Игры

В этой главе вы будете изучать, как реализовывать игры в Visual Prolog. Что и говорить, хорошая игра — это как хорошая поэма. Недостаточно выучить русский язык и правила стихосложения, для того чтобы писать как Пушкин. Кроме знания языка, на котором он собирается писать, поэту необходимо воображение, творческие способности и т.д. То же самое относится к хорошему разработчику игр. Вам понадобится масса креативности, для того чтобы сделать что-то похожее на «Тетрис» (*Tetris*) или «Космических пришельцев» (*Space Invaders*). Я использовал эти примеры для того, чтобы показать, что хорошие игры не зависят от изысканной графики или звуковых эффектов. Важными особенностями являются напряженное внимание, длительное напряжение и ощущение награды за хорошо выполненную работу.

Однажды молодой человек спросил Лопе де Вега (*Lope de Vega*), как писать стихи. Ответ был такой: *Это просто — заглавная буква в начале каждой строфы и рифма в конце*. «Что я должен вставить в середину?» — спросил юноша. *В середину, hay que poner talento*[1], сказал Лопе де Вега.

В этом уроке вам придется иметь дело с некоторой техникой, такой как заглавные буквы и рифма. Для того чтобы создать хорошую игру, вам потребуется добавить к рецепту талант.

Игра, которую вы реализуете, проста. Имеется червь, быстро ползущий по экрану. Игрок должен изменять его направление, для того чтобы предохранить червя от ударов о препятствия и стены. Вам необходимо реализовать следующие элементы:

1. Окно, по которому червь будет ползать. Окно должно иметь четыре кнопки для управления движением червя.
2. Класс состояний, который описывает направление движения и положение червя. Он содержит предикат `mov/0` для изменения положения червя и предикат `turn/2`, который используется для выполнения поворотов.
3. Класс, который рисует изображение положения червя. Он содержит предикат `snapshot(Win)`. Обратите внимание на следующую деталь: предикат `snapshot/1` не может нарисовать червя прямо в окне. Это бы заставило мерцать изображение. Он должен построить изображение и только затем нарисовать его на экране.
4. Таймер, который вызывает `mov/0` и `snapshot/1` через фиксированные интервалы, для того чтобы создать иллюзию движения.

Мы знаем, что делать — давайте сделаем это. До этого момента я избегал создания объектов. Действительно, каждый раз, когда вы создавали класс, вам рекомендовалось снимать галочку *Creates Objects*. Однако если мы жестко будем следовать этой установке, наша игра будет иметь очень неприятное свойство. Если вы проиграете, выйдете из текущей игры и попробуете начать новую, то обнаружите, что новая игра не находится в исходном состоянии. Вместо этого она будет в последнем состоянии, в котором вы ее оставили. Проблема заключается в том, что переменные, которые используются для описания состояния, имеют память, которая сохраняется от одной формы к другой. Объекты были изобретены для предотвращения такого неприятного поведения. Поэтому

[1] нужно поместить талант (*исп.*).

давайте используем прекрасную возможность изучить, как создавать объекты и работать с их состояниями.

- ***Создайте проект***: `game`.

```
Project Name: game
UI Strategy: Object-Oriented GUI (pfc/vpi)
TargetType: Exe
Base Directory: C\vip70
Sub-Directory: game
```

- ***Создайте пакет***: `playground`.

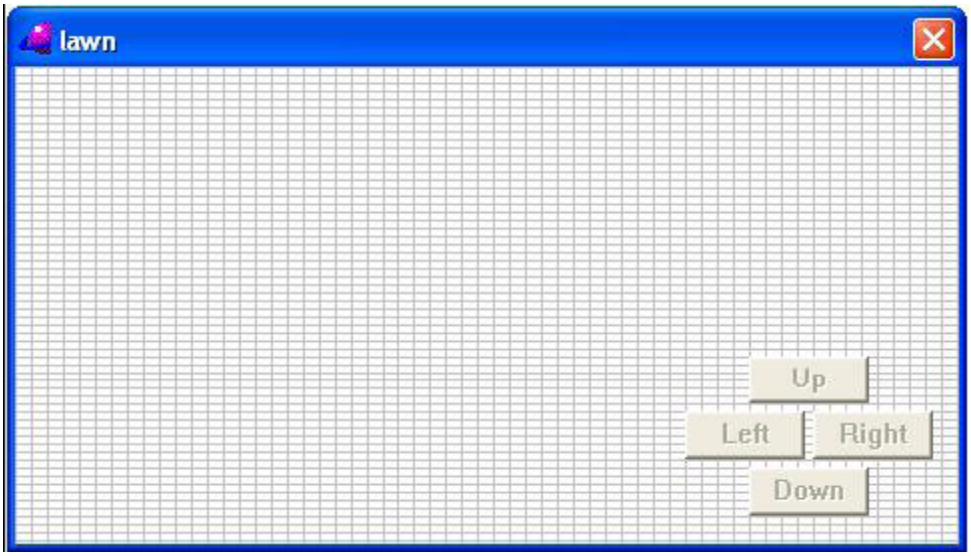


Рисунок 15.1 Форма — лужайка

- ***Создайте форму***: `lawn`. Добавьте четыре кнопки к форме `lawn` — обращайтесь к рисунку 15.1. Хорошей идеей является изменить названия кнопок в диалоговом окне *Properties* с `pushButton_ctl` на `up_ctl`, `down_ctl`, `left_ctl` и `right_ctl`. Постройте приложение, для того чтобы включить в него эту форму. Затем включите *File/New* и добавьте код

```
clauses
    onFileNew(S, _MenuTag) :- X= lawn::new(S), X:show().
```

    для *ProjWin/Code Expert/Menu/TaskMenu/id_file/id_file_new*.
- ***Создайте класс*** `objstate` внутри `playground`. Не отключайте *Create Objects* для класса `objstate`. Поместите в него код, приведенный на рисунке 15.3. **NB**: Класс `objstate` должен иметь интерфейс в файле `objstate.i`. Постройте приложение.
- ***Создайте класс*** `draw` внутри `playground`. Отключите *Create Objects*. Вставьте в него код, приведенный на рисунке 15.2. Постройте приложение.
- ***Создайте класс*** `click` для хранения часов. Снимите *Creates Objects*.

```
% Файл click.cl
class click
    open core, vpiDomains
```

```
predicates
    bip:(window).
    kill:().
end class click

% Файл click.pro
implement click
    open core
class facts
    tm:vpiDomains::timerId := null.
clauses
    bip(W) :- tm := W:timerSet(500).
    kill() :- vpi::timerKill(tm).
end implement click
```

Постройте приложение для включения класса `click` в проект.

- Добавьте код

```
clauses
    onDestroy(_Source) :- click::kill().
```

для обработчика *DestroyListener* с помощью окна *Properties* формы `lawn.frm`.

- Добавьте следующий код

```
class facts
    stt:objstate := objstate::new().
clauses
    onShow(Parent, _CreationData) :-
        stt := objstate::new(),
        stt:init(), click::bip(Parent).
```

для обработчика *ShowListener* диалогового окна *Properties* формы `lawn.frm`.

- Добавьте код для обработчика *ClickResponder*, соответствующего каждой из кнопок:

```
clauses
    onDownClick(_S) = button::defaultAction() :-
        stt:turn(0, 10).

clauses
    onLeftClick(_S) = button::defaultAction() :-
        stt:turn(-10, 0).

clauses
    onRightClick(_S) = button::defaultAction() :-
        stt:turn(10, 0).

clauses
    onUpClick(_S) = button::defaultAction() :-
        stt:turn(0, -10).
```

- Добавьте код

```
clauses
    onTimer(_Source, _TimerID) :- stt:mov(),
        R= rct(10, 10, 210, 210),
        invalidate(R).
```

для обработчика *TimeListener* из диалогового окна *Properties* формы `lawn.frm`.

- Добавьте код

```
clauses
    onPaint(_Source, _Rectangle, GDIObject) :-
        draw::snapshot(GDIObject, stt).
```

для обработчика *PaintResponder* из окна *Properties* формы `lawn.frm`.

Постройте и запустите программу. При выборе пункта меню *File/New* появится окно с червем. Вы можете управлять червем с помощью четырех кнопок.

Улучшение игры остаётся за вами. Вы можете добавить препятствия, например стены. Вы также можете положить маленькие яблоки для питания червя. Наконец, вы можете написать 3D-игру. В дествительности, 3D-игра очень проста. Сегменты имеют три координаты, и, прежде чем рисовать, вы должны найти их проекции на плоскость. Если вы поищете информацию в интернете, то, разумеется, найдёте формулы для вычисления перспективной проекции точки. Алекс Соарес (*Alex Soares*) написал оригинальную игру, на которой я основывал это руководство. Его игра является 3D-игрой, написанной на Visual Prolog 5.2 с использованием DirectX — библиотеки, содержащей массу ресурсов для 3D-игр.

Продолжим наше изучение предикатов рисования. Если вы будете рисовать фигуры на каждый тик часов, то создадите анимацию. Однако если вы выполняете рисование прямо на видимом окне, то анимация будет мигать, что не очень хорошо. Решением этой проблемы является рисование на скрытом окне и передача его в видимое окно только после того, как оно будет готово к отображению. Предикат

```
W= pictOpen(Rectangle)
```

открывает скрытое окно, для того чтобы вы рисовали, не беспокоясь о мерцании анимации.

Предикат

```
Pict= pictClose(W)
```

создаёт изображение из содержимого скрытого окна, в то же время он закрывает окно. Наконец, предикат

```
pictDraw(Win, Pict, pnt(10, 10), rop_SrcCopy)
```

передаёт изображение в видимое окно. Вот пример кода, содержащего эти три предиката:

```
snapshot(Win, S) :-
    S:mov(), !,
    Rectangle= rct(0, 0, 200, 200),
    W= pictOpen(Rectangle),
    draw(W, S),
    Pict= pictClose(W),
    Win:pictDraw(Pict, pnt(10, 10), rop_SrcCopy).
```

```
% Файл draw.cl
class draw
    open core, vpiDomains
predicates
    classInfo : core::classInfo.
    snapshot:(windowGDI Win, objstate).
end class draw

% Файл draw.pro
implement draw
    open core, vpiDomains, vpi
class predicates
    draw:(windowHandle, objstate) procedure.
clauses
    classInfo("playground/draw", "1.0").
    draw(W, S) :- S:segm(Rectangle),
        vpi::drawEllipse(W, Rectangle), fail.
    draw(_W, _S).

    snapshot(Win, S) :- S:mov(), !,
        Rectangle= rct(0, 0, 200, 200),
        W= pictOpen(Rectangle),
        draw(W, S),
        Pict= pictClose(W),
        Win:pictDraw(Pict, pnt(10, 10), rop_SrcCopy).
end implement draw
```

Рисунок 15.2 draw.cl и draw.pro

Аргумент `rop_SrcCopy` определяет режим растровых операций. Он говорит, что система должна скопировать рисунок из исходного окна в назначенное. Существуют и другие режимы передачи изображений из одного окна в другое:

- `rop_SrcAnd` выполняет операцию логического ***И*** для бит рисунка и фона. Вы можете использовать ее для создания спрайтов — независимых графических объектов, свободно перемещающихся по экрану.
- `rop_PatInvert` инвертирует цвета.
- `rop_SrcInvert` инвертирует цвета источника.

15.1. Факты объектов

Обратите особое внимание на одну специфическую деталь: когда мы имеем дело с классами, которые не создают объектов, состояние объекта устанавливается фактами класса (*class facts*). Они объявляются так:

```
class facts
    city:(string Name, pnt Position).
    conn:(pnt, pnt).

clauses
    city("Salt Lake", pnt(30, 40)).
    city("Logan", pnt(100, 120)).
    city("Provo", pnt(100, 160)).

    conn(pnt(30, 40) , pnt(100, 120)).
    conn(pnt(100, 120), pnt(100, 160)).
```

Факт класса является общим для всех объектов класса. Это означает, что если объектный предикат (предикат, объявленный в интерфейсе) изменяет факт, то он изменяется для всех объектов класса. Поэтому, если бы вы сохраняли состояние червя в факте класса, игра имела бы следующее неприятное свойство.

Неприятная особенность игры. Когда игрок потеряет червя, выйдет из текущей игры и начнёт играть в следующую, он обнаружит, что новый червь не находится в начальном положении. Фактически он будет в той позиции, в которой был потерян старый.

Проблема состоит в том, что факты класса, используемые для описания состояния червя, имеют память класса (*class memory*), которая сохраняется от одной формы к другой, от одного объекта червя к другому. Решение просто: объявляйте факты без добавления ключевого слова *class* к заголовку объявления. Тогда у каждого объекта будет собственный набор фактов. Объектные факты объявляются следующим образом:

```
facts
    w:(integer, pnt) nondeterm. % worm
    d:(integer, integer) single. % direction
clauses
    init() :- assert(w(1, pnt(110, 100))),
        assert(w(2, pnt(120, 100))), assert(w(3, pnt(130, 100))),
        assert(w(4, pnt(140, 100))), assert(w(5, pnt(140, 100))).
    d(10, 0).
```

Данная схема сохранения состояния объекта реализована в программе, приведенной на рисунке 15.3.

```
% Файл objstate.i
interface objstate
    open core, vpiDomains
```

```
predicates
    init:().
    turn:(integer, integer).
    mov:().
    segm:(rct) nondeterm (o).
end interface objstate

% Файл objstate.pro
implement objstate
    open core, vpiDomains
facts
    w:(integer, pnt) nondeterm. % worm
    d:(integer, integer) single. % direction
clauses
    init() :- assert(w(1, pnt(110, 100))),
        assert(w(2, pnt(120, 100))),
        assert(w(3, pnt(130, 100))),
        assert(w(4, pnt(140, 100))),
        assert(w(5, pnt(140, 100))).
    d(10, 0).

    mov() :- retract(w(1, P)), P= pnt(X1, Y1),
        d(DX, DY), P1= pnt(X1+DX, Y1+DY), assert(w(1, P1)),
        retract(w(2, P2)), assert(w(2, P)),
        retract(w(3, P3)), assert(w(3, P2)),
        retract(w(4, P4)), assert(w(4, P3)),
        retract(w(5, _)), assert(w(5, P4)), !.
    mov().

    segm(rct(X, Y, X+10, Y+10)) :- w(_, pnt(X, Y)).

    turn(DX, DY) :-
        assert(d(DX, DY)).
end implement objstate
```

Рисунок 15.3 Класс objstate

Глава 16: Анимация

Обратитесь к главе 8, если вы не помните, как использовать событие *painting*. Вам понадобятся маски для создания рисунков с прозрачным фоном. Эта тема также была рассмотрена в главе 8. Наконец, вспомните про использование *GDIObject* из раздела 3.2, в котором вы также найдёте материал об обновлении прямоугольника для рисования.

- Создайте проект

```
Project Name: rattlesnake
UI Strategy: Object-oriented GUI (pfc/gui)
```

В этом проекте вы приведете в движение знаменитую гремучую змею, которая присутствует на флаге *Don't thread on me*[1] Джона Пола Джонса[2] (*John Paul Jones*).

- Создайте новый пакет под названием `snake`.
- Создайте новую форму `canvas`. Вложите её в пакет `snake`.
- Создайте класс `click` для управления событиями времени. Отключите *Creates Objects*. Используйте код, приведенный на рисунке 16.1, для определения и реализации `click`.
- Постройте приложение. Включите *File/New*. Затем добавьте код

```
onFileNew(S, _MenuTag) :-
    F= canvas::new(S), F:show(), click::bip(F).
```

для *TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new*.

16.1. Класс dopaint

Класс `dopaint` будет управлять рисованием. Он работает аналогично методу `do-paint` в Java. Создайте класс, назовите его `dopaint` и отключите *Creates Objects*. Декларация класса имеет вид:

```
% Файл dopaint.cl
class dopaint
    open core
predicates
    classInfo : core::classInfo.
    draw:(windowGDI).
    invalidrectangle:(vpiDomains::rct) procedure (o).
end class dopaint
```

[1] «Не наступите на меня!»
[2] Флаг американских колонистов 1776 г.

На рисунке 16.2 приведена имплементация класса `dopaint.pro`. Постройте приложение.

```
% Файл click.cl
class click
    open core
predicates
    bip:(window).
    kill:(window).
end class click

% Файл click.pro
implement click
    open core
class facts
    tm:vpiDomains::timerId := null.
clauses
    bip(W) :- tm := W:timerSet(1000).
    kill(W) :- W:timerKill(tm).
end implement click
```

Рисунок 16.1 Класс click для работы с таймером

16.2. Управление таймером

При закрывании окна `canvas` вы должны отключить таймер. Для того чтобы добиться этой цели, добавьте следующий код

```
onDestroy(W) :- click::kill(W).
```

к обработчику события *DestroyListener* с помощью диалогового окна *Properties* формы `canvas.frm`. Затем добавьте код

```
onTimer(_Source, _TimerID) :-
    dopaint::invalidRectangle(R),
    invalidate(R).
```

к обработчику *TimeListener*, также с помощью окна *Properties* формы `canvas.frm`. Наконец, добавьте обработчик события

```
onPaint(_Source, _Rectangle, GDIObject) :-
    dopaint::draw(GDIObject).
```

к *PaintResponder*, с помощью диалогового окна *Properties* формы `canvas.frm`.

```
implement dopaint
    open core, vpiDomains

constants
    className = "snake/snakestate".
    classVersion = "".

class facts
    yesno:integer := 0.
class predicates
    flipflop:(picture Picture, picture Mask) determ (o, o).
clauses
    classInfo(className, classVersion).

    flipflop(Pict, Mask) :- yesno= 0,
        yesno := 1,
        Pict= vpi::pictLoad("figs\\n0.bmp"),
        Mask= vpi::pictLoad("figs\\n0Mask.bmp"), !.
    flipflop(Pict, Mask) :- yesno= 1,
        yesno := 0,
        Pict= vpi::pictLoad("figs\\n1.bmp"),
        Mask= vpi::pictLoad("figs\\n1Mask.bmp").

    draw(W) :-
        P= vpi::pictLoad("figs\\frogs.bmp"),
        W:pictDraw(P, pnt(10, 10), rop_SrcCopy),
        flipflop(Snake, Mask), !,
        W:pictDraw(Mask, pnt(40, 50), rop_SrcAnd),
        W:pictDraw(Snake, pnt(40, 50), rop_SrcInvert).
    draw(_).
    invalidRectangle(rct(40, 50, 100, 100)).
end implement dopaint
```

Рисунок 16.2 Имплементация класса dopaint

16.3. Как работает программа

Первое, что делает предикат `draw(W)` — это рисует несколько лягушек в качестве фона. Лягушки загружаются с помощью следующего предиката:

```
P= vpi::pictLoad("figs\\frogs.bmp"),
```

Разумеется, в папке *figs* у вас должен лежать файл `frogs.bmp`.

Затем предикат `draw(W)` получает изображение и маску змеи.

Предикат `flipflop(Snake, Mask)` предназначен для чередования двух изображений змеи, чтобы создать

иллюзию движения. Для того чтобы изобразить змею поверх фонового рисунка, вы должны использовать маски. Если вы не помните, как это делается, перечитайте главу 8.

Глава 17: Текстовый редактор

В этой главе вы научитесь использовать класс `editorControl` для создания текстового редактора.

- Создайте новый проект

```
Project Name: editor
UI Strategy: Object-oriented GUI(pfc/gui)
```

- Добавьте пакет `editorControl`, который находится в каталоге инсталляции Visual Prolog. Для этого выберите пункт меню *File/Add* и найдите пакет в

```
Visual Prolog 7.x\pfc\gui\controls\editorControl.
```

- Постройте приложение. Затем создайте новую форму `edform.frm` в корне дерева проекта так, как показано на рисунке 17.3. Для того чтобы поместить элемент управления *editorControl* в форму, вставьте на прототип формы элемент управления *custom control* — это кнопка с ключом Йале[1] (*Yale*) (см. рис. 17.1). Затем выберите *editorControl* в диалоговом окне, показанном на рисунке 17.2. Постройте приложение. Измените имя идентификатора

Рисунок 17.1 Диалоговое окно Choose Custom Control

Рисунок 17.1 Кнопка добавления пользовательского элемента управления

элемента управления на `editorControl_ctl` так, как показано на рисунке 17.4.

- Включите команду меню *File/New* и добавьте следующий код:

[1] Йале изобрёл этот вид ключа, очень популярный сегодня. Легенда гласит, что вскоре после его изобретения Гудини (*Houdini*) смог пробиться через новое устройство – *прим. авт.*

```
onFileNew(S, _MenuTag) :- F= edform::new(S), F:show().
```

17.1. Сохранение и загрузка файлов

С помощью дерева проекта откройте форму `editor.frm`, если она еще не открыта. Используйте диалоговое окно *Properties*, для того чтобы вставить код, приведенный на рисунке 17.5 для обработчика события *ClickResponder* нажатия на кнопку `button::save_ctl`. Затем добавьте код, приведенный на рисунке 17.6, для обработчика события *ClickResponder* нажатия на кнопку `button::load_ctl`. Практически это все, что вам нужно сделать, для того чтобы создать функционирующий текстовый редактор.

