

Visual Prolog for Tyros

Eduardo Costa

PREFACE

This book started as a personal project. My intention was simply to write a tutorial on Logic Programming for my son. The success of the book, however, was immense, and I received many suggestions on how to improve the text, or the code, and encouragement to keep on with the work. Mail came from Saudi Arabia, China, Russia, Spain, France, Brazil, Canada, and many other places. Although the space is short to list all people who wrote me about the book, I want to acknowledge the collaboration and enthusiasm of the following persons:

- Elena Efimova, who made many corrections on the text, and in the historical background of Mathematics and Logics.
- Mark Safronov, who translated the book to Russian, and made many corrections on the contents of the original. By the way, the Russian translation has a much better layout than the English original.
- Thomas W. de Boer, who prepared a parallel edition of the book, with longer explanations, and a text appropriate to rank beginners.
- Stuart Cumming. I often give examples that show what one should avoid from the point of software engineering. Stuart pointed out that people could be misled by these examples; he suggests that, whenever an implementation is not robust, I should emphasize its weak points. Although I agree with Stuart, I thought it better to *remove* the negative examples altogether.
- Rose Shapiro, who corrected my Latin, and my description of many historical events.
- Yuri Ilyin, who helped me with his expertise; without him, this book would not be written.
- Reny Cury, who went through the manuscript correcting typos.
- Philippos Apolinarius who helped me with his knowledge of Botanical Latin, and Chinese.
- Thomas Linder Puls and Elizabeth Safro, from PDC, for support and encouragement.

Contents

I	Savoir-faire	11
1	Introduction	13
1.1	Creating a project in VIP	13
1.1.1	Create a new GUI project: name	14
1.1.2	Compile and execute the program	15
1.2	Examples	16
1.3	Notions of Logics: Ancient Greeks	16
2	Forms	17
2.1	Create a form: folder/name	17
2.2	Enable the task menu: e.g. File/New option	17
2.3	In CodeExpert, add code to project tree item	18
2.4	Example	22
2.5	Notions of Logics: Aristotle's Syllogistic	22
2.5.1	Valid syllogisms	24
3	Mouse events	27
3.1	Add code to MouseDownListener	27
3.2	onPaint	28
3.3	Examples	30
3.4	Notions of Logics: Boolean algebra	30
3.5	Argument forms	31
4	Less Figures	33
4.1	Task menu	33
4.2	Project Tree	34
4.2.1	Code Expert	35
4.3	Create Project Item	36
4.4	Create a new class: folder/name	39
4.5	Edit field contents	39
4.6	Examples	40

4.7	Notions of Logics: Predicate calculus	40
5	Horn Clauses	41
5.1	Functions	41
5.2	Predicates	42
5.3	Solutions	42
5.4	Multi-solutions	46
5.4.1	A program that uses multi solution predicates	47
5.5	Logical junctors	48
5.6	Implication	49
5.7	Horn Clauses	49
5.8	Declarations	51
5.9	Drawing predicates	53
5.10	GDI Object	53
5.11	Examples	56
5.12	Notions of Logics: Meaning of Horn clauses	56
6	Console Applications	57
6.1	Cut	57
6.2	Lists	58
6.3	List schemes	64
6.4	String operations	68
	—Parsing into tokens: fronttoken	68
	—String concatenation: concatList	68
	—String concatenation: concat/2 and concat/3	68
	—Term to String conversion: toTerm	69
	—Term to String conversion: hasdomain	69
	—Term to String conversion: toString	69
	—Term to String conversion: format	70
6.4.1	Useful String Predicates	71
	—String processing: Useful predicates	71
6.5	Notions of Logics: Grammar for Predicates	76
7	Grammars	79
7.1	Parsing grammar	80
7.2	Generating grammar	81
7.3	Why Prolog?	83
7.4	Examples	84
7.5	Notions of Logics: Natural deduction	84

8	Painting	87
8.1	onPainting	88
8.2	Custom Control	91
8.3	Notions of Logics: Resolution principle	93
9	Data types	95
9.1	Primitive data types	95
9.2	Sets	97
9.3	Sets of numbers	97
9.4	Irrational Numbers	100
9.5	Real Numbers	102
9.6	Mathematics	102
9.7	Format	102
9.8	domains	104
9.8.1	Lists	104
9.8.2	Functors	104
9.9	Notions of Logics: Horn clauses	108
10	How to solve it in Prolog	109
10.1	Utilities	110
11	Facts	129
11.1	Class file	132
11.1.1	Reading and writing a string	132
11.2	Constants	133
12	Classes and Objects	135
12.1	Object facts	137
13	Giuseppe Peano	139
13.1	Turtle graphics	139
13.2	Turtle states	140
13.3	Recursion	142
13.4	Peano's curve	143
13.5	Latino Sine Flexione	143
13.6	Quotations from Key to Interlingua	144
13.7	Examples	149
14	L-Systems	151
14.1	Class draw	152
14.2	Examples	154

15 Games	155
15.1 Object Facts	162
16 Animation	163
16.1 dopaint	164
16.2 Handling the timer	166
16.3 How the program works	166
17 Text Editor	167
17.1 Saving and Loading files	168
18 Printing	171
19 Tab forms and other goodies	173
19.1 Famous programs	174
19.2 Botany	176
19.3 Handling Chinese	182
19.4 Regular expressions	186
19.5 A MIDI player	189
19.6 Midi format	191
20 Bugs	195
20.1 Type error	196
20.2 Non-procedure inside a procedure	197
20.3 Non-determ condition	198
20.4 Impossible to determine the type	198
20.5 Flow pattern	200
21 A Data Base Manager	201
21.1 Database manager	201
21.2 btr class	205
21.3 The database manager	207
22 Books and articles	209
22.1 Grammars	209
22.2 Databases	210
22.3 Programming Techniques	210
II Artificial Intelligence	213
23 Search	215

23.1 States	215
23.2 Search tree	216
23.3 Breadth First Strategy	218
23.4 Depth First Strategy	222
23.5 Heuristic search	223
24 Neural Networks	229
24.1 Neuron description	234
24.2 Implementation of a multi-layer network	235
24.3 Running the two layer neural network	240
24.4 A Historical Perspective	241
25 Alpha Beta Pruning	243
25.1 Move generation	244

List of Figures

1.1	Task menu: New Project	13
1.2	Project Settings	14
1.3	Project tree	14
1.4	Building the project	15
1.5	An empty application	15
2.1	Adding a new item to the Project Tree	18
2.2	Create a new form	19
2.3	Resizing the form window	19
2.4	Project Tree/Task Menu	20
2.5	Enabling the Task Menu	20
2.6	Going to the Code Expert	21
2.7	Dialog and Window Expert	21
2.8	Pop-up Form	22
2.9	Barbara	25
2.10	Darii	26
3.1	PaintResponder	29
3.2	Paint Responder	30
4.1	Create a new package	37
4.2	A form with an edit field	37
4.3	The fn class	40
5.1	Interesting predicates	43
5.2	mapDataBase/draw.cl	45
5.3	mapDataBase/draw.pro	46
5.4	draw.cl and draw.pro	50
5.5	Utah cities	51
5.6	Class draw	55
6.1	Reduce in Prolog	65

6.2	Zip in Prolog	67
7.1	Die Welt ist alles, was der Fall ist.	84
8.1	A form with a custom control	92
9.1	A thinking Greek	101
9.2	List declaration	105
10.1	Minmax tree	128
11.1	Reading a string from file	133
13.1	Files curve.cl and curve.pro	141
13.2	File turtle.cl	143
13.3	turtle.pro	144
13.4	Peano's curve	145
13.5	Inserting a file into an editor	149
13.6	A dictionary of "difficult" LsF words	150
15.1	Lawn form	156
15.2	draw.cl and draw.pro	160
15.3	objstate	161
16.1	click class handles the timer	164
16.2	dopaint.pro	165
17.1	The insert custom control button	168
17.2	Choose Custom Control dialog	168
17.3	A form with Edit Control	169
17.4	User control properties	169
17.5	Code for onSave	170
17.6	Code for onLoad	170
19.1	listEdict methods	181
19.2	File: ideograms.pro	184
19.3	Midi generator	194
20.1	A buggy program	197
20.2	A bug caused by a non deterministic predicate	199
20.3	Deterministic predicate required.	199
23.1	A search tree	216
23.2	Puzzle	217

24.1 A three input perceptron	230
24.2 A three input perceptron	234
24.3 A neural network that can learn how to calculate a xor-gate .	235

Part I

Savoir-faire

Chapter 1

Introduction

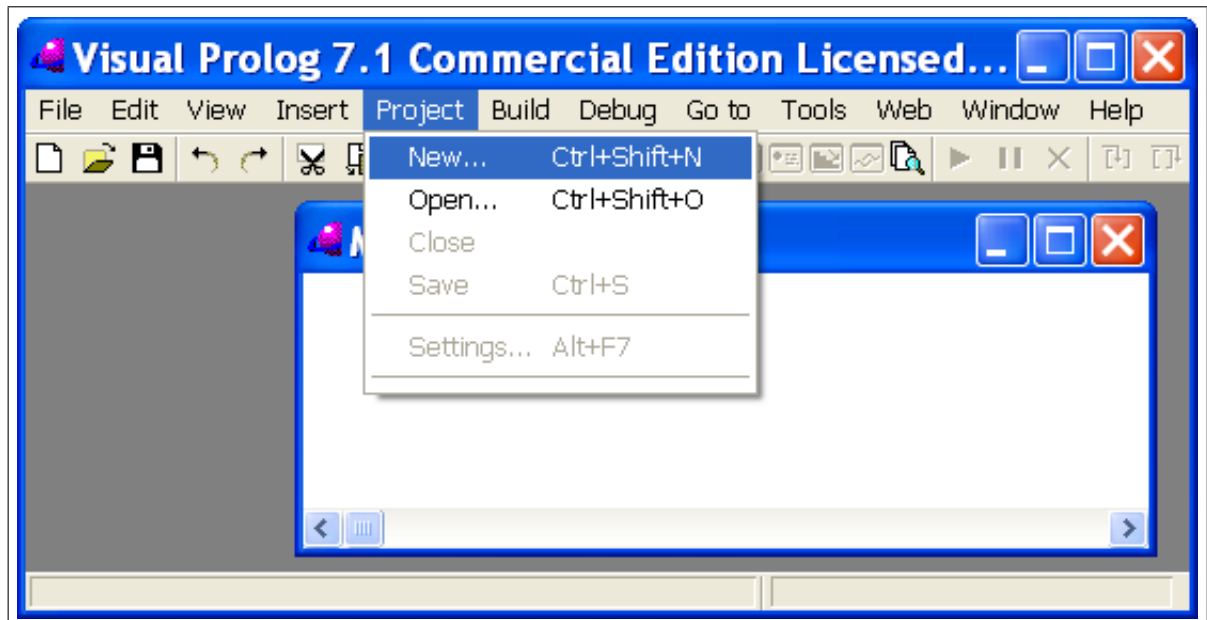


Figure 1.1: Task menu: New Project

1.1 Creating a project in VIP

Let us create an empty project, to which you will add functionality later on. The environment that you will use to develop your programs is called IDE, that is the acronym for Integrated Development Environment. When you enter the Visual Prolog IDE, you will get an environment like the one shown in figure 1.1; we will refer to the menu of the IDE as the task menu.

The system of windows and dialogs that you will create to communicate with potential users of your programs is called a Graphical User Interface, or GUI for short.

1.1.1 Create a new GUI project: name

This step is quite easy. Choose the option *Project/New* from the task menu, as shown in figure 1.1. Then, fill in the *Project Settings* dialog as in figure 1.2. Press the button *OK* and you will get the project tree (figure 1.3).

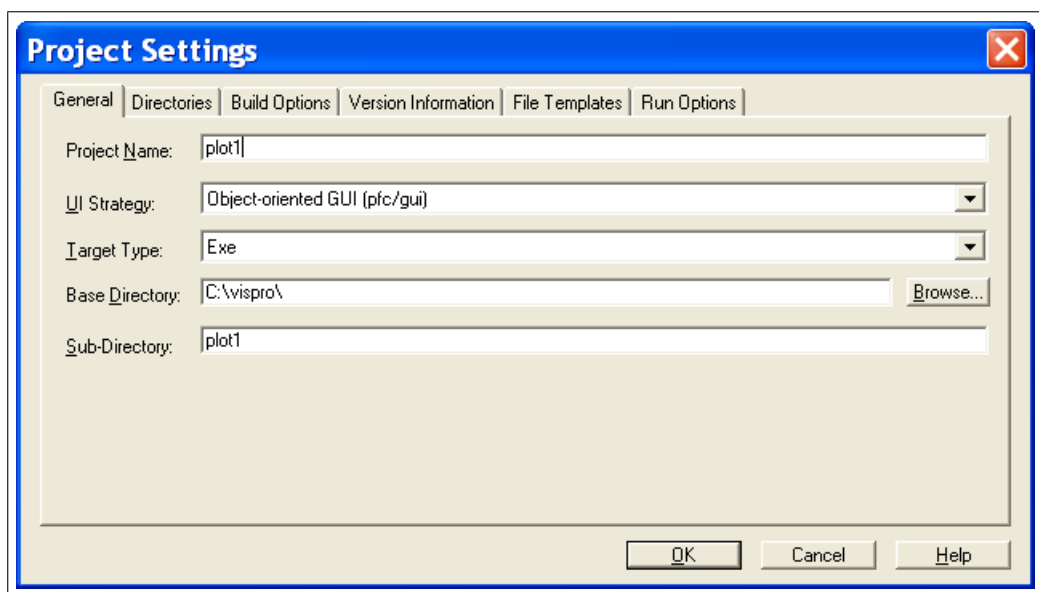


Figure 1.2: Project Settings



Figure 1.3: Project tree

1.1.2 Compile and execute the program

To compile the program, choose the option *Build/Build* from the task menu, as shown in figure 1.4. To execute it, choose *Build/Execute* (see figure 1.4) from the task menu, and a window like the one in figure 1.5 will pop up on the screen. In order to exit the program, all you have to do is to click on the X-shaped button that appears on the upper right hand corner of the Window frame; if you prefer, choose the option *File/Exit* option of the application task menu.

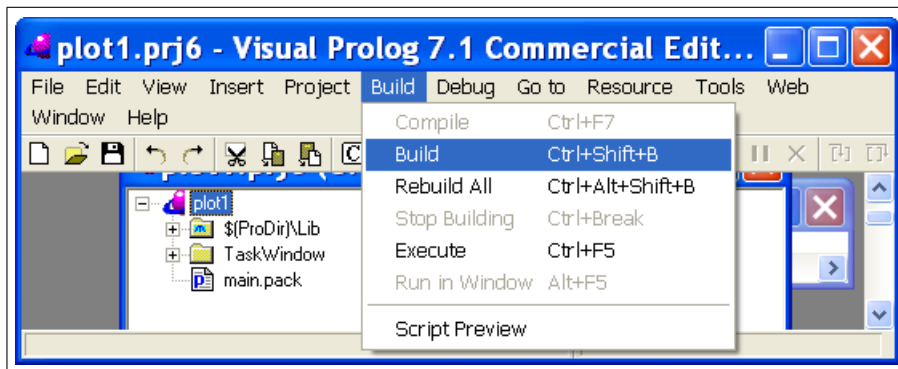


Figure 1.4: Building the project

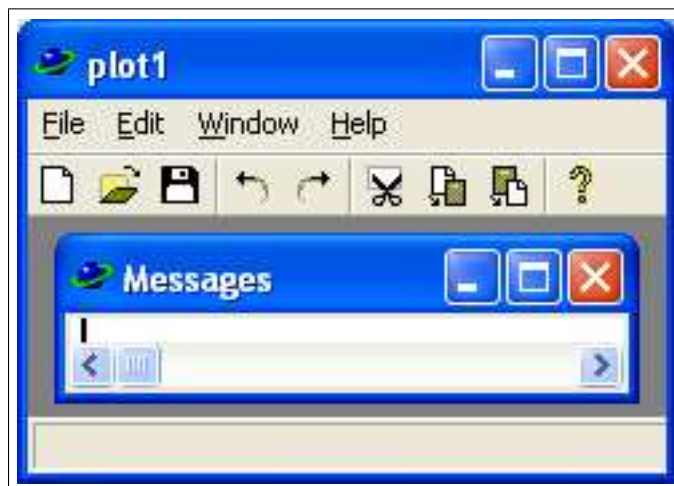


Figure 1.5: An empty application

1.2 Examples

The example in this chapter shows how to run an application, and is in directory *plot1*. In the next chapter, you will add functionality to the `plot1` project, which means that will make the corresponding application do something useful.

1.3 Notions of Logics: Ancient Greeks

The Ancient Greeks devised a kind of society that was very different from anything that existed before them. In other civilizations, decisions of political or legal nature were taken by a king, a small group of aristocrats, or a king's deputy; these people could commit to writing the basis for their decisions; when they did so, they would refer the resulting codex to a god, or to a traditional hero. As for the Greeks, the laws and political decisions were proposed by citizens for whom feats, lots, or life experience won the right to a seat in an assembly or in a court of law. The proposition was voted, and enforced only if the majority of the voters supported it. Let us see how Pericles describes the situation.

Our constitution does not copy the laws of neighboring states; we are rather a model to others than imitators ourselves. Its administration favors the many instead of the few; this is why it is called a democracy. If we look to the laws, they afford equal justice to all in their private differences; if no social standing, advancement in public life falls to reputation for capacity, class considerations not being allowed to interfere with merit; nor again does poverty bar the way: If a man is able to serve the state, he is not hindered by the obscurity of his condition from proposing laws, and political actions.

In this kind of society, if someone wanted to advance his interests, or make inroads for his views, he would need to convince others, prove that he was right, win debates and disputes. Therefore, necessity forced the Greeks to invent Logics. Of course they developed other methods of winning an argument besides Logics: The art of telling lies, crowd psychology, the science of disguising a fallacy, ornate language, and flattery were among the Greek methods of winning a debate. However, this book will focus on Logics.

Chapter 2

Forms

In this chapter, you will add a form to the empty project that you have created in chapter 1. A form is a window pane where you can add components like buttons, that trigger actions, edit fields, that you can use to input text, and canvas, where you can draw things.

2.1 Create a form: `folder/name`

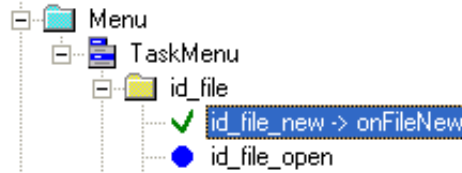
To create a form, choose the option *File/New in New Package* from the task menu, as in figure 2.1. Select *Form* from the left pane and fill in the *Create Project Item* dialog as in figure 2.2. The new form name is *query*. Since you have chosen the option *New in New Package*, Visual Prolog will create the *query* form in a package named after the form. After you press the *Create* button of the *Create Project Item* dialog, the IDE shows you a prototype of the new form, as in figure 2.3. You may want to change the size of the window rectangle, making it slightly larger than the prototype. In order to do this, click and hold the mouse at the lower right corner and drag it, as you do when you resize a normal window.

2.2 Enable the task menu: e.g. **File/New option**

When you ran the empty application, you certainly took notice of the fact that the *File/New* option is disabled. To enable it, click on the *TaskMenu.mnu* branch of the Project tree (figure 2.4). Then, open the tree that appears in the lower part of the *TaskMenu* dialog, and remove the tick from the radio button *Disabled* that corresponds to the *&New\!F7* option, as in figure 2.5.

2.3 In CodeExpert, add code to project tree item

To add code to the *File/New* option, click on the *TaskWindow.win* branch of the project tree with the right button of the mouse, which opens a floating menu. Choose the option *Code Expert* (figure 2.6). As in figure 2.7, click on



Finally, press the button *Add* (refer to figure 2.7), and double click on the branch **id_file_new->onFileNew**. This will open a text editor, with the following snippet:

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

Build the application; then modify this snippet to:

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

Build the program again, choosing the option *Build/Build* from the task menu, as in figure 1.4. Execute the program, and you will see that, whenever you choose the *File/New* option, a new form is created.

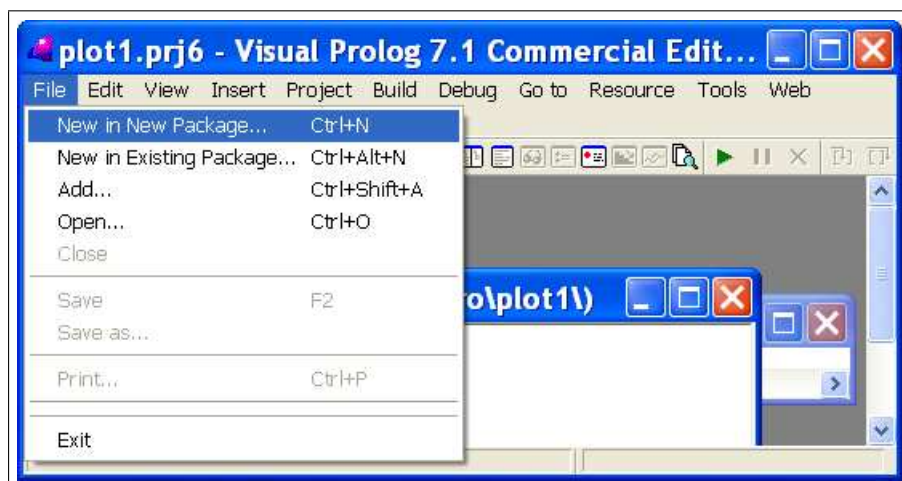


Figure 2.1: Adding a new item to the Project Tree

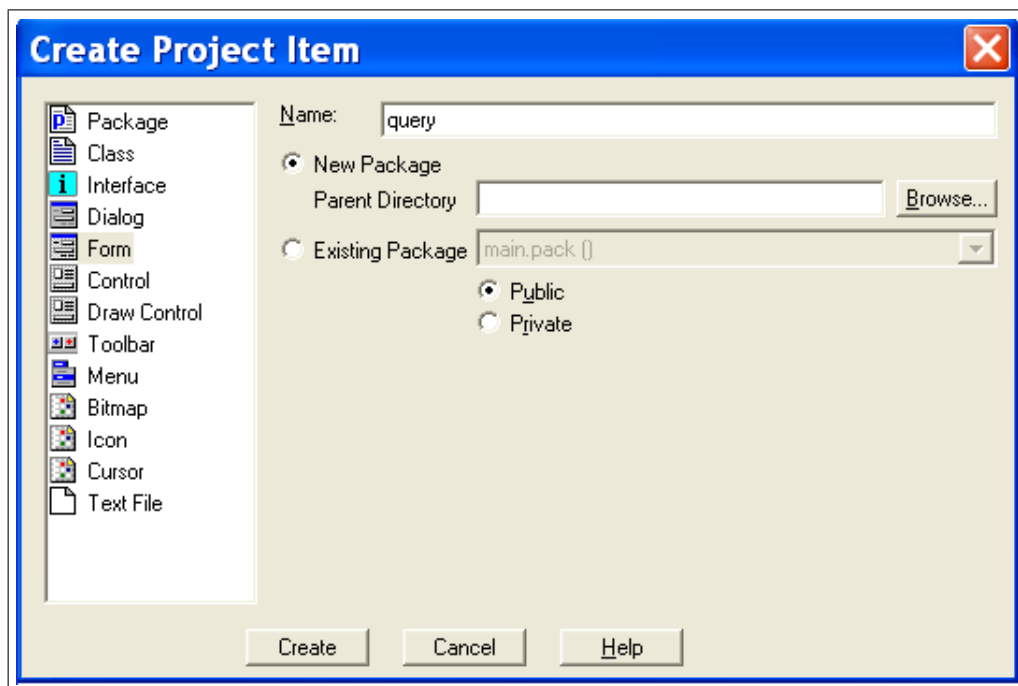


Figure 2.2: Create a new form

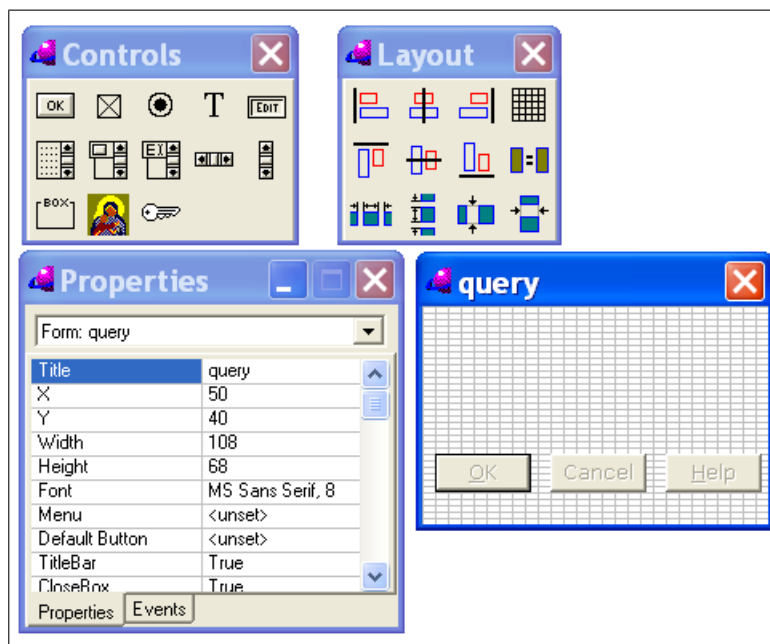


Figure 2.3: Resizing the form window

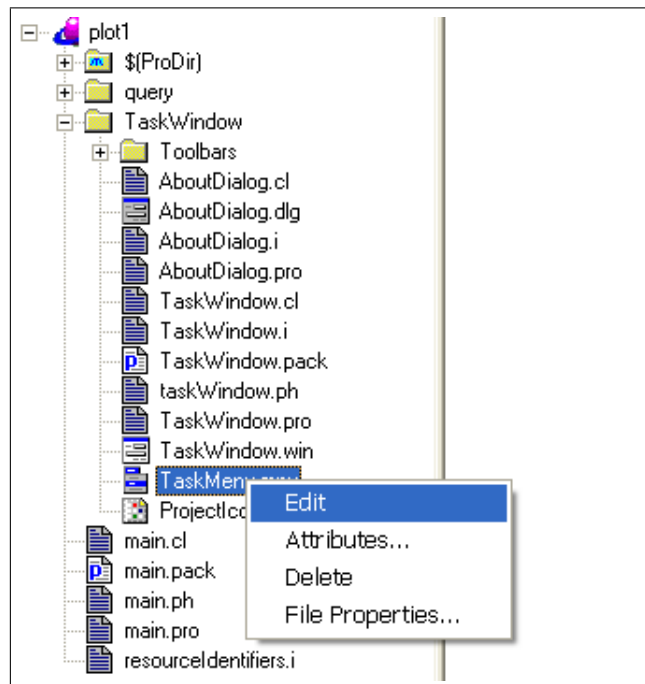


Figure 2.4: Project Tree/Task Menu

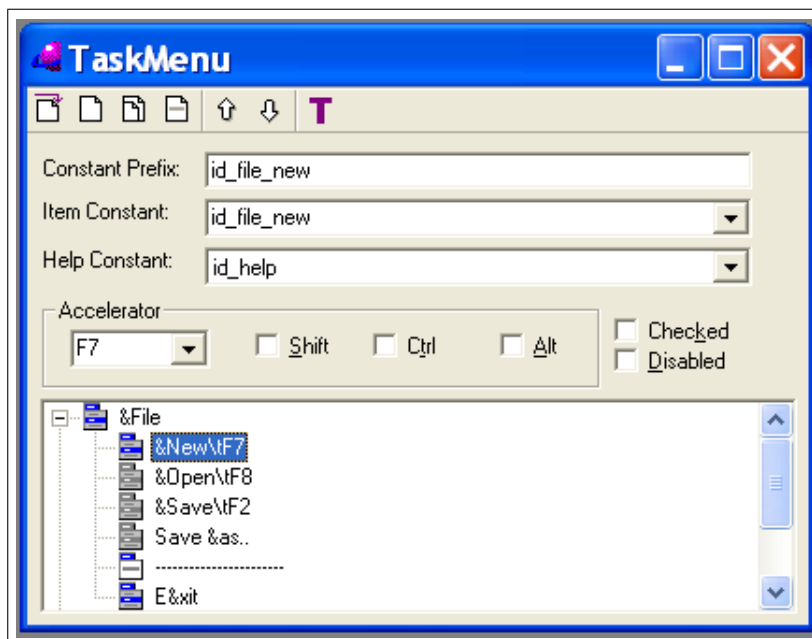


Figure 2.5: Enabling the Task Menu

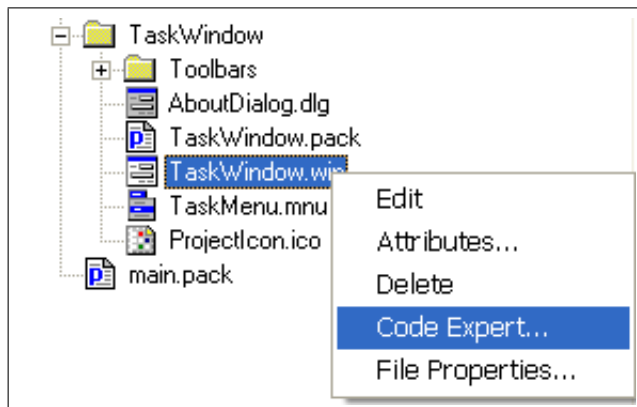


Figure 2.6: Going to the Code Expert

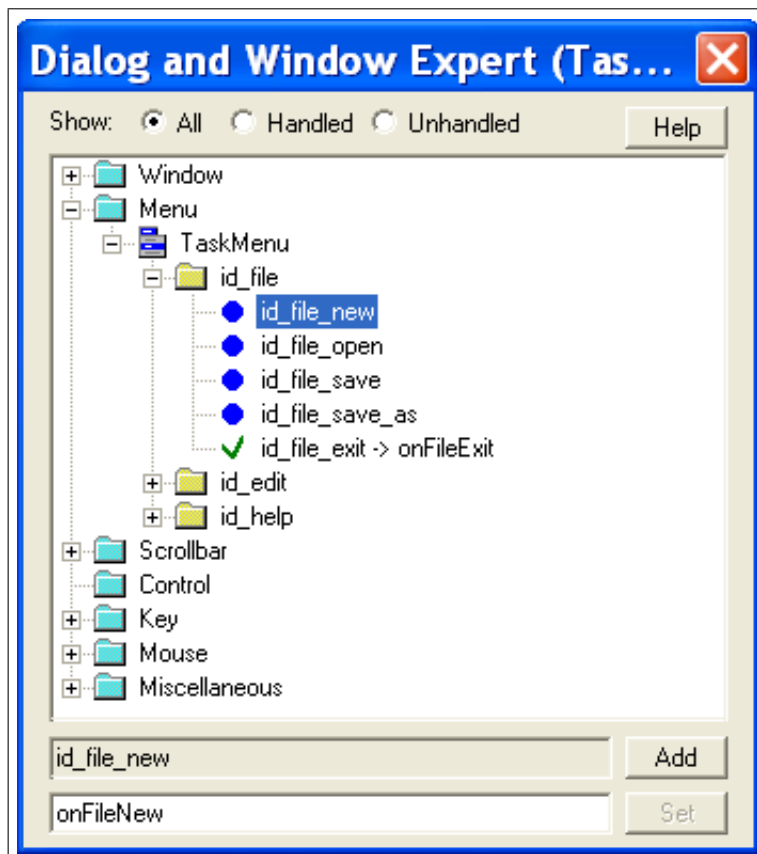


Figure 2.7: Dialog and Window Expert

2.4 Example

The example in this chapter shows how one creates a form by choosing the option *File/New*, then filling in the *Create Project Item* dialog.