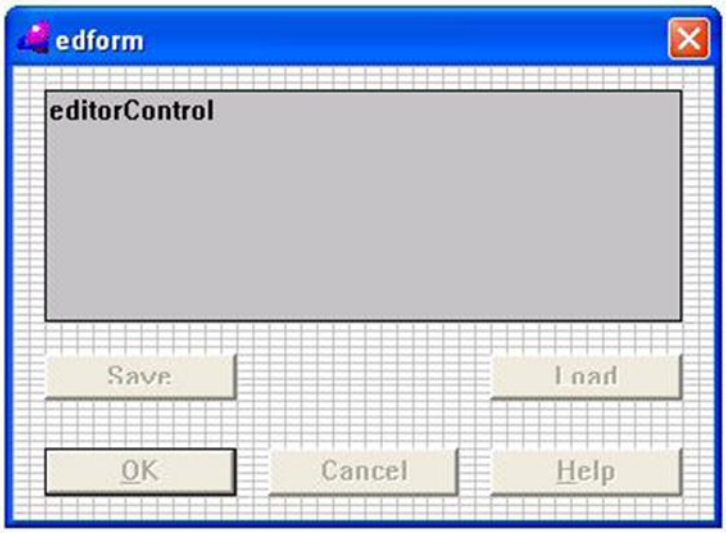


Рисунок 17.3 Форма с Edit Control

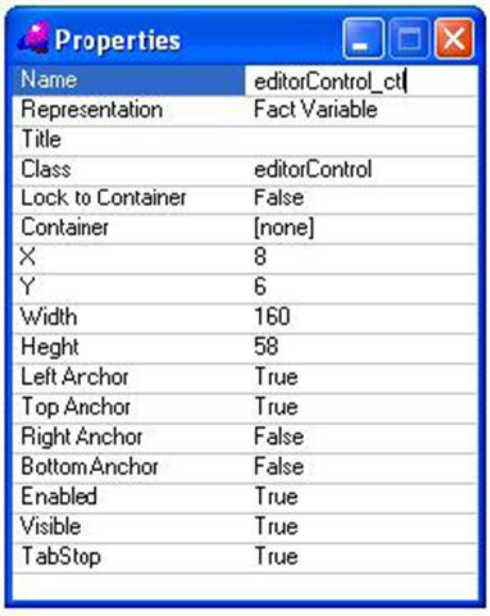


Рисунок 17.2 Свойства элемента управления

```
predicates
    onSaveClick : button::clickResponder.
clauses
    onSaveClick(_Source) = button::defaultAction() :-
        Txt= editorControl_ctl:getEntireText(),
        FName= vpiCommonDialogs::getFileName("*.*",
                            ["Text", "*.txt"],
                            "Save", [], ".", _X), !,
        file::writeString(FName, Txt).
onSaveClick(_Source) = button::defaultAction().
```

Рисунок 17.5 Код для предиката onSaveClick

```
predicates
    onLoadClick : button::clickResponder.
clauses
    onLoadClick(_Source) = button::defaultAction() :-
        FName= vpiCommonDialogs::getFileName("*.*",
                        ["Text", "*.txt"],
                        "Load", [], ".", _X),
        file::existFile(FName),
        !,
        Str= file::readString(FName, _IsUnicodeFile),
        editorControl_ctl:pasteStr(1, Str).
    onLoadClick(_Source) = button::defaultAction().
```

Рисунок 17.6 Код для предиката onLoadClick

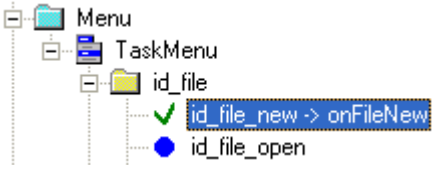
Глава 18: Печать

В предыдущей главе вы узнали, как построить текстовый редактор. В данной главе вы получите первоначальное представление о печати.

- Создайте проект

```
Project Name: testPrt
    UI Strategy: Object-oriented GUI (pfc/GUI)
    Target Type: Exe
    Base Directory: C:\vip\codeForTyros
```

- Скомпилируйте приложение, для того чтобы внести прототипы *Task Window* в дерево проекта.
- Дважды щелкните по ветви *TaskMenu.mnu* дерева проекта и включите пункт *&File/&New/tF7* из прототипа меню.
- Щелкните правой кнопкой мыши по элементу *TaskWindow.win* дерева проекта и выберите пункт *Code Expert* из контекстного меню, как показано на рисунке 2.6. Появится окно *Dialog and Window Expert*. Откройте папки *Menu*, *TaskMenu* и *id_file*, как показано ниже.
-



Выберите элемент *id_file_new* и нажмите кнопку *Add*. Затем дважды щелкните по созданной ветви *id_file_new→on_file_new*. Наконец, замените прототип `onFile-New(_Source, _MenuTag)` приведенным ниже фрагментом кода.

```
clauses
    onFileNew(_Source, _MenuTag) :-
        PW=vpi::printStartJob("Recoreco"),
        _HRES = vpi::winGetAttrVal(PW, attr_printer_hres),
        VRES = vpi::winGetAttrVal(PW, attr_printer_vres),
        V_SCR_RES=vpi::winGetAttrVal(PW, attr_screen_vres),
        FNT=vpi::fontCreate(ff_Fixed,[], VRES*40 div V_SCR_RES),
        vpi::winSetFont(PW, FNT),
        vpi::printStartPage(PW),
        vpi::drawText(PW, 100, 200, "Before the sunset!"),
        vpi::printEndPage(PW),
        vpi::printEndJob(PW).
```

Красота метода печати Visual Prolog состоит в том, что вы работаете с принтером так, как если бы он был обычным графическим окном. Действительно, первое, что вы должны сделать, это открыть рабочее окно принтера:

```
PW=vpi::printStartJob("Recoreco")
```

Раз вы имеете окно, то можете использовать его для получения информации о разрешении принтера.

```
HRES= vpi::winGetAttrVal(PW, attr_printer_hres),
VRES= vpi::winGetAttrVal(PW, attr_printer_vres),
V_SCR_RES=vpi::winGetAttrVal(PW, attr_screen_vres),
```

Методом проб и ошибок, а также используя информацию о разрешении принтера, вы можете определить шрифт, который хорошо выглядит при печати:

```
FNT=vpi::fontCreate(ff_Fixed,[], VRES*40 div V_SCR_RES),
vpi::winSetFont(PW,FNT),
```

Наконец, вы можете использовать любой предикат рисования, для того чтобы создать хорошо напечатанную страницу, например:

```
vpi::drawText(PW, 100, 200, "Before the sunset!"),
```

Глава 19: Вкладки и не только

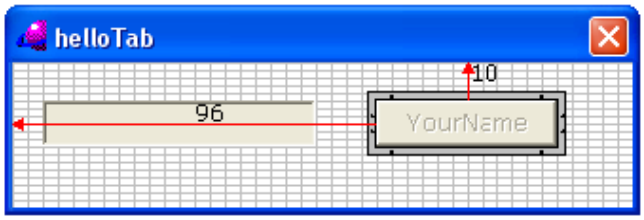
- Создайте новый проект:

```
Name: tab_example
UI Strategy: Object GUI
```

- *New in New Package*. Создайте пакет под названием `forms`
- Из каталога инсталляции с помощью *File/Add* добавьте пакет:

```
pfc\gui\controls\tabControl\tabControl.pack
```

- *New in Existing Package* (в пакете *forms*). Создайте окно *Control* под названием `hello_tab`. Для этого выберите элемент *Control* на левой панели (окна *Create Project Item*. Разместите на нем поле редактирования и кнопку. — *ред*. *пер*.). Переименуйте кнопку: `yourname_ctl`.



- Постройте приложение, для того чтобы можно было вставить код для событий. Добавьте следующий фрагмент кода для обработчика события *ClickResponder* кнопки `yourname_ctl`:

```
clauses
    onYournameClick(_Source) = button::defaultAction :-
        Name= edit_ctl:getText(),
        Ans= string::concat("Hello, ", Name, "!\n"),
        stdio::write(Ans).
```

19.1. Знаменитые программы

Существуют две программы, которые можно назвать печально известными — “*Hello, World!*” и рекурсивный поиск чисел Фибоначчи. В этой главе вы узнаете, как поместить обе эти программы на вкладки. Вы уже создали форму для приложения `hello`. Теперь мы создадим форму для последовательности Фибоначчи.

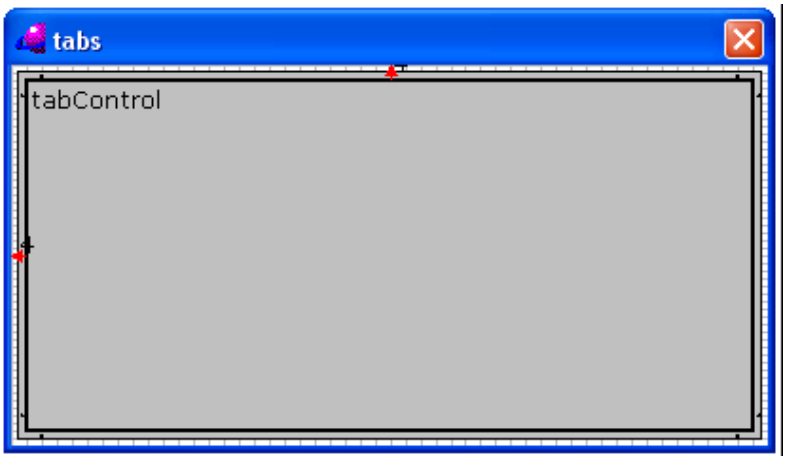
- *New in Existing Package* (в пакете *forms*). Создайте *Control* под названием `fib-Tab`. Переименуйте кнопку в `fib_ctl`.

- Постройте приложение, для того чтобы вставить код для событий. Добавьте следующий фрагмент кода для обработчика *ClickResponder* кнопки `fib_ctl`:

```
class predicates
    fibo:(integer, integer) procedure (i, o).
clauses
    fibo(N, F) :-
        if N<2 then F=1
        else
            fibo(N-1, F1), fibo(N-2, F2),
            F= F1+F2
        end if.
predicates
    onFibClick : button::clickResponder.
clauses
    onFibClick(_Source) = button::defaultAction :-
        Num= edit_ctl:getText(),
        I= toTerm(Num), fibo(I, F),
        Ans= string::format("fibo(%d)= %d", I, F),
        edit_ctl:setText(Ans).
```

Снова постройте приложение, чтобы убедиться, что все работает правильно.

- Создайте новую форму в существующем пакете с помощью команды *New in Existing Package* (существующий пакет называется *forms*) под названием `forms/tabs`. Используйте ключ Йале, для того чтобы вставить `tabControl` на форму. Постройте приложение.



- Перейдите в файл `tabs.pro`, до которого можно добраться с помощью дерева проекта, и замените предложение

```
clauses
    new(Parent):-
        formWindow::new(Parent),
        generatedInitialize().
```

следующим фрагментом кода:

```
clauses
    new(Parent):-
        formWindow::new(Parent),
        generatedInitialize(),
        %
        Page1 = tabPage::new(),
        Tab1 = helloTab::new(Page1:getContainerControl()),
        Page1:setText(Tab1:getText()),
        tabControl_ctl:addPage(Page1),
        Page2 = tabPage::new(),
        Tab2 = fibTab::new(Page2:getContainerControl()),
        Page2:setText(Tab2:getText()),
        tabControl_ctl:addPage(Page2),
        succeed.
```

- Включите пункт *File/New* меню приложения.
- Добавьте фрагмент кода

```
clauses
    onFileNew(S, _MenuTag) :-
        W= tabs::new(S),
        W:show().
```

для *TaskWindow.win/ CodeExpert/Menu/TaskMenu/ id_file/id_file_new*.

- Постройте приложение в последний раз. Неплохо, не правда ли?

19.2. Ботаника

Вы можете не иметь никакого интереса к ботанике. И все-таки существует много причин для того, чтобы отдать должное этой важной области науки. Давайте их перечислим.

Сохранение мира. Глобальное потепление является самым серьезным вызовом, стоящим перед нами сегодня. Для того чтобы защитить планету для будущих поколений, мы должны уже сегодня сократить концентрацию газов, удерживающих тепло, с помощью современных технологий и практических

решений. Оказывается, растения являются, образно говоря, единственным живительным путем выхода из этой западни, в которую мы попали сами. Поэтому мы нуждаемся в политических мерах по защите климата, чтобы сократить вред от вырубки тропического леса; понять динамику взаимозависимости растений, чтобы контролировать уменьшение биологического разнообразия, и выделить быстрорастущие культуры, чтобы возместить леса, уничтоженные в прошлом, вследствие нашей недальновидности. Поэтому если спасение мира — ваше призвание, вот хороший повод для старта. Я полагаю, что спасение мира также является и хорошим бизнесом, так как для исследователей и специалистов, занимающихся глобальным потеплением, будет выделяться все большее количество денег. Я горжусь тем, что моя семья очень активно участвует в защите природы и в биологических исследованиях. Я хочу отдать должное моей бабушке, индеанке (коренной американке, не инд**и**анке), которая вместе с моим отцом создала экологический заповедник, который мы поддерживаем до сих пор. Я помню, что когда я был ребенком, самой милой сердцу книгой в нашем доме была «Разум цветов» (*L'Intelligence des fleures*) Метерлинка (*Maeterlinck*). Мой отец заставлял меня повторять *plusieurs fois*[1] известное высказывание Метерлинка: «Когда будет спилено последнее дерево, последний человек умрет, цепляясь за его ствол». По правде говоря, я никогда не находил свидетельств того, что Метерлинк сказал такое, однако моя бабушка была в этом уверена. Я надеюсь, что бельгийский читатель сможет пролить свет на мои сомнения по этому поводу.

Компьютерная графика. В главе об Аристиде Линденмайере вы видели, что биологи разработали умные методы изображения растений с помощью компьютерной графики. Вы, вероятно, помните песню Бизе:

Votre toast, je peux vous le rendre, señors, señors, car avec
les soldats oui, les toreros peuvent s'entendre.

Я мог бы сказать, что ученый в области *computer science* и биолог могут понять друг друга, по крайней мере, в сфере компьютерной графики.

Орхидеи. Эти экзотические растения — одни из наиболее красивых созданий природы. Они являются объектами культа, который не имеет ничего общего с вином или французской кухней. Кстати говоря, серб Ниро Вульф[2] (*Nero Wolf*) любил вино, французскую кухню и орхидеи. Если вы почитаете книги про Вульфа, то заметите, что интрига для читателя создается не тем, как Вульф раскрывает тайны, а тем, как он добывает деньги для своих дорогостоящих увлечений. Он нанимает Фрица Брэнера, исключительно одаренного швейцарского повара, который готовит и подает ему всю еду и который отвечает по-французски, когда к нему обращаются. Он также нанимает Теодора Хорстмана, специалиста по орхидеям, который заботится обо всех его растениях. Наконец, наиболее важным из его служащих является Арчи Гудвин, который работает в качестве детектива, чтобы добывать деньги на дорогостоящие привычки его влиятельного босса.

Одной из самых интересных особенностей ботаники, по сравнению с другими науками, является то, что существует специальный вид латыни, который используется биологами разных стран для описания и именования растений. Многие биологи не

[1] Наизусть (*фр*.).
[2] Ниро Вульф (или Неро Вольф) — сыщик, герой детективов Рекса Стаута (Rex Stout), США.

достаточно знают латынь, поэтому программа, помогающая им иметь дело с этим непростым языком, положила бы начало для реализации целей сохранения мира, разведения орхидей и создания красивой компьютерной графики. В данном разделе я покажу, как может быть написана подобная программа. Кроме того, вы узнаете, как работать с вкладками и раскрывающимися списками (*listEdit*).

Программа, обучающая биологов латыни, приводится в корневой директории папки, содержащей примеры, которые сопровождают этот документ. Программа содержит три вкладки.

- Вкладка *Произношение* (*Pronunciation tab*). Для того чтобы научить вас правильному латинскому произношению, я использовал отрывки из очень известной книги под названием *Philosophiae Naturalis Principia Mathematica*[1]. Автором этой книги является некий Исаак Ньютон. Я не уверен, слышали ли вы об этом ученом, но английские образованные люди считают его величайшим ученым из живших когда-либо. Они, разумеется, преувеличивают. Несмотря на это, он был достаточно хорошим ученым, чтобы писать на латыни и находить читателей. Как вы помните, когда ученый является достаточно талантливым, говорят, что он мог писать на латыни так, что нашел бы читателей. Ньютон, Пеано, Гаусс и другие буквально следовали этому утверждению и писали на латыни. На этой вкладке мой сын, который бегло говорит на латыни, английском, древнегреческом и китайском, будет читать Ньютона для вас.

- Вкладка *Склонение* (*Declensiontab*). Существуют языки, в которых существительные изменяются в соответствии с их функцией в предложении. Латынь, санскрит, немецкий, русский, польский являются такими языками. Поэтому если вы думаете, что ботаническая латынь недостаточно полезна, чтобы заслужить ваше внимание, вы можете использовать основную структуру программы для того, чтобы написать программу, обучающую русскому или немецкому.

- Вкладка *Глагол* (*Verbtab*). Склонение — это не единственное свойство, которое объединяет латынь с немецким, русским и древнегреческим. Латынь имеет также очень сложную систему спряжения глаголов, которая распространилась на так называемые романские языки (испанский, итальянский, португальский, румынский, французский, гальский и каталонский).

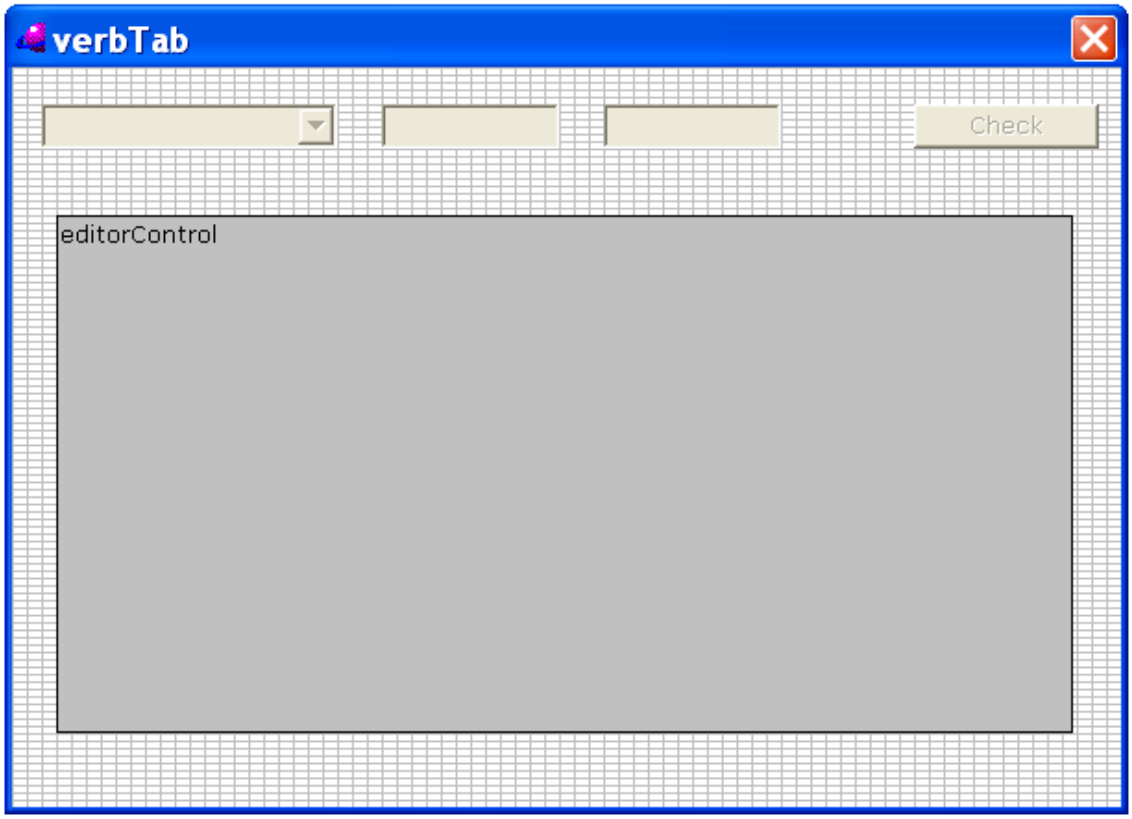
Если вы хотите выучить один из романских языков, вы можете адаптировать программу, рассчитанную на латынь, так, чтобы учить испанские или французские глаголы. Между прочим, испанский и португальский намного ближе к латыни, чем другие романские языки. Система спряжений глаголов была перенесена из латыни в испанский и португальский языки почти без изменений. Рассмотрим глагол *amare*[2]:

Латинский		**Испанский**	
Present	Imperfect	Present	Imperfect
amo	amabam	amo	amaba
amas	amabas	amas	amabas

[1] «Математические начала натуральной философии».
[2] Любить (*лат*.).

amat	amabat	ama	amaba
amamus	amabamus	amamos	amábamos

Теперь сосредоточим наши усилия на разработке программы. Я предполагаю, что вы поместите все вкладки в один и тот же пакет. В данном примере этот пакет называется *tabPackage*. Для того чтобы создать вкладку, выберите команду меню среды *New in New Package*. Затем выделите элемент *Control* на левой панели. Нажмите кнопку *Create* и сделайте следующий макет вкладки:



Вкладка *verbTab* содержит раскрывающийся список (*listEdit conrol*), заполненный глаголами, два поля редактирования (*edit conrol*), предназначенные только для чтения, чтобы показывать грамматическое время и наклонение, с которыми студент будет упражняться, и кнопку, которая будет сравнивать ответ студента с образцом. Основная причина, по которой мы предлагаем это приложение, состоит в том, чтобы показать, как работать с раскрывающимся списком. Поэтому давайте поближе посмотрим на методы, связанные с этим важным видом элементов управления. Перейдите на вкладку *Events* диалогового окна *Properties* и добавьте следующий код

```
predicates
    onShow : window::showListener.
clauses
    onShow(_Source, _Data) :-
        VerbList= listVerb_ctl : tryGetVpiWindow(),
        verbs::allVerbs(L),
        vpi::lboxAdd(VerbList, L),
        vpi::lboxSetSel(VerbList, 0, 1), !,
        verbs:: get_verb_state(Tense, Mood),
        mood_ctl:setText(Mood),
```

```
        tense_ctl:setText(Tense).
    onShow(_Source, _Data).
```

для обработчика событий *showListener*. Следующий фрагмент кода показывает, как поместить список глаголов в этот элемент управления, а также как выделить элемент списка:

```
    VerbList= listVerb_ctl : tryGetVpiWindow(), ...
    vpi::lboxAdd(VerbList, L),
    vpi::lboxSetSel(VerbList, 0, 1), ...
```

С помощью диалогового окна *Properties* вы должны добавить код, приведенный на рисунке 19.1, для события *onCheckClick*. При этом вы узнаете несколько других методов для этого элемента управления.