2.5 Notions of Logics: Aristotle's Syllogistic

For Aristotle, a proposition consists of two terms, a subject, and a predicate, and a quantifier: 'every', 'no', 'some', 'not every'. *Subject* comes from the Latin rendering of the Greek term ὑποκείμενον, which means what (or whom) the proposition is about. Predicate is what the preposition talks about in the subject.

The Portuguese Logician, medical doctor, and mathematician Pedro Julião (also known as Peter of Spain) made important contributions to many fields of knowledge, before being elected pope under the name of John XXI; for instance, he is considered the father of the ophthalmology, and of the pre-

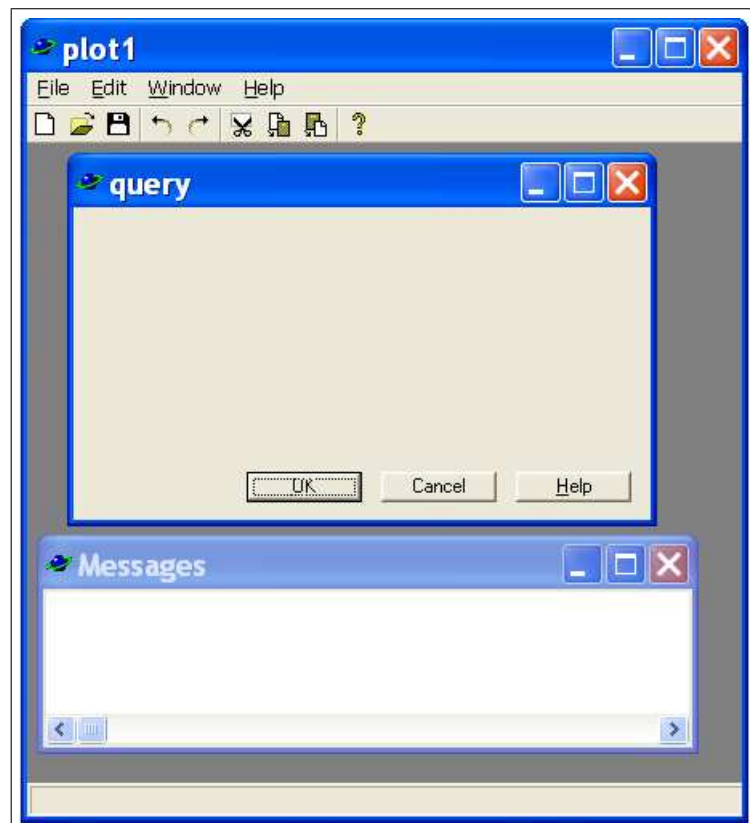


Figure 2.8: Pop-up Form

ventive medicine. He wrote a very influential Logical treatise (*Summulae Logicales*), and is credited with introducing the following abbreviations for the Logical quantifiers:

Quantifier	Type	Abbreviation	Example
<i>a</i>	Universal affirmative	<i>PaQ</i>	Every <i>P</i> is <i>Q</i>
<i>e</i>	Universal negative	<i>PeQ</i>	No <i>P</i> is <i>Q</i>
<i>i</i>	Particular affirmative	<i>PiQ</i>	Some <i>P</i> is <i>Q</i>
<i>o</i>	Particular negative	<i>PoQ</i>	Some <i>P</i> is not <i>Q</i>

Propositions are classified in contraries, contradictories, and subcontraries. Here are the definitions of each one of these classes:

- a. Two propositions *P* and *Q* are contradictories if and only if
 - i. *P* and *Q* cannot both be true, and
 - ii. necessarily, either *P*, or *Q* is true.

Types *a* and *o* are contradictories; the same happens to *e* and *i*.

- b. *P* and *Q* are contraries if and only if
 - i. *P* and *Q* cannot both be true, and
 - ii. *P* and *Q* can both be false.

a and *e* are contraries.

- c. *P* and *Q* are subcontraries if and only if
 - i. *P* and *Q* cannot both be false, and
 - ii. *P* and *Q* can both be true.

i and *o* are subcontraries.

A syllogism is an argument consisting of exactly three propositions (two premises and a conclusion) in which there appear a total of exactly three terms, each of which is used exactly twice. Let *P* represent the predicate of the conclusion (major term), and *S* the subject of the conclusion (minor term). Aristotle called these terms the *extremes*. *M* represents the term that is common to the premises, but it is absent in the conclusion; it is called the middle term. The premise containing *P* is called the major premise; the premise containing *S* is the minor premise.

Let an asterisk $*$ represent any of the four quantifiers ('a', 'e', 'i', 'o'). In this case, the syllogisms are classified in three figures, to wit,

First Figure

Second Figure

Third Figure

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

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

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

When we replace each asterisk with a quantifier, we get a tuple of propositions: {**major premise**, **minor premise**, **conclusion**}. According to Peter of Spain, each distinct tuple ($\{a, a, a\}$, $\{a, i, o\}$...) constitutes a modus, or mood. In order to calculate how many moods there exist, we need to review our lessons of combinatorics.

Permutations are the ways of building lists of things; in a list, the order matters. On the other hand, combinations are groups of things, and the order of the elements of a group does not matter. Since the order of the quantifiers in a syllogism does matter, we are dealing with permutations. We need to permute three elements from a set of four elements; repetition is not disallowed. If you have n things to choose from, and you choose p of them, then the number of possible permutations is given by

$$\underbrace{n \times n \times n \times \dots n}_{p \text{ times}} = n^p$$

In the present case, one has to choose three quantifiers from the set $\{a, e, i, o\}$. Therefore, there are $4^3 = 64$ possibilities for each figure; since we have three figures, the number of possibilities grows to 192.

2.5.1 Valid syllogisms

Not all of candidate syllogisms are logically valid. Aristotle recognized four valid moods each for the first and second figures, and six moods in the case of the third figure. On the other hand, Modern Logics recognizes four modes each for all three figures. Peter of Spain invented Latin mnemonics to help us remember the valid moods; in his scheme, the vowels indicate the mood; for instance, BARBARA indicates $\{a, a, a\}$, and is a valid mood for the first figure. The cultors of Classical Latin will notice that the names are based

on the Medieval Latin pronunciation, where a word like CÆSARE was spelled CÆSARE, and pronounced accordingly. Anyway, here is the complete list of the valid moods:

First figure	Second figure	Third figure
$\{a, a, a\}$ – 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

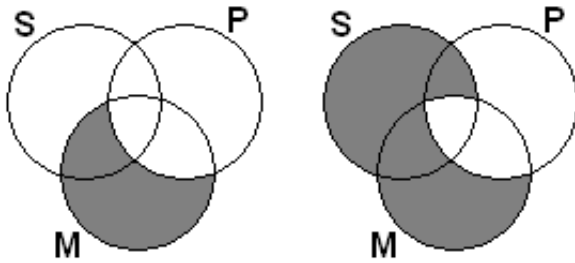


Figure 2.9: Barbara

According to [Lewis Carroll], Venn proposed a diagram to check whether a syllogism is valid. Figure 2.9 illustrates Venn's method for Barbara. The first premise of Barbara says that *all M is P*; therefore, one shades the part of M that is not in P (left hand side of the figure). The second premise says that *All S is M*; therefore, one must shade the part of S that is outside M. The remaining unshaded part of S is entirely inside P. Then, one can conclude that *All S is P*.

$$\frac{\begin{array}{c} MaP \\ SaM \end{array}}{SaP}$$

Let us now consider the Darii syllogism, that says

$$\frac{\begin{array}{c} \forall P \text{ is } M \\ \exists M \text{ is } S \end{array}}{\exists S \text{ is } P}$$

Where the symbol \forall means *All*, and \exists means *Some*. Venn claims that the universal premise should be diagrammed before diagramming a particular premise. Therefore, at the left hand side of figure 2.10, we have shaded the area of P that is not on M , as required by the premise $\forall P \text{ is } M$. The

next step is to place the \exists symbol in the area corresponding to the premise $\exists M$ is S . Figure 2.10 shows that the \exists symbol could be in the $SM\tilde{P}$ area or in the SPM area. Since we do not know exactly which area it is in, we put the \exists symbol on the line. The conclusion is that *Some S is P*.

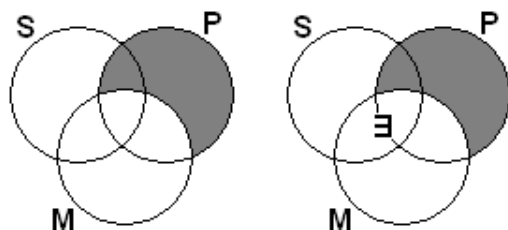
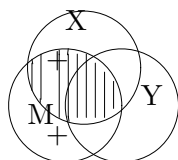


Figure 2.10: Darii

Alice Liddell's father wanted to give her a very good education. When he noticed that there did not exist any good Greek-English dictionaries, he and his friend Scott sat and prepared a very good one, that almost all students of Greek use with pleasure and profit. He also hired the great writer [Lewis Carroll] as a teacher of Logics for his two daughters. In order to make his classes interesting, [Lewis Carroll] wrote a few books for children: *Alice in the Wonderland*, *Through the Looking Glass*, and the *Game of Logics*. The example below is from the *Game of Logics*.



No philosophers are conceited;
 Some conceited persons are not-gamblers.
 Therefore some not-gamblers are not philosophers.

The following solution to the above syllogism has been kindly supplied to me by Mr. Venn himself: The Minor Premiss that some of the constituents in my' must be saved — mark it with a cross. The Major declares that all xm must be destroyed; erase it. Then as some my' is to be saved, it must clearly be my'x'. That is, there must exist my'x'; or, eliminating m, y'x'. In common phraseology, Some not-gamblers are not philosophers.

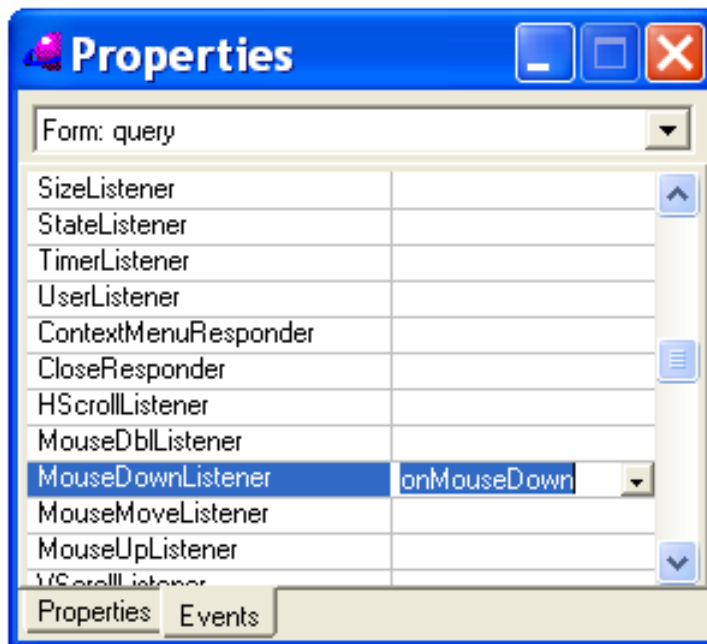
Chapter 3

Mouse events

In the first two chapters of this document, you learned how to build an application with a form, that pops up whenever you choose the option *File/New*, from the application menu.

3.1 Add code to MouseDownListener

Click on the *query.frm* branch of the project tree in order to reopen the query form, as shown on figure 2.3; press the button *Events* of the *Properties*-dialog. This should open an Event list, as shown on the figure below.



Click on the *MouseDownListener* event, and replace the *onMouseDown/4* procedure with the following code:

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



Build the program, and execute it. Choose the option *File/New* to create a new form. Whenever you click at any point within the form, the program will plot a famous greeting expression.

3.2 onPaint

You have used the event handler *onMouseDown/4* to paint something on the form. There are programmers that do not think that this is a good idea. In fact, there are languages that make this approach very difficult. Java is one of these languages, a very popular one, by the way. In Java, one should do all paintings and drawings inside the following method:

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

Of course, one can get around this restriction, even in Java. Anyway, let us learn how to put all drawing activities inside the event handler *onPaint/3*.

1. Create a project, whose name is `plot0`.
2. Add a new `query` form in a new `query` package.
3. Enable the task menu option *File/New*, and add the snippet

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

```
to ProjTree/Taskwindow.win/Menu/TaskMenu/id_file/id_file_new.
```

4. Build the application.

The next step is to add the snippet

```
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).
```

to the MouseDownListener. You are ready to implement the event handler `onPaint` for `query`. The event handler `onPaint` will be called whenever a rectangular region of the form becomes obsolete or invalid. When does a painting become invalid? When the window is covered completely or partially by another window, or when you invalidate it using the predicate `invalidate(R)`, where `R` is a rectangle. The rectangle is described by the coordinates of its upper left angle, and its lower right angle.

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

You have probably noticed that the event handler `onMouseDown/4` invalidates a rectangular region around the mouse click. Then the event handler `onPaint` will act on this invalid region by repainting it. To bring this into effect, add the snippet of figure 3.2 to the *PaintResponder*. From the Project Tree, click on the `query.frm` branch. Press the *Event* button of the Properties-dialog (see figures 3.1, and 2.3, page 19); you will find the *PaintResponder* in the list of events that will appear on the Properties-dialog. This new approach requires you to understand a lot of new things, like the **if-then-else** construction, and the idea of *global variable*. Do not worry about these things for now. You will learn about them at the appropriate time.

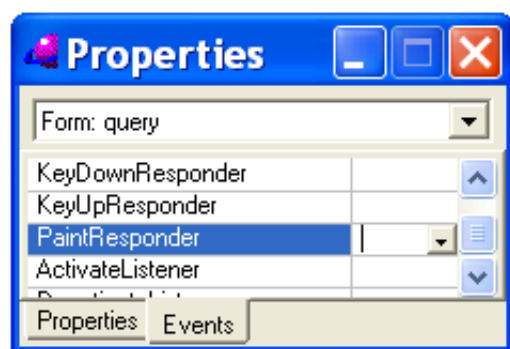


Figure 3.1: PaintResponder

```

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.

```

Figure 3.2: Paint Responder

3.3 Examples

This chapter shows how to handle a mouse event, such as a left click.

3.4 Notions of Logics: Boolean algebra

Propositional Calculus is a formal system in which formulæ representing propositions can be formed by combining atomic propositions using logical connectives. A formula of the PC obeys the following grammar rules:

1. The α -Set contains atomic formulæ, in general, lowercase letters of the Latin alphabet. Any member of the α -Set is a well formed formula.
2. If p is a formula, then $\neg p$ is a formula. The symbol \neg represents the logical negation. Therefore, $\neg p$ is true if p is false, and false otherwise.
3. If p and q are formulæ, then $p \vee q$ (logical disjunction), $p \wedge q$ (logical conjunction), $p \rightarrow q$, $p \equiv q$ are also formulae. Semantically, the formula $p \vee q$ means p **or** q , and is true if either p or q is true; $p \wedge q$ is true only if both p and q are true; $p \rightarrow q$ means that p implies q , i.e., **if** p is true **then** q is true. Finally, $p \equiv q$ means that p is equivalent to q : If p is true, then q is true as well, and if q is true, then p is also true.

Truth tables. In order to establish the truth value of a formula, one often resorts to truth tables. Let us learn about truth tables through examples. Let us consider the well formed formula $p \wedge q$. If 1 represents the constant true, and 0 represents false, the truth table of the negation($\neg p$), of logical

conjunction($p \wedge q$), and of the logical disjunction($p \vee q$) is given by

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

Informally, the formula $p \rightarrow q$ means that

if p is true, then q is also true

A reasonable truth table that conveys such a meaning is given below.

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

It is possible to prove that $P \rightarrow Q$ is equivalent to the formula $\neg P \vee Q$; let us verify this assertion.

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 Argument forms

Logical programmers often use a kind of argument called MODUS TOLLENDOPONENS. In logical operator notation, this simple and very useful argument form is written thus:

$$p \vee q$$

$$\neg p$$

$$\vdash q$$

where \vdash represents the logical assertion. Roughly speaking, we are told that either p or q is true; then we are told that p is not true; so we infer that it has to be q that is true.

A very important argument form is the MODUS PONENS. It is important because it drives Prolog's inference engine.

$$q \leftarrow p$$

$$p$$

$$\vdash q$$

You have learned that $p \rightarrow q$ means **if p then q** , i.e., p **implies** q . Therefore, $q \leftarrow p$ means that q is implied by p , or q **if** p . In fewer words, the first line of the argument says that q is true, if p is true; the second line says that p is true; then man or machine can infer that q is true. Let us consider a concrete example.

$$\text{happy}(Man) \leftarrow \text{hearsSocrates}(Man)$$

$$\text{hearsSocrates}(plato)$$

$$\vdash \text{happy}(plato)$$

Another valid argument is called MODUS TOLLENS:

$$p \rightarrow q$$

$$\neg q$$

$$\vdash \neg p$$

Chapter 4

Less Figures

Before going ahead, let us find out a way to describe our projects without having to resort to figures.

4.1 Task menu

The task menu, shown in figures 1.1 and 1.4, is the main menu of VIP IDE. The notation **A/B** refers to one of its options. For instance, use the option *Project/New* of the task menu to create a new project (figure 1.1). It has six forms, to wit: *General*, *Directories*, *Build Options*, *Version Information*, *File Templates*, and *Run Options*. In most cases, you need to fill in only the *General* form.

General

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

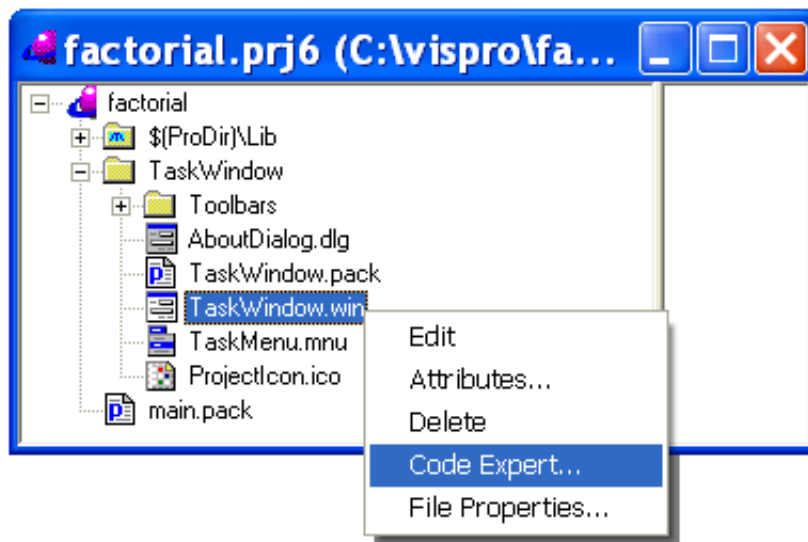
When you find the instructional step:

Create a new GUI project: factorial (section 1.1.1).

you should enter the *Project Settings* dialog (by choosing the *Project/New* option of the VDE task menu) and fill in the General form as shown above. Do it *MUTATIS MUTANDIS*, since your computer may not have space available on drive C:, or you may want to put your programs in a directory different from C:\vispro.

4.2 Project Tree

The easiest way to navigate through files and resources is to click on the corresponding item of the Project Tree:



If you left click on a folder, it will open and present its contents. If you click on an item with the right button of the mouse, a floating menu opens, as shown in the figure. If I want you to go to the code expert, and add a snippet to the *TaskWindow.win*, I will say:

In CodeExpert, add code to TaskWindow/TaskWindow.win(section 2.3)

To carry out this instruction, go to the project tree and do the following:

1. Click on the *TaskWindow* folder to open it, if it is closed.
2. Right click on the *TaskWindow.win* branch of the project tree to deploy the floating menu, which has the following options:

```

Edit
Attribute
Delete
Code Expert
File Properties...

```

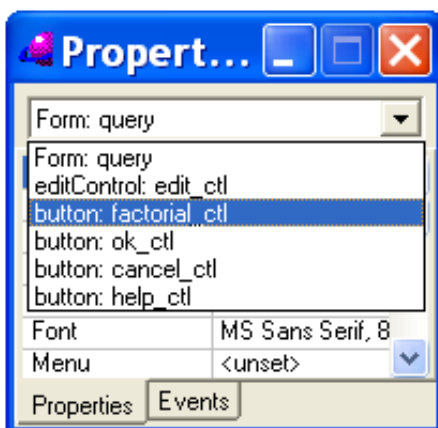
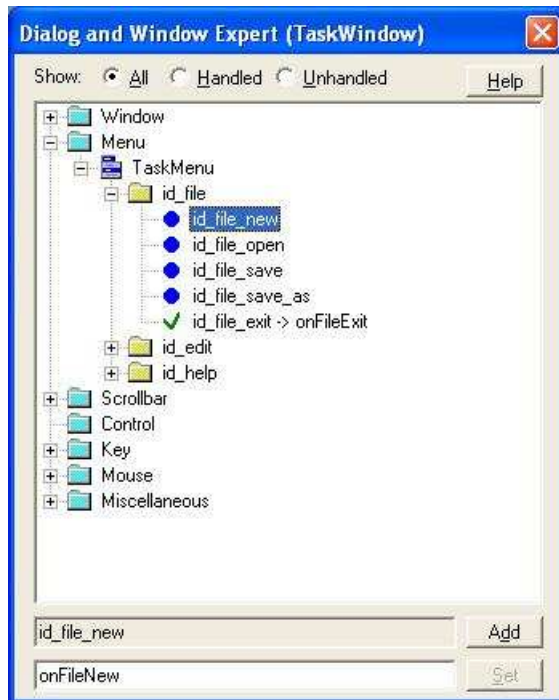
3. Finally, choose the option Code Expert.

4.2.1 Code Expert

The code expert also has a tree shape.

As its name reveals, the code expert is used to insert code, or snippets, into the many files of your project. To reach the code expert, you must right click on the item of the project tree to which you want to add code. Then, choose the option *Code Expert* from the floating menu.

In the case of forms, there is a short-cut to the Code Expert. If you left click the branch of the form on the Project Tree, a prototype of the form will pop up, together with the *Properties*-dialog, and two component boxes, one for Controls, and another for Layout schemes (see figure 2.3); if you press the button *Events* on the *Properties*-dialog, you will get a menu of events to choose from. In past chapters, you had the opportunity to work with two event handlers, to wit, the *MouseDownListener* (page 27), and the *PaintResponder* (page 29).



Forms have components, like buttons, and edit fields. On the top of the *Properties*-dialog, there is a list box for selecting these components; if you select one of these components, and press the *Events*-button you will get a list of the events that correspond to the selected component. More on that later on.

To navigate through the Code Expert tree, and reach the point where you want to insert code, left-click on the appropriate branches of the tree. If you want the code expert to add a prototype on a leaf,

click on the leaf, and press the button *Add* that will appear at the bottom of the dialog. Then, click on the leaf again, to reach the code.

If I ask you: In CodeExpert, add (section 2.3)

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

to `TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new`, here are the steps that you must follow:

- Project tree — Open the folder *TaskWindow* of the project tree, right click on *TaskWindow.win* to open its menu, and choose the option **Code Expert** (figure 2.6).
- Code Expert — Click on **Menu** → click on **TaskMenu** → click on **id_file**. Select **id_file_new** and press the button *Add* to generate prototype code. Finally, click on **id_file_new** → **onFileNew**. Refer to figures 2.6, 2.7, and to sec. 2.3. Add the requested code to the file `TaskWindow.pro`.

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

4.3 Create Project Item

To add a new item to the project tree, choose the option

File/New in New Package

from the task menu if you want to place the item inside a new package named after it. If you want to insert the item into an existing package, choose the option **File/New in Existing Package**. Be careful to place the new item, or the new package, into the desired folder. In the example below, one places the package containing the *query* form on the root of **factorial**. Always choose names that mean something. For instance, if your package holds computer graphics, you could choose a name like *canvasFolder*; if it contains queries, a good name is *formcontainer*; and so on. Example:

- **Create a new package** on the root of the **factorial**-project (figure 4.1). Let the name of the package be **formcontainer**.
- **Create a new form** (query) inside the **formcontainer**-package, by selecting the branch corresponding to the **formcontainer**-package in the Project Tree, and choosing the option *File/New in Existing Package* from the task menu (section 2.1). In order to put the window inside a package, be sure that the package folder is selected before going to *File/New in Existing Package*.

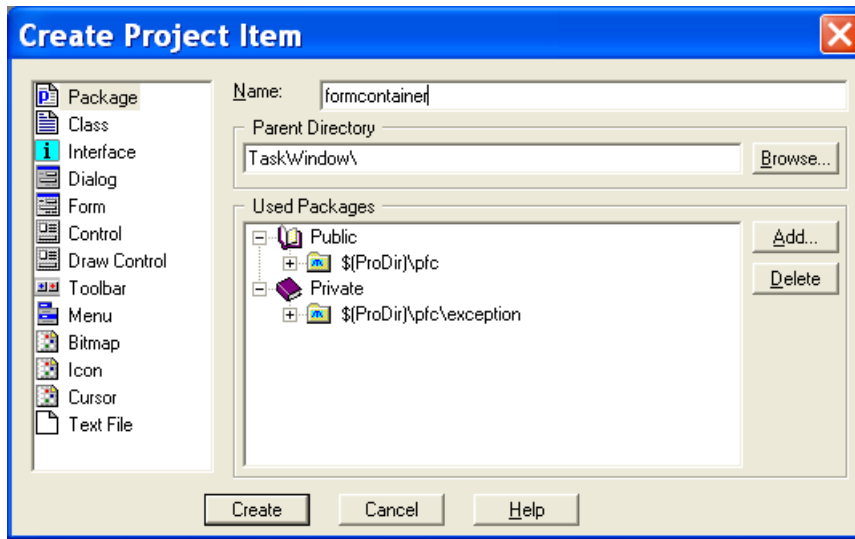


Figure 4.1: Create a new package

When you create a form, the IDE shows the *Form Window* dialog (see figure 4.2). You can take the opportunity to resize the window and add an edit field (handle: `edit_ctl`) and a button (handle: `factorial_ctl`), as shown in the figure. You can also reach the *Form Window* dialog by left clicking on the form name in the project tree.

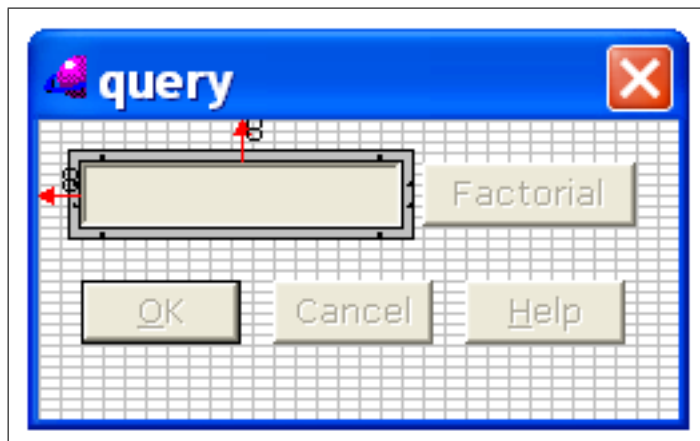
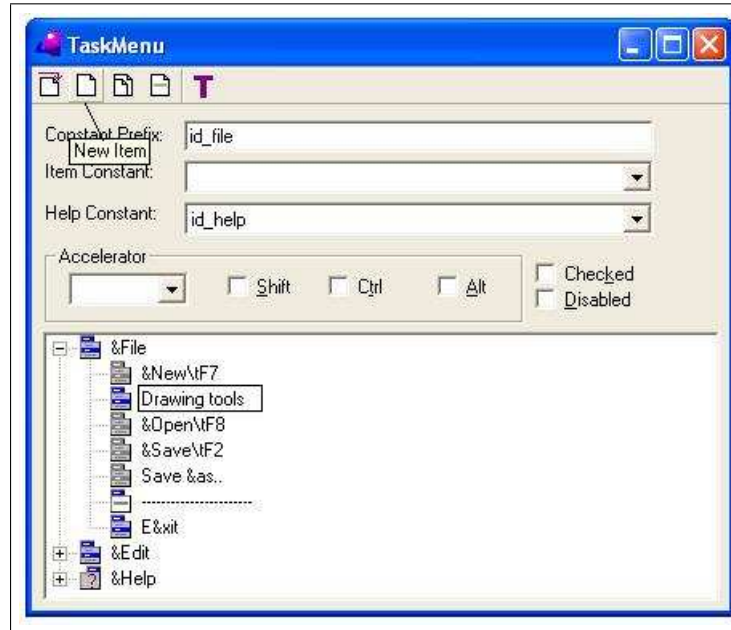


Figure 4.2: A form with an edit field

If you “double-click” on *ProjectTree/TaskMenu.mnu*, you get a dialog like the one shown in figure 2.5. You can unfold the tree of this menu specification by clicking on its option branches, as already mentioned. You may also need to enable a menu option, as in section 2.2. It is also possible to create a new

menu entry, as shown in the figure below, where I have pressed the button (icon) labelled *New Item* and created the entry *Drawing Tools* by filling in the dialog. As you can see from the figure, the new option is enabled from the start. The symbol *&* in the *&File* entry makes *F* a shortcut key.



Let us start a new project from scratch, to see what we can do with fewer figures and check whether we have a terse description of a VIP project.

Create a new GUI project: factorial (section 1.1.1).

Add a new package to the project tree: factorial/formcontainer.