- Получение окна, ассоциированного с этим элементом управления:

```
    VerbList= listVerb_ctl : tryGetVpiWindow(),
```

- Получение индекса выделенного элемента:

```
    IV= vpi::lboxGetSelIndex(VerbList)
```

- Удаление элемента с заданным индексом:

```
    vpi::lboxDelete (VerbList, IV)
```

- Вычисление количества элементов списка:

```
    C= vpi::lboxCountAll(VerbList)
```

Вкладка *declensionTab* работает точно так же, как и вкладка *verbTab*. Вкладка *pronunciationTab* обладает чем-то новым. Она вызывает функцию *playSound*[1] операционной системы Windows, для того чтобы проиграть волновой файл. Раз мы имеем дело с внешней функций, то необходимо использовать вспомогательную операцию для связи Пролог-программы с Windows API. Вы должны начать с объявления *playSound* в качестве внешней процедуры. Затем вы должны ввести предикат Пролога для представления функции API. Ниже приведено объявление *playSound* как внешней процедуры.

```
resolve playSound externally
```

```
    onCheckClick(_Source) = button::defaultAction :-
        Txt= editorControl_ctl:getEntireText(),
        Verb= listVerb_ctl:getText(),
        VerbList= listVerb_ctl : tryGetVpiWindow(),
        IV= vpi::lboxGetSelIndex(VerbList), !,
        Mood= mood_ctl:getText(),
        Tense= tense_ctl:getText(),
        verbs::pres_root_conj(Verb, Tense, Mood, L),
        if template::check_it(Txt, L, Ans) then
```

[1] Функция воспроизведения звука волнового файла API Windows.

```
            editorControl_ctl:pasteStr(" "),
            vpi::lboxDelete (VerbList, IV) ,
            C= vpi::lboxCountAll(VerbList),
            if C= 0 then
                verbs::next_verb_state(),
                verbs::get_verb_state(NewTense, NewMood),
                mood_ctl:setText(NewMood),
                tense_ctl:setText(NewTense),
                verbs::allVerbs(NewList),
                vpi::lboxAdd(VerbList, NewList)
            else
            succeed()
            end if,
            vpi::lboxSetSel(VerbList, 0, 1)
        else
            template::check_present(Txt, L, Ans),
            editorControl_ctl:pasteStr(Ans)
        end if.
    onCheckClick(_Source) = button::defaultAction.
```

Рисунок 19.1 Методы списков listEdit

Теперь мы объявим предикат, который будет вызываться всякий раз, когда вы запустите функцию *playSound*, а также несколько необходимых констант.

```
class predicates
    playSound : (string Sound,
        pointer HModule,
        soundFlag SoundFlag)
        -> booleanInt Success
        language apicall.

constants
    snd_sync : soundFlag = 0x0000.
        /* play synchronously (default) */
    snd_async : soundFlag = 0x0001.
        /* play asynchronously */
    snd_filename : soundFlag = 0x00020000.
        /* name is file name */
    snd_purge : soundFlag = 0x0040.
        /* purge non-static events for task */
    snd_application : soundFlag = 0x0080.
        /* look for application specific association */
```

Наконец, добавьте приведенный ниже фрагмент кода:

```
clauses
    onPlayClick(_Source) = button::defaultAction :-
        File= listEdit_ctl:getText(),
        _ = playSound(File,
            null, /*snd_nodefault*/ snd_async+snd_filename).
```

для обработчика события *clickResponder* кнопки *Play*.

19.3. Обработка китайского языка

После вторжения в латынь посмотрим, как работать с китайским языком. Диалект китайского, известный на Западе как мандаринское наречие, является языком с большим количеством носителей. Кроме этого, система письма этого наречия, была заимствована другими китайскими наречиями, японским языком и, в переработанном виде, корейским. В результате по крайней мере треть населения Земли имеет представление о письменном китайском.

Мандаринское наречие имеет три системы письма: пиньин (*Pinyin*[1]), который использует латынь с греческим произношением, традиционные иероглифы и упрощенные иероглифы. Когда используют компьютер, чтобы писать на китайском, то обычно выражают свои мысли на *Pinyin*, а текстовый редактор переводит с *Pinyin* на язык упрощенных или традиционных иероглифов. Я, разумеется, не знаю китайский, но его знает мой сын. Поэтому по моей просьбе он использовал класс `multiMedia_native`, для того чтобы добавить звук к некоторым пунктам меню. Ниже приведен код, который он использовал. Просмотрите класс `multiMedia_native`, чтобы выявить дополнительные ресурсы.

```
predicates
    onFilePronunciationHowAreYou : window::menuItemListener.
clauses
    onFilePronunciationHowAreYou(_Source, _MenuTag) :-
        _ = multiMedia_native::sndPlaySound("howareu.wav",
                multiMedia_native::snd_ASync).
```

Поскольку для китайского языка мой сын использовал класс `multiMedia_native`, то здесь не было необходимости в том, чтобы использовать вспомогательные средства для вызова внешней процедуры, как он делал для латыни. Вы можете спросить, почему он не использовал одно и то же решение для латыни и китайского. Дело заключается в том, что он хотел показать, как вызывать внешние функции из Пролога, и, кроме этого, он хотел продемонстрировать, что мульдимедийный аппарат API легко применим для подобного рода операций.

- Создайте проект *Chinese*.
- Добавьте в его корень *editControl*.
- Создайте пакет *chineseforms*.
- Добавьте в него форму *chineseForm*. Эта форма должна иметь поле редактирования *editControl* и кнопку *Pinyin2Ideogram*.
- Создайте класс *ideograms*, уберите галочку в поле *Creates Objects*. Этот класс вы увидите ниже. Имплементация приведена на рисунке 19.2.

```
% File: ideograms.cl
class ideograms
    open core
predicates
    classInfo : core::classInfo.
```

[1] Система транслитерации китайских иероглифов буквами латинского алфавита.

```
    ideograms:(string, string) procedure (i, o).
end class ideograms

implement ideograms
    open core
clauses
    classInfo("chineseforms/ideograms", "1.0").
class facts - pideograms
    pinyin:(string, string).
class predicates
    pinyin2ideogram:(string, string) procedure (i, o).
    converList:(string*, string) procedure (i, o).
clauses
    pinyin("ni^", "\u4f60").
    pinyin("ha^o", "\u597d").
    pinyin("ma", "\u55ce").
    pinyin("wo^", "\u6211").
    pinyin("Ni^", "\u4f60").
    pinyin2ideogram(P, Idea) :- pinyin(P, Idea), !.
    pinyin2ideogram(P, P).

    ideograms(S, Idea) :- L= string::split(S, " \n"),
        converList(L, Idea).

    converList([], "") :- !.
    converList([P|Ps], Acc) :- pinyin2ideogram(P, Idea),
        converList(Ps, Rest),
        Acc= string::concat(Idea, Rest).
end implement ideograms
```

Рисунок 19.2 Файл ideograms.pro

С помощью диалогового окна *Properties* добавьте следующий фрагмент кода:

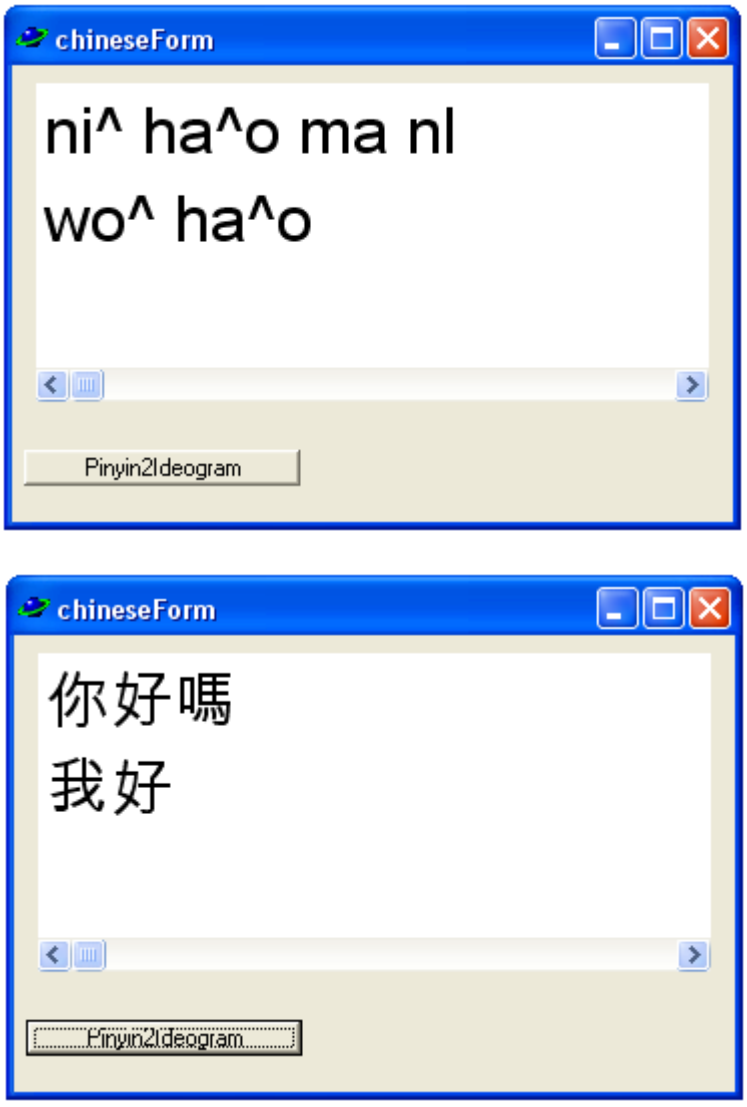
```
clauses
    onPinyinClick(_Source) = button::defaultAction :-
        Txt= editorControl_ctl:getEntireText(),
        ideograms::ideograms(Txt, Ans),
        editorControl_ctl:pasteStr(Ans).
```

для обработчика события *ClickResponder* кнопки *Pinyin2Ideogram*.
Последним шагом является добавление следующего кода:

```
predicates
    onShow : window::showListener.
clauses
    onShow(_Source, _Data) :-
        editorControl_ctl:setFontDialog(), !.
    onShow(_Source, _Data).
```

для обработчика события *ShowListener* кнопки *Pinyin2Ideogram*.

Конечно, данная программа будет работать только в том случае, если у вас установлена кодировка Unicode. Как бы там ни было, новая форма создана, и теперь вы должны выбрать кодировку Unicode. Если вы живете в Китае, Японии или Корее, это не такая уж большая проблема. Однако, если вы живете в США или в Европе, то кодировка Unicode может не иметь китайских символов. Операционная система Windows предполагает, что вы говорите на большинстве распространенных языков, за исключением китайского. Я уже говорил вам, что Парагвай — моя любимая страна. Поэтому я часто скачиваю парагвайские песни. В результате мой компьютер часто полагает, что я говорю на двух официальных языках этой страны. Так или иначе, я уверен в достаточной степени, что Arial Unicode MS обладает китайскими символами.



19.4. Регулярные выражения

Регулярные выражения являются полезным инструментом всякий раз, когда вы имеете дело с языками. В самом деле, если программист хочет найти строку, содержащую заданную подстроку, он может использовать традиционные методы поиска, такие как алгоритм Бойера-Мура (*Boyer-Moore*). Однако у него может возникнуть необходимость найти шаблон, который описывает целое число, например. В этом случае

регулярное выражение будет иметь вид `"[0-9]+"`, что означает: *последовательность одной или более цифр*. Операция «+» осуществляет положительное замыкание Клини и указывает на одно или более повторений. Существует также рефлексивное замыкание Клини, которое допускает пустую строку: `"[0-9]*"`. Это были два примера регулярных выражений, которые я использую. Для чего-либо более сложного я предпочитаю использовать всю мощь Пролога.

```
/*****************************************
    Project Name: regexpr.prj
    UI Strategy: console
*****************************************/

implement main
    open core
clauses
    classInfo("main", "regexpr").

class predicates
    test:(string) procedure (i).
clauses
    test(Str) :-
        regexp::search("-[0-9]+|[0-9]", Str, Pos, Len), !,
        stdio::write(Pos), stdio::nl,
        stdio::write(Len), stdio::nl.
    test(Str) :- stdio::write("No integer found in ", Str),
        stdio::nl.

clauses
    run():-
        console::init(),
        test( "There were 99 small Romans"),
        succeed(). % place your own code here
end implement main

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

Для того чтобы достичь более глубокого понимания регулярных выражений, было бы неплохо узнать, что представляет из себя иерархия Хомского-Шутценбергера. Ноам Хомский (*Noam Chomsky*, американский лингвист) и Марсель Поль Шутценбергер (*Marcel-Paul Schutzenberger*, французский врач) предложили, чтобы языки описывались с помощью грамматических правил, очень похожих на логические выражения, представленные в клаузальной форме. Если вы не помните, что это такое, вернитесь к соотвествующему разделу. В соответствии с формализмом Хомского-Шутценбергера простая группа существительного в английском языке описывается с помощью следующего набора правил:

```
article → [the]
noun → [iguana]
adjective → [pretty]
npr → article; adjective; noun
```

Каждый символ, который появляется с левой стороны стрелки, является нетерминальным символом. Таким образом, `article`, `npr`, `noun` и `adjective` — это нетерминальные символы, т.е. символы, которые определяет грамматика. Символы, которые возникают только в правой части правил, являются терминальными символами, например `[iguana]`. Правило для символа `npr` можно истолковать следующим образом:

```
npr →                   Группа существительного — это:
      article           артикль, за которым следует
      adjective         прилагательное, за которым следует
      noun              существительное
```

Язык является регулярным, если существует не более одного нетерминального символа с правой стороны каждого из правил, которое его описывает. Кроме этого, нетерминальный символ должен быть самым правым. Регулярные языки прекрасно подходят для описания таких токенов, как цифры, лексемы и т.д.

Регулярные выражения были придуманы Стивеном Клини (*Stephen Kleene*) в 1950 годах в качестве метода описания регулярных языков. В современных обозначениях элемент множества терминальных символах представляется с помощью квадратных скобок, внутри которых находится описание множества. Например, регулярное выражение `"[0-9]"` — это шаблон, который соответствует произвольной цифре. Программист может использовать знак + (положительное замыкание Клини), для того чтобы обозначить последовательность одного и более элементов множества. Например, выражение `"[0-9]+"` соответствует любому положительному целому числу в десятичном представлении. Последовательность ноля и более элементов множества указывается с помощью звездочки `*` (рефлексивное замыкание Клини). Выбор альтернативы обозначается с помошью вертикальной черты `|`. Например, выражение `"-[0-9]+|[0-9]+"` является шаблоном, который соответствует отрицательному или неотрицательному целому числу. Наконец, можно использовать знак ^, чтобы представить то, что не принадлежит множеству. Так, выражение `"[^a-z]"` обозначает множество всех символов, отличных от строчных букв латинского алфавита.

`"[0-9]"`	множество всех цифр
`"[0-9]+"`	последовательность одной и более цифр
`"[0-9]*"`	последовательность нуля и более цифр
`"-[0-9]+|[0-9]+"`	последовательность цифр после знака минуса или просто последовательность цифр
`"[^A-Z]"`	символ, не являющийся заглавной буквой латинского алфавита

В образец для замены можно добавить обратные ссылки и использовать `"\\1"`, для того чтобы вставить подобранную строку в этот образец. Ниже приведен пример.

```
implement main
    open core
class predicates
    test:(string) procedure (i).
clauses
    classInfo("main", "regexpr").

    test(Str) :-
        Ans= regexp::replace(Str, "<H1\\b[^>]*>(.*?)</H1>", "--\\1--"),
        stdio::write(Ans), !.
```

```
    run():- console::init(),
        test( "<H1> Viatores in Roma </H1> etc.").
end implement main
goal
    mainExe::run(main::run).
```

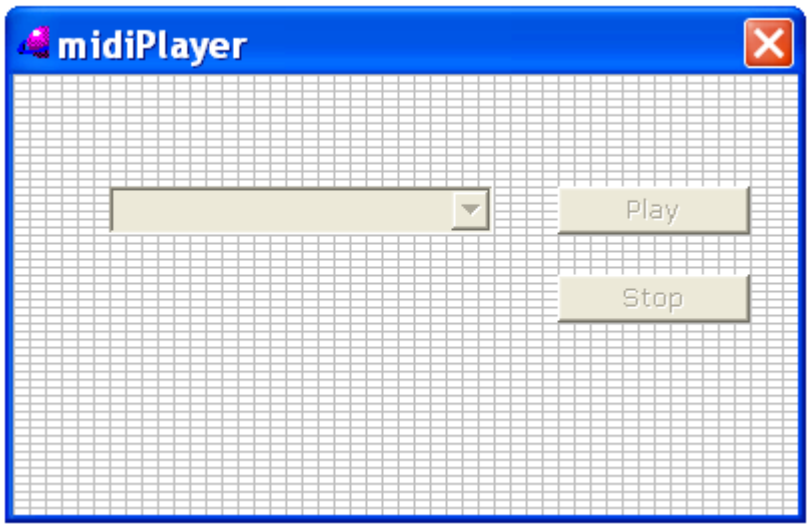
В данном примере теги HTML заменяются своим содержанием. То, что находится между тегами, охватывается первой обратной ссылкой. Знак вопроса в этом регулярном выражении дает гарантию, что звездочка остановится перед первым закрывающим тегом, а не перед последним.

19.5. MIDI-проигрыватель

Компьютеры дали музыкантам возможность единолично создавать интересные песни, исследовать новые музыкальные направления и редактировать свою работу. В данном разделе вы немного узнаете о MIDI — протоколе, который переносит музыку из компьютера в концертный зал. Мы начнем с написания программы, которая исполняет MIDI-файлы.

```
Project Name: midiplay
UI Strategy: Object-oriented GUI (pfc/gui)
```

- Создайте новую форму в новом пакете. Пусть форма называется *midiPlayer*. Добавьте в нее список *listEdit* и две кнопки: `play_ctl` и `stop_ctl`.



- В диалоговом окне *Properties* формы *midiPlayer* перейдите на вкладку *Events* и добавьте следующий фрагмент кода:

```
predicates
    onShow : window::showListener.
clauses
    onShow(_Source, _Data) :-
        SongList= listSongs_ctl : tryGetVpiWindow(),
        L= ["Wagner", "Beethoven"],
```

```
        vpi::lboxAdd(SongList, L),
        vpi::lboxSetSel(SongList, 0, 1), !.
    onShow(_Source, _Data).
```

для обработчика *ShowListener*.

- Включите пункт меню *File/New* главного окна приложения. Затем добавьте к нему следуюший фрагмент кода:

```
clauses
    onFileNew(S, _MenuTag) :-
        W= midiPlayer::new(S), W:show().
```

Постройте приложение.

- Следующим шагом является добавление приведенного ниже фрагмента кода к обработчику *ClickResponder* кнопки `play_ctl`.

```
clauses
    onPlayClick(_Source) = button::defaultAction :-
        Null = null, Len = 300,
        _= multiMedia_native::mciSendString(
            string::createCopy("Close mid"),
            string::create(300, " "), Len, Null),
        Buffer =string::create(300, " "),
        File= listSongs_ctl:getText(),
        C= "open %s.mid type sequencer alias mid",
        Command= string::format(C, File),
        _= multiMedia_native::mciSendString(
            Command, Buffer, Len, Null),
        _= multiMedia_native::mciSendString(
            "Play mid from 0",
            Buffer, Len, Null),
        _= vpi::processEvents().
```

- Последний шаг — это добавление кода к кнопке, которая будет прерывать музыку, если вы решите заняться чем-нибудь еще.

```
clauses
    onStopClick(_Source) = button::defaultAction :-
        Null= null,
        _Z= multiMedia_native::mciSendString(
                string::createCopy("Close mid"),
                string::create(300, " "), 300, Null).
```

19.6. Midi-формат

Мои отец и мать не очень любили музыку. Несмотря на это, мне повезло, и я знал много известных музыкантов. Например мой сын, который играет на фортепьяно и поёт, дружит с Хосе Берналем, а тот, в свою очередь, является сыном Сержио Берналя (*Sergio*

Bernal), дирижера Симфонического оркестра университета штата Юты. Более того, я рос по соседству с известным композитором, маэстро Жубером Карвальо (*Joubert de Carvalho*). Прекрасная соната, которую я предлагаю вам в данном проекте, была написана моим соседом. Вам будет небезынтересно узнать, что существует город, названный в честь этой западающей в память мелодии.

Я также знал маэстро Сукорского (*Sukorski*), который создал очень интересное произведение искусства. Он написал программу на Прологе (да, это был Пролог!), которая выполняет анализ настроения человека с помощью измерения сопротивления кожи, электромиографических сигналов, увеличения сердцебиения и т.д. В своей основе программа Сукорского аналогична детектору лжи. Используя эти данные, программа была в состоянии составить музыку, подходящую для конкретного случая.

Я изучал науки о космосе и классическую литературу в Корнеле. Мой кабинет до этого был занят Мугом (*Moog*). Этот Муг нацарапал свое имя на том самом столе, которым я пользовался, когда был в Корнеле. Казалось, что я унаследовал стол Муга. Я также хорошо знал маэстро Довиччи (*Dovicchi*), научным руководителем которого был ни кто иной, как известный венгерский композитор Георгий Лигетти (*Gyorgy Ligetti*). Маэстро Довиччи исследовал методы вопроизведения высококачественной музыки на компьютере в midi-формате. С помощью своих компьютеров он был способен исполнить в одиночку «Сельскую честь» (*Cavalleria Rusticana*) Масканьи (*Mascagni*). Мне случилось присутствовать при этом знаменательном событии.

Кроме маэстро Довиччи, еще одним большим экспертом в области компьютерной музыки является композитор Лучано Лима (*Luciano Lima*). Его программы анализируют произведения таких великих композиторов, как Чайковский или Бетховен, и пишут музыку в том же музыкальном стиле. Я включил одну из композиций Лимы в примеры. С помощью интернета я связался с маэстро Лимой, который является опытным Пролог-программистом, и попросил его показать вам, как создать простой MIDI-файл.

```
Project Name: midiplay
UI Strategy: Object-oriented GUI (pfc/gui)
```

- Постройте приложение, чтобы создать дерево проекта.
- С помощью дерева проекта откройте папку

```
$(Prodir)/pfc/gui/Controls/
```

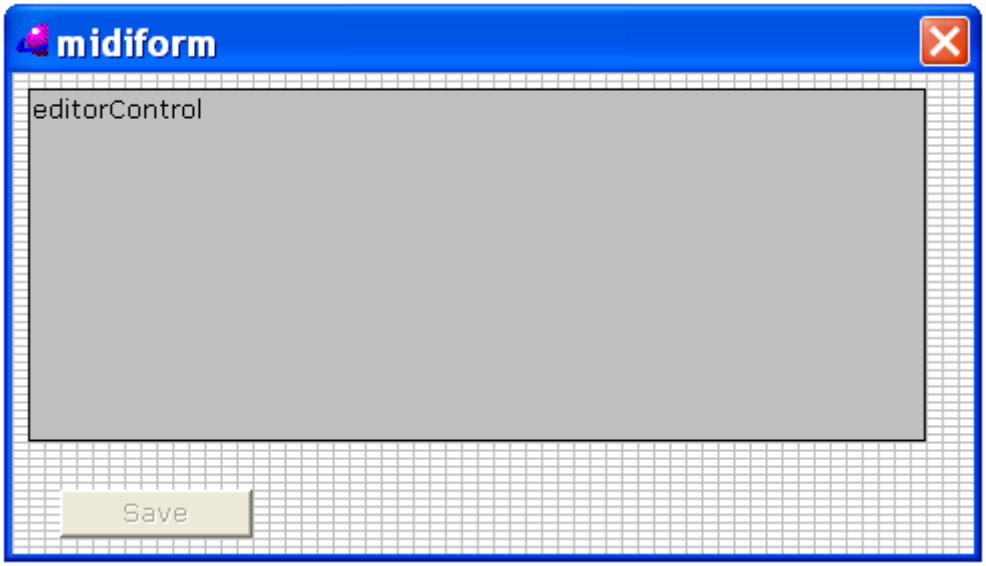
- С помощью правой кнопки мыши щелкните по папке *Controls* и добавьте пакет *editorControl* в дерево проекта. Снова постройте приложение.
- С помощью команды *New in New Package* создайте форму под названием *midiform*.
- Переименуйте идентификатор для кнопки *Save* как `save_ctl`. Постройте приложение. Включите пункт *File/New* главного меню приложения и добавьте к нему следующий фрагмент кода:

```
clauses
    onFileNew(S, _MenuTag) :-
        W= midiform::new(S), W:show().
```

- Создайте класс с именем *midi*. Ниже вы увидите декларацию этого класса. Его имплементация приведена на рисунке 19.3.

```
class midi
    open core
```

```
predicates
    classInfo : core::classInfo.
    generate:(string, string*) procedure (i, i).
end class midi
```



- Добавьте приведенный ниже фрагмент кода к обработчику *ClickResponder* кнопки `save_ctl`.

```
clauses
    onSaveClick(_Source) = button::defaultAction :-
        Txt= editorControl_ctl:getEntireText(),
        S= list::removeAll(string::split(Txt, "\n "), ""),
        FName= vpiCommonDialogs::getFileName("*.mid",
        ["Midi", "*.mid"],
        "Save in MIDI format", [], ".", _X), !,
        midi::generate(FName, S).
    onSaveClick(_Source) = button::defaultAction.
```

Вы найдете улучшенную и более завершенную версию программы, приведенной на рисунке 19.3, в примерах. Программа, показанная на рисунке 19.3, предоставляет хорошую возможность узнать, как записывать данные в файл. Первым шагом является создание файла для потока вывода:

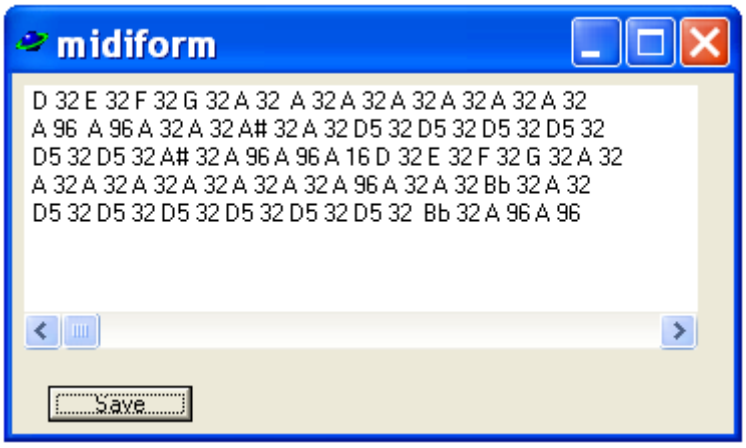
```
P= outputStream_file::create(File, stream::binary())
```

Затем вы записываете в файл список байтов, с помощью конструкции `foreach`, как показано ниже.

```
foreach X= list::getMember_nd (MidiCode) do
    P:write(X)
end foreach,
P:close().
```

С точки зрения музыканта программы, приведенные в примерах, являются чрезвычайно упрощенными. Однако вы с лёгкостью можете дополнить пропущенные

шаги. Например вы можете добавить треки, улучшить разрешение, заменить инструменты и т.д.



Кстати говоря, пример, приведенный выше на рисунке, является простейшим представлением *Индеанки* прекрасной Гуарани Хосе Асунсьона Флореса. Вот ее текст:

India, bella mezcla de diosa y pantera, doncella desnuda que habita el Guaira Arisca romanza curvo sus caderas copiando un recodo de azul Parana	Индеанка, красивая метиска богини и пантеры, Обнаженная девушка, живущая в Гуаире. Танец придал её бедрам форму Синих изгибов реки Параны.
Bravea en las sienes su orgullo de plumas, su lengua es salvaje panal de eirusu Collar de colmillos de tigres y pumas enjoya a la musa de Ybytyruzu.	На ее висках сияет гордость перьев, Ее язык – дикие соты , Ожерелье из клыков тигров и пум – Драгоценное украшение для музы Ибитирусу.

```
implement midi
    open core
class predicates
    note:(string, unsigned8) procedure (i, o).
    duration:(string, unsigned8) procedure (i, o).
    tomidi:(string*, unsigned8*) procedure (i, o).
    mSz:(integer, unsigned8, unsigned8, unsigned8, unsigned8)
        procedure (i, o, o, o, o).
class facts
    nt:(string, unsigned8).
clauses
    classInfo("midi/midi", "1.0").

    duration(X, Y) :- Y= toTerm(X), Y < 128, !.
    duration(_X, 24).

    nt("C", 60). nt("D", 62). nt("E", 64). nt("F", 65). nt("G", 67).
    nt("A", 69). nt("B", 71). nt("C5", 72). nt("D5", 74).
    nt("C#", 61). nt("D#", 63). nt("F#", 66).
    nt("G#", 68). nt("A#", 70). nt("Bb", 70).

    note(X, Y) :- nt(X, Y), !.
    note(_, 69).
```

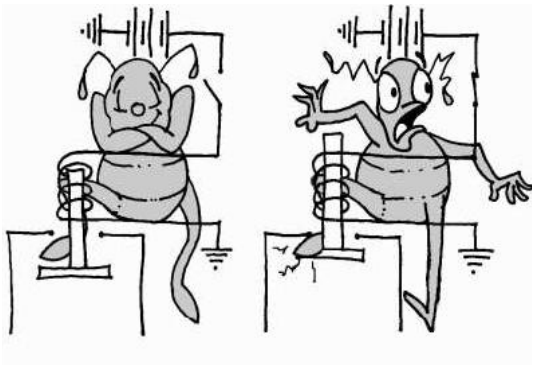
```
    tomidi([N, D|Rs], [0,192, 24, 0,144,N1, 127,F1,128,N1,0|R] ) :-
        /* guitar=24, violin= 40 voice= 52 */
        note(N, N1), duration(D, F1), !, tomidi(Rs, R).
    tomidi(_, []).

    mSz(S, 0, 0, X3, X4) :- I4= S mod 256, X4= convert(unsigned8, I4),
        S1= S div 256, I3= S1 mod 256, X3= convert(unsigned8, I3).

    generate(File, Music) :- tomidi(Music, L),
        Events= [0, 255, 89, 2, 0, 0, 0, 255, 81, 3, 7, 161,
            32, 0, 255, 88, 4, 2, 2, 24, 8],
        NC= list::length(Events), N1= list::length(L),
        Cnt= NC+ N1 + 4, mSz(Cnt, X1, X2, X3, X4),
        MainHeader= [ 77, 84, 104, 100, 0, 0, 0, 6, 0, 0, 0, 1, 0, 48,
            77, 84, 114, 107, X1, X2, X3, X4],
        H= list::append(MainHeader, Events, L, [0,255,47,0]),
        P= outputStream_file::create(File, stream::binary()),
        foreach X= list::getMember_nd (H) do
            P:write(X)
        end foreach,
        P:close().
end implement midi
```

Рисунок 19.3 Midi-генератор

Глава 20: Ошибки



Старые компьютеры были основаны на реле, которые представляют собой громоздкие электрические устройства. Обычно они содержат электромагнит, который активизируется током в одной цепи, для включения или выключения другой цепи. Компьютеры, сделанные из подобных вещей, были огромными, медленными и ненадёжными. Итак, 9 сентября 1945 года моль влетела в одно из реле компьютера Mark II в Гарварде и заклинила его. С тех пор слово *bug*[1] (***баг***) стало стандартным словом для обозначения ошибки, которая препятствует компьютеру работать как предназначено.

Из-за багов компилятор Visual Prolog часто возвращает сообщения об ошибках, вместо того, чтобы генерировать код и запускать соответствующие программы. В действительности, компилятор Visual Prolog выдаёт больше сообщений об ошибках, чем любой другой язык, кроме, возможно, Clean. Студенты не ценят это постоянное ворчание об ошибках и редко пытаются проанализировать сообщения об ошибках, но поверьте мне, вам следует быть благодарными разработчикам такого компилятора, который даёт вам шанс вычистить все пятна, которые могут повредить вашей программе. Как я уже сказал, Visual Prolog хорош в локализации ошибок. Несмотря на это, если вы не научитесь разбираться с сообщениями об ошибках, отладка покажется вам случайным хождением, которое почти никогда не ведёт к успеху. Поэтому в этой главе вы узнаете, как интерпретировать сообщения компилятора.

20.1. Ошибка типа

В следующей программе нет ошибок. Однако, если вам понадобится использовать беззнаковое целое, которое получается в результате вычисления функции факториала, для того чтобы, скажем, вычислить биномиальный коэффициент, компилятор может отказаться ее компилировать.

```
implement main
open core
clauses
    classInfo("main", "bugs").
class predicates
    fact:(unsigned) -> unsigned.
clauses
    fact(N) = F :- if N < 1 then F=1
        else F=N*fact(N-1) end if.
    run():- console::init(),
```

[1] Букашка.

```
        stdio::write("N: "),
        N= stdio::read(),
        stdio::write(fact(N)), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

Например, если вы попробуете скомпилировать программу, приведенную на рисунке 20.1, вы получите следующее сообщение об ошибке:

```
error c504: The expression has type '::integer',
which is incompatible with the type '::unsigned'
```

Ошибка исчезнет, если вы явно преобразуете каждый входной аргумент `binum/3` в беззнаковое целое. Вы должны также преобразовать выходной аргумент факториала в целое.

```
binum(N, P, R) :-
    N1= convert(unsigned, N),
    P1= convert(unsigned, P),
    R = convert(integer, fact(N1) div (fact(P1)*fact(N1-P1)))
```

```
implement main
    open core
clauses
    classInfo("main", "bugs").

class predicates
    fact:(unsigned) -> unsigned.
    binum:(integer, integer, integer) procedure (i, i, o).

clauses
    fact(N) = F :- if N < 1 then F=1 else F=N*fact(N-1) end if.

    binum(N, P, R) :- R = fact(N) div (fact(P)*fact(N-P)).

    run():- console::init(),
        stdio::write("N: "), N= stdio::read(),
        stdio::write("P: "), P= stdio::read(),
        binum(N, P, R), stdio::write(R), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

Рисунок 20.1. Программа, содержащая ошибки

20.2. Не-процедура внутри процедуры

Существует другая ошибка, которая доставляет много сложностей начинающим. Visual Prolog объявляет некоторые предикаты как процедуры. Например, в случае предиката `run()`, который имеется во всех консольных программах. Это означает, что вы не можете вызвать недетерминированный предикат из `run()`. Таким образом, если вы попытаетесь скомпилировать программу, приведенную на рисунке 20.2, вы получите следующее сообщение об ошибке:

```
error c631: The predicate 'nonprocedure::run/0',
which is declared as 'procedure', is actually 'nondeterm'
```

```
implement main
class facts
    weight:(integer, real).
clauses
    classInfo("main", "nonprocedure").

    weight(0, -1.0).
    weight(1, 2.0).
    weight(2, 3.5).

    run():- console::init(),
        weight(1, X), stdio::write(X), stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

Рисунок 20.2 Ошибка, вызванная недетерминированным предикатом

Это правда, что предикат `weight/2` индексируется в соответствии с его целым аргументом. Это означает, что для заданного целого аргумента предикат `weight/2` должен действовать как процедура. Однако Пролог глух к принятию желаемого за действительное. Поэтому вам необходимо определить процедуру `getweight/2`, которая будет возвращать весовые значения.

Ниже вы сможете увидеть, каким образом может быть определен предикат `getweight`. Используйте это в качестве модели для выхода из ситуаций, в которых вам нужно вызвать недетерминированный предикат из процедуры.

```
implement main
class facts
    weight:(integer, real).
class predicates
    getweight:(integer, real) procedure (i, o).
clauses
    classInfo("main", "nonprocedure").
```

```
    weight(0, -1.0).
    weight(1, 2.0).
    weight(2, 3.5).

    getweight(I, R) :- weight(I, R), ! or R=0.

    run():- console::init(), getweight(1, X),
        stdio::write(X), stdio::nl.
end implement main
goal  mainExe::run(main::run).
```

20.3. Недетерминированное условие

В Прологе имеются конструкции, которые не допускают недетерминированные предикаты. Например, условие в конструкции `if-then-else` обязано быть детерминированным предикатом. Поэтому программа, приведенная на рисунке 20.3, выведет сообщение об ошибке:

```
error c634: The condition part of if-then-else may not
have a backtrack point (almost determ)
```

```
implement main
class predicates
    vowel:(string, string) procedure (i, o).
    member:(string, string*) nondeterm.
clauses
    classInfo("main", "ifexample-1.0").

    member(X, [X|_]).
    member(X, [_|Xs]) :- member(X, Xs).

    vowel(X, Ans) :- if member(X, ["a", "e", "i", "o", "u"])
        then Ans= "yes" else Ans="no" end if.

    run():- console::init(),
        vowel("e", Ans),
        stdio::write(Ans).
end implement main
goal mainExe::run(main::run).
```

Рисунок 20.3 Требуется детерминированный предикат

Ошибка исчезнет, если вы определите предикат `member` как детерминированный:

```
class predicates
    member:(string, string*) determ.
clauses
    member(X, [X|_]) :- !.
    member(X, [_|Xs]) :- member(X, Xs).
```

20.4. Невозможность определения типа

Бывают случаи, когда необходимо определить тип переменной до ее конкретизации. Уточнение типа может быть произведено с помощью следующей процедуры:

```
hasdomain(type, Var)
```

Если убрать вызов `hasdomain(integer, X)` в бесполезной программе, приведённой ниже, появится сообщение об ошибке, утверждающее, что невозможно определить тип терма `X`.

```
implement main
clauses
    classInfo("main", "vartest-1.0").

    run() :- console::init(),
        hasdomain(integer, X), X= stdio::read(),
        stdio::write(X, X, X), stdio::nl.
end implement main
goal mainExe::run(main::run).
```

20.5. Схема входа-выхода аргументов предиката

Когда вы не объявляете *flow pattern*, Пролог предполагает, что все аргументы являются входными. Очень часто это допущение (что все аргументы у вашего предиката – входные – *ред*. *пер*.) бывает неверным, вы получите сообщение об ошибке, как это показано в примере, приведенном ниже.

```
implement main
class predicates
    factorial:(integer, integer).
clauses
    classInfo("main", "dataflow-1.0").

    factorial(0, 1) :- !.
    factorial(N, N*F1) :- factorial(N-1, F1).

    run():- console::init(), N= toTerm(stdio::readLine()),
        factorial(N, F), stdio::write(F).
end implement main
goal mainExe::run(main::run).
```

Вы можете исправить ошибку, если объявите `factorial` как процедуру с входом и выходом:

```
class predicates
    factorial:(integer, integer) procedure (i, o).
```

Глава 21: Управление базой данных

Настоящая глава показывает, как создавать форму, содержащую поля редактирования, и использовать их содержимое для заполнения базы данных. Однако самое важное состоит в том, что вы научитесь получать доступ к полям из формы. Поле редактирования хранится в факте-переменной, имя которой вы должны узнать и затем использовать его для управления окном, как показано в приведенном ниже примере.

```
onSave(_Source) = button::defaultAction() :-
    Key= keyvalue_ctl:getText(),
    Email= email_ctl:getText(), T= listEdit_ctl:getText(),
    emails::insert_info(Key, Email, T).
```

21.1. Управление базами данных

Базы данных очень полезны в любом коммерческом приложении. Если вы не особенно много знаете о базах данных, я очень рекомендую вам поискать в интернете книгу по этой теме. Ниже приведены основные понятия баз данных, основанных на ***B+деревьях***.

Indexes. ***Индексы*** хранятся в специальной структуре данных под названием ***B+Tree***. Все, что вы должны знать про ***B+Tree***, — это то, что они хранят вещи упорядоченно (например, в алфавитном порядке). Когда вещи упорядочены, как, например, слова в словаре, их легче искать.

Domains. ***Домены*** определяют структуру записей. В нашем примере домен *contact* определяется в виде:

```
domains
    contact=email(string, string, string).
```

References. ***Ссылки*** являются указателями на места, в которых хранятся записи.

После этого краткого введения перейдем к программе.

- Создайте проект под названием *emailBook* с объектно-ориентированным GUI.
- Используя пункт меню *File/Add*, добавьте пакет `chainDB`. Этот пакет находится внутри папки *pfc* в установочном каталоге. Постройте приложение.
- Создайте класс `emails` в корне дерева проекта. Уберите галочку для *Creates Objects*. В разделе 21.2 вы найдете код для файлов `emails.cl` и `emails.pro`, который вы должны в них вставить. Постройте приложение.
- Добавьте новый элемент в главное меню приложения. Для того чтобы это сделать, пройдите в окно дерева проекта и дважды щёлкните по элементу `TaskMenu.mnu`. В диалоговом окне *TaskMenu* щёлкните правой кнопкой мыши и выберите *New* из всплывающего меню. В соответствующем поле напишите *Create* — название пункта меню. Постройте приложение.
- Добавьте следующий код:

```
class predicates
    createNewDB:(string FNAME).
predicates
    onCreate : window::menuItemListener.
clauses
    onCreate(_Source, _MenuTag) :-
        FName= vpiCommonDialogs::getFileName("*.*",
                            ["Email", "*.dbs"],
                            "Create", [], ".", _X), !,
        createNewDB(FName).
    onCreate(_, _).

    createNewDB(Fname) :- file::existfile(Fname), !.
    createNewDB(Fname) :- emails::data_base_create(Fname).
```

для *Project Window\TaskMenu.win\Code Expert\Menu\id_create*.

Вы должны разрабатывать свои программы шаг за шагом, проверяя, верен ли каждый новый шаг, перед тем, как двигаться дальше. Сейчас вы добавили новый пункт *Create* в меню `TaskMenu.mnu`. Вы также вставили код в `onCreate`, для того чтобы прочитать имя файла и создать соответствующую базу данных. Постройте приложение, запустите его и выберите пункт *Create* меню приложения. Когда диалоговое окно выбора файла спросит вас об имени файла, наберите `mails.dbs`. Если всё будет идти по сценарию, программа создаст файл `mails.dbs`. Когда вы хотите использовать базу данных, вам нужно её открыть. После её использования вы должны закрыть её, до того как будет выключен компьютер. Для того чтобы быть уверенными в том, что мы не забудем закрыть базу данных до выхода из приложения, добавим следующий код:

```
clauses
    onDestroy(_) :- emails::data_base_close().
```

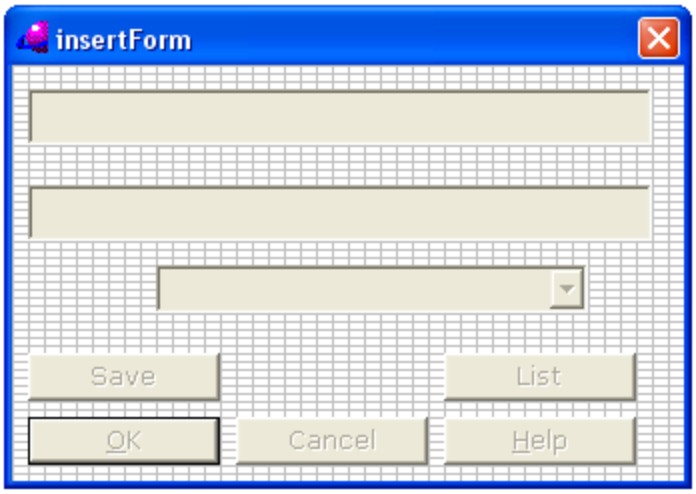
для *Project Tree/TaskWindow.win/Code Expert/Window/onDestroy*.