Create a new form: formcontainer/query (section 4.3). Add an Edit Field (`edit_ctl`) and a Button (`factorial_ctl`) to it, as in figure 4.2.

Enable the TaskMenu File/New option. Enable *File/New* (section 2.2), and add (section 2.3):

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

    to TaskWindow.win/Menu/TaskMenu/id_file->id_file_new->onFileNew.
```

Build the application to insert the new classes into the project.

4.4 Create a new class: *folder/name*

To create a new class and put it inside the package *formcontainer*, select the folder *formcontainer* in the *Project Tree*, and choose the option *File/New in Existing Package* from the IDE task menu. Select the option *Class* from the Project Item dialog, and fill in the class *Name* field with **fn**. Make sure to untick the *Create Objects* box.

When you press the *Create* button, VIP will display the files **fn.pro** and **fn.cl**, that contain a prototype of the *fn* class. It is our job to add functionality to these files, by replacing the contents of listing of page 40 for them. Then, build the application once more, to make sure that Visual Prolog inserts the **fn**-class into the project.

4.5 Edit field contents

It is quite tricky to get the contents of an edit field. This is true for Visual C, Delphi, Clean, etc. To do justice to Visual Prolog, I must say that it offers the easiest access to controls, as compared to other languages. However, it is still tricky. To make things somewhat easier, the IDE stores a handle to the edit field window in a fact variable. To see how to make good use of this convenience, open the **query.frm** by clicking on its branch in the project tree.



Choose the **button:factorial_ctl** from the *Properties*-dialog list box, press the *Event*-button, and click on the **ClickResponder** entry of the Event-list; then add the following snippet to the **factorial_ctl** button code:

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

Build and run the program.

4.6 Examples

In this chapter, you have learned how to create a form with a button(`factorial_ctl`, and an edit field (`edit_ctl`); you have also learned how to access the contents of the edit field. Below is the `fn`-class, that implements the factorial function.

```
%File: fn.cl
class fn
predicates
    classInfo : core::classInfo.
    calculate:(string) procedure.
end class fn

% File 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
```



Figure 4.3: The `fn` class

4.7 Notions of Logics: Predicate calculus

Propositional calculus does not have variables or quantifiers. This was fixed when Friedrich Ludwig Gottlob Frege revealed Predicate Calculus to mankind. However, Frege's notation was hard to deal with; modern notation was introduced by Giuseppe Peano. Let us see how Aristotle's propositions look like in the modern Predicate Calculus notation.

All a is b	$\forall X(a(X) \rightarrow b(X))$
Some a is b	$\exists X((a(X) \wedge b(X))$

Chapter 5

Horn Clauses

5.1 Functions

I am sure you know what a function is. It is possible that you do not know the mathematical definition of function, but you have a feeling for functions, acquired by the use of calculators and computer programs or by attending one of those Pre-Algebra courses. A function has a functor, that is the name of the function, and arguments. E.g.: $\sin(X)$ is a function. Another function is $\text{mod}(X, Y)$, that gives the remainder of a division. When you want to use a function, you substitute constant values for the variables, or arguments. For instance, if you want to find the remainder of 13 divided by 2, you can type $\text{mod}(13, 2)$ into your calculator (if it has this function, of course). If you want the sinus of $\pi/3$, you can type $\sin(3.1416/3)$.

One could say that functions are maps between possible values for the argument and a value in the set of calculation results. Domain is the set of possible values for the argument. Image is the set of calculation results. In the case of the sinus function, the elements of the domain is the set of real numbers. There is an important thing to remember. Mathematicians insist that a function must produce only one value for a given argument. Then, if a calculation produces more than one value, it is not a function. For instance, $\sqrt{4}$ can be 2 or -2 . Then, the square root of a number is not a function; however, you can force it into the straightjack of the function definition, by saying that only values greater or equal to zero are part of the image.

What about functions of many arguments? For example, the function $\max(X, Y)$ takes two arguments, and returns the greatest one. In this case, you can consider that it has only one argument, that is a tuple of values. Thus, the argument of $\max(5, 2)$ is the pair $(5, 2)$. Mathematicians say that the domain of such a function is the Cartesian product $\mathbb{R} \times \mathbb{R}$.

There are functions whose functor is placed between its arguments. This is the case of the arithmetic operations, where one often writes $5 + 7$ instead of $+(5, 7)$.

5.2 Predicates

Predicates are functions whose domain is the set `{verum, falsum}`, or, if you do not like the Latin names used by logicians, you can always opt for the English equivalent: `{true, false}`. There are a few predicates that are well known to any person who tried his hand on programming, or even by a student who is taking Pre-Algebra. Here they are:

$X > Y$ is **true** if X greater than Y , **false** otherwise
 $X < Y$ is **true** if X less than Y , **false** otherwise
 $X = Y$ is **true** if X equal to Y , **false** otherwise

A predicate with one argument tells of an attribute or feature of its argument. One could say that such a predicate acts like an adjective. In C, `~X` is **true** if X is **false**, and **false** otherwise. In other computer languages, there are predicates equivalent to this one. More examples of one slot predicates:

`positive(X)`: **true** if X is positive, **false** otherwise
`exists("bb.txt")`: **true** if file "bb.txt" exists, **false** otherwise

A predicate with more than one argument shows that there exists a relationship between its arguments. In the case of $X = Y$, the relationship is the equality between X and Y . It would be interesting to have a programming language, with predicates stating attributes and relationships other than the few offered by calculators and main stream computer languages. For instance, many people, during their lifetime, have felt a compelling need for the prediction of one of the predicates listed in figure 5.1, specially the third one. Prolog is a computer language that was invented to fulfill such a need.

5.3 Solutions

Suppose that you have a predicate `city(Name, Point)` that gives the coordinates of a city on a map. The `city/2` predicate has the domain ¹

`city:(string Name, pnt Position).`

¹N.B. Predicates are functions whose domain can be any Cartesian product, but whose image is always the set `{true, false}`.

and can be implemented as a database of facts:

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

This predicate checks whether the given position of a given city is correct, when one is not sure about it. Here are some queries that one could pose to the `city/2` predicate.

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

There is no question that you could find a use for such a predicate. However, a predicate that returns the coordinates of a city, given the name, would be even more useful.

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

In this new kind of predicate, symbols that start with uppercase are called variables. Examples of variables: *X*, *Y*, *Wh*, *Who*, *B*, *A*, *Xs*, *Temperature*, *Humidity*, *Rate*. Therefore, if you want to know whether a symbol is a variable, check its first letter. If it is uppercase or even the underscore sign (`_`), you are dealing with a variable.

positive(X) is true if *X* is positive, false otherwise.

rain(Temperature, Pressure, Humidity) is true if it is likely to rain at a given temperature, pressure, and humidity. For instance

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

returns true if it is likely to rain when your instruments show $100^{\circ}F$, 1.2 atmospheres, and 90% of relative humidity.

invest(Rate, StandardDeviation, Risk) Given a return Rate, Standard-Deviation, and acceptable Risk, this predicate returns **true** if you should choose an investment.

Figure 5.1: Interesting predicates

When you use a variable, as *P* in the query `city("Salt Lake", P)`, you want to know what one must substitute for *P* in order to make the predicate `city("Salt Lake", P)` true. The answer is `P= pnt(30, 40)`. Sophocles said that your hand should not be faster than your mind, nor fail to keep pace with it. Therefore, let us implement the `city/2` predicate.

Project Settings. Enter the *Project Settings* dialog by choosing the option *Project/New* from the VDE task menu, and fill it in.

General

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

Create Project Item: Form. Select the *root* node of the Project Tree. Choose the option *File/New in New Package* from the task menu. In the *Create Project Item* dialog, select the option *Form*. Fill in the dialog:

```
Name: map
```

Add the following buttons to the new form: `Logan`, `Salt Lake`, `Provo`.

Window Edit. Resize the new form rectangle. The form window must have a sufficient size to show our “map”. Leave a generous empty space in the center of the form.

Build/Build This is important: From the task menu, **Build/Build** the project, otherwise the system will accuse an error in the next step.

Project Tree/TaskMenu.mnu. Enable *File/New*.

Project Tree/TaskWindow.win/Code Expert. Add

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

to `Menu/TaskMenu/id_file/id_file_new/onFileNew`. **Build/Build** the project again (it is better to be safe than sorry).

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

Figure 5.2: mapDataBase/draw.cl

Create class. Create a *draw* class, as explained in section 4.4. To create a new class, select the root of the *Project Tree*, and choose the option *File/New in New Package* from the IDE task menu. The class name is *draw* and the *Create Objects* box is off. Build the project, in order to insert a prototype of the new class into the project tree. Then, edit *draw.cl* and *draw.pro* as shown in figures 5.2 and 5.3.

To call the *drawThem* predicate using the city-buttons, go to the project tree, and open the *map.frm* form, if it is not already open; in the *Properties*-dialog, choose the *logan_ctl* button from the component list box; press the Event button, and add the following snippet

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

to the *ClickResponder*.

Repeat the operation for "Salt Lake" and "Provo". Don't forget to change the names of the buttons to *logan_ctl* and *provo_ctl*, and *saltlake_ctl* respectively; change also the city name "Logan" in "drawThem" to "Provo" or "Salt Lake". Build the project and execute the program. If you do not remember how to build and execute the program, take a look at section 1.1.2. In the new application, choose the option *File/New*. A new form will pop up. Whenever you click on a button, the program draws the corresponding city.

```

% 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

```

Figure 5.3: mapDataBase/draw.pro

5.4 Multi-solutions

In the last section, you saw predicates that are used for attributing solutions to their variable, not for checking whether a relationship is true or false. In the example, the predicate `city/2` is used to retrieve only one solution. However, there are situations that call for more than a single solution. Let `conn/2` be a predicate that establishes a connection between two cities.

```

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

```


You can use it to retrieve all connections between cities, as the example shows:

```

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)

```

Consider the query:

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

The answer could be `W= pnt(100,120)`, but it could also be `W= pnt(200, 100)`.

5.4.1 A program that uses multi solution predicates

Let us create a program to demonstrate how nice the multi solution feature of Prolog is, a feature that it does not share with other languages.

Project Settings. Enter the *Project Settings* dialog by choosing the option *Project/New* from the IDE task menu, and fill it in.

```

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

```

Create Project Item: Package. Select the *drawMap* node of the project tree. Choose the option *File/New in New Package* from the task menu. In the *Create Project Item* dialog, select *Package* and fill in the form:

```

Name: plotter
Parent Directory:

```

Create Project Item: Form. Select the *plotter* node of the project tree. Choose the option *File/New in Existing Package* from the task menu. In the *Create Project Item* dialog, select the option *Form*, and fill it.

```

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

```

Include the form. From the task menu, choose Build/Build.

Project Tree/TaskMenu.mnu. Enable *File/New*.

Project Tree/TaskWindow.win/Code Expert. Add

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

to Menu/TaskMenu/id_file/id_file_new/onFileNew
```

Create class. Create a *draw* class inside the package *plotter*, as explained in section 4.4. Untick the box *Create Objects*. Put class code in files *draw.cl* and *draw.pro*, as shown in figure 5.4. Build the application.

ProjectTree/map.frm Open the *map.frm* and add the following snippet to the *PaintResponder*:

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

If you build and run this program, you should obtain a window with a map, such as the one in figure 5.5, whenever you press *File/New*.

5.5 Logical junctors

I guess that you have had contact with the logical *and*, that appears in languages like C or Pascal:

```
if ((X > 2) && (X < 4)) {...}
```

If P_1 and P_2 are predicates, the expression “ P_1 **and** P_2 ” is true if both P_1 and P_2 are true. A sequence of predicates linked by logical *and*-s is called conjunction. In C,

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

is a conjunction. In Prolog, the predicates of a conjunction are separated by commas. Then, the expression $(X>2) \ \&\& \ (X<4)$ becomes

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

The logical *and* is called a junctor.

5.6 Implication

An implication is a junctor represented by the symbol `:-`, that means *if*. Therefore,

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

means that you `drawThem` on `Win` if you draw `connections` on `Win` and `drawCities` on `Win`.

5.7 Horn Clauses

A Horn clause can be a sole predicate. For instance, in the listing below, there are four one predicate Horn clauses.

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

One predicate Horn clauses are called *facts*. In the example, the facts state a relationship between a city and its coordinates. The domain of these predicates is a set of pairs, each pair containing the name of the city and its coordinates. A Horn clause can also take the form

$$H :- T_1, T_2, T_3 \dots$$

where T_i and H are predicates. Thus,

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

is an instance of Horn clause. In a Horn clause, the part that comes before the `:-` sign is called the *head*. The part that comes after the sign `:-` is the *tail*. In the example, the head is `drawThem(W)`, and the tail is

```
connections(Win), drawCities(Win)
```

A set of Horn clauses with the same head defines a predicate. For instance, the four city Horn clauses define the `city/2` predicate, and

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

defines `drawThem/1`.

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

% File: 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
```

Figure 5.4: draw.cl and draw.pro

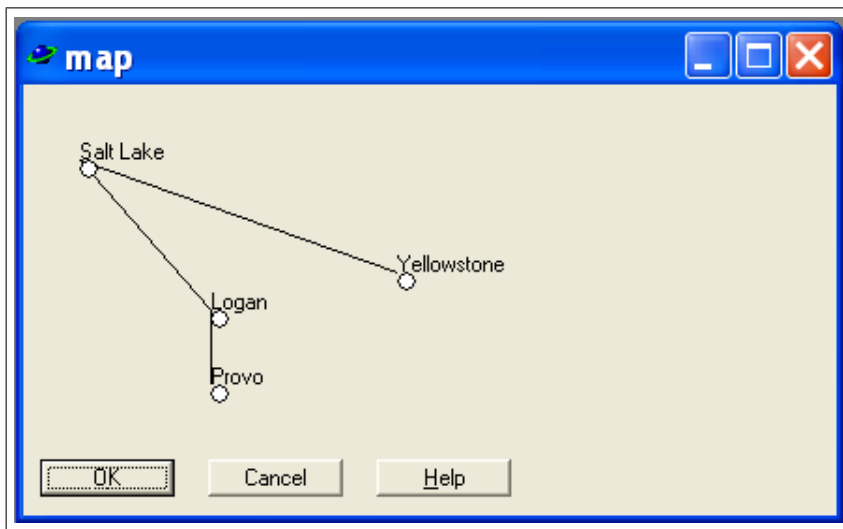


Figure 5.5: Utah cities

5.8 Declarations

The definition of a predicate is not complete without type/flow pattern declaration. Here is the type and flow (or mode) declaration of `drawThem/1`.

```

predicates
  drawThem:(windowHandle) procedure (i).

```

The type declaration says that the argument `Win` of `drawThem(Win)` is of type `windowHandle`. The mode declaration states that the argument of `drawThem/1` must be an input, i.e., it is a constant, not a free variable. In this case, the predicate inputs data from its argument. In general, you do not need to provide mode declaration. The compiler will infer the modes. If it fails to perform the inference, it will issue an error message, and you can add a mode declaration. If a predicate is supposed to be used only inside its class, it is declared as *class predicate*. Examples:

```

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

```

One predicate Horn clauses may be declared as facts. For example:

```

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

```

Later on, you will see that facts can be asserted or retracted from the data base. The arguments of `conn/2` are a pair of `pnt` values. The type `pnt` is defined in class `vpiDomains`. To use it in class `draw`, I have two options. I can spell out the class to which `pnt` belongs

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

or else, I can open that class inside `draw`. I have chosen the second option:

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

Determinism declarations

To declare that a predicate has a single solution, or many solutions, one uses the following keywords:

determ. A `determ` predicate can fail, or succeed with one solution.

procedure. This kind of predicate always succeeds, and has a single solution. The predicates `connections/1` and `drawCities/1` defined in figure 5.4 are procedures, and could have been declared thus:

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

multi. A `multi` predicate never fails, and has many solutions.

nondeterm. A `nondeterm` predicate can fail, or succeed in many ways. The facts `city/2` and `connection/2` are `nondeterm`, and accept the following declaration:

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

If a predicate has many solutions, and one of its solution fails to satisfy a predicate in a conjunction, Prolog backtracks, and offer another solution in an effort to satisfy the conjunction. Consider the Horn clause:

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

The nondeterm fact `conn(P1, P2)` provides two points, that are used by the procedure `drawLine(Win, P1, P2)` to draw a straight line. Then, Prolog tries to satisfy the predicate `fail`, that always fails, as its name shows. Therefore, Prolog backtracks, and tries another solution, until it runs out of options; then it tries the second clause of `connection/1`, which always succeeds.

5.9 Drawing predicates

My sixteen year old son thinks that the only use for computer languages is games, drawings, graphics, etc. If you are of the same opinion, you must learn drawing predicates in depth. They require a handle² to a window that will support figures and plots. This is how you may obtain the handle:

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

In the *draw* class, the handle *W* is passed around. For instance, in the clause

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

it is passed to `connections/1`, where it is the first argument of `drawLine/3`:

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

The predicate `drawLine(Win, P1, P2)` draws a line from *P1* to *P2* on the window *Win*. As you know, *P1* and *P2* are points, such as `pnt(10, 20)`. Another predicate that you are acquainted with is

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

that draws an ellipse on the window *W*. The ellipse is inscribed inside the rectangle `rct(X1, Y1, X2, Y2)`, where *X1*, *Y1* are the coordinates of the upper left corner, and *X2*, *Y2* are the coordinates of the lower right corner.

5.10 GDI Object

In the last example, one performed the drawing from the `onPaint` event handler. When this happens, it may be a good idea to use methods from the so called `GDI Object`. The example that follows shows how this works.

²You do not need to know what a handle is for the time being.

Project Settings. Create the following project:

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

Create a package: `plotter`.

Create a form inside `plotter`: `map`.

Project Tree/TaskMenu.mnu. Enable *File/New*. From the task menu, Build/Build the application, in order to include the *map* into the project.

Project Tree/TaskWindow.win/Code Expert. Add

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

to Menu/TaskMenu/id_file/id_file_new/onFileNew
```

Create class. Create a *draw* class inside package `plotter`. Remember to untick “Create objects”. You will find the new version of class `draw` in figure 5.6. Build the application.

Add

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

to the *PaintResponder* of the `map.frm`.




```

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

% File: 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

```

Figure 5.6: Class draw

5.11 Examples

There are three examples in this chapter: *mapDataBase* draws the position of the three main cities of Utah, when one presses the appropriate button; *drawMap* draws a coarse road map of Utah; *drawMapObj* is the GDI version of *drawMap*.

5.12 Notions of Logics: Meaning of Horn clauses

A Horn clause has the form

$$H : -T_1, T_2, T_3 \dots$$

The meaning of this clause is

If $T_1 \& T_2 \& T_3 \dots$ are true,
then H is also true.

You have seen before that this statement is equivalent to

H is true or the conjunction $T_1 \& T_2 \& T_3 \dots$ is not true.

This last interpretation sheds light on the meaning of the neck symbol, where the semicolon stands for the connective **or**, and the minus sign stands for the logical negation. Then,

$$H : -T$$

means: H is true, or not T . To finish our study of the Prolog connectives, let us remind you that the comma stands for **and**, the semicolon stands for **or**, and \rightarrow stands for **if...then**.

Chapter 6

Console Applications

Since people like graphic applications, we started with them. However, they add details that sway your attention from what really matters. Thus, let us see a few basic programming schemes without resorting to a graphic interface.

6.1 Cut

What should you do if you don't want the system either to backtrack or try another clause after finding a solution? In this case, you insert an exclamation mark at a given point in the tail of the Horn clause. After finding the exclamation mark, the system interrupts its search for new solutions. The exclamation mark is called *cut*. Let us illustrate the use of cuts with an algorithm that is very popular among Prolog programmers. Let us assume that you want to write a predicate that finds the factorial of a number. Mathematicians define factorial as:

$$\begin{aligned}\text{factorial}(0) &\rightarrow 1 \\ \text{factorial}(n) &\rightarrow n \times \text{factorial}(n-1)\end{aligned}$$

Using Horn clauses, this definition becomes:

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

The cut prevents Prolog from trying the second clause for $N = 0$. In other words, if you pose the query

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

the program succeeds in using the first clause for $N = 0$, and returns $F = 1$. Without the cut, it will assume that the second clause of the definition also

provides a solution. Then, it will try to use it, and will go astray. The cut ensures that factorial is a function and, as such, maps each value in the domain to one and only one value in the image set. To implement the factorial function, follow the following directive:

Create a New Project. Choose the option *Project/New* and fill the *Project Settings* dialog thus:

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

Pay attention to the fact that we will use the console strategy, not GUI.

Build. Choose the option *Build/Build* from the task menu to add a prototype of class *facfun* into the project tree. Edit *facfun.pro* as shown below. Build the project again, and execute it using *Run in Window*. Write a number to the prompt, and you will get its factorial. NB: To test a console program, use *Run in Window*, not *Execute*.

```
%File: main.pro
implement main

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).
```

6.2 Lists

There is a book about Lisp that says: A list is a an ordered sequence of elements, where *ordered* means that the order matters. In Prolog, a list is

put between brackets, and is supposed to have a head and a tail:

List	Type	Head	Tail
[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]

You can match a pattern of variables with a list. For instance, if you match

[X|Xs]

with the list [3.4, 5.6, 2.3] you will get $X = 3.4$ and $Xs = [5.6, 2.3]$, i.e., X matches the head of the list and Xs matches the tail. Of course, you can use another pair of variables instead of X and Xs in the pattern $[X|Xs]$. Thus, $[A|B]$, $[X|L]$, $[Head|Tail]$, $[First|Rest]$ and $[P|Q]$ are equivalent patterns. More examples of matches for the pattern $[X|Xs]$:

Pattern	List	X	Xs
$[X Xs]$	[3.4, 5.6, 2.3]	$X = 3.4$	$Xs = [5.6, 2.3]$
$[X Xs]$	[5.6, 2.3]	$X = 5.6$	$Xs = [2.3]$
$[X Xs]$	[2.3]	$X = 2.3$	$Xs = []$
$[X Xs]$	[]	<i>Does not match.</i>	

As you can see, the pattern $[X|Xs]$ only matches lists with at least one element. Here is a pattern that only matches lists with at least two elements:

Pattern	List	$X1, X2$	Xs
$[X1, X2 Xs]$	[3, 5, 2, 7]	$X1 = 3, X2 = 5$	$Xs = [2, 7]$
$[X1, X2 Xs]$	[2, 7]	$X1 = 2, X2 = 7$	$Xs = []$
$[X1, X2 Xs]$	[7]	<i>Does not match.</i>	
$[X1, X2 Xs]$	[]	<i>Does not match.</i>	

Let `avg` be a small project that calculates the length of a list of real numbers. As in the case of `factorial`, make sure that `avg` is a console application.

General

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

Build the project to insert the `avg` class into the project tree. Then, edit `avg.pro` as shown below. Execute using *Run in Window*, as before.

```
/* File: 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).
```

Let us examine the concepts behind this short program. The first thing that you have done is to create a list domain:

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

Then, you define a predicate `len/2`, that inputs a list from the first argument, and outputs its length to the second argument. Finally, you define a `run()` predicate to test your program:

```
run():- console::init(),           % Initialize the console.
        hasdomain(rList, List), % Create a rList variable.
        List= read(),           % Read List from console.
        len(List, A),           % Find the length of List.
        write(A), nl.          % Output the length.
```

The domain declaration is handy if you are dealing with complex algebraic data types. However, it is not required for such a simple structure as a list of reals. Below you will find the same program without the domain declaration.

```

/* File: main.pro */
implement main
open core, console
class predicates
  len:(real*, 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).

```

This program has a drawback: One can use *len/2* only for calculating the length of a list of reals. It would be nice if *len/2* could take any kind of list. This should be possible if one substitutes a type variable for **real***. However, before checking whether this scheme really works, it is necessary to devise a way of informing *L= read()* which will be the eventual type of the input list, and feed the information to *len/2*. Fortunately there is a predicate that creates a variable for any given type. You will discover below how to use it.

```

implement main /* in file 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).

```

The next step in our scheme of things is to insert `sum/2` clauses to `main.pro`.

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

Let us examine what happens if one calls `sum([3.4, 5.6, 2.3], S)`.

1. `sum([3.4, 5.6, 2.3], S)`, matches
`sum([X|Xs], S+X) :- sum(Xs, S)`, with $X=3.4$, $Xs=[5.6, 2.3]$,
yielding `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. `sum([5.6, 2.3], S[5.6, 2.3])`, matches
`sum([X|Xs], S+X) :- sum(Xs, S)`, with $X=5.6$, $Xs=[2.3]$,
yielding `sum([5.6, 2.3], S[2.3] + 5.6) :- sum([2.3], S[2.3])`
3. `sum([2.3], S[2.3])` matches
`sum([X|Xs], S+X) :- sum(Xs, S)`, with $X=2.3$, $Xs=[]$,
yielding `sum([2.3], S[] + 2.3) :- sum([], S[])`
4. `sum([], S[])` matches `sum([], 0.0)` yielding $S_{[]} = 0$.

After reaching the bottom of the list, the computer must roll back to the start of the computation. What is worse, it must store the values of every X that it finds on its way down, in order to perform the addition $S + X$ when it rolls back. The traditional way to prevent the roll back is to use accumulators. Here is a definition that uses an accumulator to sum the elements of a list:

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

Let us see how the computer calculates this second version of list addition.

1. `add([3.4, 5.6, 2.3], 0.0, S)`
matches `add([X|Xs], A, S) :- add(Xs, X+A, S)`
yielding `add([5.6, 2.3], 0+3.4, S)`
2. `add([5.6, 2.3], 0.0+3.4, S)`
matches `add([X|Xs], A, S) :- add(Xs, X+A, S)`
yielding `add([2.3], 0+3.4+5.6, S)`
3. `add([2.3], 0.0+3.4+5.6, S)`
matches `add([X|Xs], A, S) :- add(Xs, X+A, S)`
yielding `add([], 0+3.4+5.6+2.3, S)`
that matches `add([], A, A)` yielding $S = 11.3$.

You could use `add/3` to calculate the average value of a list of numbers.


```

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).

```

The above program goes through the list twice, first to calculate the sum, and second to calculate the length. Using two accumulators, you can calculate the length and the sum of the list together.

```

%File: 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,           % add X to the sum acc
        L+1.0,         % increments the length acc
        A).

  run():- console::init(),
    List= read(),
    avg(List, 0, 0, A),
    write(A), nl.
end implement main

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

```

The above listing shows a program to calculate the average value of a list of real numbers. One makes use of two accumulators, one for the sum, and the other for the number of elements.

6.3 List schemes

Let us see a few important schemes of list programming. I thought that was obvious that I didn't invent these schemes. However, since a few persons wrote me asking questions about their origin, I added two entries to the reference list: [John Hughes], and [Wadler & Bird].

Reduction

The scheme used to calculate the sum of the elements of a list is called reduction, as it reduces the dimension of the input. In fact, lists can be thought of as one-dimensional data, and the sum is a zero-dimensional datum. There are two ways of performing a reduction: recursive and iterative.

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

The term *iterative* comes from the Latin word *ITERUM*, that means *again*. You can also imagine that it comes from *ITER/ITINERIS*= *way, road*. An iterative program goes through the code again, and again.

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

Let us assume that, instead of performing an addition, you want to perform a multiplication. In this case, you can write the following program:

```
%Iterative reduction
red([], A, A) :- !.
red([X|Xs], A, S) :- red(Xs, X*A, S).
product(Xs, S) :- red(Xs, 0.0, S).
```

The two pieces of code are identical, except for a single detail: one of them has $X+A$ in the second argument of the iterative call of the predicate `red/3`, while the other has $X*A$. [Wadler & Bird] and [John Hughes] suggests that one capture this kind of similarity in higher order predicates or functions. Before them, when he received the Turing award, [John Backus] made the

same proposition. Let us see that one can use this prestigious method of programming in Visual Prolog.

In the program of figure 6.1, I have declared a domain that is a two-place function. In the section containing the class predicates, I have defined a few functions belonging to this domain. Finally, I defined a `red/4` predicate that capture the structure of the reduction operation.

```

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).
```

Figure 6.1: Reduce in Prolog

ZIP

There is a device used for continuous clothing closure invented by the Canadian electrical engineer Gideon Sundback. The device drives one crazy when it does not work, and is the cause of so many accidents with young boys, that doctors and nurses need special training to deal with it.

```
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).
```

The example shows a zipping scheme followed by a reduction step, in order to perform the dot product of two lists. In the expression $X*Y+R$, one uses the multiplication to close one clasp of the zipper.

We can capture the zipping scheme in a higher-order predicate, the same way we have captured the reducing scheme. In fact, we can close each clasp of the zipper with the help of the same type of functions that we have used to perform a reduction step. In figure 6.2, you can find a (not very efficient) definition of `zip` in Prolog. I added this listing because more than one person wrote me requesting for an in-depth treatment of this important topic; I believe that I cannot provide an in-depth treatment, since this would go beyond the scope of this book; after all, my goal is to avoid subjects that present steep learning curves to beginners; however, the examples in figures 6.1, and 6.2 should be enough for showing how to write higher-order predicates in Visual Prolog. People who want to explore the possibilities of higher-order Logics, and functions may refer themselves to books like [Wadler & Bird], or in papers like [John Hughes].

The predicate `zip` has four arguments. The first argument is a two-place function; in listing 6.2, I provide two of such functions, but you can define others. The second and third arguments contain the lists to be zipped together. The fourth argument contains the output of the operation. The definition of `zip/4` is simple:

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

If either input list becomes empty, the first or the second clause stop the recursion, producing the empty list as output; otherwise, the third clause

clasps the heads of the two input lists, and proceeds to the rest of the input lists. I believe that this should be enough for a starting point.

```

implement 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).
```

Figure 6.2: Zip in Prolog

6.4 String operations

You will find a few examples of string operations below. These examples should be enough to show you how to deal with such data structures. You can find more operations using the help of 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).
```

Try to understand how `frontToken` works, because it is a very useful predicate. One uses it to break a string into tokens, in a process called lexical analysis. For instance, if you execute

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

you will get `T="break"`, and `R=" December"`.

One often has to transform a string representation of a term into an operational representation. In this case, s/he can use the function `toTerm`. In the example below, `Sx= stdio::readLine()` reads a string from the command line into `Sx`; then `toTerm` transforms the string representation into a real number; a call to `hasdomain(real, IX)` makes sure that `toTerm` will yield a real number.

```

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).

```

This scheme works even for lists, and other complex data structures, as you can see below. Unfortunately, `hasdomain` does not accept the *star* notation for list types; therefore, you must declare a list domain, or use a pre-defined domain, like `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).