Теперь, когда вы уверены, что никто не забудет закрыть базу данных, можно переходить к тому, как её открывать. Дважды щёлкните по элементу `TaskMenu.mnu` дерева проекта и включите элемент меню *&Open\tF8*. Добавьте следующий фрагмент кода:

```
class predicates
    openDB:(string FNAME).
predicates
    onFileOpen : window::menuItemListener.
clauses
    onFileOpen(_Source, _MenuTag) :-
        FName= vpiCommonDialogs::getFileName("*.*",
                            ["Email", "*.dbs"],
                            "Open", [], ".", _X), !,
        openDB(FName).
    onFileOpen(_, _).
    openDB(Fname) :- file::existfile(Fname), !,
        emails::data_base_open(Fname).
    openDB(_).
```

для *ProjWin/TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_open*.

Постройте приложение. Затем откройте пункт *File/New in New Package* меню среды и вставьте форму, приведенную ниже, в корень дерева проекта.



На этой форме имеется раскрывающийся список *listEdit*. Поэтому необходимо добавить в него список элементов. Для того чтобы сделать это, добавьте фрагмент кода

```
clauses
    onListEditShow(_Source, _CreationData) :-
        WlboxHandle=listEdit_ctl:tryGetVpiWindow(), !,
        vpi::lboxAdd(WlboxHandle, ["Person", "Commerce"]).
    onListEditShow(_, _).
```

для обработчика *ShowListener* списка `listEdit:listEdit_ctl`.

Последним шагом в этом проекте является добавление кода для кнопок *Save* и *List*. Добавьте следующий код:

```
onSaveClick(_Source) = button::defaultAction() :-
    Key= key_ctl:getText(),
    Email= address_ctl:getText(),
    T= listEdit_ctl:getText(),
    emails::insert_info(Key, Email, T).
```

для обработчика *ClickResponder* кнопки `button:save_ctl`. Наконец добавьте код

```
clauses
    onListClick(_Source) = button::defaultAction() :-
        emails::db_list().
```

для обработчика *ClickResponder* кнопки `button:list_ctl`. Постройте приложение еще раз.

21.2. Класс emails

```
% Файл emails.cl
```

```
class emails
    open core
predicates
    classInfo : core::classInfo.
    data_base_create:(string FileName) procedure (i).
    data_base_open:(string FileName).
    data_base_close:().
    insert_info:(string Name, string Contact, string Tipo).
    data_base_search:(string Name, string Contact, string Tipo)
        procedure (i, o, o).
    db_list:().
    del:(string).
end class emails

% Файл emails.pro
implement emails
    open core
domains
    contact= email(string, string, string).
class facts - dbState
    dbase:(chainDB, chainDB::bt_selector) determ.

clauses
    classInfo("db/emails/emails", "1.0").
    data_base_create(FileName) :-
        ChainDB= chainDB::db_create(FileName, chainDB::in_file),
        ChainDB:bt_create("email", ContactName, 40, 21, 1),
        ChainDB:bt_close(ContactName),
        ChainDB:db_close(), !.
    data_base_create(_).

    data_base_open(FileName) :-
        ChainDB= chainDB::db_open(FileName, chainDB::in_file),
        ChainDB:bt_open("email", ContactName),
        assert(dbase(ChainDB, ContactName)).

    data_base_close() :-
        retract(dbase(ChainDB, BTREE)), !,
        ChainDB:bt_close(BTREE),
        ChainDB:db_close().
    data_base_close().

    insert_info(Key, Email, Tipo) :-
        dbase(ChainDB, BTREE), !,
        ChainDB:chain_insertz("Contacts",
            email(Key, Email, Tipo),
            RefNumber),
        ChainDB:key_insert(BTREE, Key, RefNumber).
    insert_info( _, _, _).

    data_base_search(Name, Contact, Tipo) :-
        dbase(ChainDB, BTREE),
```

```
        ChainDB:key_search(BTREE, Name, Ref),
        ChainDB:ref_term(Ref, Term),
        Term= email(Name, Contact, Tipo), !.
    data_base_search(_Name, "none", "none").

class predicates
    getInfo:(chainDB, chainDB::ref, string, string)
        procedure (i, i, o, o).
    dbLoop:().
clauses
    getInfo(ChainDB, Ref, Name, Email) :-
        ChainDB:ref_term(Ref, Term),
        Term= email(Name, Email, _), !.
    getInfo(_, _, "none", "none").

    db_list() :-
        dbase(ChainDB, BTREE),
        ChainDB:key_first(BTREE, Ref), !,
        getInfo(ChainDB, Ref, Name, Email),
        stdio::write(Name, ", ", Email), stdio::nl,
        dbLoop().
    db_list() :- stdio::write("None"), stdio::nl.

    dbLoop() :- dbase(ChainDB, BTREE),
        ChainDB:key_next(BTREE, Ref), !,
        getInfo(ChainDB, Ref, Name, Email),
        stdio::write(Name, ", ", Email), stdio::nl,
        dbLoop().
    dbLoop().

    del(Key) :-
        dbase(ChainDB, BTREE),
        ChainDB:key_search(BTREE, Key, Ref), !,
        ChainDB:key_delete(BTREE, Key, Ref),
        ChainDB:term_delete("Contacts", Ref).
    del(_Key).
end implement emails
```

21.3. Объект базы данных

Функция, которая создаёт базу данных, очень проста:

```
data_base_create(FileName) :-
    ChainDB= chainDB::db_create(FileName, chainDB::in_file),
    ChainDB:bt_create("email", ContactName, 40, 21, 1),
    ChainDB:bt_close(ContactName),
    ChainDB:db_close(), !.
data_base_create(_).
```

Метод db_create/2 делает то, что он означает, т.е. он создаёт базу данных внутри FileName и возвращает объект для работы с ней. Объект хранится в ChainDB. Из ChainDB вызывается bt_create для того чтобы получить BTree. Пакет BTree конкретизирует переменную ContactName указателем на BTree. Наконец закрывается и BTree, и база данных.

База данных открывается с помощью предиката, который почти так же прост, как и тот предикат, что использовался для её создания.

```
data_base_open(FileName) :-
    ChainDB= chainDB::db_open(FileName, chainDB::in_file),
    ChainDB:bt_open("email", ContactName),
    assert(dbase(ChainDB, ContactName)).

data_base_close() :-
    retract(dbase(ChainDB, BTREE)), !,
    ChainDB:bt_close(BTREE),
    ChainDB:db_close().
data_base_close().
```

Предикат db_open открывает базу данных, хранящуюся внутри FileName, и возвращает объект, который может быть использован для управления ею. Объект сохраняется в ChainDB, который используется для открытия BTree email. Как ChainDB, так и указатель на BTree сохраняются в базе фактов, для использования в дальнейшем.

```
assert(dbase(ChainDB, ContactName))
```

Когда пользователь закрывает приложение, предикат data_base_close удаляет объект базы данных и указатель на BTree из базы фактов и закрывает как BTree, так и ChainDB. До этого места мы описывали предикаты, которые управляют базой данных. Теперь мы опишем метод, который можно использовать для внесения сведений в базу данных.

```
insert_info(Key, Email, Tipo) :- dbase(ChainDB, BTREE), !,
    ChainDB:chain_insertz("Contacts",
            email(Key, Email, Tipo),
            RefNumber),
    ChainDB:key_insert(BTREE, Key, RefNumber).
insert_info( _, _, _).
```

База данных — это наполняемая система. Как и любая наполняемая система, она имеет два компонента: цепь папок, каждая содержит данные, которые необходимо хранить, и алфавитный указатель, который указывает на различные папки и позволяет потенциальному пользователю обнаружить, где хранится желаемая информация. В нашем примере цепь папок называется Contacts. Каждая папка имеет следующий вид: email(string, string, string). Первый аргумент email хранит название контакта, второй содержит электронное письмо, и третий информирует о типе контакта, который ассоциирован с этой записью базы данных.

Предикат chain_insertz сохраняет папку в цепи Contacts. Он возвращает указатель RefNumber на ее местоположение. Указатель RefNumber будет сохранен в алфавитном порядке в BTree.

Для того чтобы найти электронное письмо в `Contact`, соответствующее заданному имени `Name`, ищется ссылка в `BTree`. Обладая номером ссылки, можно прямо получить папку.

```
data_base_search(Name, Contact, Tipo) :- dbase(ChainDB, BTREE),
    ChainDB:key_search(BTREE, Name, Ref),
    ChainDB:ref_term(Ref, Term),
    Term= email(Name, Contact, Tipo), !.
data_base_search(_Name, "none", "none").
```

Предикат, который используется для удаления элемента, легко понять, и я не буду на нём останавливаться. Предикат, который перечисляет содержимое базы данных в алфавитном порядке, является гораздо более сложным, но не трудным для понимания, если вы знаете основные принципы управления базами данных.

Глава 22: Книги и статьи

В этой главе я буду говорить о некоторых книгах по Прологу и технике программирования. Вы заметите, что книги несколько устарели и больше не издаются. Одна из причин этого состоит в том, что люди не покупают техническую литературу так часто, как раньше, так как большинство студентов надеются получить материал, как и это руководство, из интернета. Другая причина заключается в том, что многие великие учёные в области *computer science* были с энтузиазмом увлечены Прологом в восьмидесятые и девяностые годы. Эти учёные написали книги, которые тяжело превзойти. Например, я не могу найти ни одной современной книги, которая бы приблизилась к [Coelho/Cotta].

22.1. Грамматики

Пролог великолепно подходит для обработки языков, написания компиляторов и подготовки документов. Фактически этот язык был изобретён с расчётом на обработку естественных языков. Большинство книг работают со стандартным Прологом, который несколько отличается от Visual Prolog. В то время как стандартный Пролог был разработан для быстрого прототипирования маленьких приложений и для проверки идей, Visual Prolog используется в крупных коммерческих и промышленных системах. Уэсли Баррос (*Wellesley Barros*), например, разработал и запрограммировал в Visual Prolog крупную систему для управления больницей. Будьте уверены, что в подобной системе нельзя позволить себе аварии из-за ошибок типа. В критических ситуациях проверки и объявления типов очень важны. В любом случае, несмотря на то, что вы программируете в Visual Prolog, вы можете извлечь пользу из идей, представленных в книгах, основанных на стандартном Прологе. Одна старая книга, которую вы можете найти в своей местной библиотеке, была написана аргентинским ученым Вероникой Даль (*Veronica Dahl* [Abramson/Dahl]), одним из величайших современных лингвистов.

Доктор Седрик де Карвальо (*Cedric de Carvalho*) написал компилятор Пролога, который генерирует байт-код Java. Этот компилятор позволяет писать в Прологе апплеты. Его компилятор — ещё не до конца оперившийся инструмент, но вы легко можете его усовершенствовать. Между тем вы многое изучите о компиляции, грамматическом разборе, языке Java и т. д. Вы можете начать читать статьи Седрика в [Cedric et all]. Вы также можете портировать компилятор, написанный в Visual Prolog 5.2, в систему Visual Prolog 7.x. Вот адрес:

http://netProlog.pdc.dk

Другой книгой по обработке естественных языков со множеством интересных подходов к грамматическому формализму, является книга [Pereira/Shieber]. Эта книга является более дидактической, чем книга [Abramson/Dahl]. В действительности книга [Abramson/Dahl] была написана для специалистов, а [Pereira/Shieber] говорит со студентами, которые ещё только учатся сложному искусству обработки языков. Книгой, которая не специализируется на обработке естественных языков, но дает хорошее введение в этот предмет, является [Sterling and Shapiro]. Если вы хотите прочитать более продвинутую книгу по этой теме, хорошим выбором является книга [Gazdar/Mellish].

22.2. Базы данных

Дэвид Уоррен, учёный в области *computer science*, является главным сторонником использования Пролога как движка для баз данных. Книга [Warren] больше не издается, но вы должны достать копию, если вы хотите изучать базы данных. Это классика.

Дэвид Уоррен руководит командой, которая разрабатывает XSB, Пролог, ориентированный на базы данных, в университете Стони Брук (*Stony Brook University*). Хотя я не думаю, что эта версия Пролога может использоваться для серьёзных приложений, так как она полагается на иные инструменты для построения GUI, не компилируется (написать красивое приложение в XSB нелегко) и не типизирована, я очень рекомендую посетить страницу XSB. Там вы найдёте множество хороших идей о базах данных и Прологе, а также об инструментах для их разработки. Вот адрес:

http://www.cs.sunysb.edu/~sbProlog/xsb-page.html

Между прочим, XSB предлагает набор инструментов для экспорта его функциональности в другие языки. Таким образом, если вам нравятся его свойства, вы можете вызвать DLL, написанную на языке XSB, из Visual Prolog! Страница XSB показывает, как вызвать DLL из Visual Basic. Вы можете использовать тот же подход, чтобы вызвать её из Visual Prolog.

22.3. Техника программирования

Если вам сложно разобраться с такими понятиями, как *рекурсивное программирование, операции со списками, отсечение, недетерминированное программирование* и *грамматический разбор*, вам следует прочитать книгу «Искусство программирования на языке Пролог» [Sterling and Shapiro]. Идеи, представленные в этой книге, легко могут быть перенесены в Visual Prolog. В Visual Prolog вы даже можете улучшить оригинальные программы. Например, один из моих студентов сумел создать диаграммы для электрических цепей, описанных в главе 2 [Sterling and Shapiro].

Моей самой любимой книгой по Прологу по-прежнему является «Пролог в примерах» [Coelho/Cotta]. Она имеет вид FAQ: авторы ставят проблему и показывают ее решение. Это также антология. Проблемы и решения были предложены и решены великими учёными *computer science*. Эта книга заслуживает нового издания. Настойчиво пытайтесь достать копию. Если вы не можете получить копию, попросите разрешения авторов и сделайте ксерокопию.

Часть II
Искусственный интеллект

Глава 23: Поиск

Патрик Генри Уинстон (*Patrick Henry Winston*), автор очень известной книги про язык LISP, начал свои исследования в области искусственного интеллекта с методов поиска, утверждая, что поиск — это вещь повсеместная. Точнее не скажешь. И действительно: всё, от планирования движений роботов до нейросетей, может быть сведено к поиску. Поэтому, подобно Уинстону, мы начнём своё изучение искусственного интеллекта с методов поиска.

23.1. Состояния

Применительно к искусственному интеллекту, выражение *решение задач* относится к анализу того, как компьютеры могут использовать поиск для решения проблем в хорошо определенных доменах. Из этого определения возникает один вопрос: что интеллектуальная программа просматривает при поиске решения задачи? Она бродит по лабиринту состояний, связанных операциями, приводящими к изменению свойств.

Объект обладает состояниями, то есть множеством свойств, которое определяет его положение в данный момент. Например, Евклид определял точку как сущность, имеющую лишь одно свойство — местоположение. Следовательно, состояние точечного объекта задаётся его координатами. В общем случае для определения состояния объекта используются и другие свойства, например:

Цвет: красный, зелёный, синий и др.
Целостность сообщает, является ли объект целым или несобранным.
Позиция задаёт направление, в котором повёрнут объект.
Безопасность говорит о том, защищён ли объект от разрушения.

Каждое свойство имеет один или несколько атрибутов, а также значения этих атрибутов. Например, местоположение имеет три атрибута, которые являются координатами точки. В двумерном пространстве атрибуты местоположения могут принимать значения в декартовом произведении $\mathbf{R} \times \mathbf{R}$, поэтому кортеж `(50, 200)` дает верные значения для местоположения. Если говорить кратко, то можно сказать, что состояние — это запечатление во времени. Каждый атрибут, который можно изменять, чтобы решить задачу, называется степенью свободы, так что точка в трехмерном пространстве имеет три степени свободы.

Оператором называется процедура, которая может быть использована для совершения перехода из одного состояния в другое. Программа, обладающая искусственным интеллектом, начинает от начального состояния и использует допустимые операторы для продвижения в направлении целевого состояния. Множество всех состояний, которые могут принимать рассматриваемые объекты, и все переходы, которые ведут из одного состояния в другое, называется пространством состояний.

23.2. Дерево поиска

Дерево поиска — это распространённая диаграмма, на которой каждое состояние представляется в виде помеченной вершины (см. идентификатор внутри круга ниже на рисунке), а каждый допустимый переход из одного состояния в другое описывается в виде ветви перевернутого дерева. На рисунке 23.1 вы можете увидеть изображение дерева поиска.

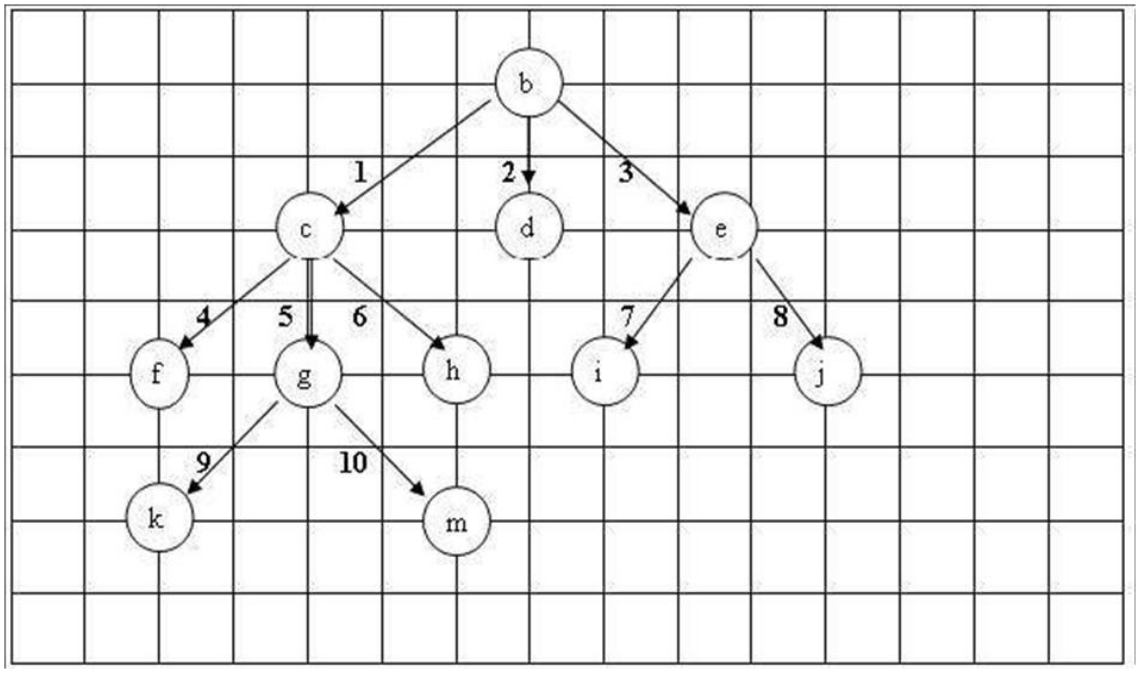
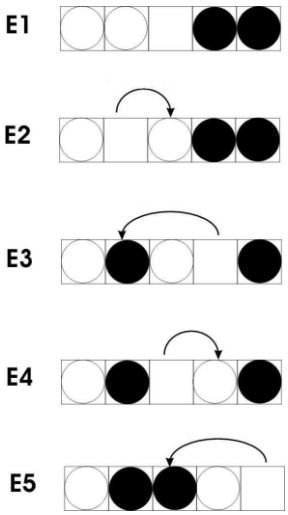


Рисунок 23.1 Дерево поиска

Вершины b, c, d, t, f, g, h, i, j, k и m (см. рис. 23.1) являются состояниями, через которые может пройти объект. Ветви, соединяющие две вершины, соответствуют операторам, которые вызывают переход из одного состояния в другое. Например, ветвь под номером 3 на рисунке 23.1 — это оператор, который вызывает переход из состояния b в состояние e. Если вершина X может быть достигнута из другой вершины Y с помощью только одного оператора, говорят, что X — это дочь Y и что Y — это мать X. Таким образом, вершина c является дочерью вершины b, так как c можно достичь из b с помощью оператора 1. Если у вершины нет дочери, она называется листом. Вершина, у которой нет матери, является корнем дерева поиска.



Более конкретный пример поможет вам понять идеи, которые мы изучили выше. Две черные фишки, подобные шашкам, и две белые фишки расположены так, как показано на рисунке. Черные фишки отделены от белых пустым местом. Нужно расположить черные фишки между белыми. Допустимы две операции: (1) подвинуть фишку на пустое место и (2) перепрыгнуть через фишку на пустое место. Рисунок справа показывает пошаговое решение этой задачи.

В задаче, которую мы хотим решить, интересующие нас объекты ограничены четырьмя фишками и пустым местом. Позиции фишек и пустого места определяют

состояние. Начальное состояние E1 показано в верхней части рисунка. Цель E5 находится в нижней части рисунка.

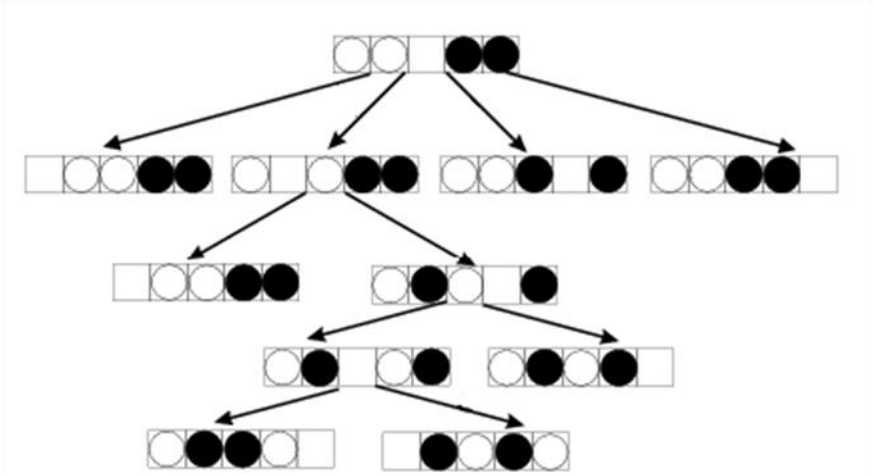


Рисунок 23.2 Головоломка

На рисунке 23.2 вы можете увидеть дерево поиска для описанной выше головоломки. Можно сказать, что состояния подобны городам на карте, операторы — дорогам, соединяющим города. Задача решена, когда найден путь, ведущий от корня к цели, и самый очевидный способ найти путь — это ходить от города к городу, пока не будет найден путь, соответствующий цели. Этот процесс называется поиском.

23.3. Поиск в ширину

Поиск называется исчерпывающим, если он гарантирует порождение всех достижимых состояний до того, как он завершится по неудаче. Одним из способов осуществить исчерпывающий поиск является генерация всех вершин на определённом уровне, до того как произойдет продвижение к следующему уровню дерева. Этот метод называется поиском в ширину. Он гарантирует, что пространство допустимых операций будет рассматриваться систематически.

Для дерева, приведенного на рисунке 23.1, поиск начинается с рассмотрения уровня `(c-d-e)`, содержащего вершины, расположенные на расстоянии одной ветви от корня. Затем он переходит к уровню `(f-g-h-i-j)`, расположенному на расстоянии двух ветвей от корня. Наконец, механизм поиска посещает уровень `(k-m)`. Программа, выполняющая операцию поиска, обладает параметром, который называется *Очередь*. Он содержит список путей – кандидатов. Каждый путь представляется в виде списка вершин, при этом вершина описывается в виде `r(integer, string)`.

```
implement main
domains
    node= r(integer, string).
    path= node*.
    queue= path*.

class facts
    operator:(integer, string, string).

class predicates
```

```
        bSearch:(queue, path) determ (i, o).
        nextLevel:(path, path) nondeterm (i, o).
        solved:(path) determ.
        prtSolution:(path).

    clauses
        classInfo("main", "breath").
        solved(L) :- L= [r(_, "k")|_].

        prtSolution(L) :-
            foreach P= list::getMember_nd(list::reverse(L)) do
                stdio::write(P), stdio::nl
            end foreach.

        operator(1, "b", "c").
        operator(2, "b", "d").
        operator(3, "b", "e").
        operator(4, "c", "f").
        operator(5, "c", "g").
        operator(6, "c", "h").
        operator(7, "e", "i").
        operator(8, "e", "j").
        operator(9, "g", "k").
        operator(10, "g", "m").


        bSearch([T|Queue],Solution) :-
            if solved(T) then Solution= T
            else Extentions= [Daughter || nextLevel(T, Daughter)],
            ExtendedQueue= list::append(Queue, Extentions),
            bSearch(ExtendedQueue, Solution) end if.

        nextLevel([r(Branch, N)|Path], [r(Op, Daughter),r(Branch, N)|Path])
    :-
            operator(Op, N, Daughter),
            not(list::isMember(r(Op, Daughter),Path)).

        run():- console::init(),
            if bSearch([[r(0, "b")]], L) then prtSolution(L)
            else stdio::write("No solution!"), stdio::nl end if.
    end implement main /* breath*/
    goal mainExe::run(main::run).
```

Изначально `Queue` содержит единственный путь, который начинается и заканчивается в корне:

```
    bSearch([[r(0, "b")]], L)
```

Так как этот путь не ведёт к цели, он продолжается до каждой дочери вершины “b”, при этом продолжения дописываются в конец очереди:

```
bSearch([ [r(1, "c"),r(0, "b")],
        [r(2, "d"),r(0, "b")],
        [r(3, "e"),r(0, "b")]], L)
```

Пока что робот посетил только одну вершину, а именно вершину “b”. Затем в конец очереди дописываются продожения из вершины “c”:

```
bSearch([ [r(2, "d"),r(0, "b")],
        [r(3, "e"),r(0, "b")],
        [r(4, "f"),r(3, "c"),r(0, "b")],
        [r(5, "g"),r(3, "c"),r(0, "b")],
        [r(6, "h"),r(3, "c"),r(0, "b")]], L)
```

Вершина “d” достигнута, но, так как дочерей у нее нет, этот путь просто удаляется из очереди:

```
bSearch([ [r(3, "e"),r(0, "b")],
        [r(4, "f"),r(3, "c"),r(0, "b")],
        [r(5, "g"),r(3, "c"),r(0, "b")],
        [r(6, "h"),r(3, "c"),r(0, "b")]], L)
```

Теперь дочери вершины “e” дописываются в конец очереди:

```
bSearch([ [r(4, "f"),r(3, "c"),r(0, "b")],
        [r(5, "g"),r(3, "c"),r(0, "b")],
        [r(6, "h"),r(3, "c"),r(0, "b")],
        [r(7, "i"), r(3, "e"),r(0, "b")],
        [r(8, "j"), r(3, "e"),r(0, "b")]], L)
```

Вершина “f”, не имеющая потомков, удаляется из очереди, и в конец очереди добавляются дочери вершины “g”:

```
bSearch([ [r(6, "h"),r(3, "c"),r(0, "b")],
      [r(7, "i"), r(3, "e"),r(0, "b")],
      [r(8, "j"), r(3, "e"),r(0, "b")],
      [r(9, "k"), r(5, "g"),r(3, "c"),r(0, "b")],
      [r(10, "m"), r(5, "g"),r(3, "c"),r(0, "b")]], L)
```

Вершины “h”, “i” и “j” посещаются поочередно и удаляются из очереди. Поиск заканчивается, когда алгоритм находит путь

```
[r(9, "k"), r(5, "g"),r(3, "c"),r(0, "b")]
```

Процедура

```
prtSolution(L) :-
    foreach P= list::getMember_nd(list::reverse(L)) do
        stdio::write(P), stdio::nl
    end foreach.
```

разворачивает этот путь и распечатывает каждый его элемент. Конечный результат, который выводится на экран, приведён ниже.

```
r(0,"b")
r(1,"c")
r(5,"g")
r(9,"k")

D:\vipro\aityros\aiprogs\breath\Exe>pause
Press any key to continue...
```

Теперь вы готовы к тому, чтобы попробовать применить алгоритм для решения настоящей задачи, например, нашей головоломки.

```
implement slide
domains
    node= r(string, stt).
    path= node*.
    queue= path*.
    piece= e;b;w.
    stt= st(piece, piece, piece, piece, piece).
class predicates
    bSearch:(queue, path) determ (i, o).
    nextLevel:(path, path) nondeterm (i, o).
    solved:(path) determ.
    prtSolution:(path).
    operator:(string, stt, stt) nondeterm (o, i, o).
clauses
    classInfo("breath", "1.0").
    solved(L) :- L= [r(_, st(w,b,b,w,e))|_].
    prtSolution(L) :-
        foreach P= list::getMember_nd(list::reverse(L)) do
            stdio::write(P), stdio::nl
        end foreach.

    operator("slide", st(e,A,B,C,D), st(A,e,B,C,D)).
    operator("jump", st(e,A,B,C,D), st(B,A,e,C,D)).
    operator("slide", st(A,e,B,C,D), st(A,B,e,C,D)).
    operator("slide", st(A,e,B,C,D), st(e,A,B,C,D)).
    operator("jump", st(A,e,B,C,D), st(A,C,B,e,D)).
    operator("slide",st(A,B,e,C,D),st(A,e,B,C,D)).
    operator("slide",st(A,B,e,C,D),st(A,B,C,e,D)).
    operator("jump",st(A,B,e,C,D),st(e,B,A,C,D)).
    operator("jump",st(A,B,e,C,D),st(A,B,D,C,e)).
    operator("slide",st(A,B,C,e,D),st(A,B,e,C,D)).
    operator("jump",st(A,B,C,e,D),st(A,e,C,B,D)).
    operator("slide",st(A,B,C,e,D),st(A,B,C,D,e)).
    operator("slide",st(A,B,C,D,e),st(A,B,C,e,D)).
    operator("jump",st(A,B,C,D,e),st(A,B,e,D,C)).

    bSearch([T|Queue],Solution) :-
        if solved(T) then Solution= T
        else Extentions= [Daughter || nextLevel(T, Daughter)],
            ExtendedQueue= list::append(Queue, Extentions),
            bSearch(ExtendedQueue, Solution) end if.
```

```
    nextLevel([r(Branch, N)|Path], [r(Op, Daughter),r(Branch, N)|Path])
 :-
        operator(Op, N, Daughter),
        not(list::isMember(r(Op, Daughter),Path)).

    run():- console::init(),
        if bSearch([[r("0", st(w,w,e,b,b))]], L) then prtSolution(L)
        else stdio::write("No solution!"), stdio::nl end if.
end implement slide
goal mainExe::run(slide::run).
```

Решение головоломки, напечатанное компьютером, приведено ниже.

```
r("0",st(w(),w(),e(),b(),b()))
r("slide",st(w(),e(),w(),b(),b()))
r("jump",st(w(),b(),w(),e(),b()))
r("slide",st(w(),b(),e(),w(),b()))
r("jump",st(w(),b(),b(),w(),e()))

D:\vipro\aityros\aiprogs\slide\Exe>pause
Press any key to continue...
```

23.4. Поиск в глубину

Проанализируйте рисунок 23.1. Поиск в глубину после посещения вершины “b” перемещается к ее самой левой дочери — дочери “c”. После посещения “c” он идёт к вершине “f”. Только после посещения всех потомков “c” поиск в глубину посещает вершину “g”. Для того чтобы реализовать эту стратегию, всё, что вам нужно сделать — это дописывать потомков текущей вершины в начало очереди Queue, а не в ее конец. Вот единственная необходимая модификация, сделанная для предыдущей программы:

```
dSearch([T|Queue],Solution) :-
    if solved(T) then Solution= T
    else Extentions= [Daughter || nextLevel(T, Daughter)],
        ExtendedQueue= list::append(Extentions, Queue),
        dSearch(ExtendedQueue, Solution) end if.
```

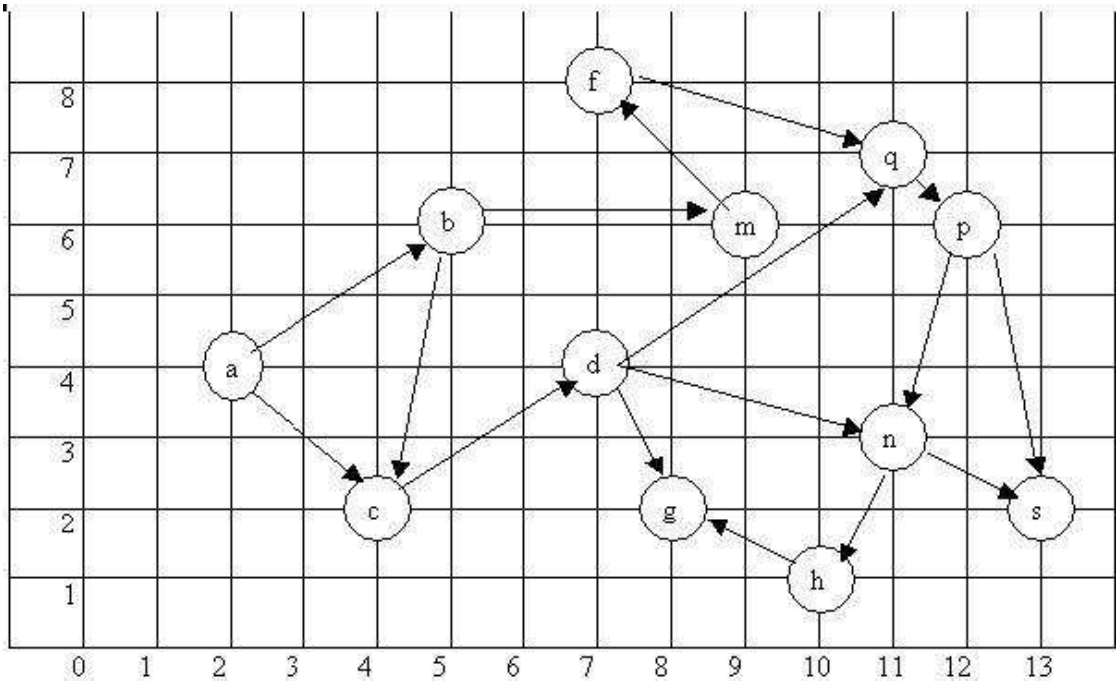
По очевидным причинам я также сменил название предиката на dSearch.

23.5. Эвристический поиск

Стратегии поиска, которые мы изучали до сих пор, никак не использовали знание предметной области для выбора рассматриваемой ветви. Этот недостаток исправляется с помощью эвристической стратегии, которая оценивает стоимость перемещения от корня к цели через вершину N до перехода в N. Функция стоимости определяется по следующей формуле:

$$f(N) = g(N) + h(N).$$

В этой формуле используются следующие обозначения: $g(N)$ — это известная стоимость перемещения из корня до вершины N; функция $h(N)$ возвращает оценку стоимости перемещения из N до целевой вершины. Эвристическая стратегия сортирует очередь таким образом, чтобы пути с наименьшими значениями для $f(N)$ посещались первыми. Используем эвристическую стратегию, для того чтобы найти путь между двумя городами на карте, приведенной ниже.



Однако перед тем, как начать, я хотел бы рассказать одну историю. Я люблю малоизвестные языки и выучил довольно много из них. Например, я знаю древнеегипетский, греческий язык Гомера, эсперанто, латынь и др. Когда я был подростком, я даже выучил гуарани, второразрядный язык, на котором говорят только в Парагвае, малонаселенной стране.

Какой-то парень или девушка перевел книгу некоего В.Н. Пушкина с русского на один из тех второразрядных языков, которые я знал. Название книги было «Эвристика, наука о творческом мышлении» (*Heuristics, the Science of Creative Thought*). Когда я читал перевод книги Пушкина, я понял, как трудно найти хорошую эвристическую функцию и насколько еще более сложно перевести текст с одного языка на другой, особенно если ты не очень хорошо разбираешься в предмете, затрагиваемом в тексте. Обработка естественного языка сложна не только для компьютеров, но также и для людей. Например, доктор Пушкин пытался объяснить правила игры в шахматы, для того чтобы использовать эту игру для тестирования искусственного интеллекта. В своих объяснениях он сказал, что конь ходит буквой Г, где Г — буква русского алфавита. Переводчик привёл следующее переложение оригинала: *Конь ходит вдоль перевёрнутой буквы L* (*The knight moves along an upsidedown L-shaped path*). После этого долгого экскурса в сферу перевода вернёмся к эвристическому поиску. Города будут представляться с помощью своих координат:

```
xy("a", 2, 4). xy("b", 5, 6).
```

Двухместные факты описывают односторонние дороги, которые соединяют города:

```
op("a", "b" ). op("b", "m"). op("m", "f").
```

Вы уже знаете, что при эвристическом поиске очередь должна быть отсортирована так, чтобы наиболее многообещающие пути были поставлены вперёд. Алгоритм сортировки, который вы узнаете, очень интересен. Он был изобретен Тони Хоаре. Каждый путь представляется с помощью следующей конструкции:

```
t(real, real, it),
```

в которой первый аргумент хранит значение эвристической функции *f* для вершины, а второй аргумент хранит значение *g*(*N*). Алгоритм сортировки имеет предикат сравнения в качестве первого аргумента. Он передвигает самые многообещающие ветви в начало очереди.

```
search([T|Queue],S):-
    if goalReached(T) then S= T
    else Extension= [E || toDaughter(T,E)],
        NewQueue= list::append(Queue,Extension),
        BestFirst= list::sortBy(cmp, NewQueue),
        search(BestFirst, S)
    end if.
```

Ниже приведен полный вариант программы.

```
implement main /*heureka*/
domains
    item=r(string, string).
    it= item*.
    node=t(real, real, it).
    tree=node*.
class facts
    xy: (string, integer, integer).
    op: (string, string).
class predicates
    getXY:(string, integer, integer) determ (i, o, o).
    cost: (string, string) -> real.
    hn: (string) -> real.
    not_in_circle: (string, it) determ (i,i).
    theGoal: (string) procedure (o).
    toDaughter: (node, node) nondeterm (i,o).
    init:(string) procedure (o).
    goalReached:(node) determ.
    search: (tree, node) determ (i,o).
    prtSolution: (node) procedure (i).
    solve: () procedure.
    cmp:(node, node) -> core::compareResult.
clauses
    classInfo("main", "heureka-1.0").

    cmp(t(A, _, _), t(B, _, _)) = core::greater() :- A > B, !.
    cmp(t(A, _, _), t(B, _, _)) = core::equal() :- A=B, !.
    cmp(_, _) = core::less().

    op("a", "b" ). op("b", "m"). op("m", "f").
```

```
op("f", "q"). op("q", "p"). op("p", "n").
op("p", "s"). op("b", "c"). op("c", "d").
op("d", "q"). op("d", "n"). op("d", "g").
op("n", "h"). op("n", "s"). op("h", "g").

init("a").

goalReached(t(_,_,[r(_,M)|_])):- theGoal(R), R=M.

theGoal("s").

not_in_circle(Stt, Path):-not(list::isMember(r("", Stt), Path)).

xy("a", 2, 4). xy("b", 5, 6).
xy("c", 4, 2). xy("d", 7, 4).
xy("f", 7, 8). xy("g", 8, 2).
xy("h", 10, 1). xy("m", 9,6).
xy("n", 11, 3). xy("p", 12, 6).
xy("q", 11, 7). xy("s", 13, 2).

getXY(M, X, Y) :- xy(M, X, Y), !.

cost(No, NoFilho) = C:-
    if getXY(No, XN, YN), getXY(NoFilho, XF, YF)
    then
        C = math::sqrt(((XN-XF)*(XN-XF)) + ((YN - YF)*(YN - YF)))
    else
        C= 0.0
    end if.

hn(N) = HN :- theGoal(S),
    if getXY(S, XS, YS), getXY(N, XN, YN)
    then HN= math::sqrt(((XN-XS)*(XN-XS)) + ((YN - YS)*(YN - YS)))
    else HN= 0.0 end if.

search([T|Queue],S):-
    if goalReached(T) then S= T
    else Extension= [E || toDaughter(T,E)],
        NewQueue= list::append(Queue,Extension),
        BestFirst= list::sortBy(cmp, NewQueue),
        search(BestFirst, S)
    end if.

toDaughter(t(_F,G,[r(B,N)|Path]),t(F1,G1,
                    [r(Op, Child),r(B,N)|Path])):-
    op(N, Child),
    Op= string::format("%s to %s", N, Child),
    not_in_circle(Child, Path),
    G1 = G + cost(N, Child), F1 = G1 + hn(Child).

prtSolution( t(_,_,T)):-
    foreach X= list::getMember_nd(list::reverse(T)) do
```

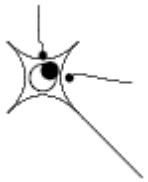
```
            stdio::write(X), stdio::nl
        end foreach.

    solve():- if init(E), search([t(hn(E),0,[r("root",E)])],S)
        then prtSolution(S)
        else stdio::write("No solution") end if.

    run():- console::init(), solve().
end implement main /*heureka*/
goal mainExe::run(main::run).
```

Глава 24: Нейронные сети

Испания не отличается своими научными достижениями. Действительно, во Франции существуют семьи со вдвое большим количеством нобелевских лауреатов, чем во всей Испании. В Европе не много стран с меньшим количеством нобелевских лауреатов, чем в Испании. Одной из них является Сербия, но одним из сербов, не получивших нобелевскую премию, был Никола Тесла (*Nicola Tesla*). А страна, у которой есть Никола Тесла, не нуждается в нобелевских лауреатах. Несмотря на это, единственные два испанских нобелевских лауреата находятся среди величайших учёных всех времён, вместе с Ньютоном и Архимедом. В этой главе вы узнаете об одном из них.



Испанский ученый дон Сантьяго Рамон-и-Кахаль (*Santiago Ramón y Cajal*) изучал физиологию нервной системы. Он заключил, что она работает благодаря клеткам, которые он назвал нейронами. В наши дни учёные считают, что заданный нейрон принимает входные сигналы от других нейронов через связи, называемые синапсами. Кстати говоря, синапс — это греческое слово, означающее **связь**.

Обычно нейрон собирает сигналы от остальных нейронов через сеть структур, которые называются дендритами, что означает **ветви дерева** на греческом. Если интенсивность входных сигналов превосходит определённый порог, нейрон запускается и посылает электрические раздражители через ствол, известный как аксон, который разделяется на древовидные дендриты, синапсы которых активизируют другие нейроны. Итак, подводя итоги… на конце каждого дендрита синапс преобразует деятельность нейрона в электрические раздражители, которые возбуждают или затормаживают другой нейрон.