```

You have learned how to transform a string representation of a data type to a Prolog term. Now, let us see how to do the inverse operation, i.e., to transform a term to its string representation. If you need the standard representation of a term, then you can use the predicate `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).

```

If you want to format numbers and other simple terms nicely into a string, you can use the `string::format` format function; an example is given below.

```
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).
```

In the above example, the format `"%8.3f\n"` means that I want to represent a real number, and a carriage return; the field width is 8, and the number must be represented with three digits of precision. The format `"%010d\n"` requires an integer right justified on a field of width 10; the field empty positions must be filled in with zeroes. The format `"%-10d\n"` specifies the representation of a left justified integer on a field of width 10; the minus sign assures left justification; right justification is the default. The format `"%10.1e\n"` specifies scientific notation for real numbers. Below, you can see the output of the example.

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

Below, you will find a listing of the data types accepted by the format string. Note that when converting real values into a text representation they are truncated and rounded with 17 digits, unless another precision is specified.

- f Format reals in fixed-decimal notation (such as 123.4).
- e Format reals in exponential notation (such as 1.234e+002).
- g Format reals in the shortest of f and e format.
- d Format as a signed decimal number.
- u Format as an unsigned integer.

- x Format as a hexadecimal number.
- o Format as an octal number.
- c Format as a char.
- B Format as the Visual Prolog binary type.
- R Format as a database reference number.
- P Format as a procedure parameter.
- s Format as a string.

6.4.1 Useful String Predicates

In this section, you will find a listing of string predicates that you may find useful in the development of applications; `adjustBehaviour`, `adjustSide`, and `caseSensitivity` can be considered as having the following definitions:

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,

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
-----
```

```
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.
```

```
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).
        /* Convert string characters to lowercase */

toUpperCase : (string Source) ->
        string Output procedure (i).
        /* Convert string character to uppercase */

trim : (string Source) ->
        string Output procedure (i).
        /* Removes both leading and trailing
        whitespaces from the string. */

trimFront : (string Source) ->
        string Output procedure (i).
        /* Removes leading whitespaces from the string. */

trimInner : (string Source) ->
        string Output procedure (i).
        /* Removes groups of whitespaces from the Source line. */

trimRear : (string Source) ->
        string Output procedure (i).
        /*Removes trailing whitespaces from the string. */
```

```

/*****
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

```

6.5 Notions of Logics: Grammar for Predicates

Let us define informal grammar rules for the Predicate Calculus.

- The symbol $p(t_1, t_2, \dots, t_{n-1}, t_n)$ is called literal. In a literal, p/n is a n -place symbol distinct from all other symbols of the language, and t_i are terms; a term can be
 - Functional symbols, i.e., symbols of the form $f(x_1, x_2 \dots x_n)$. A functional symbol may have arity zero, which means that a, x, b , and $rosa$ can be functional symbols.
 - Numerals, strings, and characters can also be terms.

A formula consisting of a sole literal is called an atomic formula.

- If t_1 and t_2 are terms, then $t_1 = t_2$, $t_1 > t_2$, $t_1 < t_2$, $t_1 \geq t_2$, and $t_1 \leq t_2$ are literals, and one can use them to create formulæ.

- If P and Q are formulae, then $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, and $P \equiv Q$ are formulae. Besides this, if P is a formula, and X is a variable, $\forall X(P)$ (for any X , P is true) and $\exists X(P)$ (there exists X , that makes P true) are both formulae.
- A variable introduced by the symbol \exists is called existential variable. Therefore, the variable X in the formula

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

is an existential variable. One can remove all existential variables of a formula, by substituting constants for them. In the example, one can substitute π for X :

$$\sin(X) = 0$$

Clausal form Any sentence of the predicate can be re-written into the clausal form, i.e., as a disjunction of conjunctions, with no existential quantifier(\exists), and no universal quantifier inside parentheses. Since $A \vee B$ is a disjunction, and $A \wedge B$ is a conjunction, any formula can be re-written as

$$\forall X \forall Y \forall Z \dots ((L_{11} \vee L_{12} \vee L_{13} \dots) \wedge \\ (L_{21} \vee L_{12} \vee L_{13} \dots) \wedge \\ (L_{31} \vee L_{12} \vee L_{13} \dots) \wedge \dots)$$

where L_{ij} are literals, or negations of literals. We have already seen how to remove existential quantifiers from a formula. Let us examine the remaining necessary steps to obtain the clausal formula.

1. Remove implications, and equivalences.

$$\begin{array}{ll} \alpha \rightarrow \beta & \neg \alpha \vee \beta \\ \alpha \equiv \beta & (\alpha \wedge \beta) \vee (\neg \alpha \wedge \neg \beta) \end{array}$$

2. Distribute negations. Use truth tables to prove that

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

and that

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

Then, use these results to make the following re-writing operations:

$$\begin{array}{ll} \neg(\alpha \vee \beta) & \neg \alpha \wedge \neg \beta \\ \neg(\alpha \wedge \beta) & \neg \alpha \vee \neg \beta \end{array}$$

3. Thoralf Albert Skolem (1887-1963) is a Norwegian mathematician, who invented the method of removing existential variables named after him. Skolemization substitutes constants for existential variables. E. g.

$$\frac{\exists X(p(X))}{\forall X(\text{man}(X) \rightarrow \exists W(\text{loves}(W, X)))} \left| \frac{p(s_0)}{\forall X(\text{man}(X) \rightarrow \text{loves}(w(X), X))} \right.$$

4. Distribute conjunctions using the following re-write rules:

$$\begin{array}{ll} (\alpha \wedge \beta) \vee \gamma & (\alpha \vee \beta) \wedge (\beta \vee \gamma) \\ \neg(\alpha \wedge \beta) & \neg\alpha \vee \neg\beta \end{array}$$

Chapter 7

Grammars

The philosopher Wittgenstein, in his *Tractatus Logicus Philosophicus*, expressed the opinion that the World is a state of things. Things and objects have attributes, like position, integrity¹, color, etc. The set of values that these attributes take is called state. For example, according to Euclid, a point has no attribute other than position. Then, the state of a point can be given by its coordinates.

The first sentence of the *Tractatus* is: *The world is all, that is the case*. The last one is: *Whereof one cannot speak, thereof one must be silent*. It is practically impossible to create a data structure to represent Wittgenstein's ideas on a computer. However, if ever devised by a linguistic genius, that data structure will certainly have three constructors: the *world* node, the *case* node, and the *silence* node. Here is a semantic class whose sole use is to export the semantic domain of the *Tractatus*:

```
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
```

There is a novelty in this listing: The use of constructors of data structures. A constructor has the shape of a function, i.e., it has a functor and arguments.

¹Is it whole, or broken?

For instance, the constructor `case/2` has two arguments, a grammatical category and a string representing a token.

7.1 Parsing grammar

Acting is to change the attributes of the objects. Thus, the result of an act is a change of state. For instance, if a predicate parses a list of words, the world will undergo a change of state. In the final state, the parser has consumed part of the list. To understand this point, let us create a *german* class to parse the first sentence of the Tractatus. Using it as a model, you can write a more powerful class to parse the whole book. It is a cinch, because the Tractatus is very short.

```
%File: german.cl
class german
  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

%File: 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), S, End) :-
    article(T1, S, 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

```

If you pay attention to the declaration of `article`, you will notice that its flow pattern is (o, i, o), id est, it inputs a list of words, and outputs a parse tree² and the partially consumed input list. The definition of `noun` works exactly like `article`. An example is worth one thousand words; let us feed ["die", "Welt", "ist", "alles"] to `article/3`.

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

This call will match the first clause of `article/3`, with

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

The output will be `T1= case(art, "")`, and `S1= ["Welt", "ist", "alles"]`. Next, let us feed `S1= ["Welt", "ist", "alles"]` to `noun/3`.

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

The above call will match the first clause of `noun/3`, with `R=["ist", "alles"]`. The output will be `T2=case(nom, "Welt")`, and `End=["ist", "alles"]`. The definition `nounphr` makes these two calls in succession, parsing a noun phrase like ["die", "Welt"]; `copula/3`, and `phr/3` behaves like `nounphr`.

7.2 Generating grammar

While the German parse takes a sentence, and outputs a node, the English parse, that has been given below, takes a node, in order to generate a sentence. One could say that the node is the meaning of the sentence. The German parse consumes a German sentence, and outputs its meaning. The English parser inputs the meaning, and produces an English translation.

²If you do not know these linguistic terms, take a look at a book about natural language processing, or compiler construction, whichever pleases you most.

```

%File: 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

% File: 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

```

Horn clauses that yields a phrase according to the grammar of a given language are called production rules. In the *english*-class, `article/2`, `noun/2` and `nounphr` are examples of production rules.

Each English production rule outputs a list of words. Keeping this in mind, let us take a look at the production rule for `phr/2`.

```

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

```

The second argument of the head uses the `list::append(Start, S1)` function to concatenate a noun phrase with a copula, yielding a phrase.

To implement and test this program, create the *tratactus* console project. Insert the classes *german*, *english* and *semantic* into the project tree. Do not forget to untick the *Create Objects* box. Add the given code to the appropriate class files. The testing program is given in figure 7.1. After compiling it, if you want to test it from inside VIP IDE, use the option *Build/Run in Window*. By the way, if you do not want to see a lot of errors, introduce the `semantic` class first of all; build the application; then introduce the `german` class, and build the application again; the next step is to add the `english` class, and rebuild the application. Finally, you can insert the code for the `main` class, and build the application for the last time.

7.3 Why Prolog?

One of the goals of a high level computer language is to increase the reasoning abilities of programmers, in order to achieve higher productivity, and artificial intelligence. Although there is a consensus that Prolog meets these goals better than any other language, there are several controversies surrounding these terms. Therefore, let us see what we mean by higher productivity, and artificial intelligence.

Higher productivity. A productive programmer delivers more functionality in shorter time. In a software project, there are parts that take most of the programming effort. For instance, data structures like lists and trees are needed everywhere, but are difficult to code and maintain in languages like C, or Java. In Prolog, there isn't anything easier than dealing with lists, or trees. Parsing is another difficult subject that one cannot live without; by the way, parsing is what you have done in order to implement the small English grammar that you have learned in this chapter. The design of the application logic is also very unwieldy, but I do not need to elaborate on this subject, because Prolog has Logic even in its name.

Artificial Intelligence makes your programs highly adaptable to change, and more user friendly. Prolog, alongside with Lisp and Scheme, is very popular among people who work with Artificial Intelligence. This means that you will find Prolog libraries for almost any AI algorithm.

7.4 Examples

The example of this chapter (*tratactus*) shows how one could write a program that translates philosophical German into English.

```

/*****

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).
```

Figure 7.1: Die Welt ist alles, was der Fall ist.

7.5 Notions of Logics: Natural deduction

Both [Gries] and [Dijkstra] propose that one uses Logics to build correct programs, or to prove that a given program is correct. Many people say that these authors are dated, because the languages (Pascal-like structured languages) used in their books are dated. This is a stupid thing to say, because the methods they propose are based on Predicate Calculus, that is unlikely to become dated; besides this, one can use the Dijkstra/Gries system to program in a modern language, like Visual Prolog, Clean, or Mercury. In fact, since [Dijkstra] method is based on Predicate Calculus, one can apply

it more easily to a Prolog dialect than to a Pascal dialect. In fact, [Cohn] and [Yamaki] have devised a very simple procedure to do it:

- Write the specification of your algorithm in clausal form.
- Transform the specification into a Prolog programming by introducing control mechanisms, like cuts, and **if-then-else** structures.
- Prove that the introduction of control mechanisms does not change the semantic of the algorithm.

To prove that a program is correct, [Gries] proposes that one specifies the program with Predicate Calculus, and prove its correctness using a method known as Natural Deduction. In Logics, natural deduction is an approach to proof theory that tries to provide a more natural model for logical reasoning than the one proposed by Frege and Russel.

Epistemology says that a judgment is something that one can know. An object is evident if one in fact knows it. Something is evident if one can observe it; for instance, global warming is evident because scientists have records that shows it happening. Many times, evidence is not directly observable, but rather deduced from basic judgments. The steps of deduction is what constitutes a proof. There are many kinds of judgement:

Aristotelic judgement: *A is true, A is false and A is a proposition.*

Temporal logic: *A is true at time t*

Modal logic: *A is necessarily true or A is possibly true*

We have seen how to proffer a judgement of the kind *A is a proposition*: The grammar rules given on pages 30 and 76 provide inference rules for deciding whether a string of symbols is a valid formular of Propositional Calculus, or Predicate Calculus. However, the given grammar for Propositional Calculus is informal. In a formal treatment, grammar rules are called *formation rules*, and are represented by $\bigwedge F$. Then the inference rules to recognize a well formed formula of the propositional calculus can be written thus:

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

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

In the Natural Deduction system, one has formation rules for the appropriate calculus, and one rules of insertion, and one rule of elimination for each

one of the logic connectives present in the calculus one is working with. If one is working with Predicate Calculus, there are insertion, and elimination rules for negation ($\neg P$), conjunction ($P \wedge Q$), disjunction ($P \vee Q$), equivalence ($P \equiv Q$), implication ($P \rightarrow Q$), universal quantification ($\forall X(P)$), and existential quantification ($\exists X(P)$). In an inference rule, one calls prosthesis the expressions above the line, and apodosis what comes below the line. The turnstile symbol \vdash is not a part of a well-formed formula: but a meta-symbol. One can see $P \vdash Q$ as meaning *P entails Q*, and $\vdash Q$ as *Q is a theorem*. Here are a few inference rules of the Natural Deduction system that one can use with Predicate Calculus sentences in the clausal form:

$$\begin{array}{ll}
\text{and Insertion} & \wedge / I : \frac{E_1, E_2, \dots E_n}{E_1 \wedge E_2 \dots E_n} \\
\text{and Elimination} & \wedge / E : \frac{E_1 \wedge E_2 \dots E_n}{E_i} \\
\text{or Insertion} & \vee / I : \frac{E_i}{E_1 \vee E_2 \dots E_{n-1} \vee E_n} \\
\text{or Elimination} & \vee / E : \frac{E_1 \vee E_2, E_1 \rightarrow E, E_2 \rightarrow E}{E} \\
\text{implication Insertion} & \rightarrow / I : \frac{E_1, E_2 \vdash E}{E_1 \wedge E_2 \rightarrow E} \\
\text{MODUS PONENS} & \rightarrow / E : \frac{E_1 \rightarrow E_2, E_1}{E_2} \\
\text{not Insertion} & \neg / I : \frac{E \vdash E_1 \wedge \neg E_1}{\neg E} \\
\text{MODUS PONENS} & \neg / E : \frac{\neg E \vdash E_1 \wedge \neg E_1}{E}
\end{array}$$

Since we intend to work with the clausal form, you will not need the rules corresponding to the quantifiers. Let us play a little with our new toy. Prove that $p \wedge q \vdash p \wedge (r \vee q)$:

	$p \wedge q \vdash p \wedge (r \vee q)$	
1	$p \wedge q$	premise 1
2	p	$\wedge / E, 1$
3	q	$\wedge / E, 1$
4	$r \vee q$	$\vee / I, 3$
5	$p \wedge (r \vee q)$	$\wedge / I, 2, 4$

In the proof shown above, each step always contains the inference rule, and the lines on which it was used. For instance, line $p \wedge (r \vee q)$ because of the \wedge / I inference rule, and of lines 2 and 4.

Chapter 8

Painting

In this chapter, you will learn how to draw pictures using the `onPaint` event handler. Start by creating a project with a form, where you will draw things.

- Create a project:

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

- Create `paintPack` *package* attached to the root of the Project Tree.
- Put `canvas.frm` inside `paintPack`. Build/Build the application.
- Enable the option `File/New` of the application task menu, and add

clauses

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

to `TaskWindow/Code Expert/Menu/TaskMenu/id_file/id_file_new`. After this step, if you compile and run the program, whenever you choose the `File/New` option of the application task menu, the above predicate will be executed. The command `X= canvas::new(S)` creates a new `X` object of class `window`. This window will be the child of the task window `S`. The command `X:show()` sends a message to the object `X` to show the window.

What will you do next? If I were you, I would draw something on `canvas`, to create an interesting background.

8.1 onPainting

Whenever a window needs painting, it calls the event handler `onPainting`. Therefore, if you add instructions to `onPainting`, they will be executed.

- Create a class `dopaint` inside `paintPack`. Untick *Creates Objects*.
- Add the code below to `dopaint.cl` and `dopaint.pro`.

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

%File: 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
```

- Build/Build the application to add the class *dopaint* to the project.
- Go to the *Properties*-dialog of the `canvas.frm`, press the *Events*-button, and add the following code to the *PaintResponder*:

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

If the `canvas.frm` is closed, click on its branch in the Project Tree.

As I said before, whenever the window needs painting, it calls `onPaint`, that calls `dopaint::bkg(GDIObj)`. The class `dopaint` has a `bkg(GDIObj)` method, that draws things on the window pointed to by `GDIObj`.

Let us understand what the method `bkg(W)` does. As you can see, `P` holds a list of points. Every point is defined by its coordinates. For instance, `pnt(0, 0)` is a point on `(0, 0)`. When `bkg(W)` calls `W:drawPolygon(P)`, it sends a message to the object `W`, asking it to draw a polygon whose vertices

are given by the list `P`. Compile the program, and check whether it will act exactly as said.

Let us improve method `bkg(W)`. It draws a white triangle on the pane of `canvas`. To make the triangle red, one must change the painter's brush. A brush is a two argument object.

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

Colors are represented by integers that specify the amount of red, green, and blue. As you probably know, you can get a lot of colors by combining red, green, and blue palettes. Let us represent the integers in hexadecimal basis. In hexadecimal, numerals have fifteen digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The red color is given by the integer `0x000000FF`, where `0x` shows that we are dealing with the hexadecimal basis. Representing colors by integers is fine, but you have the option of using identifiers. For instance, `color_Red` represents the red color, as you can guess. There are also identifiers for patterns:

- `pat_Solid`: Solid brush.
- `pat_Horz`: Horizontal hatch.
- `pat_Vert`: Vertical hatch.
- `pat_FDiag`: 45-degree downward hatch.

Here is a modification of `bkg(W)` that provides a red triangle:

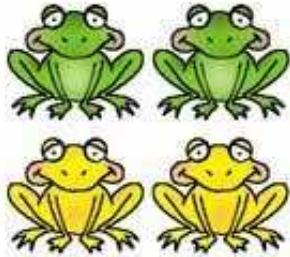
```
bkg(W) :- Brush= brush(pat_Solid, color_Red),
        P= [pnt(0,0), pnt(10,80), pnt(150, 100)],
        W:setBrush(Brush), W:drawPolygon(P).
```

Finally, here is a version of `bkg(W)` that draws an ellipse, and clears a green rectangle inside it.

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

The last thing that you will learn in this chapter is how to add a bmp picture to your window. Here is how to modify method `bkg(W)`:

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



The first step is to load the picture from a file. This is done by using the predicate `pictLoad/1`. The next step is to draw the picture. In the example, the picture `P` is put at the point `pnt(10, 10)`.

One often needs to draw a small picture on top of another, removing the background of the small picture. For instance, you may want to put an eagle among the frogs of the previous examples.



The first step is to draw a mask of the eagle using `rop_SrcAnd`. The mask shows a black shadow of the eagle on a white background. The next step is to draw the eagle itself using `rop_SrcInvert`. The eagle must fit perfectly well onto its mask.

The program that put the eagle among the frogs is given below, and is implemented in directory *paintEagle*, of the examples.

The *paintEagle* project is exactly alike *painting*, except for the definition of `bkw(W)`.



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

%File: 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 Custom Control

We have been painting directly on the canvas of a form. However, it is a good idea to create a custom control, and use it for painting.

- Create a new project.

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

- Choose the option **File/New** in **New Package** from the IDE task menu. At the *Create Project Item* dialog, select the **Control** tag from the left hand side pane. The name of the new custom control will be **canvas**.

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

Leave the *Parent Directory* field blank. Press the *Create* button.

- A prototype of a canvas control will appear on the screen. You can close it to avoid cluttering the IDE. Build the application to include the new control.
- Add a new form to the project tree. Choose **File/New** in **New Package** from the task menu, and select the *Form* tag from the left hand side pane of the *Create Project Item* dialog. Let the new form's name be **greetings**. After pressing the *Create*-button, you will obtain a prototype of the **greetings**-form.
- Choose the key-symbol that appears on the *Controls*-dialog. Click on the pane of the **greetings**-form. The system will show you a menu containing a list of custom controls; the **canvas**-control is the first one.



- Remove the *Ok*-button from the **greetings**-form, and replace a **hi_ctl**-button for it, as in the figure 8.1.

- Build the application.
- Go to the *Properties*-dialog of the `hi_ctl`-button, and add the following snippet to the *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!").
```

- Build the application again. Enable the `TaskMenu.mnu` → `File/New` choice. Add

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

to `TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new`. Build and execute the application. Choose the option `File/New`, and a new form will appear. Whenever you click on the canvas control of the new form, it will print warm greetings.

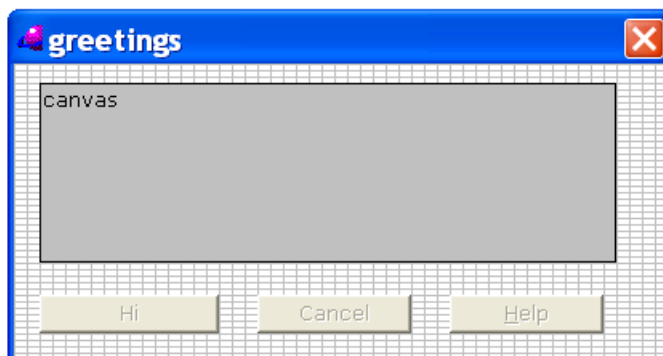


Figure 8.1: A form with a custom control

8.3 Notions of Logics: Resolution principle

The most important inference rule for Prolog programmers is the resolution principle, proposed in 1963 by [Robinson]:

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

To prove this principle, we will need two syllogisms. One of them is the first distributive law, that says

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

There is also a second distributive law, that we are not going to use in the proof of Robinson's resolution principle.

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

Let us take a concrete case of these laws. Lysias was an Athenian lawyer that lived in the time of Socrates. In fact, he was the son of that Kephalos who entertains Socrates in Plato's Republic. He was a metic, or resident alien; as such, he did not have civil rights; nevertheless his family was very rich, as Plato explains in the Republic. His defences and prosecutions are very interesting. For instance, during the dictatorship of the thirty tyrants, the rulers of country decided to take 8 rich metics, and two poor ones, kill them, and rob them; Lysias and his brother Polemarchus were among those condemned to death. However, Lysias managed to flee, and returned to his country to help restore the democracy, and to prosecute one of the tyrants that killed his brother.

In another occasion, Lysias defended a man who killed his wife's lover. To understand his defence, you need to know a few facts about the Greek society. Man and woman lived in different quarters: Man lived in the androecium, and women lived in the gynoecium. You probably remember that angiosperms also have androecia (housing for males) and gynoecia (housing for females); the reason is that Carl von Linné named these parts of the vegetal physiology after the housing arrangements of the Ancient Greeks. The other Greek custom was the dowry. When the woman left her father's house, she would take a handsome amount of money to the house of her husband. Therefore, no man was inclined to divorce his wife, because this would entail returning the money to her family.

When Eratosthenes discovered that his wife was cheating him, he decided to kill her lover, for the divorce would mean losing her rich dowry. He feigned a trip to his farm in the outskirts of the city, but returned on time to kill the

villain on the unfaithful woman's bed. However, the prosecutor claimed that the ravenous husband killed the rival on the domestic altar, to where he fled seeking the protection of Zeus. If Eratosthenes killed the man on the altar, the jury would condemn him to death; if he killed him on the bed, he would be acquitted of all charges. Lysias's defense of his client was based on the following argument:

*He was on the altar or he was on the bed, and he was on the altar,
or he was killed.*

The prosecutor accepted the argument. Then Lysias simplified it, using the first distributive law:

*He was on the altar, or he was on the bed and he was killed. Since
you know that he was killed, he was on the bed.*

Eratosthenes was acquitted. After this long digression, let us prove the resolution principle.

	$p \vee q, \neg p \vee r \vdash q \vee r$	
1	$p \vee q \vee r$	\vee/I premise 1
2	$\neg p \vee q \vee r$	\vee/I premise 2
3	$(p \vee q \vee r) \wedge (\neg p \vee q \vee r)$	$\wedge/I, 1, 2$
4	$(q \vee r) \vee (p \wedge \neg p)$	first distributive law, 3
5	$q \vee r$	MODUS TOLLENDO PONENS

Chapter 9

Data types

Most modern languages work with strongly typed data. This means that the compiler checks whether the data fed to a predicate or a procedure belong to the right type. For instance, the arithmetic operations ($x + y$, $x \times y$, $a - b$, $p \div q$) work on integer, real or complex numbers. Thus, the compiler makes sure that your program will feed numbers, and nothing else, to these operations. If there is a logical mistake in your code, and you try to divide a string by an integer, the compiler balks. I am sure that you are saying to yourself that this is the only reasonable thing to do. I agree. However, not all language designers had the common sense to hardwire type checking into their compilers. For instance, Standard Prolog does not check type until the faulty code is executed. The program fails when it is already in the computer of the final user.



9.1 Primitive data types

Visual Prolog has a lot of primitive data types:

integer : 3, 45, 65, 34, 0x0000FA1B, 845. By the way, 0x0000FA1B is an hexadecimal integer, i.e., a numeral expressed in basis 16, which has the following digits: 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, etc.

string : "pencil", "John McKnight", "Cornell University", etc.

symbol : "Na", "Natrium", "K", "Kalium", etc.

You have seen that symbols look a lot like strings: Both are a sequence of UNICODE characters. However, they are stored differently. Symbols

are kept in a look-up table and Prolog uses their addresses in the table to represent them internally. Therefore, if a symbol occurs many times in your program, it will use less storage than a string. Below, there is a very simple example of using real numbers, symbols, and strings. This example is taken from chemistry.

```

/*****
Project Name: primetype1
UI Strategy:  console
*****/
% File main.pro
implement main
    open core
constants
    className = "main".
    classVersion = "primetype1".
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 Sets

A set is a collection of things. You certainly know that collections do not have repeated items. I mean, if a guy or girl has a collection of stickers, s/he does not want to have two copies of the same sticker in his/her collection. If s/he has a repeated item, s/he will trade it for another item that s/he lacks in his/her collection.

It is possible that if your father has a collection of Greek coins, he will willingly accept another drachma in his set. However, the two drachmas are not exactly equal; one of them may be older than the other.

9.3 Sets of numbers

Mathematicians collect other things, besides coins, stamps, and slide rules. They collect numbers, for instance; therefore you are supposed to learn a lot about sets of numbers.

\mathbb{N} is the set of natural integers. Here is how mathematicians write the elements of \mathbb{N} : $\{0, 1, 2, 3, 4 \dots\}$.

\mathbb{Z} is the set of integers, i.e., $\mathbb{Z} = \{\dots - 3, -2, -1, 0, 1, 2, 3, 4 \dots\}$.

Why is the set of integers represented by the letter \mathbb{Z} ? I do not know, but I can make an educated guess. The set theory was discovered by Georg Ferdinand Ludwig Philipp Cantor, a Russian whose parents were Danish, but who wrote his *Mengenlehre* in German! In this language, integers may have some strange name like *Zahlen*.

You may think that set theory is boring; however, many people think that it is quite interesting. For instance, there is an Argentinean that scholars consider to be the greatest writer that lived after the fall of Greek civilization. In few words, only the Greeks could put forward a better author. You probably heard Chileans saying that Argentines are somewhat conceited. *You know what is the best possible deal? It is to pay a fair price for Argentines, and resell them at what they think is their worth.* However, notwithstanding the opinion of the Chileans, Jorge Luiz Borges is the greatest writer who wrote in a language different from Greek. Do you know what was his favorite subject? It was the Set Theory, or *Der Mengenlehre*, as he liked to call it.

Please, my Argentinian friends, no hard feelings. After all, your country ranks fourth in my list of favorite countries, after Greece, France, and Paraguay. The high position of Paraguay in my list is solely due to the fact that José Asunción Flores was Paraguayan; for those who do not know José

Asunción Flores, listen to his song *India*, and I am sure that you will put Paraguay in your map. As for Argentina, it is the fatherland of Bernardo Houssay, Luis Federico Leloir, César Milstein, Adolfo Pérez Esquivel, Carlos Saavedra Lamas, Jorge Luiz Borges, Alberto Calderon, Alberto Ginastera, José Cura, Dario Volonté, Verónica Dahl, Carolina Monard, Che Guevara, Diego Maradona, Juan Manuel Fangio, Osvaldo Golijov, Eva Peron, Hector Panizza, José Cura etc. Before I forget, José Asunción Flores lived in Buenos Aires. But let us return to sets.

When a mathematician wants to say that an element is a member of a set, he writes something like

$$3 \in \mathbb{Z}$$

If he wants to say that something is not an element of a set, for instance, if he wants to state that -3 is not an element of \mathbb{N} , he writes:

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

Let us summarize the notation that Algebra teachers use, when they explain set theory to their students.

Double backslash. The weird notation $\{x^2 || x \in \mathbb{N}\}$ represents the set of x^2 , *such that* x is a member of \mathbb{N} , or else, $\{0, 1, 4, 9, 16, 25 \dots\}$

Guard. If you want to say that x is a member of \mathbb{N} on the condition that $x > 10$, you can write $\{x || x \in \mathbb{N} \wedge x > 10\}$.

Conjunction. In Mathematics, you can use a symbol \wedge to say **and**; therefore $x > 2 \wedge x < 5$ means that $x > 2$ *and* $x < 5$.

Disjunction. The expression

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

means $x < 2$ **or** $x > 5$.

Using the above notation, you can define the set of rational numbers:

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

In informal English, this expression means that a rational number is a fraction

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

such that p is a member of \mathbb{Z} and q is also a member of \mathbb{N} , submitted to the condition that q is not equal to 0.

Visual Prolog does not have a notation for sets, but you can use lists to represent sets. For instance, choose the option *Project/New* from the task menu, and fill the *Project Settings* dialog thus:

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

Pay attention that we are going to use the console strategy, not GUI. Choose the option *Build/Build* from the task menu to put a prototype of class *zermelo* into the project tree. Edit *zermelo.pro* as shown below. Build the project again, and execute it using *Build/Run in Window*.

```
implement main
    open core

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

    run():-
        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).
```

You may wonder why I named the above program after the German mathematician Ernst Friedrich Ferdinand Zermelo. One reason could be that he was appointed to an honorary chair at Freiburg im Breisgau in 1926, which he resigned in 1935 because he disapproved of Hitler's regime. Another reason was that he invented the notation for set that we have been using.

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

In Zermelo's notation, this expression is written thus:

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

In plain English, Q is a list of pairs (X, Y) such that X belongs to $[1, 2, 3, 4]$, and Y belongs to $[1, 2, 3, 4, 5]$. The snippet

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

tells the computer to `write(Num, "/", Den, ", ")` for each tuple (Num, Den) that is member of the list Q .

The set of rational numbers is so named because its elements can be represented as a fraction like

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

where $p \in \mathbb{N}$ and $q \in \mathbb{N}$.

The next section is somewhat hard to digest. Therefore, if you want to skip it, feel free to do so. In the section about Real Numbers, I will summarize the important results, so you won't miss anything worth the effort of understanding a whole page of nasty mathematics.

9.4 Irrational Numbers

At Pythagora's time, Ancient Greeks claimed that any pair of line segments is commensurable, i.e., you can always find a meter, such that the lengths of any two segments are given by integers. The following example will help you understand the Greek theory of commensurable lengths at work. Consider the square of figure 9.1.

If the Greeks were right, I would be able to find a meter, possibly a very small one, that produces an integer measure for the diagonal of the square, and another integer measure for the side. Suppose that p is the result of measuring the side



of the square, and q is the result of measuring the diagonal. The Pythagorean theorem states that $\overline{AC}^2 + \overline{CB}^2 = \overline{AB}^2$, i.e.,

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

You can also choose the meter so that p and q have no common factors. For instance, if both p and q were divisible by 2, you could double the length of the meter, getting values no longer divisible by 2. E.g. if $p = 20$ and $q = 18$, and you double the length of the meter, you get $p = 10$, and $q = 9$. Thus let us assume that one has chosen a meter so that p and q are not simultaneously even. But from equation 9.1, one has that q^2 is even. But if q^2 is even, q is even too. You can check that the square of an odd number is always an odd number. Since q is even, you can substitute $2 \times n$ for it in equation 9.1.

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

Equation 9.1 shows that q is even; equation 9.2 proves that p is even too. But this is against our assumption that p and q are not both even. Therefore, p and q cannot be integers in equation 9.1, which you can rewrite as

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

The number $\sqrt{2}$, that gives the ratio between the side and the diagonal of any square, is not an element of \mathbb{Q} , or else, $\sqrt{2} \notin \mathbb{Q}$. It was Hypasus of Metapontum, a Greek philosopher, who proved this.

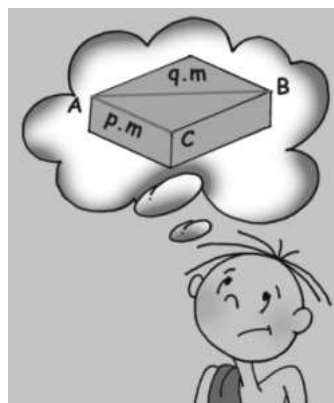


Figure 9.1: A thinking Greek

The Greeks were a people of wise men and women. Nevertheless they had the strange habit of consulting with an illiterate peasant girl at Delphi, before doing anything useful. Keeping with this tradition, Hyppasus asked the Delphian priestess—the illiterate girl—what he should do to please Apollo. She told him to measure the side and the diagonal of the god's square altar using the same meter. By proving that the problem was impossible, Hyppasus discovered a type of number that can not be written as a fraction. This kind of number is called irrational.

9.5 Real Numbers

The set of all numbers, integer, irrational, and rational is called \mathbb{R} , or the set of real numbers. In Visual Prolog, \mathbb{Z} is called **integer**, although the set **integer** does not cover all integers, but enough of them to satisfy your needs. A Visual Prolog real number belongs to the set **real**, a subset of \mathbb{R} .

If $x \in \mathbf{integer}$, Visual Prolog programmers say that x has type **integer**. They also say that r has type **real** if $r \in \mathbf{real}$. There are other types besides **integer**, and **real**. Here is a list of primitive types.

integer — Integer numbers between -2147483648 and 2147483647 .

real — Reals must be written with a decimal point: 3.4 , 3.1416 , etc.

string — A quoted string of characters: `"3.12"`, `"Hippasus"`, `"pen"`, etc.

char — Characters between single quotes: `'A'`, `'b'`, `'3'`, `' '`, `'.'`, etc.

9.6 Mathematics

I always thought it very hard to satisfy mathematicians, since they are so picky, and I have an appalling difficulty in keeping even a pretense of being formal. In particular, my definitions of the different numerical sets are not *rigorous* as most mathematicians would require. However, I have no intention of writing for an AMS journal, or making major contributions to number theory; I will be content if I am able to give non-mathematicians a working knowledge of the concepts involved. By working knowledge, I mean that I hope that my readers will be able to understand what a type is, and how type are related to Set Theory. As a bonus, I hope that students of literature will find Borges' books easier to digest, after reading my text; hence my Argentinian friends may forgive me for telling jokes about them.

9.7 Format

The method **format** creates nice string representations of primitive data types for output. For instance, it is nice to represent colors using hexadecimal integers. In this representation, the fundamental colors become `0x00FF0000` (red), `0x0000FF00` (green), and `0x000000FF` (blue). However, if you try to print these numerals using `stdio::write/n`, you will get them in the decimal basis. Using `string::format/n`, as shown below, you will get a printout of the primitive colors in hexadecimal.


```

/*****
Project Name: primtype2
UI Strategy:  console
*****/
% File 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).

```

The first argument of function `format` specifies how you want the printout. In the example, the argument is `"%x\n"`: You want a hexadecimal representation of a number, followed by a carriage return (`\n`). The arguments that follow the first show the data that you want to print. Here are a few examples of formatting:

```

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

```

The format fields specification is `%[-][0][width][.precision][type]`, where the hyphen (-) indicates that the field is to be left justified, right justification being the default; the zero before *width* pads the formatted string with zeros until the minimum width is reached; *width* specifies a minimum field size; *precision* specifies the precision of a floating-point numeral; finally *type* can be `f` (fixed point numeral), `e` (scientific notation for reals), `d` (decimal integer), `x` (hexadecimal integer), `o` (octal). Strings are represented as `%s`. You can add any message to the format specification; you can also add carriage returns (`"\n"`).

9.8 domains

You may need other data types besides the primitives provided by Visual Prolog. In this case, you need to specify the new data types that you will insert in your code. In previous chapters you have already learned how to declare domains; you also learned how to create complex data structures, like lists. Therefore, all we have to do in this section is to recall concepts already acquired, and to make a synthesis of what we have learned.

9.8.1 Lists

You have learned that a list is an ordered sequence of elements. You also know how to create list domains, and declare predicates on lists. For instance, here are a few list domains:

```
domains
    reals= real*. % a list of reals
    integers= integer*. % a list of integers
    strings= string*. % a list of strings
    rr= reals*. % a list of lists of reals
```

Once you have a domain declaration, you can use it like any primitive type. In the case of list, `domain` declaration is not necessary, since one can use list types like `real*`, `string*` or `integer*` directly in the predicate declaration, as in the program of figure 9.2.

9.8.2 Functors

Predicate calculus, a branch of Logics, has a data type called functional symbol. A functional symbol has a name, or functor, followed by arguments.

```

/*****
Project Name: newdomains
UI Strategy:  console
*****/
% File 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).

```

Figure 9.2: List declaration

The arguments are written between parentheses, and separated from each other by commas. To make a long story short, they have exactly the same notation as functions. Here are a few example of functional symbols:

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

The last example shows that a functional symbol can lack arguments. This said, let us see how to declare data types for a few functors. The example calculates the day of the week for any date between the years 2000 and 2099. It has the following domain declarations:

DOMAINS	Comments
year= integer.	<i>year</i> is synonym of <i>integer</i> . The use of synonyms makes things clearer.
day= sun; mon;tue; wed; thu; fri; sat; err.	This domain has many functional symbols without arguments. Each functional symbol is separated from the next by semicolon, and represents a day of the week.
month= jan;feb;mar;apr; may;jun;jul;aug;sep; oct;nov;dec.	Another domain (type) with more than one functional symbol.
month_code= m(month, integer).	Here, one has a functional symbol with two arguments.
numberOfDays= nd(integer); february.	Another type with two functional symbols; the first functional symbol has one argument; the second one has no argument at all.

```

/*****
Project Name: dayOfWeek
UI Strategy:  console
*****/
% File: main.pro
% This program calculates the day of the week
% for the years between 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, % Last digits of Year.
    monthCode(Month, m(_M, C), _), % Get month code.
    !,
    S= Y+Y div 4+C+MD,
    R= S mod 7, /* Day of the week as an integer
                  between 0 (sunday) and 6 (saturday) */
    dayOfWeek(R, D). % Get functional symbol of the day
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 Notions of Logics: Horn clauses

A Horn clause has at most one positive literal. For mathematicians, Horn clauses have interesting properties. However, their appeal is even greater for computer scientists, because they can be put in the following form:

$$\begin{aligned}
 H_1 &\leftarrow +L_{11} \wedge +L_{12} \wedge +L_{13} \dots, \\
 H_2 &\leftarrow +L_{21} \wedge +L_{12} \wedge +L_{13} \dots, \\
 H_3 &\leftarrow +L_{31} \wedge +L_{12} \wedge +L_{13} \dots, \dots
 \end{aligned}$$

where \leftarrow is the implication symbol, and $+L_{ij}$ is the negation of the negative literals. The proof that a clausal clause (page 77) $H \vee \neg T$ can indeed be put in the form $H \leftarrow T$ was given on page 30. The form $H \leftarrow T_1, T_2, T_3 \dots$ is appealing because it resembles informal rules that people use to express knowledge:

The rain is certain if swallows fly close to the ground.

From what you have seen, it is no wonder that the first trial to use Logic as programming language was focused on Horn clauses. However, the designers of Prolog faced the fact that Horn clauses are not complete enough for most applications. This difficulty was solved with the introduction of control mechanisms, like the cut, into the Horn clauses.

Chapter 10

How to solve it in Prolog

The title of this chapter is a homage to Helder Coelho, the first author of the best book on Prolog ever written: [HowToSolveItWithProlog]. This book was printed by the Laboratorio Nacional de Engenharia Civil, where Helder Coelho works in Portugal. Later, Coelho published *Prolog by Example* (see [Coelho/Cotta]), that is good too, but not as good as his first book.

Coelho's book is a collection of short problems, proposed and solved by great programmers and computer scientists. The whole thing is organized as a kind of FAQ. The problems are interesting, and the solutions are illuminating. More important, the book is also very amusing. I hope that it will see a new edition soon, since it shows the history of the creators of Logic Programming, their discoveries, their explorations.

All programs in this chapter will be console applications. Besides this, I will only provide the listing of the main class implementation. It is up to you to complete the red tape. For instance, if I give you the listing

```
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).
```

you must create a console project, whose name is `hello`.

10.1 Utilities

Verbal statement: Search all members of a list L. Logic program:

```

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

The example shows that you can define a predicate with different domains. For instance, there is a definition of *member/2* for **string***, and another for **integer***. The same thing happens to *test/1*.

The definition of *test/1* uses the predicate *fail* to backtrack, and print all solutions of the **nondeterm** predicate *member*. As a result, the program will print all elements of the list.

Verbal statement: Check whether U is the intersection of lists L1 and L2;
search the intersection of lists L1 and 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).
```

Verbal Statement: Specify the relationship *append/3* between three lists which holds if the last one is the result of appending the first two.

```

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).

```

This problem is an all time favorite. If you run the program, *test1/1* will find and print all ways of cutting the list ["a", "b", "c", "d"]. On the other hand, *test2/2* will concatenate two lists. Both predicates use *app/3*, which perform two operations, concatenation, and cutting down lists.

Verbal Statement: Write a program that reverses a list.

```

/*****
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).

```

This program is useful and illuminating. It is illuminating because it shows how to use accumulating parameters in a very efficient way. Assume that you feed the goal

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

to Prolog. Then the inference engine will try to make `rev([1, 2, 3], R)` equal to the head of one of the Horn clauses that are in the program. In the present case, it will succeed if it makes `L=[1,2,3]` in the clause

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

But if $L=[1,2,3]$, this clause becomes

$$\text{rev}([1,2,3], R) \text{ :- rev1}([1,2,3], [], R)$$

This process of making a goal equal to the head of a clause is called unification, and one says that $\text{rev}([1,2,3], R)$ unifies with

$$\text{rev}(L, R) \text{ :- rev1}(L, [], R), \text{ for } L=[1,2,3]$$

producing $\text{rev1}(L, [], R)$, that is equal to $\text{rev1}([1,2,3], [], R)$, since $L=[1,2,3]$. The product of unification is the tail of a Horn clause, because you must prove the tail, in order to prove the head. For example, consider the following Horn clause in English notation:

X is grandfather of Y if X is father of Z and Z is father of Y.

If you want to prove that X is grandfather of Y, you must prove that Y is father of Z, and Z is father of Y. In fewer words, one must prove the tail, in order to prove the head of a Horn clause. All this long discussion may be summarized thus

- $\text{rev}([1,2,3], R)$ unifies with

$$\text{rev}(L, R) \text{ :- rev1}(L, [], R), \text{ for } L=[1,2,3]$$

yielding $\text{rev1}([1,2,3], [], R)$.

- $\text{rev1}([1,2,3], [], R)$ unifies with

$$\text{rev1}([H|T], L1, L) \text{ :- rev1}(T, [H|L1], L) / H=1, L1=[], T=[2,3]$$

producing $\text{rev1}([2, 3], [1], L)$.

- $\text{rev1}([2, 3], [1], L)$ unifies with

$$\text{rev1}([H|T], L1, L) \text{ :- rev1}(T, [H|L1], L) / H=2, L1=[1], T=[3]$$

yielding $\text{rev1}([3], [2,1], L)$.

- $\text{rev1}([3], [2,1], L)$ unifies with

$$\text{rev1}([H|T], L1, L) \text{ :- rev1}(T, [H|L1], L) / H=3, L1=[2,1], T=[]$$

yielding $\text{rev1}([], [3,2,1], L)$.

- $\text{rev1}([], [3,2,1], L)$ unifies with $\text{rev1}([], L, L) \text{ :- !}$ for $L=[3,2,1]$, producing the result, that is, $L=[3,2,1]$.

Laplace used to say that Newton was not only the greatest scientist of all times and lands, but also a very lucky man, for he was born before Laplace. Of course, Le Marquis would discover the Celestial Mechanics before Newton, if he had the chance of doing so. I suppose that you would write the quick sort algorithm before Van Emden, if he was not lucky enough to put his hands on a Prolog compiler before you. The quick sort algorithm was invented by Tony Hoare, that wanted a fast method for sorting lists and vectors. This was a time when B+Trees did not exist, and the only way to perform a fast search in a list of customers was to have it sorted.

Let us see how Van Emden explained his algorithm to his friend Coelho. First of all, he implemented a *split/4* algorithm, that splits a list L in two sublists: S(mall) and B(ig). The S sublist contains all of L that are smaller of equal to P(ivot). The B sublist contains the elements that are larger than P. Here is how Van Emden implemented the *split/4*:

```
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).
```

If you run this program, the computer will print

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

which is what it should do. Now, let us understand the idea behind the quick-sort algorithm. Let us assume that you have the list `[5, 3, 7, 4, 5, 8, 2]`, and you want to sort it.

- Take the first element of `L= [5, 3, 7, 4, 5, 8, 2]` as pivot, and split it in `Small=[3,4,5,2]` and `Big=[7,8]`.
- Sort `Small` using the same algorithm; you will get

`Sorted_Small= [2, 3, 4, 5]`

- Sort `Big` and get `Sorted_Big= [7,8]`.
- Append `Sorted_Small` to `[Pivot|Sorted_Big]`. You will get the sorted list: `[2,3,4,5,5,7,8]`.

This algorithm needs a predicate that appends one list to another. You can use `app/3`, that you learned on page 112. However, that algorithm is `nondeterm`, which we do not need, since we do not need the functionality of cutting a list in two; all we need is a vanilla concatenation. Therefore, you can insert a cut after the first clause of `app/3`.

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

Declare `app` as a procedure with two input arguments, and one output.

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

To test the algorithm, we are going to use a list of integers. Of course, it will be more useful with a lists of strings.

```

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).

```

Prolog was invented by Alain Colmerauer, a French linguist. His student Philippe Roussel implemented an interpreter. It seems that Roussel's wife gave the name *Prolog* to the computer language. Warren was the man who implemented the first Prolog compiler. The problem below was proposed by this very same Warren.

Verbal Statement: The number of days in a year is 366 each four years; otherwise it is 365. How many days have years 1975, 1976, and 2000?

```
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).
```


Luis Moniz Pereira is a Portuguese who works with Artificial Intelligence. He was one of the first persons to use Prolog for Artificial Intelligence. He is co-author of the book *How to Solve it in Prolog*.

Verbal statement: Write a program for coloring any planar map with at most four colors, such that no two adjacent regions have the same color.

The problem is a classical generate and test program. In `test(L)`, the predicate calls

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

generate colors for the six regions of the map. Then, `test(L)` builds a the map as a list of neighboring countries:

```
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)],
```

Finally, the predicate `aMap(L)` checks whether L is an admissible map. It will be admissible if there is no two neighboring countries painted with the same color. If predicate `aMap(L)` fails, the proposed colors are not correct. Therefore, the program backtracks to `generateColor(X)` in order to get a new set of colors.

The four color problem is interesting for two reasons.

1. The generate and test scheme was one of the first Artificial Intelligence techniques.
2. There is a very famous theorem, that states:

Every map can be colored with four colors so that adjacent countries are assigned different colors.

This theorem was proved in 1976 by Kenneth Appel and Wolfgang Haken, two mathematicians at the University of Illinois.

```

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).

```

Verbal statement: Write a program for the game of Nim. A position of the game of Nim can be visualized as a set of heaps of matches; we will represent each heap as a list of integers.

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

Therefore, the set of heaps will be a list of lists; e.g.

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

Two players, you and the computer, alternate making moves. As soon as a player is unable to make a move, the game is over, and that player has lost. A move consists of taking at least one match from exactly one heap. In the example, if you take three matches from the third heap, you make a valid move, and the board becomes

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

where you have removed three matches from the third heap. To implement this project, we will use a programming technique called *incremental development of systems*. First, you will implement and test a program that appends two lists. Since you are going to use this program both to split and to concatenate lists, you need to test both possibilities. In fact, if you run the first program (next page), you will get

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

The first line is the result of concatenating two sets of heaps; the lines that follow shows the many possibilities of splitting a list of heaps. Then the first program seems to be working properly.

```

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).

```

The next step is to write a predicate that takes at least one match from a heap; of course, it can take more than one match. After removing one or more matches from the heap, `takesome/3` inserts the modified heap into a set of heaps. If you test the program, it will print the following result:

```

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

```

NB: the first clause of `takesome/3` makes sure that the predicate will not insert an empty heap into the set of heaps.

```

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).

```

Now, you are ready to generate all possible moves from a given position. If you run the test program, it will produce the following screen:

```

Possible moves from [[1,1,1],[1,1]]:
[[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([], 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).

```

```

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([], 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).

```

The heart of this program is the predicate

```
winningMove(X, Y) :- move(X, Y), not(winningMove(Y, _)).
```

If there is a winning strategy, this amazing one line predicate will find it!

You will find below an example of a game against the computer. As you can see, if there is a chance of winning, the machine will take it.

```

[[1,1,1],[1,1]]
Your move: [[1], [1,1]]
My move: [[1],[1]]
Your move: [[1]]
My move: []
I win

```

To implement the final program, you will create a class for the game itself. Let the name of this class be `nim`. Then, the main program is given below.

```

% File: 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).

```

The class interface exports the method `game/1`, and the domains that you need to describe a game position. It is defined below.

```

% File: nim.cl
class nim
    open core
domains
    ms= integer*.
    hs= ms*.
predicates
    classInfo : core::classInfo.
    game:(hs) determ (i).
end class nim

```



```

% File: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

```

a tree, that is called search tree. When it is the computer's turn, it generates states of minimum value for the opponent. On the other hand, it tries to follow tree branches that leads to a winning position. In a simple game like Nim, the computer is able to find a path to sure victory. In more complex games, like chess, this is not always possible; one must interrupt the search before being sure of winning the game. In this case, game strategists design functions that assign values to a position even when it is not a sure win, or defeat.

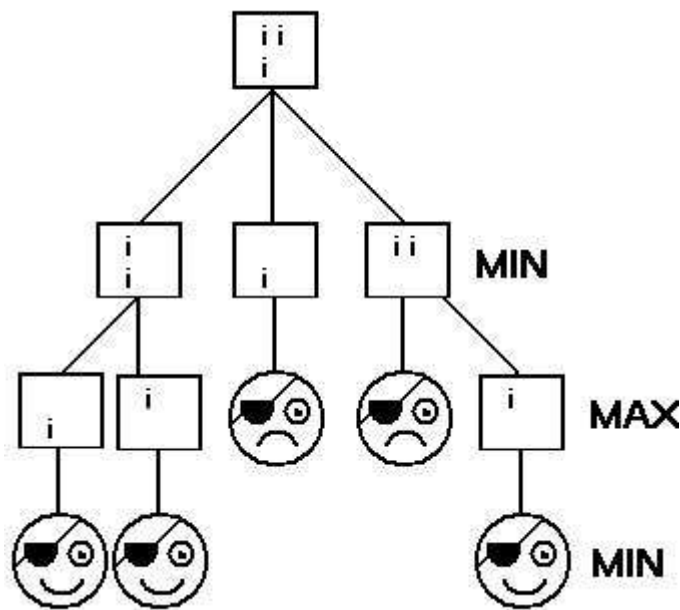


Figure 10.1: Minmax tree

Chapter 11

Facts

A fact is a Horn clause without a tail. Facts can be asserted, modified, or retracted dynamically, at runtime. An example will help you to understand why *facts* are necessary in Prolog. Let us assume that you want to create a small database of publications. When you have a new book, you want to add it to the database.

```
addItem(journal("AI in focus", "MIT Press")),
```

The predicate `addItem/1` can be defined thus:

```
class predicates
  addItem:(volume).
clauses
  addItem(V) :-
    num := num+1,
    assert(item(V, num)).
```

The execution of `addItem(journal("AI", "AldinePress"))` adds the clause

```
item(journal("AI", "AldinePress")).
```

to the database, and increments the variable `num`. The declaration

```
class facts - bib
  num:integer := 0.
  item:(volume, integer) nondeterm.
```

creates the fact database `bib`, with a variable *num* and a predicate *item/2*; the predicate is `nondeterm`, and the variable is single.

The domain *volume*, that will provide typing to the records of this small database, is defined as

```
domains
    name= string.
    author= n1(name); n2(name, name); etal(name).
    publisher= string.
    title= string.
    volume= journal(title, publisher); book(title, author).
```

The program below shows how to define *facts*, and how to save the fact database to a file.

```
% File 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).

```

After creating the database and saving it to a file, you can use it in another program. To test this possibility, create a console project *factread*, and add the following program to it:

```

% File: 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).

```

You need to transfer the file

```
bibliography.fac
```

created by *facttest* to *factread\exe*. Then, run the program

```
factread.exe.
```

You will see that the program will read the database, and print it.

11.1 Class file

You have used class *file* to save facts to a file. There are many other useful predicates in class *file*, that you will learn in this section.

11.1.1 Reading and writing a string

You often need to read a text file, do something to it, and write it back. In the past, when computers did not have much memory, it was necessary to read streams of characters. Nowadays, even a personal computer memory capacity is on the order of gigabytes. Therefore, the best approach to the problem is to read the whole file into a string, and use the powerful *string* class to do the processing. Prolog has two predicates to do this kind of job:

```

readString:(string FileName, boolean IsUnicodeFile)
            -> string String procedure (i,o).
writeString:( string Filename, string Source,
              boolean IsUnicodeFile) procedure (i,i,i).

```

Both predicates have a version that does not require specifying whether the file is unicode.

```

readString:(string FileName)
            -> string String procedure (i).
writeString:(string Filename, string Source) procedure (i,i).

```

```

%File: 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).

```

Figure 11.1: Reading a string from file

Figure 11.1 shows an example of using these predicates. In the example, one reads the string from a file, converts it to uppercase, and write the uppercase version to another file.

I believe that you will be able to figure out all the functionality of class *file* from the help menu. Therefore, I will not linger on this topic.

11.2 Constants

In order to learn how to use constants, let us build a form with a task menu; Visual Prolog, and most other languages as well, represent menu options by integer numbers; this representation is fine for the computer, but not as good for people, who will analyze the program. Therefore, Visual Prolog represent these handles to menu options by constants with meaningful names.

- Create a new project.

Project Name: janua

UI Strategy: Object-oriented GUI (pfc/gui)

By the way, JANUA is the Latin word for door; *window* is a small door, or JANELLA; the door of the year is MENSIS JANUARIUS, or *january*.

- Create a New *Form* in New Package; let the form be named `txtWin`. From the *Properties*-dialog, add a TaskMenu to `txtWin`.



- Build the project. Use the *Events*-tab of the *Properties*-dialog to introduce the following snippet into the *MenuItemListener*:

```

clauses
  onMenuItem(_Source, M) :-
    Item= resourceIdentifiers::id_help_contents,
    M= Item, !,
    stdio::write("Hello").
  onMenuItem(_Source, _M) .

```

Enable the *File/New* option of the Taskmenu, and add the snippet

```

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

```

to TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.

Chapter 12

Classes and Objects

Let us create an exploratory console project, in order to get a better understanding on how classes and objects work.

- Create a new console project.

Project Name: `classexample`
UI Strategy: `console`

Build/Build the application. Create a new class: *account*. Do not untick the *Creates Objects* box.

- Modify the file *account.i* as shown:

```
% File:account.i
interface account
    open core
    predicates
        ssN:(string SocialSecurity) procedure (o).
        setSocialSecurity:(string SSN) procedure(i).
        deposit:(real).
end interface account
```

Class *account* will produce a new *account* object whenever you call the method

```
A= account::new()
```

You can send messages to this object. For example, you may set the SocialSecurity Number of the customer:

```
A:setSocialSecurity("AA345"),
```

Afterwards, you may retrieve this very same Social Security Number.

```
A:ssN(SocialSecurity),
```

Both `ssN/1` and `setSocialSecurity/1` must be declared in the interface, which is in file `account.i`. An account needs other methods, besides the `ssN/1` and `setSocialSecurity/1`. For instance, you will need methods for deposits, for passwords, etc. I will leave to your imagination the creation of these methods. An interesting addition would be methods for cryptography.

- Modify `account.pro` as shown below.

```
% File: 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
```

- Modify `run/0` in file `main.pro`, in order to test the new class.

```
run():- console::init(),
  A= account::new(), A:setSocialSecurity("AA345"),
  A:ssN(SocialSecurity),
  stdio::write(SocialSecurity), stdio::nl.
```

12.1 Object facts

Objects have facts, that can be used to hold their state. The philosopher Wittgenstein was very fond of facts, and state of things. Let us take a look at the first few sentences of his *Tractatus Logico-Philosophicus*.

1. *The world is everything that is the case.* In German, the philosopher used the word *Fall*: *Die Welt ist alles, was der Fall ist*. The Latin word *casus* means *fall*, which explains the translations. However, what is falling in the world? The dice of Fortune are falling. There are a lot of possible events. The Fortune chooses with her dice what will happen. At least, this is what the Romans believed. Do you remember that Julius Cesar said *Alea jacta est*? I know! He said it in Greek, and Suetonius had it translated to Latin, but the point remains.
 - *The world is the totality of facts, not of things.* = *Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge*. Did you remarked how Wittgenstein liked facts? Later on, you will see that he thought that facts must have states.
 - *The world divides into facts.* From now on, I will spare you the German original.
 - *The world is determined by the facts, and by these begin all the facts.*
 - *For the totality of facts determines both what is the case, and also all that is not the case.* Pay attention that Wittgenstein takes into consideration both facts that happen (what is the case), and also those that are mere possibilities (all that is not the case).
2. *What is the case, the fact, is the existence of atomic facts.*
 - *An atomic fact is a combination of objects.*
 - *The object is simple.*
 - *Objects contain the possibility of all states of affairs.* In Visual Prolog, object facts, and variables are used to represent the state of an object.
 - *The possibility of its occurrence in atomic facts is the form of the object.*

One can see that the idea of classes and objects is an old idea in Western Philosophy. The same goes for states. In the example, the state objects

are individual accounts. The state of the account (our object) specifies the funds, the name of the account holder, his/her name, his/her Social Security Number, the password, etc. This state is represented by a fact, and a variable:

```
facts - customer
    funds:real := 3.0.
    personalData:(string Name,string SocialSecurity) single.
```

Interface predicates are used to access the facts that give the state of the object.

```
% File:account.i
interface account
    open core
predicates
    ssN:(string SocialSecurity) procedure (o).
    setSocialSecurity:(string SSN) procedure(i).
    deposit:(real).
end interface account
```

About eight persons wrote me saying the name of Wittgensteing book should be “Tratactus Logicus-Philosophicus”. Dr. Mary Rose Shapiro went into a long discussion on the differences between Classical and Philosophical Latin. She claims that the name as I wrote it was a mistake committed by Ogden, when he translated the book into English. My knowledge of the subject does not go to such a depth, but I decided to keep the title as it is in every bookshelf.

Chapter 13

Giuseppe Peano

Giuseppe Peano was one of the greatest mathematicians of all time. He wrote a small book whose title is

Arithmetices Principia Novo Methodo Exposita

From this, you can figure out how good he was. Not many people write things in Latin at this time and age, and find readers. Gauss was one of these few Mathematicians that could afford writing in Latin. Peano was another one. Of course, Gauss wrote mainly in Latin. Peano wrote mostly in French, the language of science at the beginning of the last century, and in Italian, his mother tongue. The only thing he wrote in Latin was the *Arithmetices Principia*, a very short book of 29 pages. There, he proposes to the world the modern notation for formal logics, a branch of Mathematics that gave origin to Prolog. He also talks about other things, like number theory, and recursion. I strongly suggest that you learn Latin, and read Peano's book.

13.1 Turtle graphics

Before diving into Peano's contributions to science (and there are many), let us implement turtle graphics in Prolog. To this end, create an Object-oriented GUI, with the following project settings:

General

Project name: `peano`

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

Target type: `Exe`

- Create a New in New Package form: `canvas` (section 2.1). The name of the form is *canvas*, and it is at the root of the *peano* project.