В 1947 году Уоррен Маккаллок (*Warren McCullogh*) и Вальтер Питтс (*Walter Pitts*) предложили первую математическую модель нейрона. Согласно этой модели, нейрон является бинарным устройством с фиксированным порогом. Он получает множество входов от возбуждающих синапсов. Каждый входной источник имеет некоторый вес, являющийся положительным целым числом. Тормозящие входы имеют абсолютное право вето над любыми возбуждающими входами. На каждом шаге обработки нейроны синхронно обновляются посредством суммирования взвешенных возбуждающих входов, при этом выход полагается равным 1 тогда и только тогда, когда сумма больше или равна порогового значения (в противном случае выход равен 0 – *ред. пер*).

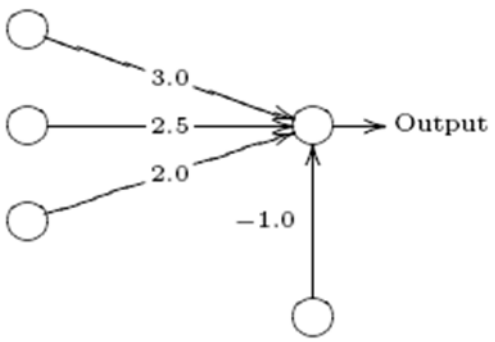
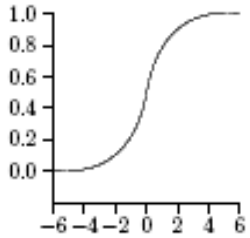


Рисунок 24.1. Перцептрон с тремя входами

Следующим значительным продвижением в теории нейросетей был перцептрон, предложенный Франком Розенблаттом (*Frank Rosenblatt*) в статье 1958 года. Розенблатт был профессором Корнельского университета, моей альма-матер. Однако, мне не посчастливилось быть знакомым с ним, так как он скончался за много лет до того, как я пришёл в этот университет.

Перцептрон складывает входные сигналы, помноженные на вес (для каждого входного сигнала свой вес), и, если результат больше порогового значения, он производит ненулевой выход.



Этот метод был улучшен следующим образом: сумма взвешенных входов подается на вход в сигмоидную функцию, а выход персептрона полагается равным полученному значению этой функции. Если вы начертите график сигмоида, он примет форму греческой буквы ς, которая известна как конечная сигма (*final sigma*), отсюда и название сигмоид. Сигмоид определяется так:

$$\varsigma(x) = \frac{1}{1+e^{-x}}$$

В Прологе это определение принимает следующий вид:

```
sigmoid(X)= 1.0/(1.0 + math::exp(-X)).
```

На рисунке 24.1, приведенном выше, изображен перцептрон с тремя входами. Обозначим входы через x_1, x_2 и x_3. В данном случае выход будет равен результату выражения

$$\varsigma(3.0x_1 + 2.5x_2 + 2.0x_3 - 1).$$

Одним из интересных свойств перцептрона является его способность к обучению: вы можете обучить перцептрон распознавать входные образы. На самом деле, существует алгоритм, который модифицирует веса перцептрона так, что тот сможет различать два и более образов, если имеется достаточное количество примеров. Например, перцептрон с двумя входами можно научить определять, когда конъюнктор (and-gate) выдаст 1, а когда 0.

Алгоритм обучения перцептрона должен иметь набор обучающих примеров. Предположим, что перцептрон изучает работу конъюнктора. Вот возможные примеры:

```
facts
    eg:(unsigned, real, real, real).
clauses
    eg(0, 1, 1, 1).
    eg(1, 1, 0, 0).
    eg(2, 0, 1, 0).
    eg(3, 0, 0, 0).
```

Перцептрон обладает процедурой предсказывания, которая использует формулу $\varsigma(\sum x_i w_i)$ для предсказания выходного значения по заданному множеству входов. В Прологе процедура предсказывания может иметь следующий вид:

```
predicted_out(E, V) :-
    eg(E, I_1, I_2, _),
    getWgt(0, W_0),
    getWgt(1, W_1),
    getWgt(2, W_2),
    X = W_0+W_1* I_1 + W_2* I_2,
    V= sigmoid(X).
```

Веса хранятся в базе данных.

```
facts
    weight:(unsigned, real).
clauses
    weight(0, 1.5). weight(1, 2.0). weight(2, 1.8).
```

Начальные веса являются фиктивными. Настоящие значения весов должны быть найдены обучающим алгоритмом. Процедура `getWgt` имеет очень простое определение:

```
getWgt(I, W) :- weight(I, W), !.
getWgt(_I, 0.0).
```

Вы можете спросить, зачем вообще нужно определять `getWgt`. Было бы легче напрямую использовать `weight`. Проблема заключается в том, что `weight` определяет недетерминированный предикат, а нам требуется процедурный предикат для извлечения веса. Кстати говоря, ещё нужна и процедура для извлечения примеров:

```
egratia(Eg, I1, I2, Output) :-
    eg(Eg, I1, I2, Output), !.
egratia(Eg, I1, I2, Output) :-
    Ex = Eg rem 4,
    eg(Ex, I1, I2, Output), !.
egratia(_Eg, 1, 1, 0).
```

Алгоритм обучения основан на вычислении и исправлении ошибки, сделанной предсказывающей функцией. Ошибку для заданного примера вычисляет следующий предикат:

```
evalError(Obj, E) :- nn:predicted_out(Obj, VC),
    nn:egratia(Obj, _I1, _I2, V),
    E= (VC-V)*(VC-V).
```

В действительности нам нужна не ошибка данного примера, а сумма ошибок по всем примерам. Общую ошибку можно вычислить так, как показано ниже.

```
errSum(Exs, _Partial_err, Total_err) :- acc := 0.0,
    foreach Eg= list::getMember_nd(Exs) do
        evalError(Eg, Err), acc := acc + Err
    end foreach,
    Total_err= acc.
```

Схема вычислений `foreach` выбирает каждый пример из списка `Exs`, вычисляет соответствующую ошибку и прибавляет результат вычисления к накопителю `acc`, который хранится в глобальной переменной.

```
class facts
    acc:real := 0.0.
```

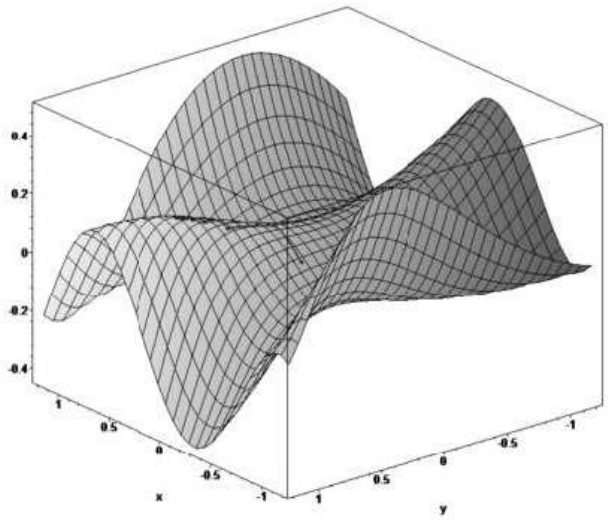
Веса будут обновляться по методу градиентного спуска, таким образом, чтобы свести ошибку до минимума.

```
updateWeight(DX, LC, Exs) :- errSum(Exs, 0.0, Err0),
    PN= [tuple(I, W) || nn:getWgt_nd(I, W)],
    nn:setWeight([ tuple(P, NV) ||
        tuple(P, V) = list::getMember_nd(PN),
        V1= V+DX,
        nn:assertWgt(P, V1),
        errSum(Exs, 0.0, Nerr),
        nn:popweight(P, V1),
        NV= V+LC*(Err0-Nerr)/DX]).
```

Градиентный спуск — это алгоритм оптимизации, который находит локальный минимум функции с помощью шагов, пропорциональных значению градиента функции в заданной точке, взятому со знаком минус. На каждом шаге метода градиентного спуска веса обновляются по следующей формуле:

$$\omega_{n+1} = \omega_n - \gamma \nabla error(\omega).$$

Если γ — достаточно маленькое положительное число, то $error(\omega_{n+1}) < error(\omega_n)$. Если начать с ω_0, то последовательность ω_0, ω_1, ω_2, … будет сходиться к минимуму. Минимум будет достигнут тогда, когда градиент станет равным нулю (или близким к нулю, в практических ситуациях). Вы можете представлять себе это как ландшафт с горами и долинами. Местоположение в этом ландшафте задаётся координатами ω_1 и ω_2. Высота гор и глубина долин изображает ошибки. Обучающий алгоритм перемещает сеть с исходного положения в долину, то есть в позицию, ошибка в которой достигает локального минимума.



Ситуация является идеальной, когда сеть достигает глобального минимума. Американцы, которые служат в контрразведке, любят говорить, что они лучшие из лучших. Это и есть то, что требуется для решения любой задачи оптимизации: лучшее из лучших. Несмотря на это, градиентные методы чаще всего попадают в ловушку локального минимума. Это происходит потому, что когда сеть достигает минимума, не имеет значения, локальный это минимум или глобальный, его градиент становится близким к нулю. В самом деле, алгоритм узнает, что он достиг минимума, потому что градиент достигает значения, близкого нулю. Проблема состоит в том, что этот критерий не говорит о том, глобальный это минимум или локальный. В словаре говорится, что *градиент* определяет наклон дороги или железнодорожных путей. На вершине горы или на дне долины градиент равен нулю. Если использовать *уровень*[1] на вершине горы или на дне долины, никакого наклона зафиксировано не будет.

Алгоритм обучения нейросети обновляет веса нейрона до тех пор, пока ошибка не будет оставаться в некоторой маленькой окрестности 0, затем он остановится.

[1] Инструмент для измерения степени наклона поверхности.

```
train(DX, LC, Exs) :- errSum(Exs, 0, Err0),
    if (Err0 < 0.1) then
        stdio::write("Total Err: ", Err0), stdio::nl
    else
        updateWeight(DX, LC, Exs),
        train(DX, LC, Exs)
    end if.
```

24.1. Описание нейрона

Нейрон, приведенный на рисунке 24.2, можно описать с помощью следующего уравнения:

$$\omega_1 x_1 + \omega_2 x_2 + \omega_0 = 0.$$

Это уравнение прямой на плоскости. Если использовать больше сущностей, получатся прямые линии в пространствах высших размерностей, но они все равно останутся прямыми линиями (гиперплоскостями — *ред. пер.*). Построим график логической функции, известной как дизъюнктор (or-gate). Она имеет два входа, x_1 и x_2.

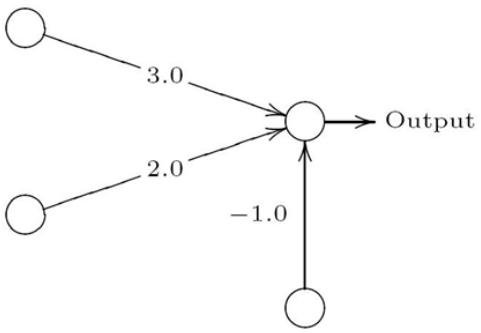
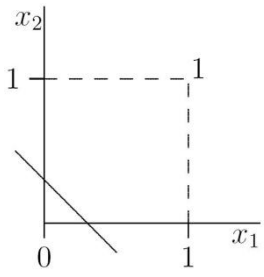


Рисунок 24.2. Перцептрон с тремя входами



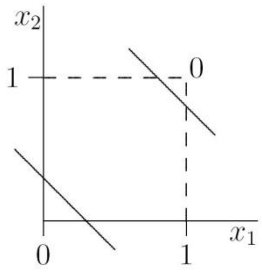
Функция дизъюнктора возвращает 1, если $x_1 = 1$ или $x_2 = 1$; она также возвращает 1, если как $x_1 = 1$, так и $x_2 = 1$. График, приведенный на рисунке, демонстрирует поведение дизъюнктора. Все, что делает изображённая на рисунке 24.2 нейросеть, – это рисует прямые линии, отделяющие точки в которых дизъюнктор равен 1, от точек, в которых он равен 0, как вы можете видеть из рисунка. Значит всё, что перцептрон может сделать, это выучить, как рисовать прямые линии в пространстве состояний, для того чтобы выделять кластеры точек с заданным свойством. Проблема состоит в том, что часто нельзя нарисовать подобные прямые линии в многомерном пространстве.

Функция исключающего ИЛИ (xor-gate) намного более полезна, чем дизъюнктор, по двум причинам. Во-первых, два входа схемы исключающего ИЛИ могут представлять логические высказывания вида:

x_1 = двигатель включён; x_2 = двигатель выключен
x_1 = дверь открыта; x_2 = дверь закрыта

Двигатель не может быть одновременно включен и выключен, поэтому $x_1 = 1$ *И* $x_2 = 1$ не может быть *истиной*, также как не может быть *истиной* $x_1 = 0$ *И* $x_2 = 0$. Для описания логических схем часто используют таблицы истинности, которые определяют выходные значения для всех значений входных переменных. Ниже вы можете увидеть таблицу истинности для схемы исключающего ИЛИ.

x_1	x_2	**выход**
1	1	0
1	0	1
0	1	1
0	0	0



Как видно по рисунку, одной прямой линии недостаточно, для того чтобы разделить значения функции исключающего ИЛИ. Однако две прямые линии могут отделить значения, равные нулю, от значений, равных единице, как вы можете видеть на рисунке. Точно так же, один перцептрон не сможет научиться вычислять выходные значения схемы исключающего ИЛИ, но сеть, содержащая большее количество перцептронов, как, например, сеть, изображённая на рисунке 24.3, может выполнить эту задачу. В следующих параграфах вы узнаете, как использовать объекты для реализации многослойной нейронной сети в Прологе.

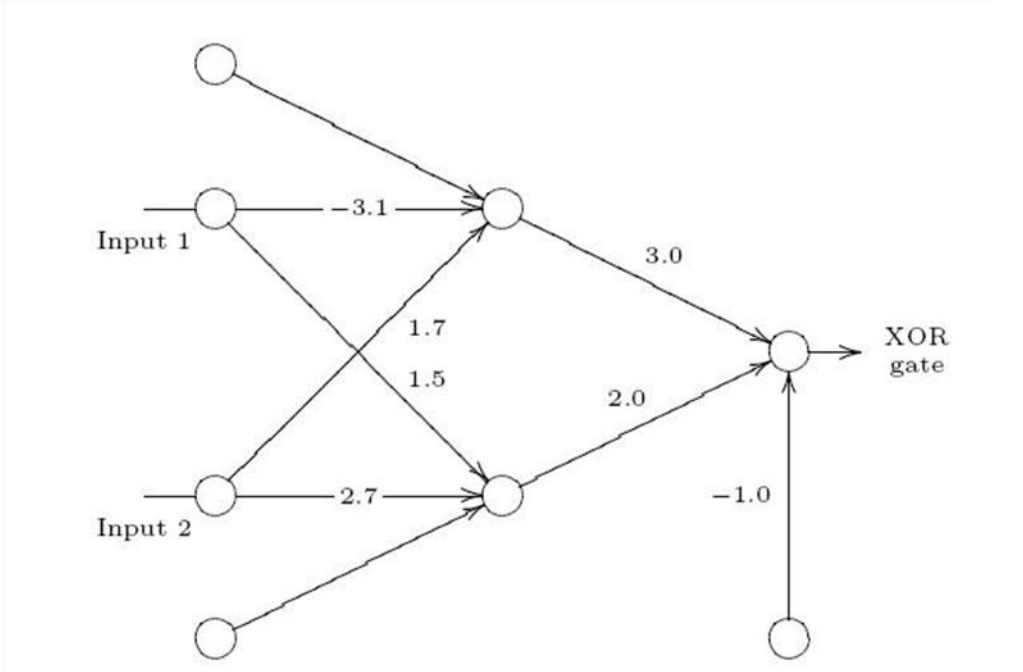


Рисунок 24.3. Нейросеть, способная научиться вычислять исключающее ИЛИ

24.2. Реализация многослойной сети

- Создайте новый консольный проект

```
Project Name: neurons
UI Strategy: console
```

- Выделите корень дерева проекта. Выберите пункт *File/New* главного меню среды. Откроется диалоговое окно *Create Project Item*. Выделите элемент *Class* в левом поле. Создайте класс `network`, который может создавать объекты. Нажмите кнопку *Create*. Добавьте спецификацию класса в файлы `network.cl`, `network.i` и `network.pro`.

```
% Файл network.cl
class network : network
predicates
    sigmoid:(real) -> real.
end class network
```

- Добавьте интерфейс класса в файл `network.i`:

```
% Файл network.i
interface network
    open core
domains
    example= e(real, real, real).
    examples= example*.
predicates
    setWeight:(tuple{unsigned, real}*) procedure.
    predicted_out:(unsigned, real) procedure (i, o).
    egratia:(unsigned, real, real, real) procedure (i, o, o, o).
    getWgt_nd:(unsigned, real) nondeterm (o, o).
    assertWgt:(unsigned I, real W) procedure (i, i).
    popweight:(unsigned I, real W) procedure (i, i).
    usit:( real*, real) procedure ( i, o).
    setExamples:(examples) procedure (i).
    getWgt:(unsigned, real) procedure (i, o).
    hidden:(unsigned, unsigned*, real) procedure (i, i, o).
    setEx:(unsigned, examples).
end interface network
```

Добавьте код, приведенный ниже, в файл `network.pro`. Постройте приложение.

```
% Файл network.pro
implement network
open core
constants
    className = "network/network".
    classVersion = "".
facts
    weight:(unsigned, real).
    eg:(unsigned, real, real, real).
clauses
    sigmoid(X)= 1.0/(1.0+math::exp(-X)).
```

```
getWgt_nd(I, R) :- weight(I, R).

assertWgt(I, R) :- asserta(weight(I, R)).

popweight(I, W) :- retract(weight(I, W)), !.
popweight(_, _).

setExamples(Es) :- retractall(eg(_,_,_,_)),
    setEx(0, Es).

setEx(_N, []) :- !.
setEx(N, [e(I1, I2,R)|Es]) :-
    assertz(eg(N, I1, I2, R)),
    setEx(N+1, Es).

egratia(Eg, I1, I2, Output) :- eg(Eg, I1, I2, Output), !.
egratia(Eg, I1, I2, Output) :- Ex = Eg rem 4,
    eg(Ex, I1, I2, Output), !.
egratia(_Eg, 1, 1, 0).

setWeight(Qs) :-
    retractall(weight(_,_)),
    foreach tuple(X, WX)= list::getMember_nd(Qs) do
        assert(weight(X, WX))
    end foreach.

getWgt(I, W) :- weight(I, W), !.
getWgt(_I, 0.0).

predicted_out(Obj, V) :-
    hidden(Obj, [3,4,5], I_1),
    hidden(Obj, [6,7,8], I_2),
    getWgt(0, W_0),
    getWgt(1, W_1),
    getWgt(2, W_2),
    X = W_0+W_1* I_1 + W_2* I_2,
    V= sigmoid(X).

hidden(Obj,[A,B,C],V) :- eg(Obj, I_1, I_2, _), !,
    getWgt(A, WA),
    getWgt(B, WB),
    getWgt(C, WC),
    XX = WA+WB* I_1+WC* I_2,
    V=sigmoid(XX).
hidden(_, _, 0).

usit( [I1, I2], V) :-
    getWgt(0, W_0), getWgt(1, W_1), getWgt(2, W_2),
    getWgt(3, W_3), getWgt(4, W_4), getWgt(5, W_5),
    getWgt(6, W_6), getWgt(7, W_7), getWgt(8, W_8), !,
    X1 = W_3 + W_4*I1 + W_5*I2,
    V1=sigmoid(X1),
```

```
        X2 = W_6 + W_7*I1 + W_8*I2,
        V2= sigmoid(X2),
        VX = W_0 + W_1*V1 + W_2*V2,
        V= sigmoid(VX).
    usit( _, 0).
end implement network
```

Объекты класса `network` могут играть роль нейронов. В главном модуле вы найдёте обучающую программу, а также тестовый предикат `run()` для конъюнктора и схемы исключающего ИЛИ. Постройте приложение и используйте команду меню *Run in Window* для запуска.

```
% Файл main.pro
implement main
open core
constants
    className = "main".
    classVersion = "nnxor".
clauses
    classInfo(className, classVersion).

class facts
    acc:real := 0.0.
    nn:network := erroneous.
class predicates
    evalError:(unsigned Obj, real Err) procedure (i, o).
    errSum:(unsigned* Exs, real Partial_err, real Total_err)
        procedure (i, i, o).
    updateWeight:(real DX, real LC, unsigned* Exs) procedure (i, i, i).
clauses
    evalError(Obj, E) :- nn:predicted_out(Obj, VC),
        nn:egratia(Obj, _I1, _I2, V),
        E= (VC-V)*(VC-V).

    errSum(Exs, _Partial_err, Total_err) :- acc := 0.0,
        foreach Eg= list::getMember_nd(Exs) do
            evalError(Eg, Err),
            acc := acc + Err
        end foreach,
        Total_err= acc.

    updateWeight(DX, LC, Exs) :- errSum(Exs, 0.0, Err0),
        PN= [tuple(I, W) || nn:getWgt_nd(I, W)],
        nn:setWeight([tuple(P, NV) ||
            tuple(P, V)= list::getMember_nd(PN),
            V1= V+DX,
            nn:assertWgt(P, V1),
            errSum(Exs, 0.0, Nerr),
            nn:popweight(P, V1),
            NV= V+LC*(Err0-Nerr)/DX]).
```

```
class predicates
    train:(real DX, real LC, unsigned* Exs) procedure (i, i, i).
clauses
    train(DX, LC, Exs) :- errSum(Exs, 0, Err0),
        if (Err0 < 0.1) then
            stdio::write("Total Err: ", Err0), stdio::nl
        else
            updateWeight(DX, LC, Exs),
            train(DX, LC, Exs)
        end if.

class predicates
    training:(network, real, real) procedure (i, i, i).
clauses
    training(NN, _DX, _LC) :- nn := NN,
        WWW= [tuple(0,0),tuple(1,2),tuple(2,0),
        tuple(3,0),tuple(4,1),tuple(5,0),
        tuple(6,0),tuple(7,0),tuple(8,0)],
        nn:setWeight(WWW),
        train(0.001,0.1,[0,1,2,3, 1, 2, 3, 0, 3, 2, 1, 0]).

    run():- console::init(),
        XOR = network::new(), AND = network::new(),
        XOR:setExamples([ network::e(1,1,0), network::e(1,0,1),
            network::e(0,1,1), network::e(0,0,0)]),
        AND:setExamples([ network::e(1,1,1), network::e(1,0,0),
            network::e(0,1,0), network::e(0,0,0)]),
        training(XOR, 0.001, 0.5),
        training(AND, 0.001, 0.5),
        XOR:usit( [1,1], XOR11), stdio::nl,
        stdio::write("xor(1,1)=", XOR11), stdio::nl,
        XOR:usit( [1,0], XOR10), stdio::nl,
        stdio::write("xor(1,0)=", XOR10), stdio::nl,
        XOR:usit( [0,1], XOR01), stdio::nl,
        stdio::write("xor(0,1)=", XOR01), stdio::nl,
        XOR:usit( [0,0], XOR00), stdio::nl,
        stdio::write("xor(0,0)=", XOR00), stdio::nl, stdio::nl,
        AND:usit( [1,1], AND11), stdio::nl,
        stdio::write("1&&1=", AND11), stdio::nl,
        AND:usit( [1,0], AND10), stdio::nl,
        stdio::write("1&&0=", AND10), stdio::nl,
        AND:usit( [0,1], AND01), stdio::nl,
        stdio::write("0&&1=", AND01), stdio::nl,
        AND:usit( [0,0], AND00), stdio::nl,
        stdio::write("0&&0=", AND00), stdio::nl.
end implement main
goal
    mainExe::run(main::run).
```

24.3. Запуск двуслойной нейросети

После компоновки приложения `neurons` и его запуска вы получите следующий результат:

```
Total Err: 0.09983168220091619
 Total Err: 0.09968265082000113

xor(1,1)=0.08040475166406771

xor(1,0)=0.9194034557462105

xor(0,1)=0.8913291572885467

xor(0,0)=0.0922342035736988

1&&1=0.8712088309977442

1&&0=0.08891005182518963

0&&1=0.09297532089700676

0&&0=0.009538209942478315

D:\vipro\aityros\aiprogs\neurons\Exe>pause
 Press any key to continue. . .
```

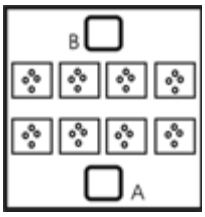
Как вы можете видеть, обучающий алгоритм очень приблизился к желаемому поведению исключающего ИЛИ.