- Create a class named *curve*: Go to the project window, select the folder *canvas*, choose the option *File/New in Existing Package* from the task window, and create the class. Untick the *Create objects* box. Build the application.
- Create a package called *turtleGraphics* in the directory `C:\vispro`, and insert a class named *turtle* into it. To accomplish this step, choose the option *File/New in New Package* from the task menu; when in *Create project Item* dialog, click the *Browse* button for the *Parent Directory*, and choose the `C:\vispro\` directory from the *Set New Directory* dialog. Do not forget to check off the *Create objects* box. Be sure to put the package *turtleGraphics* outside the project *peano*. This way, it will be easier to reuse it. Build the application.
- Enable the *File/New* option of the application's *TaskMenu.mnu*.
- Add the following code

```

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

```

```

to TaskWindow.win/CodeExpert/Menu/TaskMnu/id_file/id_file_new.

```

- Edit *curve.cl* and *curve.pro* as in figure 13.1. Edit files *turtle.cl* and *turtle.pro*, as shown in figures 13.2 and 13.3. Build the application.
- Add the following code to the *PaintResponder* of *canvas.frm*:

```

clauses
  onPaint(S, _Rectangle, _GDIObject) :-
    W= S:getVPIWindow(),
    curve::drawCurve(W).

```

Build and execute the project. Whenever you choose the option *File/New* from the application menu, a window with a star will pop up on the screen.

13.2 Turtle states

The class *turtle* implements a drawing turtle, like in Logo, a language created by Papert in an effort to forward the ideas of Jean Piaget. The turtle is an object that moves on a window, leaving a track behind itself.

```

% File: curve.cl
class curve
    open core, vpiDomains

    predicates
        classInfo : core::classInfo.
        drawCurve:(windowHandle).
end class curve

%File: 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

```

Figure 13.1: Files curve.cl and curve.pro

A turtle has two state variables, i.e., its position and the direction of its movement. These state variables are given by a fact:

```

class facts
    turtle:(pnt, real) single.
clauses
    turtle(pnt(200, 200), -pi/2).
...

```

Consider the predicate:

```

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)).

```

It implements the forward movement of the turtle. If the turtle is at position $P1 = \text{pnt}(X1, Y1)$, and moves a distance L in a direction that it is **Facing**, one can find its new position by projecting L on the X and Y axis:

$$X_2 = X_1 + L \times \cos(F) \quad (13.1)$$

$$Y_2 = Y_1 + L \times \sin(F) \quad (13.2)$$

After getting the new position $P2 = \text{pnt}(X2, Y2)$, **forward/2** uses

```
drawline(Win, P1, P2)
```

to connect point $P1$ to $P2$. Finally, information about the new position of the turtle is asserted into the database:

```
assert(turtle(P2, Facing))
```

The predicate **move/1** is similar to **forward/2**, but does not connect the new position of the turtle to the old one. Predicates **right/1** and **left/1** turn the turtle to the right and to the left, respectively. Their implementations are straightforward.

13.3 Recursion

Peano's recursion postulate can be stated as follows: In order to prove that a predicate is true for any natural number,

- Prove that it is true for 0.
- Prove that it is true for N , if it is true for $N - 1$.

Of course, this version of the postulate runs towards 0, and it is written in English. The original runs towards ∞ and is in Latin. Cute, isn't it? In any case, let us examine the **star/4** predicate (see figure 13.1) under the light of Peano's postulate. The first clause

```
star(_Win, 0, _A, _L) :- !.
```

says: You draw 0 points to a star if you do nothing. The second clause

```
star(Win, N, A, L) :- turtle::right(A),
                      turtle::forward(Win, L),
                      star(Win, N-1, A, L).
```

says: You draw N points to a star if you draw a point (turn A radians to the right, and go L pixels forward), and proceed to draw the other $N - 1$ points.


```
% File: 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
```

Figure 13.2: File turtle.cl

13.4 Peano's curve

Peano designed a recursive curve that fills the space, which the mathematicians consider to be a very interesting property. What I really wanted to draw when I created the `peano` project was the Peano's curve, then I decided to start with an easier example. In any case, to get Peano's curve, all you have to do is to replace `curve.pro` given in figure 13.4 for the file given in 13.1.

13.5 Latino Sine Flexione

You have seen that Peano created the modern notation for Logic. He also came out with the idea of recursion. Since recursion is a basic Prolog programming scheme, you could say that Peano brought two important contributions to Logic Programming. However, he did not stop here. Prolog was invented by Colmerauer for Natural Language Processing:

The programming language, Prolog, was born of a project aimed not at producing a programming language but at processing natural languages; in this case, French (Colmerauer).

In the case of Natural Language, Peano had an interesting proposal. Since most European languages have borrowed a large number of Latin words, why not use Latin for scientific communication? All you need to do is select Latin words that are common to the major European languages and you have a vocabulary large enough to express almost any idea. The problem is that Latin is a difficult language, with all those declensions and verb forms. Peano took care of this too: In his brand of Latin, he eliminated all inflexions. Let

```

% File: 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

```

Figure 13.3: turtle.pro

us take a look at a few quotations from *Key to Interlingua*, where Interlingua is the official name for Latin without inflexions.

13.6 Quotations from Key to Interlingua

1. *Interlingua adopts every word common to English, German, French, Spanish, Italian, Portuguese, Russian, and every Anglo-Latin word;*
2. *Every word has the form of the Latin stem or root or radical.*

The vocabulary of Interlingua is composed of a great many Latin words which have come into English, and therefore are easily comprehended by an English-speaking person. It includes all terms used in the scientific nomenclatures of botany, chemistry, medicine, zoology, and other sciences. Many of these

```

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

```

Figure 13.4: Peano's curve

terms are either Greek or Greco-Latin. A few necessary classical Latin words without international equivalents are a part of the vocabulary, to wit:

Latino	English	Example
que	that	Interlingua adopta vocabulo <i>que</i> existe in Anglo, Germano, Franco et Russo.
de	of	Latino habe praecisione <i>de</i> expressione.
et	and	et cetera, et al.
sed	but	Vocabulario de Latino non es formato ad arbitrio, <i>sed</i> consiste de vocabulo internationale.
isto	this, that	pro isto (because of this)
illo	he, it	Illo es Americano.
illa	she	Illa es Americana
me	I, me	Me programma in Prolog.
te	you	Te programma in Prolog.
nos	we	Nos programma in Prolog.
vos	you guys	Vos programma in Prolog.
id	it	i.e., id est
qui?	who?	Qui programma in Prolog?
que?	what, which	Te programma in que?
omne	all	Illo es omnipotente= He is omnipotent.

Latino should not have any kind of inflection. The plural is formed by using the word *plure* as in:

Plure vocabulo in Latino es in usu in Anglo
The words of Latino are in use in English.

However, Peano was a democrat, and since most people wanted to form the plural with *s*, he gave his approval to an optional s-plural. Adjectives can be created from nouns with help of preposition *de*:

- de fratre = fraternal
- de amico = friendly, amicable

Of course, there are many words that are adjectives by nature: internationale, nationale, breve (short), magno (large, big), commune (common), etc. You can use these adjectives as adverbs, without modification. You can also form adverbs using the word *modo*: in modo de fratre (fraternally), in modo de patre (paternally), in modo rapido (rapidly, speedily). I guess that this is enough. Here is an example of text in 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 scientifico 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.

Why do we need an international language, like Peano's Interlingua, if we already have English? I think that we don't. However, people who propose Interlingua have a few interesting arguments to defend their project. For instance, humankind cannot let the international language change as economic hegemony and cultural prestige shift from one nation to the other. In the XIX century, French was the language of diplomacy. Then, the American economy surpassed the French, and English became the international language. In this century, China will be the dominant superpower, and will impose written Chinese to other nations. There is an interesting story about this point. People still use Latin to describe plants. Thus, if you discover a new plant, you must describe it in Latin. This was also the case of zoologists and anatomists, but they shifted to English not long ago. Therefore, a few

botanists proposed to follow suit, and adopt a modern language in Botany too. The problem is that the language they wanted was Chinese, not English!

There is another argument in favor of Latin. Languages like French, English and Chinese are difficult to learn even by people that speak them as their mother tongues. Not many Frenchmen can write acceptable French. Very few Americans can come close to English written communication. My son is at the eighth grade in Mount Logan Middle School, thus I often go there to check his progress. Most of his fellow students cannot read, let alone write. If this happens to Americans, what can I say about a Japanese who tries to learn English? Summary: Ten years is not enough to teach an American to read and write English prose, but in ten days he can write more than acceptable Latino. But let us leave this discussion to the wise, and build a project that takes a text in LsF (*Latino sine Flexione*), and explains difficult words and expressions.

- Create a new GUI project: **LsF**.
- Add a new package to the project tree: **LsF/editor**. Add the

editorControl

package to the project root. To this end, choose the option **File/Add** from the task menu, browse for the Visual Prolog installation directory, and select the file

`\pfc\gui\controls\editorControl\editorcontrol.pack`

Build the application.

- Create a new form: **editor/txtWin**. Resize the form window, since the **editorControl** must be quite large. Add a custom control to the form. The custom control is the one represented by the icon of the key. The system will offer you a list of custom control names; choose **editorControl**. Resize the editor pane.



- Build/Build the application.
- Create an `editor/grammar` class with the *Creates Objects* box unticked. If you do not remember how to do it, take a look at section 4.4. Build the application once more.

Modify the `grammar.cl` and `grammar.pro` files as shown below.

```
%File: grammar.cl
class grammar
  open core
  predicates
    classInfo : core::classInfo.
    dic:(string).
end class grammar

%File: grammar.pro
implement grammar
  open core
  clauses
    classInfo( "editor/grammar", "1.0").
    dic(S) :- stdio::write(S), stdio::nl.
end implement grammar
```

Enable the *File/New* option, of the application task menu. Add the snippet

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

to `ProjWin/TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_new`.

Add the snippet

```
clauses
  onHelpClick(_Source) = button::defaultAction :-
    editorControl_ctl:getSelection(Start, End),
    Txt=editorControl_ctl:getText(Start, End),
    grammar::dic(Txt).
```

to the *ClickResponder* of the `help_ctl`-button of `txtWin.frm`. In the same file you added the last code for *onHelpClick* (`txtWin.pro`), modify the definition of predicate `new` as shown in figure 13.5. Finally, create a file `LsF.txt` in the `\exe` directory with the text below.

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.

```
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).
```

Figure 13.5: Inserting a file into an editor

If you open a new `txtWin` from the application, the above text will appear in the window. If you select part of the text, and press the button **Help** from the TaskMenu of the application, you will echo the selected text on the message window. The next step is to add dictionary entries to the selected text, before writing it on the message window (see figure 13.6). To this end, we start by breaking down the selection into a list of words. In figure 13.6, this is done by the following predicate:

```
strTok:(string, string_list) procedure (i, o).
```

Natural Language Processing (NLP) is the most difficult branch of Computer Science, and outside the scope of this doc. If you want to learn the state of the art, you can start by adding more entries to the program of figure 13.6.

13.7 Examples

The examples of this chapter are in directories *peano*, and *LsF*.

```

%File: 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

```

Figure 13.6: A dictionary of “difficult” LsF words

Chapter 14

L-Systems

Aristid Lindenmayer was a Dutch botanist who invented a clever way to describe plant shapes. As you know, botanists spend a great deal of their time describing plant shapes. Their descriptions are usually made in a simplified version of Latin, invented by Carl Linaeus.

Lindenmayer's method of describing plant shapes is based on the plotting turtle that you learned in section 13.1 on page 139 . On the internet, you will find a vast quantity of material on the subject. Therefore, I will limit myself to an example, that you can use as a starting point to your own projects.

- Create a GUI project: `Lsys`
- Add a package to the root of the project tree: `aristid`
- Add a form to package `aristid`: `canvas`.
- Build the application.
- Add a class to `aristid`: `draw`. Untick the *Create Objects* box. Build the project to insert the class into the project tree.
- Enable the option *File/New* of the TaskMenu. Add

```
clauses
```

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

```
to TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.
```

14.1 Class draw

Then, edit the draw class as shown below. Build the application.

```
%File: draw.cl
class draw
  open core, vpiDomains
predicates
  classInfo : core::classInfo.
  tree:(windowHandle).
end class draw

% File: 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

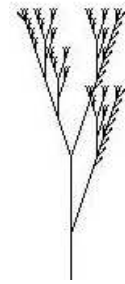
```

Finally, add the snippet

```
clauses
    onPaint(S, _Rectangle, _GDIObject) :-
        draw::tree(S:getVPIWindow()).
```

to the *PaintResponder*. Build and execute the application.

Open a canvas form by choosing the option *File/New* from the application menu. It will present a tree as shown in the figure on the left hand side.



14.2 Examples

The example of this chapter is in directory *Lsys*.

Chapter 15

Games

In this chapter, you will learn how to implement games in Visual Prolog. Of course, a good game is like a good poem. It is not enough to learn Russian, and versification rules in order to write like Pushkin. A poet, besides knowing the language in which he will write, needs imagination, creativity, etc. The same goes for a good game writer. You will need a lot of creativity to come up with something like Tetris or Space Invaders. I used these examples to show you that good games do not depend on fancy graphics or sound effects. The important features are suspense, continued tension and a sense of reward for good performance.

Once, a young man asked Lope de Vega how to write a poem. The answer was: *It is simple: Uppercase in the beginning of each verse, and a rhyme at the end.* “What do I put in the middle?”, asked the boy. In the middle, “*hay que poner talento*”, stated Lope de Vega.

This lesson has to do with a few techniques, like uppercase letters and rhymes. To get a good game, you will need to add talent to the recipe.

The game you will implement is simple. There is a worm moving fast on the screen. The player must change its direction in order to prevent it from hitting obstacles and walls. You will need to implement the following items:

1. A window, where the worm will move. The window must have four buttons, to control the movement of the worm.
2. A state class, that describes the direction of the movement, and the position of the worm. It exports the predicates `mov/0`, to change the position of the worm, and `turn/2`, that can be used to turn it around.
3. A draw class, that draws a snapshot of the state of the worm. It exports the `snapshot(Win)` predicate. Pay attention to this detail: The predicate `snapshot/1` cannot draw the worm directly onto the

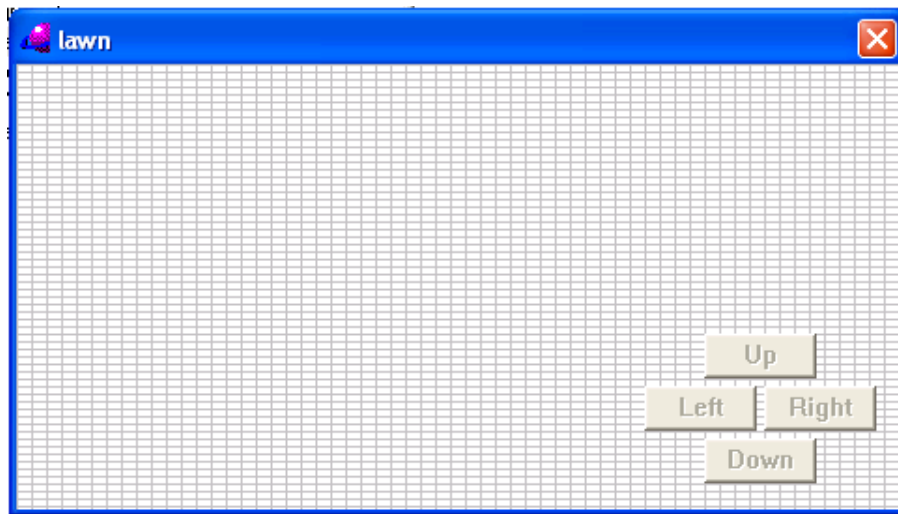


Figure 15.1: Lawn form

window. This would make the screen flicker. It must build a picture, and then draw it onto the screen.

4. A timer, that calls `mov/0` and `snapshot/1`, at fixed intervals, in order to create an illusion of movement.

We know what to do; let us do it. I have avoided creating objects until now. In fact, every time that you created a class, you were recommended to untick the *Creates Objects* radio button. However, if we rigidly maintain this state of mind, our game will have a very nasty property. If you lose, quit the current game, and try to start another one, you will find out that the new game is not in the initial state. In fact, it is in the state where you lost the old one. The problem is that the variables used to describe a state have memory that lasts from one form to another. Objects were invented to prevent this kind of nasty behavior. Therefore, let us take this very good opportunity to learn how to create objects, and deal with object states.

- Create a project: `game`.

```
Project Name: game
UI Strategy: Object-Oriented GUI (pfc/vpi)
TargetType: Exe
Base Directory: C\vip70
Sub-Directory: game
```

- Create a package: `playground`.
- Create a form: `lawn`. You will add four buttons to the *lawn* form; refer to figure 15.1 for the design. In the *Properties* dialog, it is a good idea to change the names of the buttons from `pushButton_ctl` to `up_ctl`, `down_ctl`, `left_ctl`, and `right_ctl`. Build the application in order to include the form. Then enable *File/New*, and add

```
clauses
```

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

```
to ProjWin/Code Expert/Menu/TaskMenu/id_file/id_file_new.
```

- Create class *objstate* inside *playground*. Do not untick *Create Objects* for the class *objstate*. Insert code shown in figure 15.3; NB: Class *objstate* must have an interface in file *objstate.i*. Build the application.
- Create class *draw*, inside *playground*. Untick *Create Objects*. Insert code of figure 15.2. Build the application.
- Create a class *click* to hold the clock. Untick *Creates Objects*.

```
%File: click.cl
```

```
class click
```

```
    open core, vpiDomains
```

```
predicates
```

```
    bip:(window).
```

```
    kill:().
```

```
end class click
```

```
% File: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
```

Build the application to include `click` into the project.

- Add

```

    clauses
        onDestroy(_Source) :- click::kill().

```

to the *DestroyerListener* from the *Properties*-dialog of *lawn.frm*.

- Add

```

class facts
    stt:objstate := objstate::new().
clauses
    onShow(Parent, _CreationData) :- stt := objstate::new(),
                                     stt:init(), click::bip(Parent).

```

to *ShowListener* from the *Properties*-dialog of *lawn.frm*.

- Add code for the *ClickResponder* corresponding to each of the buttons.

```

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).

```

- clauses


```

onTimer(_Source, _TimerID) :- stt:mov(),
    R= rct(10, 10, 210, 210),
    invalidate(R).

```

to *TimeListener* from the *Properties*-dialog of *lawn.frm*.

- Add

clauses

```
onPaint(_Source, _Rectangle, GDIObject) :-
    draw::snapshot(GDIObject, stt).
```

to the *PaintResponder* frm the *Properties*-dialog of `lawn.frm`.

Build and run the program. After choosing the option **File/New**, a window with a worm pops up. You can control the worm using the four buttons.

It is up to you to improve this game. You can add obstacles, like walls. You can also put little apples for the worm to eat. Finally, you can write a 3D game. The 3D game is very simple indeed. The segments have three coordinates, and you must find their projections on the plane before drawing them. If you info-fish using the Internet, you certainly will find formulae to calculate the perspective projection of a point. Alex Soares wrote the original game on which I based this tutorial. His game is in 3D, in Visual Prolog 5.2, using Direct X, a library that has a lot of resources for 3D games.

Let us keep to our study of drawing predicates. If you draw figures at every tick of the clock, you will have created an animation. However, if you perform the drawings directly on a visible window, the animation will blink, which is not very nice. The solution for this problem is to perform the drawing on a hidden window, and transfer it to the visible window only after it is ready for showing. The predicate

```
W= pictOpen(Rectangle)
```

opens a hidden window for you to draw things without worrying about a blinking animation. The predicate

```
Pict= pictClose(W)
```

creates a picture from the contents of a hidden window; at the same time, it closes the window. Finally, the predicate

```
pictDraw(Win, Pict, pnt(10, 10), rop_SrcCopy)
```

transfers the picture to a visible window. Here is an example of code containing these three important predicates:

```
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).
```

The argument `rop_SrcCopy` specifies the mode of the raster operations. It says that the system should copy the figure from the source to the destiny. There are other modes of transferring figures from one window to another:

- `rop_SrcAnd`, performs a logical-AND between the bits of the figure, and the background. You can use it for creating sprites.
- `rop_PatInvert`, inverts the colors.
- `rop_SrcInvert` inverts the colors of the source.

```
%File draw.cl
class draw
  open core, vpiDomains
predicates
  classInfo : core::classInfo.
  snapshot:(windowGDI Win, objstate).
end class draw

%File 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
```

Figure 15.2: draw.cl and draw.pro

```

%File:objstate.cl
class objstate : objstate
end class objstate

%File:objstate.i
interface objstate
    open core, vpiDomains
    predicates
        init:().
        turn:(integer, integer).
        mov:().
        segm:(rct) nondeterm (o).
end interface objstate

%File: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

```

Figure 15.3: objstate

15.1 Object Facts

Pay careful attention to one particular detail: When dealing with classes that do not create objects, the state of a class is established by class facts; here is how a class fact is declared:

```
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)).
```

A class fact is shared by all objects of the class. This means that, if an object predicate (a predicate declared in the interface) changes a fact, it is changed for all objects of the class. Therefore, if you had kept the state of the worm in class facts, the game would have the following nasty feature:

Nasty Game Property — Whenever the player loses a worm, quits the current game, and plays another one, he will find that the new worm is not in the initial state. In fact, it is in the state where he lost the old one.

The problem is that the class facts used to describe the state of the worm have class memory that lasts from one form to another, from one worm object to another. The solution is simple: Declare the *facts* without adding the keyword *class* to the head of the declaration; then there will be a private set of facts for each object. Here is how to declare object facts:

```
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).
```

This scheme of keeping an object state is implemented in the program of figure 15.3, that appears on the next page.

Chapter 16

Animation

Please, take a look at chapter 8, page 87, if you do not remember how to use the event *painting*. You will also need to use masks to create figures with a transparent background; this theme is also explained in chapter 8. Finally, you may want to recall to memory the use of *GDIObject*, which is dealt with in section 3.2, page 28, where you will find material about invalidating a rectangle for painting as well.

- Create a project.

Project Name: rattlesnake

UI Strategy: Object-oriented GUI (pfc/gui)

In this project, you will animate the famous rattlesnake that appears on the John Paul Jones' *Don't tread on me* flag.

- Create a new package, and let its name be *snake*.
- Create a new *canvas* form. Put it inside the *snake* package.
- Create class `click` to hold clock events. Untick *Creates Objects*. Use the code given in figure 16.1 to define and implement *click*.
- Build the application. Enable *File/New*. Then, add

```
onFileNew(S, _MenuTag) :-  
    F= canvas::new(S), F:show(), click::bip(F).
```

```
to TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.
```

```

%File: click.cl
class click
    open core

predicates
    bip:(window).
    kill:(window).
end class click

%File: 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

```

Figure 16.1: click class handles the timer

16.1 dopaint

Class *dopaint* will handle painting, and works more or less like the *dopaint* method in Java. Create a class, name it **dopaint**, and untick the *Creates Objects* option. Here is the class definition:

```

%File: dopaint.cl
class dopaint
    open core
predicates
    classInfo : core::classInfo.
    draw:(windowGDI).
    invalidrectangle:(vpiDomains::rct) procedure (o).
end class dopaint

```

Figure 16.2 shows the implementation of *dopaint.pro*. Build the application.

```
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
```

Figure 16.2: dopaint.pro

16.2 Handling the timer

When you close *canvas*, you must kill the timer. To achieve this goal, add

```
onDestroy(W) :- click::kill(W).
```

to the *DestroyListener* of the *Properties*-dialog of *canvas.frm*. Then, add

```
onTimer(_Source, _TimerID) :-
    dopaint::invalidRectangle(R),
    invalidate(R).
```

to the *TimeListener* of the *Properties*-dialog of *canvas.frm*. Finally, add the event handler

```
onPaint(_Source, _Rectangle, GDIObject) :-
    dopaint::draw(GDIObject).
```

to the *PaintResponder* of the *Properties*-dialog of *canvas.frm*.

16.3 How the program works

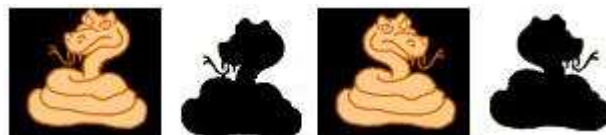
The first thing that predicate *draw(W)* does is to paint a few frogs onto the background. The frogs are loaded by the following predicate:

```
P= vpi::pictLoad("figs\\frogs.bmp"),
```

Of course, you must have the file *frogs.bmp* in directory *figs*. Then, *draw(W)* obtains the picture and the mask of a rattlesnake.



The *flipflop(Snake, Mask)* predicate is designed to alternate two snake paintings, in order to get the illusion of motion. To achieve the insertion of the snake onto the background, you must use masks. If you do not remember how to do it, read chapter 8 again.



Chapter 17

Text Editor

In this chapter, you will learn how to use the *editControl* class to create a text editor.

- Create a new project.

Project Name: `editor`

UI Strategy: Object-oriented GUI(pfc/gui)

- Add the *editorControl* package, that is found in the installation directory of Visual Prolog. Choose the task menu option *File/Add* and look for the package in

`Visual Prolog 7.x\pfc\gui\controls\editorControl.`

- Build the application. Then create a new form `edform.frm` at the project root, as shown in figure 17.3. To insert the *editorControl* into the form, choose the custom control option in the form prototype; it is the button that has a Yale key¹ (figure 17.1); and then choose `editorControl` in the Dialog shown in figure 17.2. Change the name of the control to `editorControl_ctl` as shown in figure 17.4. Build/Build the application.
- Enable *File/New* and add the snippet below to it.

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

¹Yale invented this kind of key, that is very popular nowadays. Legend has it that soon after his invention, Houdini was able to break through the new device.

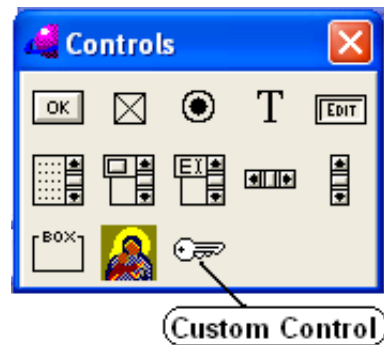


Figure 17.1: The insert custom control button

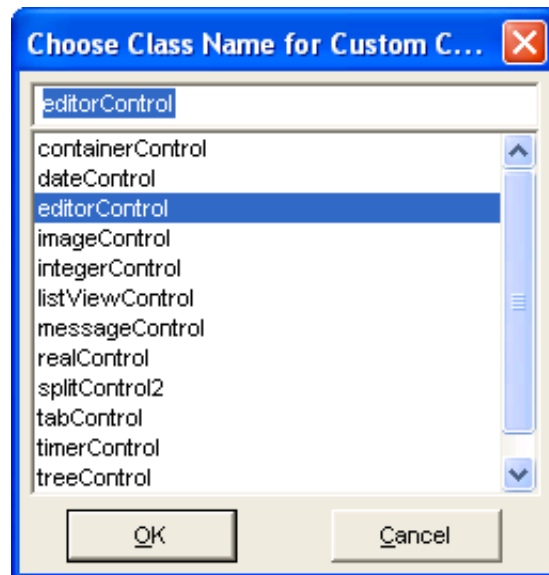


Figure 17.2: Choose Custom Control dialog

17.1 Saving and Loading files

Go to the project tree, and open `edform.frm`, if it is not already open. Use the *Properties*-dialog to add the snippet of figure 17.5 to the *ClickResponder* of `button:save_ctl`; then add the code shown in figure 17.6 to the *ClickResponder* of `button:load_ctl`. This is about all you have to do in order to create an operational text editor.

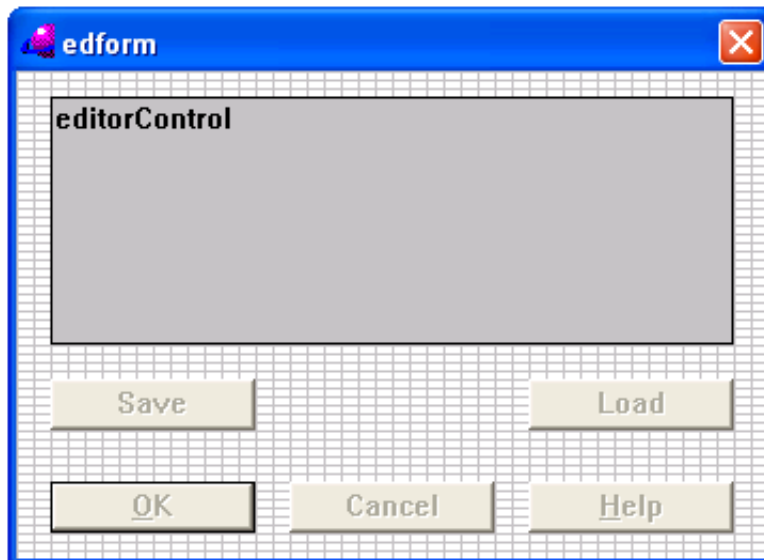
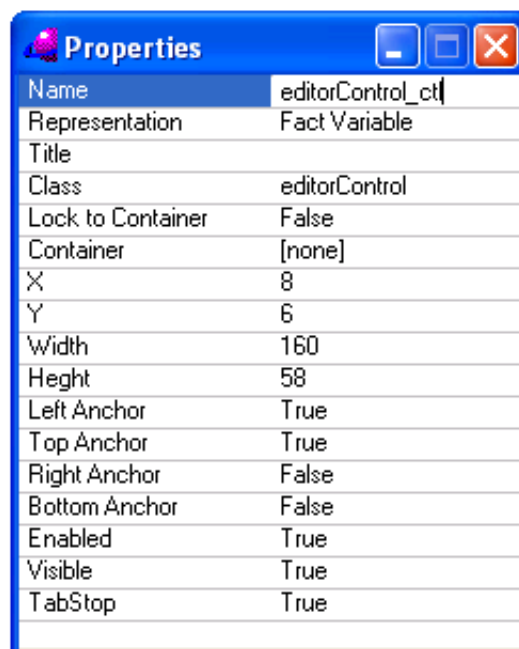


Figure 17.3: A form with Edit Control



Name	editorControl_ct
Representation	Fact Variable
Title	
Class	editorControl
Lock to Container	False
Container	[none]
X	8
Y	6
Width	160
Heght	58
Left Anchor	True
Top Anchor	True
Right Anchor	False
Bottom Anchor	False
Enabled	True
Visible	True
TabStop	True

Figure 17.4: User control properties

```

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().

```

Figure 17.5: Code for onSave

```

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().

```

Figure 17.6: Code for onLoad

Chapter 18

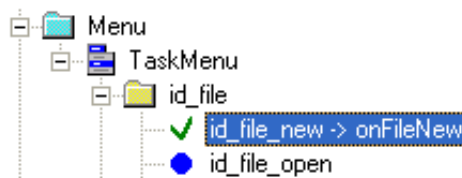
Printing

In the previous chapter, you learned how to build a text editor. In this chapter, you will acquire introductory notions of printing.

- Create a project.

```
Project Name: testPrt
UI Strategy:  Object-oriented GUI (pfc/GUI)
Target Type:  Exe
Base Directory: C:\vip\codeForTyros
```

- Build the application, in order to insert prototypes of the Task Window onto the *Project Tree*.
- On the *Project Tree*, double click on the branch `TaskMenu.mnu`, and enable the `&File/&New/tF7` option from the menu prototype.
- On the *Project Tree*, right click on the `TaskWindow.win`, and choose the option *Code Expert* from the floating menu, as in figure 2.6, page 21. The *Dialog and Window Expert* will pop up. Open the folders `Menu`, `TaskMenu`, and `id_file`, as shown below.



Select the branch `id_file_new`, and press the button *Add*. Then, double click on the newly created `id_file_new->OnFileNew` branch; finally, replace the `onFileNew(_Source, _MenuTag)` prototype with the snippet given below.

```

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).

```

The beauty of Visual Prolog printing scheme is that you deal with the printer as if it were a normal graphic window. In fact, the first thing you do, in order to print something, is to open a printer job window.

```
PW=vpi::printStartJob("Recoreco")
```

Once you have a window, you can use it to obtain information about the printer resolution.

```

_HRES= vpi::winGetAttrVal(PW,attr_printer_hres),
VRES= vpi::winGetAttrVal(PW,attr_printer_vres),
V_SCR_RES=vpi::winGetAttrVal(PW,attr_screen_vres),

```

By trial and error, and using information about the printer resolution, you can define a font, that looks good on print.

```

FNT=vpi::fontCreate(ff_Fixed,[], VRES*40 div V_SCR_RES),
vpi::winSetFont(PW,FNT),

```

Finally, you can use any drawing predicate to produce a fine printed page.

```
vpi::drawText(PW, 100, 200, "Before the sunset!"),
```

Chapter 19

Tab forms and other goodies

- Create a new project:

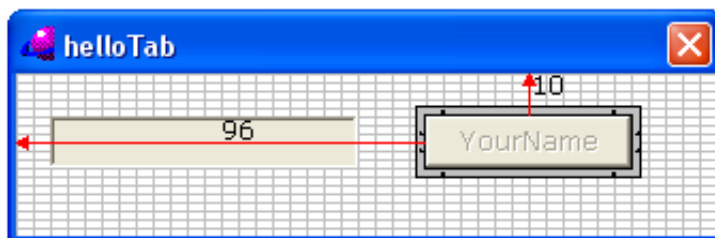
Name: `tab_example`

UI Strategy: `Object GUI`

- New in New Package: Create a package with the name `forms`
- From the installation directory, File/Add:

`pfc\gui\controls\tabControl\tabControl.pack`

- New in Existing Package (forms): Create a `Control` with the name `helloTab`; to achieve this goal, select the option *Control* on the left hand side pane. Rename the button to `button:yourname_ctl`.



- Build the application, in order to insert snippets for events. Add the following snippet to the *ClickResponder* belonging to `button:yourname_ctl`:

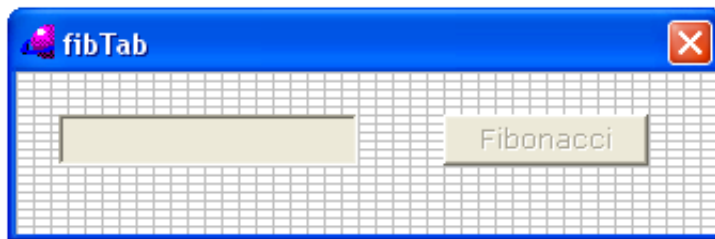
clauses

```
onYournameClick(_Source) = button::defaultAction :-  
    Name= edit_ctl:getText(),  
    Ans= string::concat("Hello, ", Name, "!\n"),  
    stdio::write(Ans).
```

19.1 Famous programs

There are two programs that one can call infamous: "Hello, world", and recursive Fibonacci. In this chapter, you will learn how to place both programs in a control with tabs. You already created the form to hold the hello-application. Now, let us create a form for the Fibonacci sequence.

- New in Existing Package (forms): Control with name `fibTab`; rename the button to `button:fib_ctl`.



- Build the application, in order to insert code for events. Add the following snippet to the *ClickResponder* belonging to `button:fib_ctl`:

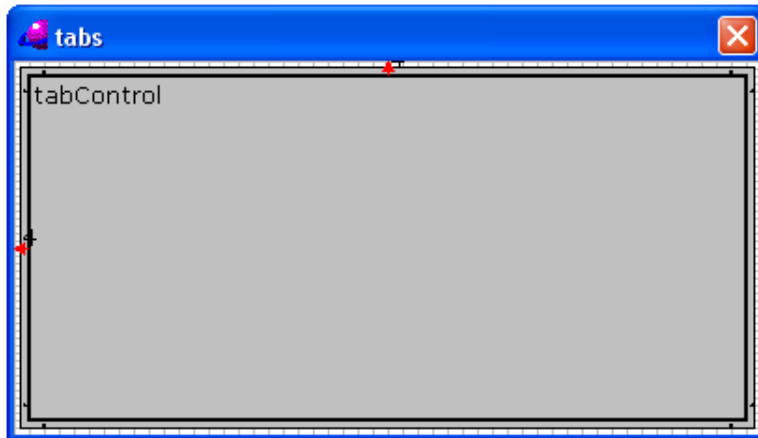
```
class predicates
    fibo(integer, integer) procedure (i, o).

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).
```

Build the application again, to make sure that everything is working properly.

- Create a New form in Existing Package (the existing package is *forms*) — Name it **forms/tabs**. Use the Yale key to insert a **tabControl** into the form. Build the application.



- Go to the file **tabs.pro**, which you can access by clicking the corresponding branch of the project tree, and replace the clause

```
clauses
    new(Parent):-
        formWindow::new(Parent),
        generatedInitialize().
```

with the following snippet:

```
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.
```

- Enable the option **File/New** of the application file menu.

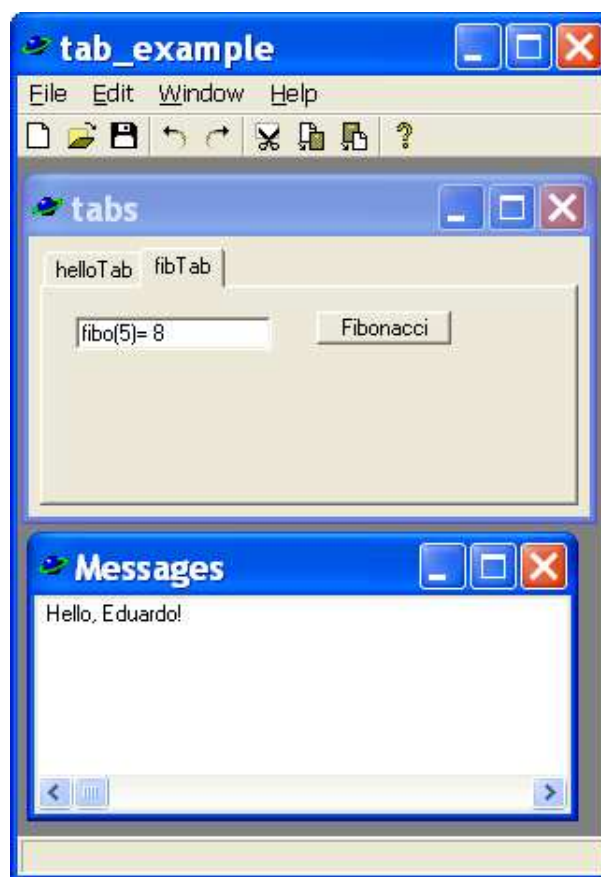
- Add the snippet

```
clauses
```

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

```
to TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new.
```

- Build the application one last time. Cute, isn't it?



19.2 Botany

Your interest in Botany may be none. However, there are many reasons for cultivating this important branch of science. Let us enumerate them.

Saving the world. Global warming is the most serious challenge facing us today. To protect the planet for future generations, we must reduce

the concentration of heat-trapping gases by using the technology, and practical solutions already at our disposal. It turns out that plants offer poetically the only viable way out of this trap that we set ourselves. Therefore, we need a climate policy initiative to reduce emissions from tropical deforestation, understand the dynamics of plant interdependency, in order to check the decrease of bio-diversity, and select rapidly growing new vegetable species to replace forests that our lack of vision and direction destroyed in the past. Therefore, if saving the world is your vocation, here is a good place to start. I believe that saving the world is also good business, since there will be plenty of money for researchers and specialists in global warming. I am proud to say that my family is very active in protecting nature, and bio-diversity. I want to pay homage to my grandmother, an Indian woman (Native American, not from India) who with my father created an ecological sanctuary that we maintain until today. I remember that when I was a child, the most cherished book in our house was Maeterlinck's *L'Intelligence des fleurs*. My father used to make me repeat *plusieurs fois* Maeterlinck's famous saying: *When the last tree has been thrown down, the last man will die clinging to its trunk*. To tell you the truth, I never found any evidence that Maeterlinck said such a sentence; however, my grandmother was sure that he did. I hope that a Belgian reader may shed light on this point.

Computer graphics. In the chapter about Aristid Lindenmeyer you have seen that Botanists invented clever methods to create computer graphics depicting flowers. You probably remember Bizet's song:

*Votre toast, je peux vous le rendre, señors, señors, car avec
les soldats oui, les toreros peuvent s'entendre.*

I could say that Computer Scientist and Botanists can understand each other, at least in the realms of computer graphics.

Orchids. These exotic plants are among the most beautiful creations of Nature. They are objects of a cult that owes nothing to wine, or French cuisine. By the way, the Serbian Nero Wold was fond of wine, French cuisine, and orchids. If you read Wolf's books, you will notice that the suspense is not created by how Wolf solves mysteries, but by how he will get the money necessary to maintain his expensive cults. He hires Fritz Brenner, an exceptionally talented Swiss cook who prepares and serves all his meals, and speaks French when addressing him; he also hires Theodore Horstmann, an orchid expert who takes care of

the plant rooms; finally, the most important of his employers is Archer Goodwin, who works as a detective to earn the money necessary to keep the expensive habits of his overweight boss.

One of the most interesting features of Botany, as compared with other sciences, is that there is a special kind of Latin internationally used by botanists for the description, and naming of plants. Since many Botanists do not know enough Latin, a program to help them deal with this difficult language would give a head start to the goals of saving the world, cultivating orchids, or creating cool computer graphics. In this section, I will see how such a program can be written. Besides this, you will learn how to work with tab-controls, and listEdit-controls.

The program to teach Botanical Latin is given in the skinner directory of the folder containing the examples that accompany this document. The program contains three tab-controls:

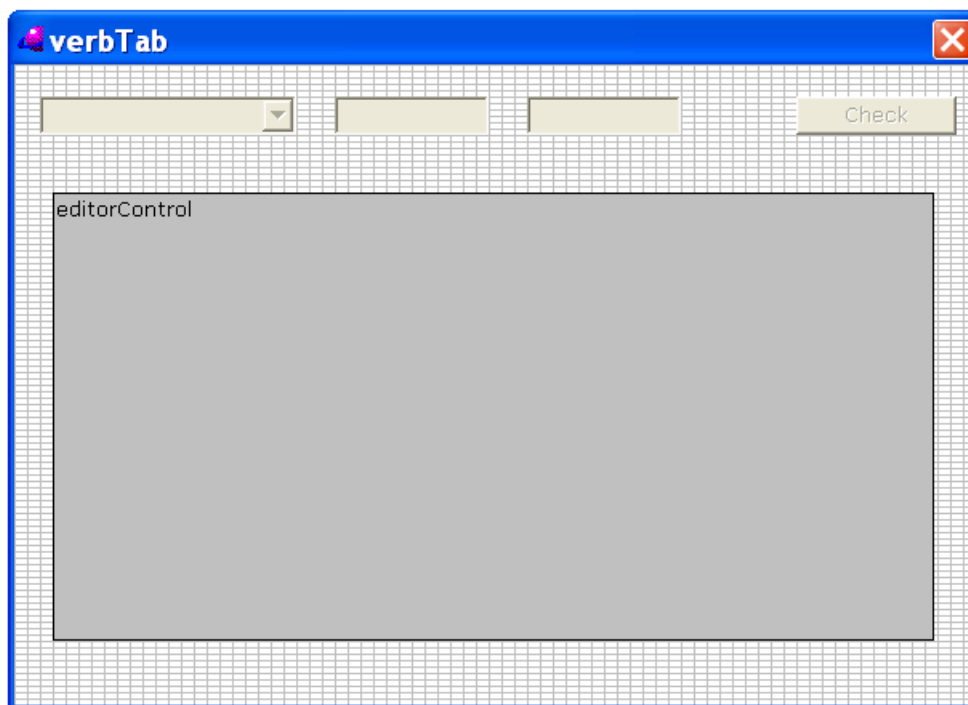
- Pronunciation tab. In order to teach you how to pronounce Latin, I used excerpts from a very famous book called *Philosophiae Naturalis Principia Mathematica*; the author of the book is a certain Isaac Newton; I am not sure whether you heard about this scientist, but English scholars consider him to be the greatest scientist who ever lived. They certainly are exaggerating; notwithstanding Newton was good enough to write in Latin, and find readers; as you remember, when a scientist is very good, one says that he could have written in Latin, that he would find readers; Newton, Peano, Gauss, and others took this saying literally, and wrote in Latin. In this tab, my son, who speaks fluent Latin, English, Ancient Greek, and Chinese, will read Newton for you.
- Declension tab. There are languages where nouns change to show their function in a sentence. Latin, Sanskrit, German, Russian, Polish are among these languages. Therefore, even if you do not think that Botanical Latin is useful enough to merit your attention, you can use the underlying structure of the Latin program to write programs to teach Russian, or German.
- Verbtabs. Declension is not the only feature that Latin shares with German, Russian, and Ancient Greek. Latin has also a very complex verbal system, that it passed down to the so called Romance Languages (Spanish, Italian, Portuguese, Rumanian, French, Galician, and Catalan).

If you want to learn one of the Romance languages, you can adapt the support provided for Latin, in order to study Spanish, or French verbs. By the

way, Spanish and Portuguese are much closer to Latin than other Romance languages. The Latin verbal system was transmitted to Spanish and Portuguese almost without alteration. Consider the verb *amare*:

Latin		Spanish	
Present	Imperfect	Present	Imperfect
amo	amabam	amo	amaba
amas	amabas	amas	amabas
amat	amabat	ama	amaba
amamus	amabamus	amamos	amábamos

Now, let us concentrate our efforts on the design of the program. I suggest that you place all tab controls in the same package; in the example, the package is called *tabPackage*. In order to create a tab control, choose the option *New in New Package* from the IDE task menu; select the option *Control* from the left hand pane; press the button *Create*, and mount the following layout:



The *verbTab* control contains a *listEdit* control, that will hold the verb drilling list; two read-only edit controls, to show the tense, and mood that the student will be practicing; and a button, that will compare the student's answer with a template. The main reason behind offering you this application is to show

how to handle the listEdit control; therefore, let us take a close look at the methods associated with this important kind of control. From the *Properties*-dialog, you must press the *Events*-button, and add the following snippet

```

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).

```

to the *showListener*. The snippet shows you how to add a list of options to the control; it shows also how to select an entry:

```

VerbList= listVerb_ctl : tryGetVpiWindow(), ...
vpi::lboxAdd(VerbList, L),
vpi::lboxSetSel(VerbList, 0, 1),...

```

From the *Properties*-dialog, you must also add the code from figure 19.1 to *Events-onCheckClick*; then you will learn a few other listEdit methods:

Get the window associated with the control :

```
VerbList= listVerb_ctl : tryGetVpiWindow()
```

Get the selected index :

```
IV= vpi::lboxGetSelIndex(VerbList)
```

Delete entry, by giving its index :

```
vpi::lboxDelete (VerbList, IV)
```

Obtain the number of entries :

```
C= vpi::lboxCountAll(VerbList)
```

The *declensionTab* control works exactly like the *verbTab*-control. The *pronunciationTab* control has something new. It triggers the Windows' `playSound` in order to play a wave file. Since we are dealing with an external function, it is necessary to go through the red tape of linking a Prolog program with a Windows' API; you must begin by declaring that `playSound` is an external routine. Then you must introduce a Prolog predicate to represent the API entry. Below one finds the declaration that `playSound` is an external routine:

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
        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.
```

Figure 19.1: listEdict methods

Now, let us declare a Prolog predicate that will be called whenever you want to trigger `playSound`, and a few necessary constants.

```
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 */
```

Finally, introduce the snippet given below

```
clauses
    onPlayClick(_Source) = button::defaultAction :-
        File= listEdit_ctl:getText(),
        _ = playSound(File,
                      null, /*snd_noddefault*/ snd_async+snd_filename).
```

into the *clickResponder* of the *Play* button.

19.3 Handling Chinese

After this incursion into Latin, let us see how to deal with Chinese. A Chinese dialect known in the West as Mandarin is the language with the largest number of speakers; besides this, the writing system used in Mandarin was adopted by other Chinese languages, by the Japanese, and by the cultivated Koreans. As a result, at least one third of the World's population has some knowledge of written Chinese.

Mandarin itself has three writing system: Pinyin, that uses the Latin alphabet with Greek accents, Tradition Ideograms, and Simplified Ideograms.

When using a computer to write Chinese, one usually express his/her ideas in Pinyin, and a Word Processing Editor translates Pinyin into Simplified, or Traditional Ideograms. Of course, I do not know Chinese, but my son knows. Therefore, at my request, he has used the `multiMedia_native` class to add sound to a couple of menu entries. You will find the snippet that he has used below; browse the multimedia class to uncover additional resources.

```
predicates
```

```
  onFilePronunciationHowAreYou : window::menuItemListener.
```

```
clauses
```

```
  onFilePronunciationHowAreYou(_Source, _MenuTag) :-
    _ = multiMedia_native::sndPlaySound("howareu.wav",
      multiMedia_native::snd_ASync ).
```

In the case of Chinese, my son used the `multiMedia_native` class. Therefore it was not necessary to go through the red tape required to call an external routine, as he has done in the case of Latin. You may ask why he did not adopt the same solution for Latin and Chinese. The point is that he wanted to show you how to call external routines from Prolog, and the multimedia API provides an easy example of such an operation.

- Create a project: `chinese`.
- Add the `editControl` to its root.
- Create a package `chineseforms`.
- Add a `chineseForm` to `chineseforms`. The `chineseForm` has

```
  editControl, and
  a Pinyin2Ideogram button.
```

- Create an `ideograms` class; un-tick the *Creates Objects* radio button. You will find the class file below. The implementation is listed in figure 19.2.

```
% File: ideograms.cl
class ideograms
  open core
predicates
  classInfo : core::classInfo.
  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

```

Figure 19.2: File: ideograms.pro

From the *Properties*-dialog, add the following snippet

```

clauses
  onPinyinClick(_Source) = button::defaultAction :-
    Txt= editorControl_ctl:getEntireText(),
    ideograms::ideograms(Txt, Ans),
    editorControl_ctl:pasteStr(Ans).

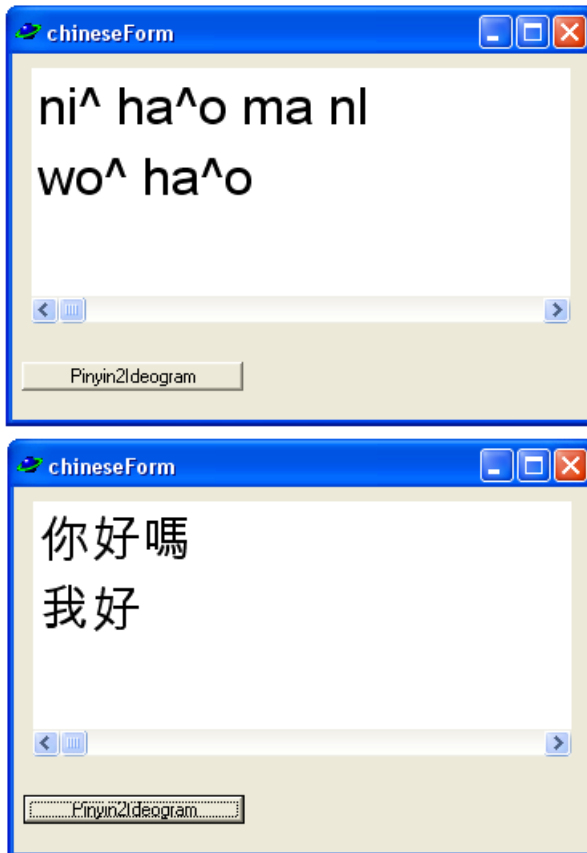
```

to the *ClickResponder* belonging to Pinyin2Ideogram-button.

The last step is the addition of the following snippet

```
predicates
    onShow : window::showListener.
clauses
    onShow(_Source, _Data) :-
        editorControl_ctl:setFontDialog(), !.
    onShow(_Source, _Data).
```

to the *ShowListener* of the *Pinyin2Ideogram*-button. Of course this program will work only if you have a Unicode font installed; whenever a new form is created, you must choose a Unicode font. If you live in China, Japan, or Korea, this is not a huge problem. However, if you live in the United States, or in Europe, your Unicode Font may not have the Chinese characters. Windows assumes that you speak the most weird languages, except Chinese. I have told you before that Paraguay is my favorite country. Therefore I often download Paraguayan songs; the result is that my computer often assumes that I speak one of the two official languages of that country. Anyway, I am pretty sure that Arial Unicode MS does have Chinese characters.



19.4 Regular expressions

Whenever one is dealing with languages, regular expressions are useful tools. In fact, if the programmer wants to search a string for a substring, s/he can use traditional search methods, like the Boyer-Moore algorithm. However, s/he may want to find a pattern that describes an integer, for instance. In this case, the regular expression would be "[0-9]+", which means: *A sequence of one or more digits*. The plus sign is called positive Kleen closure, and indicates one or more repetitions. There is also a reflexive Kleen closure, that accepts the empty string: "[0-9]*". These are the two instances of regular expressions that I use; for anything more complex I prefer the full power of Prolog.

```

/*****
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).
```

For a deeper understanding of regular expressions, it is a good idea to learn what is a Chomsky-Schutzenberger hierarchy. Noam Chomsky (an American Linguist) and Marcel Paul Schutzenberger (a French Medical Doctor) proposed that languages should be described by grammar rules very similar to logical expressions in the clausal form; if you do not remember what a clausal form is, go back to page 77, and you will find something about the subject. In Chomsky-Schutzenberger formalism, a simple English noun phrase is described by the following set of rules:

```

article → [the]
noun → [iguana]
adjective → [pretty]
npr → article, adjective, noun

```

Any symbol that appears on the right hand side of an arrow is a non-terminal symbol. Therefore, **article**, **npr**, **noun**, and **adjective** are non-terminal symbols, i.e., symbols that the grammar defines. Symbols that appears only on the right hand side of the rules are terminal symbols; e.g. **[iguana]**. The rule for **npr** can be understood thus:

npr →	A noun-phrase is defined as
article	an article followed by
adjective	an adjective followed by
noun	a noun

A language is regular if there is at most one non-terminal symbol at the right hand side of any one of the rules that describe it; besides this, the non-terminal symbol must be the rightmost one. Regular languages are good to describe tokens, like numerals, lexical entries, etc.

Regular expressions were introduced in the 1950s by Stephen Kleene as a method of describing regular languages. In modern notation, a member of a set of terminal symbols is represented by brackets around a description of the set; for instance, the regular expression "[0-9]" is a pattern that matches any digit; the programmer may use the + sign (positive Kleene closure), in order to indicate a sequence of one or more members of the set; e. g. "[0-9]+" matches any numeral representing a positive integer. A sequence of zero or more members of the set is indicated by an asterisk (the reflexive

Kleene closure). Alternation is indicated by a vertical `|` bar; for instance, `"-[0-9]+|[0-9]+"` is a pattern that matches a negative numeral, or an unsigned integer. Finally, one can use the hat character to represent characters that do not belong to a set; then `"[^a-z]"` is the set of all characters that are not lowercase letters.

<code>"[0-9]"</code>	a member the set of all digits
<code>"[0-9]+"</code>	a sequence of one or more digits
<code>"[0-9]*"</code>	a sequence of zero or more digits
<code>"-[0-9]+ [0-9]+"</code>	a sequence of digits prefixed with minus, or a sequence of digits without prefix
<code>"[^A-Z]"</code>	a character that is not an uppercase letter

In replacements, one can add back references by enclosing the pattern in parentheses, and using `"\1"` to insert the matched string into the replacement. An example is given below.

```
implement main
  open core
class predicates
  test:(string) procedure (i).
clauses
  classInfo("main", "regexpr").

  test(Str) :-
    Ans= regexpr::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).
```

The example replaces an HTML tag with its contents. Anything between the tags is captured into the first back reference. The question mark in the regular expression makes sure that the star stops before the first closing tag rather than before the last one.

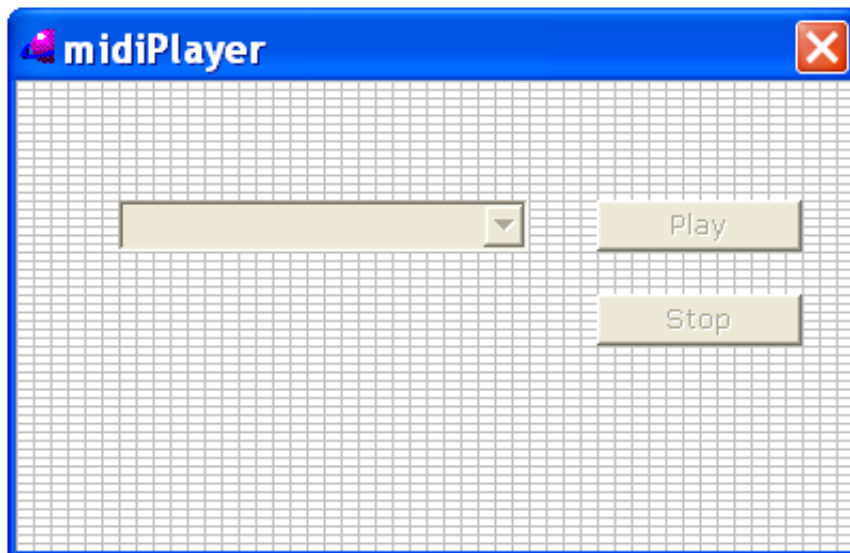
19.5 A MIDI player

Computers made it possible for a musician to create interesting songs, explore new artistic venues, and perform his/her work single-handedly. In this section, you will learn a few facts about MIDI, a protocol that brings computer music to the concert hall. Let us start by writing a program that executes MIDI files.

Project Name: `midisplay`

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

- Add a New form in a new package. Let the form name be `midisplay`. Add a `listEdit` box to it, and two buttons: `play_ctl` and `stop_ctl`.



- From the *Properties*-dialog of the `midisplay` form, press the *Events*-button, and add the following snippet

```
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).
```

to the *showListener*.

- Enable the File/New option of the application task menu. Then, add the snippet

```

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

```

to it. Build the application.

- The next step is to add the snippet below to the ClickResponder of the play_ctl-button.

```

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().

```

- The final step is to add code to the button that will interrupt the music, if you decide to do something else.

```

clauses
  onStopClick(_Source) = button::defaultAction :-
    Null= null,
    _Z= multiMedia_native::mciSendString(
        string::createCopy("Close mid"),
        string::create(300, " "), 300, Null).

```


19.6 Midi format

My father, and mother didn't like music very much. Notwithstanding, I was lucky enough to know many great musicians. For instance, my son, who is a pianist, and singer, is the best friend of José Bernal, who happens to be the son of Sergio Bernal, the conductor of the Utah State University Symphony Orchestra. What is more, I was raised next door to the house of a great composer, Maestro Joubert de Carvalho; the beautiful sonata that I offer you in the **skinner** project was composed by my neighbor; it may interest you to know that there exists a city named after that haunting piece of music.

I also knew Maestro Sukorski who created a very interesting work of art; he wrote a program in Prolog (yes, it was in Prolog!) that performs an analysis of the mood of a person, by measuring skin resistance, electromyograph signals, heartbeat increases, etc. Basically, Sukorski program was something like a lie detector. Using these data, the program was able to compose a music suited to the occasion.

I studied Space Science, and Classical Literature in Cornell; my office there had been occupied by Moog, before my arrival; in fact, Moog scribbled his name on the very same table that I used while in Cornell; it seems that I have inherited Moog's table. I also know well Maestro Dovicchi, whose adviser was no other than the famous Hungarian composer Gyorgy Ligetti. Maestro Dovicchi researches methods of producing high quality computer music in the midi format; using his computers, he was able to play Mascagni's *Cavalleria Rusticana* single handedly; I happened to be present during this memorable performance.

Besides Maestro Dovicchi, another great expert in computer music is the composer Luciano Lima. His programs analyze the work of great composers like Tchaikovsky, or Beethoven, and write music according to the same musical style. I included one of Lima's compositions in the examples. I used the Internet to contact Maestro Lima, who is an accomplished Prolog programmer, and asked him to show you how to create a simple MIDI file.

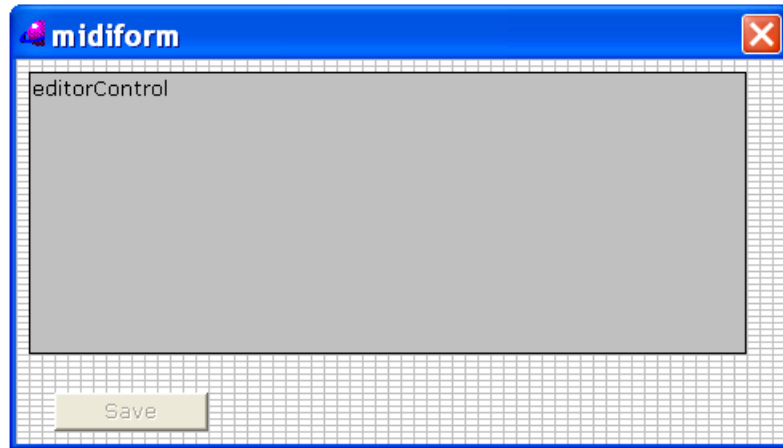
```
Project Name: midisplay
UI Strategy: Object Oriented GUI (pfc/gui)
```

- Build the application in order to construct the Project Tree.
- From the Project Tree, open the folders

```
$(Prodir)/pfc/gui/Controls/
```

Right click on the **Controls** folder, and add the **editorControl** package to the Project Tree. Rebuild the application.

- Add a New in New Package form with the name `midiform`.



- Rename the *Save*-button to `save_ctl`. Rebuild the application. Enable the **File/New** option of the application task menu, and add the snippet below to it.