24.4. Историческая перспектива

Существуют люди, которые заявляют, что, *когда Розенблатт представил на рассмотрение концепцию перцептрона, Минский (Minsky) очень грубо набросился на него на личном уровне, причем он связывался с каждой из групп, которые финансировали исследования Розенблатта, для того чтобы объявить о нем как о шарлатане, в надежде уничтожить Розенблатта как специалиста и полностью прекратить финансирование его исследований в области нейросетей*. Как я уже говорил, я не имел счастья знать Розенблатта, но я хорошо знаю Минского (я брал у него интервью, когда подрабатывал газетчиком), и я уверен, что подобные наговоры на Минского являются попросту неправдой. Что Минский и его соратник Паперт сделали — это указали на слабость теории о том, что интеллект может возникнуть из одной лишь обучающей деятельности нейронной сети. В действительности Минский на практике продемонстрировал теорему о том, что нейронная сеть не может научиться делать некоторые вещи. Это ограничение не относится исключительно к искусственным нейросетям. Существуют вещи, которым не могут научиться и наши естественные нейросети: на обложке книги Минского «Перцептроны» (*Perceptrons*) имеется изображение, похожее на лабиринт. Читатель должен определить, соединены перекрывающие друг друга стены лабиринта или нет. Минский утверждает, что обучить нейронную сеть выполнить эту задачу невозможно, и неважно, естественная это нейросеть или искусственная.

Глава 25: Альфа-бета отсечение

Манкала — это семейство настольных игр, которые очень популярны в Африке и Азии; это игра подсчёта и захвата. Самой известной на Западе игрой этого семейства является калах. Игры семейства манкала играют во многих африканских и азиатских обществах роль, сравнимую с ролью шахмат на Западе. Они также популярны в Индии, особенно среди женщин Тамила.



В варианте манкалы, который мы собираемся изучать, на доске имеется набор отверстий, расположенных в два ряда, по одному на каждого игрока. Отверстия называются *ямы*, *горшки* или *дома*. Два больших отверстия на концах доски, называемые *склады*, используются для хранения захваченных фишек. Игровые фишки — зерна, бобы или камешки — кладутся в отверстия и перемещаются между ними в течение игры. В начале своего хода игрок выбирает отверстие на своей стороне и высеивает из него зерна по доске.

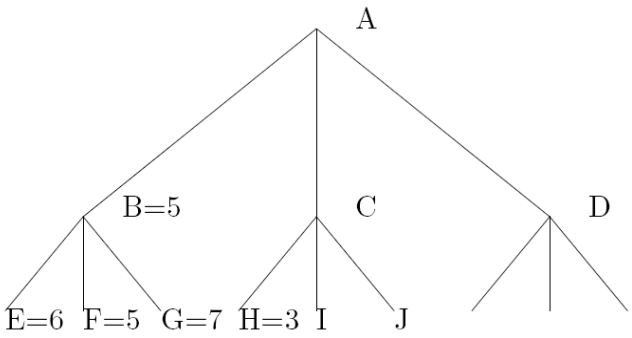
В процессе сеяния все зерна из отверстия опускаются один за другим в последующие отверстия при движении вокруг доски. В зависимости от содержимого каждого засеянного отверстия, игрок может захватить зерна с доски. Если отверстие содержит менее трех зерен, игрок может захватить его содержимое, также как и зерно, которое он собирался положить в это отверстие. Захваченные зерна отправляются в склад игрока. Игра заканчивается, если один игрок захватит как минимум на десять зерен больше, чем его оппонент. Игра также заканчивается, если на стороне одного из игроков не останется зерен. В любом случае выигрывает игрок, который захватит больше зерен.

25.1. Генерация хода

Самые любимые игры — это те, в которых один оппонент сражается с другим в битве умов. Однако я хотел бы отметить, что даже в таких играх Эраст Фандорин всегда выигрывал, как вы можете убедиться, прочитав «Пиковый валет». Но компьютеры не настолько удачливы, как статский советник. Поэтому программистам нужен инструмент для анализа игры. Этим инструментом является дерево минимакса (см. рис. 10.1). В такой диаграмме один игрок называется максимизатор, а другой — минимизатор. При анализе подразумевается, что максимизатор будет стремиться достичь максимального значения оценочной функции; отсюда и название. Минимизатор пытается свести оценочную функцию к минимальному значению. По правде говоря, дерево, представленное на рисунке 10.1, не используется. Для того чтобы понять почему, предположим, что максимизатор анализирует ход B на дереве, приведенном ниже. Он знает, что минимизатор сделает ход F, так как F — это контрход наименьшего веса. Таким образом он заключает, что B с его точки зрения стоит 5. Теперь он переходит к анализу C. Когда он обнаружит, что H=3, он может приостановить поиск, потому что ход в C будет стоить не более трех, а он уже имеет 5 при ходе в B. Процесс пропуска ветвей, в данном случае I и J, называется альфа-отсечением.

При рассуждениях с точки зрения минимизатора всё, что нужно сделать — это инвертировать знак оценочной функции и применить тот же алгоритм, который был использован для максимизатора. В этом случае пропуск ветвей называется бета-

отсечением. Наконец, если вершина является листом, то оценка производится с помощью статической функции, то есть функции, которая не спускается по ветвям.



Реализация игры со стратегией — это сложная задача. Для того чтобы довести это до конца, мы сохраним генерацию хода в отдельном классе, так чтобы сложность самой игры не пересекалась с нашими рассуждениями относительно стратегии. Создайте новый консольный проект.

```
Project Name: kalahGame
UI Strategy: console
```

Постройте приложение, для того чтобы вставить класс `main` в дерево проекта. Затем создайте класс `mv` с помощью пункта *New in New Package* меню среды.

```
% Файл mv.cl
class mv
    open core
domains
    side= a;b.
    board= none ; t(side, integer LastMove,
        integer Pot, integer Pot,
        integer* Brd).
predicates
    prtBoard:(board) procedure (i).
    newBoard:(side, board) procedure (i, o).
    addPot:(side, integer, integer, integer,
        integer, integer) procedure (i,i,i,i, o,o).
    changeSides:(side, side) procedure (i, o).
    sowing:(side, integer, integer, positive,
        integer, integer, integer*,
        board) procedure (i, i, i, i, i, i, i, o).
    sow:(integer, board, board) procedure (i,i,o).
    play:(board, board) nondeterm (i, o).
    stateval:(board, integer) procedure (i, o).
end class mv

% Файл mv.pro
implement mv
    open core
clauses
    prtBoard(t(S, LM, P1, P2, [I0, I1, I2, I3, I4, I5, I6, I7])) :- !,
```

```
    stdio::write("Move: ", LM), stdio::nl,
    if S= a then T= "a" else T= "b" end if,
    stdio::writef(" %4d %4d %4d %4d\n", I3, I2, I1,I0),
    stdio::writef("%4d %4d %s\n", P1, P2, T),
    stdio::writef(" %4d %4d %4d %4d\n", I4, I5, I6, I7).
prtBoard(_) :- stdio::write("None\n").

newBoard(Side, t(Side, -1, 0, 0, [6,6,6,6, 6,6,6,6])).

addPot(a, C, P1, P2, P1+C+1, P2) :- !.
addPot(b, C, P1, P2, P1, P2+C+1).

changeSides(a, b) :- !.
changeSides(b, a).

sow(I, t(Side, _LastMove, P1, P2, B), NewBoard) :-
    J= I rem 8,
    Counts= list::nth(J, B), !,
    list::setNth(J, B, 0, BA),
    K= (J+1) rem 8,
    sowing(Side, J, Counts, K, P1, P2, BA, NewBoard).
sow(_, T, T).

sowing(S, LM, N, K, P1, P2, BA, NuB) :- N>0,
    C= list::nth(K, BA),
    C > 0, C < 3, !,
    list::setNth(K, BA, 0, BB),
    J= (K+1) rem 8,
    addPot(S, C, P1, P2, PP1, PP2),
    sowing(S, LM, N-1, J, PP1, PP2, BB, NuB).
sowing(S, LM, N, K, P1, P2, BA, NuB) :- N > 0, !,
    list::setNth(K, BA, list::nth(K, BA) + 1, BB),
    J= (K+1) rem 8,
    sowing(S, LM, N-1, J, P1, P2, BB, NuB).
sowing(S, LM, _, _, P1, P2, BB, t(S1, LM, P1, P2, BB)) :-

changeSides(S, S1).

stateval(t(_, _, P1, P2, _), P2-P1) :- !.
stateval(_, 0).

play(t(a, LM, P1, P2, L), X) :- list::nth(0, L) > 0,
    sow(0, t(a, LM, P1, P2, L), X).
play(t(a, LM, P1, P2, L), X) :- list::nth(1, L) > 0,
    sow(1, t(a, LM, P1, P2, L), X).
play(t(a, LM, P1, P2, L), X) :- list::nth(2, L) > 0,
    sow(2, t(a, LM, P1, P2, L), X).
play(t(a, LM, P1, P2, L), X) :- list::nth(3, L) > 0,
    sow(3, t(a, LM, P1, P2, L), X).
play(t(b, LM, P1, P2, L), X) :- list::nth(4, L) > 0,
    sow(4, t(b, LM, P1, P2, L), X).
play(t(b, LM, P1, P2, L), X) :- list::nth(5, L) > 0,
```

```
        sow(5, t(b, LM, P1, P2, L), X).
    play(t(b, LM, P1, P2, L), X) :- list::nth(6, L) > 0,
        sow(6, t(b, LM, P1, P2, L), X).
    play(t(b, LM, P1, P2, L), X) :- list::nth(7, L) > 0,
        sow(7, t(b, LM, P1, P2, L), X).
end implement mv
```

Теперь вы готовы реализовать основной класс, который будет искать в дереве игры наилучший ход.

```
implement main /* kalahGame */
    open mv
clauses
    classInfo("kalahGame", "1.0").
class predicates
    maxeval:( mv::board, integer, integer,
        integer, mv::board, integer) procedure (i, i, i, i, o, o).
    mineval:( mv::board, integer, integer,
        integer, mv::board, integer) procedure (i, i, i, i, o, o).
    bestMax:( mv::board*, integer, integer, integer,
        mv::board, integer) determ (i, i, i, i, o, o).
    bestMin:( mv::board*, integer, integer, integer,
        mv::board, integer) determ (i, i, i, i, o, o).
    betaPruning:( mv::board*, integer, integer, integer,
        mv::board, integer, mv::board, integer)
        procedure (i, i, i, i, i, i, o, o).
    alphaPruning:( mv::board*, integer, integer, integer,
        mv::board, integer, mv::board, integer)
        procedure (i, i, i, i, i, i, o, o).
clauses
    maxeval(Pos, LookAhead, Alpha, Beta, Mov, Val) :-
        LookAhead > 0,
        Lcs = [P || play(Pos, P)],
        bestMax(Lcs, LookAhead - 1, Alpha, Beta, Mov, Val), ! ;
        stateval(Pos, Val), Mov= none.

    bestMax([Lnc|Lncs], LookAhead, Alpha, Beta, BestMove, BestVal) :-
        !,
        mineval(Lnc, LookAhead, Alpha, Beta, _, Val),
        betaPruning(Lncs, LookAhead, Alpha, Beta, Lnc, Val, BestMove, B
estVal).

    betaPruning([], _, _, _, Lnc, Val, Lnc, Val) :- !.
    betaPruning(_, _, _Alpha, Beta, Lnc, Val, Lnc, Val) :-
        Val > Beta, !.
    betaPruning(Lncs, LookAhead, Alpha, Beta, Lnc, Val, MLnc, MVal) :-
        if Val > Alpha then NewAlpha= Val
        else NewAlpha= Alpha end if,
        if bestMax(Lncs, LookAhead, NewAlpha, Beta, Lnc1, Val1),
            Val1 > Val then MLnc= Lnc1, MVal= Val1
        else MLnc= Lnc, MVal= Val end if.
```

```
    mineval(Pos, LookAhead, Alpha, Beta, Mov, Val) :-
        LookAhead > 0,
        Lcs = [P || play(Pos, P)],
        bestMin(Lcs, LookAhead - 1, Alpha, Beta, Mov, Val), ! ;
        stateval(Pos, Val), Mov= none.

    bestMin([Lnc|Lncs], LookAhead, Alpha, Beta, BestMove, BestVal) :-
        !,
        maxeval(Lnc, LookAhead, Alpha, Beta, _, Val),
        alphaPruning(Lncs, LookAhead, Alpha, Beta, Lnc, Val, BestMove,
BestVal).

    alphaPruning([], _, _, _, Lnc, Val, Lnc, Val) :- !.
    alphaPruning(_, _, Alpha, _Beta, Lnc, Val, Lnc, Val) :-
 Val < Alpha, !.
    alphaPruning(Lncs, LookAhead, Alpha, Beta, Lnc, Val, MLnc, MVal) :-
        if Val < Beta then NewBeta= Val
        else NewBeta= Beta end if,
        if bestMin(Lncs, LookAhead, Alpha, NewBeta, Lnc1, Val1),
            Val1 < Val then MLnc= Lnc1, MVal= Val1
        else MLnc= Lnc, MVal= Val end if.

class predicates
    readMove:(board, board) procedure (i, o).
    legalMove:(core::positive, board) determ.
    game:(board, integer) procedure (i, i).
    endGame:(board) determ (i).
    announceWinner:(integer) procedure (i).

clauses
    legalMove(I, t(a, _LM, _P1, _P2, X)) :-
        I >= 0, I < 4,
        list::nth(I, X) > 0.

    readMove(Pos, ReadPos) :-
        stdio::write("Present situation: "), stdio::nl,
        prtBoard(Pos),
        %stdio::write("Your move: "),
        SR= stdio::readLine(),
        trap(I1= toTerm(SR), _, fail),
        stdio::nl,
        legalMove(I1, Pos),
        sow(I1, Pos, ReadPos),
        prtBoard(ReadPos), !.
    readMove(Pos, ReadPos) :-
        stdio::write("Something failed"), stdio::nl,
        readMove(Pos, ReadPos).

    endGame(X) :- stateval(X, V), math::abs(V) > 10, !.
    endGame(t(a, _LM, _P1, _P2, [I0, I1, I2, I3|_])) :-
        I0 < 1, I1 < 1, I2 < 1, I3 < 1, !.
    endGame(t(b, _LM, _P1, _P2, [_I0, _I1, _I2, _I3, I4, I5, I6, I7]))
```

```
    :-
            I4 < 1, I5 < 1, I6 < 1, I7 < 1, !.

        game(X, _Val) :- endGame(X), !, stateval(X, V),
            prtBoard(X),
            announceWinner(V).
        game(none, Val) :- stdio::write("End game: ", Val),
            announceWinner(Val), !.
            game(Pos, _Val) :-readMove(Pos, NewPos),
            maxeval(NewPos, 10, -10, 10, BestMove, BestVal), !,
            game(BestMove, BestVal).

        announceWinner(Val) :- Val > 0,
            stdio::write("\n I won!\n", Val), !.
        announceWinner(Val) :- Val < 0,
            stdio::write("\n You won!\n", Val), !.
        announceWinner(_Val) :- stdio::write("\n It is a draw!\n").
            run():- console::init(),
            newboard(a, B),
            stateval(B, Val),
            game(B, Val).
    end implement main /* kalahGame */

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

Как и Эраст Фандорин, я не люблю игры. Кроме того, я не обладаю его удачей, что означает, что я всегда проигрываю. Однако я всё же поставил свой ум против компьютера и, разумеется, проиграл. Вот начало игры:

```
 Present situation:
 Move: -1
        6   6   6   6
    0                   0   a
        6   6   6   6
0

Move: 0
        7   7   7   0
    0                   0   b
        7   7   7   6

Present situation:
Move: 7
        8   8   8   1
    0                   0   a
        8   8   7   0
1

Move: 1
        9   9   1   0
    2                   0   b
        9   9   8   1

Present situation:
Move: 4
```

```
    10  10  0   1
2                   4   a
    1   11  9   0
```

Библиография

[Abramson/Dahl] Harvey Abramson and Veronica Dahl, "Logic Grammars", Springer Verlag, 1989.

[Cedric et all] de Carvalho, C. L., Costa Pereira, A. E. and da Silva Julia, R. M. "Data-flow Synthesis for Logic Programs". In: System Analysis Modeling Simulation, Vol 36, pp. 349-366, 1999.

[Warren] David S. Warren. "Computing With Logic: Logic Programming With Prolog". Addison-Wesley / January 1988.

[Lewis Caroll] Lewis Caroll. The Game of Logic. Hard Press, 2006.

[Sterling and Shapiro] Sterling and Shapiro. "The Art of Prolog". MIT Press / January 1986.

(имеется русский перевод: Стерлинг Л., Шапиро Э. «Искусство программирования на языке Пролог». М.: Мир, 1990.)

[Coelho/Cotta] Coelho, H. and Cotta, J. "Prolog by Example". (1988) Berlin: Springer-Verlag.

[HowToSolveItWithПролог] Helder Coelho, Carlos Cotta, Luis Moniz Pereira. "How To Solve It With Prolog". Ministerio do Equipamento Social. Laboratorio Nacional de Engenharia Civil.

[Pereira/Shieber] F.C.N. Pereira and S.M. Shieber. "Prolog and Natural-Language Analysis". CSLI Publications, 1987.

[Gazdar/Mellish] Gazdar, G., and C.S. Mellish. "Natural Language Processing in Prolog". Addison-Wesley.

[John Hudges] John Hughes. Why Functional Programming Matters. Computer Journal, vol. 32, number 2, pages 98-107, 1989.

Рисунок 0.2 Настройки проекта

[Wadler & Bird] Richard Bird and Philip Wadler. Introduction to Functional Programming. Prentice Hall (1988).

[John Backus] John Backus. Can Programming be Liberated from the Von Newman Style? Communications of the ACM, 21(8):613-641, August 1978.

[Gries] David Gries. The Science of programming – New York : Springer-Verlag, cop.1981.- X,366p.

[Dijkstra] E. Dijkstra, W. Feijen: A Method of Programming, Addison-Wesley Publishing Company.

[Cohn] Cohn, P. G. Logic Programming, Expressivity versus Efficiency. State University of Campinas. 1991.

[Yamaki] Kimie Yamaki. Combining partial evaluation and automatic control insertion to increase the efficiency of Prolog programs. State University of Campinas. 1990.

[Gentzen] Gerhard Karl Erich Gentzen. "Die Widerspruchsfreiheit der reinen Zahlentheorie",Mathematische Annalen, 112: 493-565.(1936)

[Robinson] Robinson, J. A. Theorem Proving on the Computer. Journal of the Association for Computing Machinery. Vol. 10, N. 12, pp 163-174.

[William Stearn] William T Stearn. Botanical Latin. Timber Press. Paperback edition, 2004.