```

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

```

- Create a class with the name `midi`. You will find the class declaration below, and the implementation on figure 19.3.

```

class midi
  open core
predicates
  classInfo : core::classInfo.
  generate:(string, string*) procedure (i, i).
end class midi

```

- Add the snippet below to the `ClickResponder` belonging to `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.

```

```
P= outputStream_file::create(File, stream::binary())
```

```
foreach X= list::getMember_nd (MidiCode) do
    P:write(X)
end foreach,
P:close().
```

midiform

D 32 E 32 F 32 G 32 A 32 A 32 A 32 A 32 A 32 A 32 A 96 A 96 A 32 A 32 A# 32 A 32 D5 32 D5 32 D5 32 D5 32 D5 32 D5 32 A# 32 A 96 A 96 A 16 D 32 E 32 F 32 G 32 A 32 A 32 A 32 A 32 A 32 A 32 A 96 A 32 A 32 Bb 32 A 32 D5 32 D5 32 D5 32 D5 32 D5 32 D5 32 D5 32 Bb 32 A 96 A 96

Save

India, bella mezcla de diosa y pantera,
doncella desnuda que habita el Guairá.
Arisca romanza curvó sus caderas
copiando un recodo de azul Paraná.

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 Ybvytruzú.

On her temples shine the pride of feathers,
her language is a wild honeycomb.
A necklace from teeth of tigers and pumas
is a jewel for the muse of Ybtyruz.

```

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).

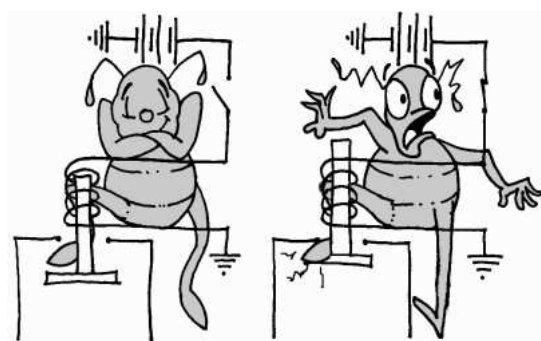
generate(File, Music) :- tomidi(Music, L),
  Events= [0, 255, 89, 2, 0, 0, 0, 255, 81, 3, 7, 161,
          32, 0, 255, 88, 4, 2, 2, 24, 8],
  NC= list::length(Events),  N1= list::length(L),
  Cnt= NC+ N1 + 4, mSz(Cnt, X1, X2, X3, X4),
  MainHeader= [ 77, 84, 104, 100, 0, 0, 0, 6, 0, 0, 0, 1, 0, 48,
               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

```

Figure 19.3: Midi generator

Chapter 20

Bugs



Old computers were based on relays, that are bulky electrical devices, typically incorporating an electromagnet, which is activated by a current in one circuit to turn on or off another circuit. Computers made of such a thing were enormous, slow, and unreliable. Therefore, on September 9th, 1945, a moth flew into one of the relays of

Harvard Mark II computer and jammed it. From that time on, *bug* became the standard word to indicate an error that prevents a computer from working as intended.

Due to bugs, the Visual Prolog compiler frequently returns error messages, instead of generating code and running the corresponding programs. In fact, the Visual Prolog compiler gives more bug alerts than any other language, except perhaps Clean. Students don't appreciate this constant nagging about errors, and seldom try to analyze the error messages, but believe me, you should be thankful to the designers of such a compiler that gives you a chance to clean up all the stains that can mar your program. As I have said, Visual Prolog is good at pinpointing bugs. Notwithstanding, if you do not learn how to deal with its error messages, you will find debugging a random walk that almost never leads to success. Therefore, in this chapter, you will learn how to interpret messages from the compiler.

20.1 Type error

The program below has no bugs; however, if you need to use the unsigned integer which results from the factorial function in order to, say, calculate a binomial number, the compiler may refuse to comply.

```

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(),
           stdio::write("N: "),
           N= stdio::read(),
           stdio::write(fact(N)), stdio::nl.
end implement main

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

For instance, if you try to compile the program of figure 20.1, you will get the following error message:

```

error c504: The expression has type '::integer',
which is incompatible with the type '::unsigned'
```

The error will disappear if you explicitly convert each input argument of `binum/3` to an unsigned integer; you must also convert the output of factorial to an integer.

```

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).

```

Figure 20.1: A buggy program

20.2 Non-procedure inside a procedure

There is another bug that poses many difficulties to beginners. Visual Prolog declares certain predicates as procedures. This is the case of the `run()` predicate that appears in all console programs. This means that you cannot call a **nondeterm** predicate from `run()`. Therefore, if you try to compile the program of listing 20.2, you will get the following error message:

```

error c631: The predicate 'nonprocedure::run/0',
which is declared as 'procedure', is actually 'nondeterm'

```

It is true that the `weight/2` fact predicate is indexed by its integer argument. This means that, given its integer argument, `weight/2` should act like a procedure. However Prolog is impervious to wishful thinking; then you need to define a `getweight/2` procedure, that will retrieve the weight values.

Below, you can see how `getweight` can be defined. Use it as a model for getting around a situation, where you need to call a **nondeterm** predicate from a procedure.

```

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 Non-determ condition

Prolog has constructions that do not accept **nondeterm** predicates. For instance, the condition of an **if-then-else** construction must be a **determ** predicate. Therefore, the program in figure 20.3 will issue an error message:

```

error c634: The condition part of if-then-else may not
have a backtrack point (almost determ)

```

The error will disappear if you define **member** as a **determ** predicate:

```

class predicates
    member:(string, string*) determ.
clauses
    member(X, [X|_]) :- !.
    member(X, [_|Xs]) :- member(X, Xs).

```

20.4 Impossible to determine the type

There are occasions when one needs to specify the type before the instantiation of a variable; type specification can be carried out by the procedure

```

hasdomain(type, Var)

```



```

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).

```

Figure 20.2: A bug caused by a non deterministic predicate

```

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).

```

Figure 20.3: Deterministic predicate required.

In the useless program listed below, if one removes `hasdomain(integer, X)`, s/he will get an error message stating that it is impossible to determine the type of the term `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 Flow pattern

When you do not declare a flow pattern, Prolog assumes that all arguments are input variables. Since this simplified assumption is more often than not wrong, you will get an error message, as in the example below.

```
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).
```

You can fix the bug by declaring *factorial* as an input/output procedure.

```
class predicates
  factorial:(integer, integer) procedure (i, o).
```

Chapter 21

A Data Base Manager

This chapter shows how to create a form with edit fields, and use its contents to feed a data base. However, the most important thing that you will learn is how to access the fields from the form. Edit field windows are stored as fact variables, whose names you must discover, and use to handle the window, as in the example below.

```
onSave(_Source) = button::defaultAction() :- Key= keyvalue_ctl:getText(),
    Email= email_ctl:getText(), T= listEdit_ctl:getText(),
    emails::insert_info(Key, Email, T).
```

21.1 Database manager

Databases are very useful in any commercial application. If you do not know much about databases, I strongly recommend that you browse the Internet for a book on the subject. The main concept of a B+Tree database are:

Indexes that are stored in a special data structure called B+Tree. All you need to know about B+Trees is that they store things in order (for instance, in alphabetical order). When things are in order, like words in a dictionary, it is easy to search for them.

Domains give the register structure. In our example, the contact domain is defined as

```
domains
    contact= email(string, string, string).
```

References are pointers to the places where registers are stored.

After this brief introduction, let us describe the program.

- Create a project whose name is *emailBook* with Object-oriented GUI.
- Using the option *File/Add* from the task menu, add the package *chainDB*. This package is inside the folder *pfc*, in the installation directory. Build the application.
- Create a class *emails* at the root of the Project Tree. Untick the *Creates Objects* option. In section 21.2, you will find the listing of *emails.cl* and *emails.pro*, that you should insert in the appropriate files. Build the application.
- Add a new item to the application task menu. In order to accomplish this task, go to the Project Tree window, left click on **TaskMenu.mnu**. On the TaskMenu dialog, right click the mouse, and choose **New** from the pop up menu. On the appropriate field, write **Create**, that is the name of the menu item. Build the application.
- Add the following code

```
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).

to Project Window\TaskMenu.win\Code Expert\Menu\id_create.
```

You must develop your programs step by step, checking whether each new step is correct before going forward. In the present step, you have added a new **Create** menu item to **TaskMenu.mnu**. You have also inserted code into **onCreate**, in order to read a file name, and create a corresponding database. Build the application, run it, and click on the **Create** option of the application menu. When the file selection dialog asks you for a file name,

type `mails.dbs`. If everything follows the script, the program will create a `mails.dbs` file. Whenever you want to use a data base, you must open it. After using it, you must close it, before turning off the computer. To make sure that we will not forget to close the data base before exiting the application, let us add the snippet

```
clauses
    onDestroy(_) :- emails::data_base_close().
```

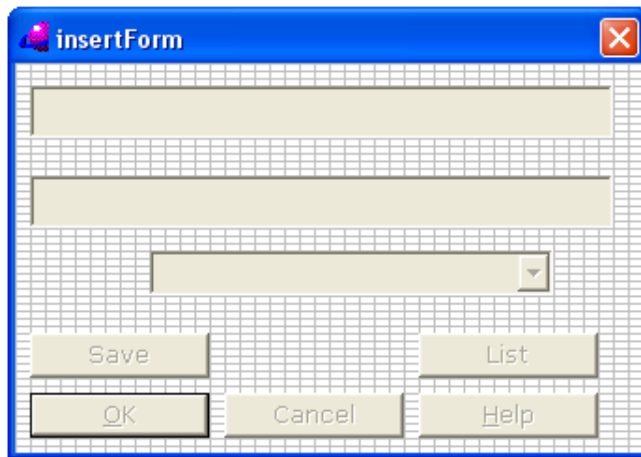
to `Project Tree/TaskWindow.win/Code Expert/Window/onDestroy`.

Now that you know for sure that nobody will forget to close the data base, you can proceed to open it. From the Project Window, click on `TaskMenu.mnu`, and enable the `&Open\tF8` menu item. Add the snippet

```
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(_).
```

to `ProjWin/TaskWindow.win/Code Expert/Menu/TaskMenu/id_file/id_file_open`. Build the application. Then, from the VDE task menu, choose the option `File/New` in `New Package`, and insert the form below onto the root of the project.



The form has a list box. Therefore, it is necessary to add a list of options into it. In order to achieve this goal, add the snippet

```
clauses
    onListEditShow(_Source, _CreationData) :-
        WlboxHandle=listEdit_ctl:tryGetVpiWindow(), !,
        vpi::lboxAdd(WlboxHandle, ["Person", "Commerce"]).
    onListEditShow(_, _).
```

to the *ShowListener* of listEdit:listEdit_ctl.

The last step of this project is to add code to the *Save*, and *List* buttons. Therefore, add

```
onSaveClick(_Source) = button::defaultAction() :-
    Key= key_ctl:getText(),
    Email= address_ctl:getText(),
    T= listEdit_ctl:getText(),
    emails::insert_info(Key, Email, T).
```

to *ClickResponder* of button:save_ctl. Finally, add

```
clauses
    onListClick(_Source) = button::defaultAction :-
        emails::db_list().
```

to the *ClickResponder* of button:list_ctl. Enable the File/New option of the application task menu; add the snippet

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

to TaskWindow/Code Expert/Menu/TaskMenu/id_file/id_file_new. Build the application once more, and that is it.

21.2 btr class

```
%File: 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

%File: 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 The database manager

The function that creates the database is very simple:

```
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(_).
```

The method `db_create/2` does what it means, i.e., it creates a database inside `FileName`, and returns an object to manipulate it. The object is stored in `ChainDB`. From `ChainDB`, one calls `bt_create` in order to obtain a BTree; the BTree package instantiates the variable `ContactName` with a pointer to the BTree. Finally, one closes both the BTree, and the database.

One opens the database with the help of a predicate that is nearly as simple as the one used to create it.

```
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().
```

The predicate `db_open` opens the database stored inside `FileName`, and returns an object that can be used to handle it; the object is stored in `ChainDB`, which is used to open the `email`-Btree. Both `ChainDB` and the BTree pointer are stored in a fact predicate, for future references.

```
assert(dbase(ChainDB, ContactName))
```

When the user kills the application, the `data_base_close` predicate retracts the database manager, and the Btree pointer from the fact database, and closes both the BTree, and `ChainDB`. Up to this point, we have been describing predicates that administrate the database; now we will describe the method that one can use to insert data into the database.

```
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( _, _, _).
```

The database is a filing system. As any filing system, it has two components: a chain of portfolios, each one containing data that one wants to store; and an alphabetical index, that points to the different portfolios, and allows a potential user to discover where the desired information is stored. In the example database, the chain of portfolios is named **"Contacts"**; each portfolio has the following form: **email(string, string, string)**. The first argument of **email** holds the name of a contact, the second argument contains an email, and the third argument informs the type of contact one has with the entity associated with the database entry.

The **chain_insertz** predicate stores the portfolio onto the **"Contacts"** chain, and returns a **RefNumber** to its location. The **RefNumber** will be stored in alphabetical order into the BTree.

To search a **Contact**-email, corresponding to a given **Name**, one searches the BTree, in order to find the reference; once in possession of the reference number, it is straightforward to obtain the portfolio.

```
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").
```

The predicate used to delete an entry is very easy to understand, and I will not linger on it. The predicate that lists the contents of the database in alphabetical order is much more complex, but not difficult to understand, once you know the basic principle of database management.

Chapter 22

Books and articles

In this chapter, I will talk about a few books on Prolog, and programming techniques. You will notice that the books are somewhat old, and out of print. One reason for this is that people do not buy technical books as often as before, since most students hope to get material like this tutorial from the Internet. Another reason is that great computer scientists were very enthusiastic about Prolog in the eighties and nineties. These scientists wrote books that are hard to beat. For instance, I cannot find any modern book that comes close to [Coelho/Cotta].

22.1 Grammars

Prolog excels in language processing, compiler writing, and document preparation. In fact, the language was invented with language processing in mind. Most books deal with standard Prolog, that is slightly different from Visual Prolog. While standard Prolog was designed for fast prototype of small applications and for testing ideas, Visual Prolog is used in large commercial and industrial critical systems. Wellesley Barros, for instance, has designed and programmed a large system for Hospital administration in Visual Prolog. You bet that one cannot afford a failure in such a system due to type errors. In critical situations, type checking and type declarations are very important. In any case, although programming in Visual Prolog, you can profit from ideas presented in books based on standard Prolog. An old book, that you may find in your local library, was written by the Argentinean scientist Veronica Dahl(see [Abramson/Dahl]), one of the greatest modern linguists.

Dr. Cedric de Carvalho wrote a Prolog compiler that generates Java byte-code. This compiler allows you to write applets in Prolog. His compiler is not a fully fledged tool, but you can easily improve it. In the mean time, you

will learn a lot about compilation, parsing, Java, etc. You can start reading Cedric's paper in [Cedric et al]. You can also port the compiler, written in VIP 5.2, to VIP 6.2. The address is

<http://netprolog.pdc.dk>

Another book on Natural Language Processing with many interesting approaches to grammar formalism is [Pereira/Shieber]. This book is more didactic than [Abramson/Dahl]. In fact, [Abramson/Dahl] wrote it for specialists, and [Pereira/Shieber] talk to students that are learning the difficult art of Language Processing. A book that is not specialized in Natural Language Processing, but gives a good introduction to it is [Sterling and Shapiro]. If you want to read an advanced book on the subject, a good option is the book by [Gazdar/Mellish].

22.2 Databases

David Warren, a computer scientist, is the main proponent of using Prolog as a database engine. The book [Warren] is out of print, but you should get a used copy, if you need to learn about databases. It is a classic.

David Warren leads a team that develops XSB, a database oriented Prolog, at Stony Brook University. Although I do not think that his version of Prolog is usable for serious applications, since it relies on other tools for building GUI, it is not compiled (it is not easy to write a beautiful application in XSB), and it is not typed, I still strongly recommend a visit to the XSB page. There you will find many good ideas on databases and Prolog, and tools to prototype them. The address is

<http://www.cs.sunysb.edu/~sbprolog/xsb-page.html>

By the way, XSB provides a set of tools to export its functionality to other languages. Therefore, if you like its features, you can call the XSB DLL from Visual Prolog! The XSB page shows how to call the DLL from Visual Basic. You can use the same approach to call it from Visual Prolog.

22.3 Programming Techniques

If you are having a hard time with concepts such as *Recursive Programming*, *List Operations*, *Cuts*, *Non Deterministic Programming*, and *Parsing*, you should read the book *The Art of Prolog* (see [Sterling and Shapiro]). The ideas presented in this book can be easily ported to Visual Prolog. In Visual

Prolog, you can even improve the original programs. For instance, one of my students was able to create diagrams for the circuits described in Chapter 2 of [Sterling and Shapiro].

My favorite book on Prolog by far is *Prolog by Example*, by [Coelho/Cotta]. It is a kind of FAQ: The authors state a problem, and present the solution. It is also an anthology. The problems and solutions were proposed and solved by great computer scientists. This book deserves a new edition. Try, and try hard, to get a used copy. If you are not able to get a copy, ask permission from the authors, and make a xerox copy.

Part II

Artificial Intelligence

Chapter 23

Search

Patrick Henry Winston, author of a very famous book of LISP, starts his exploration of Artificial Intelligence with search, claiming that search is ubiquitous. Nothing could be closer to the truth. In fact, everything, from robot motion planning to neural network, can be reduced to search. Therefore, like Winston, let us start our study of Artificial Intelligence from search.

23.1 States

In artificial intelligence, the expression *problem solving* refers to the analysis of how computers can use search to solve problems in well-circumscribed domains. One question arises from this definition: What does an intelligent program inspect in its search for the solution to a problem? It inspects a maze of states connected by operations that cause feature transitions.

An object has states, i.e., it has a set of features that determines its situation at a given instant. For example, Euclid defined point as an entity whose sole feature is its position. Therefore, the state of a punctual object is given by its coordinates. However, in general, one can use other features besides position, in order to define the state of an object:

Color: Red, green, blue, etc.

Integrity informs whether the object is whole, or unmounted.

Attitude gives the direction the object is facing.

Safety says whether the object is protected against destruction.

Each feature has one or more attributes, and values for the attributes. For instance, the position has three attributes, which are the coordinates of a

point; in a 2D space, the position attributes can have values in the Cartesian product $\mathbb{R} \times \mathbb{R}$; therefore, the tuple (50, 200) gives valid values for the position. To make a long story short, one can say that state is a snapshot in time. Each attribute that one can change to solve a problem is called degree of freedom. Then a point in a 3D space has three degrees of freedom.

Operator is a procedure that can be used to make the transition from one state to another. The artificially intelligent program starts at an initial state and uses the allowable operators to move towards a goal state. A set of all states that the objects under consideration can take, and all transitions that lead from one state to another is called state space.

23.2 Search tree

Search tree is a popular diagram, where each state is represented by a labeled node (identifier inside a circle), and each possible transition from one state to another is described by a branch of an inverted tree. In figure 23.1, you can see a picture of a search tree.

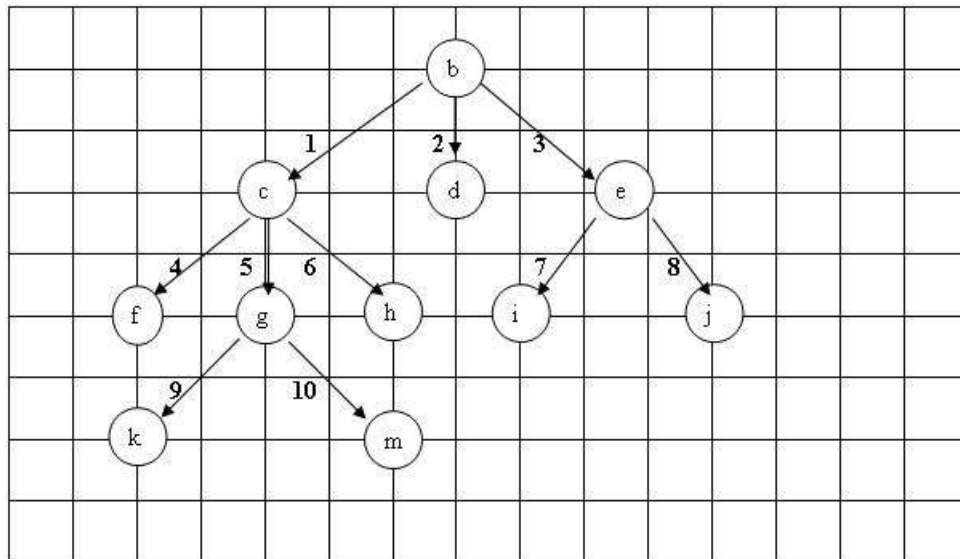


Figure 23.1: A search tree

In figure 23.1, the nodes b, c, d, e, f, g, h, i, j, k, and m are the states that the object may go through. The branches that connect two nodes are the operators that cause the transition from one state to another. For instance, the branch that is labeled 3 in figure 23.1 is the operator that changes state b into e. When a node X can be obtained from another node Y using only one

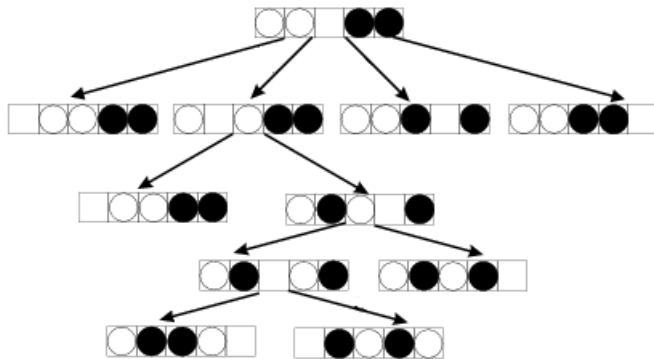


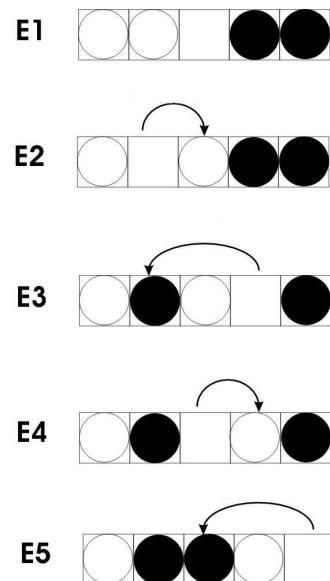
Figure 23.2: Puzzle

operator, one says that X is daughter of Y , or that Y is mother of X . Therefore, the node c is daughter of b , since one can obtain c using the operator 1. When a node has no daughter, it is called a leaf. A node that has no mother is the root of the search tree.

A more concrete example will help you to understand the concepts that we have studied above. Two black pieces, like the ones used in the game of checkers, and two white pieces are arranged as shown in the figure. The black pieces are separated from the white ones by a space. One wants to place the black pieces between the white pieces. Two operations are available: (1) Slide a piece to an empty space, and (2) Jump over a piece, landing on an empty space. The figure on the right hand side shows a step by step solution of this problem.

In the problem that we want to solve, the objects of interest are restricted to the four pieces, and the empty space. The relative positions of the pieces, and of the empty space, define a state. The initial state **E1** is shown at the top of the figure. The goal **E5** is at the bottom of the figure.

In figure 23.2, you can see a search tree for the puzzle that was described above. One could say that the states are like cities on a map, and the operators are similar to roads connecting the cities. The problem is solved when one finds a path that leads from the root to the goal, and the most obvious way to find the path is to go from city to city until finding one that matches the goal. This process is called search.



23.3 Breadth First Strategy

The search is said to be exhaustive if it is guaranteed to generate all reachable states before it terminates in failure. One way to perform an exhaustive search is by generating all the nodes at a particular level before proceeding to the next level of the tree; this method is called the Breadth First Strategy, and ensures that the space of possible operations will be examined systematically.

In the tree of figure 23.1, one starts by examining section (c-d-e), containing the nodes that are one branch away from the root. Then one goes to section (f-g-h-i-i), which are two nodes away from the root. Finally, the search engine visits section (k-m). The program that performs the search operation has a parameter called *Queue*, containing a list of candidate paths. Each path is represented by a list of nodes, where a node has the form `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 Extensions= [Daughter || nextLevel(T, Daughter)],
        ExtendedQueue= list::append(Queue, Extensions),
        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).

```

Initially, the `Queue` has a single path, that starts and ends at the root:

```
bSearch([[r(0, "b")]], L)
```

Since this path does not lead to the goal, it is extended to every daughter of "b", and the extensions are appended to the end of the `Queue`:

```

bSearch([ [r(1, "c"),r(0, "b")],
          [r(2, "d"),r(0, "b")],
          [r(3, "e"),r(0, "b")]], L)

```

At this point, the robot has visited only one node, to wit, "b". Then the extensions of node "c" are appended to the end of the `Queue`.

```

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)

```

Node "d" is visited, but since it has no daughter, it is simply removed from the Queue.

```

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)

```

The daughters of "e" are appended to the end of the Queue:

```

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)

```

Node "f" is removed from the Queue, without leaving offspring. The daughters of "g" are added to the end of the Queue.

```

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)

```

Nodes "h", "i", and "j" are visited in turns, and removed from the Queue. The search ends when the algorithm finds the path

```

[r(9, "k"), r(5, "g"),r(3, "c"),r(0, "b")]

```

The procedure

```

prtSolution(L) :-
    foreach P= list::getMember_nd(list::reverse(L)) do
        stdio::write(P), stdio::nl
    end foreach.

```

reverse this path, and prints each one of its members. The final result which comes on the screen is shown below.

```

r(0,"b")
r(1,"c")
r(5,"g")
r(9,"k")

```

```

D:\vipro\aityros\aiprogs\breath\Exe>pause
Press any key to continue...

```

At this point, you are ready to try the algorithm on a real problem, to wit, the slide puzzle.

```

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).

```

The solution of the puzzle, as printed by the computer, is presented below.

```

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
Pressione qualquer tecla para continuar. . .

```

23.4 Depth First Strategy

Analyze¹ figure 23.1. The depth first strategy, after visiting "b", goes to its leftmost daughter, which is "c". After visiting "c", it goes to "f". Only

¹Yes, I know that *analyse* should be written with an *S*, since it comes from ἀνάλυσις, a Greek word. The problem is that the Americans insist on writing it with a *Z*. That makes me remember Professor Wolfgang Fuchs, who has an Erdős number equal to one, since he worked directly with Paul Erdős. That makes my own Erdős number be as low as 2,

after visiting all descendants of "c", the depth first strategy visits "g". To implement this kind of strategy, all you need to do is to append the children of a given node to the front of the `Queue`, instead of appending them to the tail. Here is the only necessary modification made to the previous program:

```
dSearch([T|Queue],Solution) :-
    if solved(T) then Solution= T
    else  Extensions= [Daughter || nextLevel(T, Daughter)],
          ExtendedQueue= list::append(Extensions, Queue),
          dSearch(ExtendedQueue, Solution) end if.
```

I also changed the name of the predicate to `dSearch` for obvious reasons.

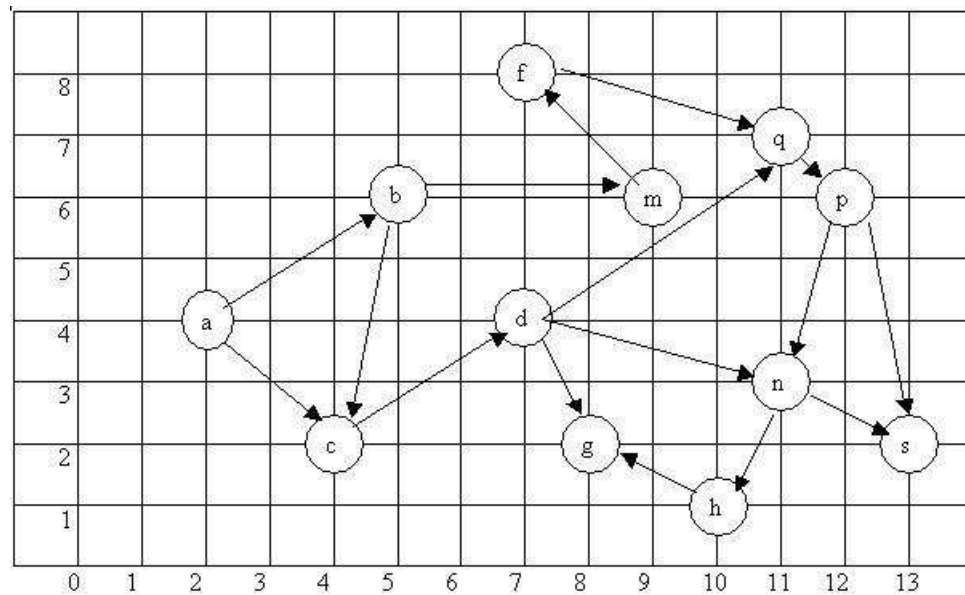
23.5 Heuristic search

The search strategies that we have examined until now do not make any use of domain specific knowledge to choose a branch to investigate. This weakness is corrected by the heuristic strategy, that evaluates the cost of going from the root to the goal passing through the node N , before jumping into N . The evaluation function is given by the following formula:

$$f(N) = g(N) + h(N)$$

In this formula, $g(N)$ is the known cost of going from the root until the N node; the function $h(N)$ gives the estimated cost of going from N to the goal node. The heuristic strategy sorts the `Queue` in such a way that paths with lower values of $f(X)$ are visited first. Let us use the heuristic strategy to find the path between two cities on the map below.

because Dr. Fuchs was my minor adviser. However, it is not to brag that I have a low Erdős number that I am writing this footnote, but to tell the opinion that Dr. Fuchs had about the French language. He said that French is certainly the most beautiful and sweet sounding language in the world; the only problem is the awful pronunciation that French people insist on maintaining.



However, before starting, I would like to tell a story. I like obscure languages, and have learned quite a few of them. For instance, I know Ancient Egyptian, Homeric Greek, Esperanto, Latin, etc. When I was a teenager, I even studied Guarani, a second rate language spoken only in Paraguay, an underpopulated country.

A guy or gal has translated a book by a certain V. N. Pushkin from Russian into one of those second rate languages that I learned. The book's title was: *Heuristics, the Science of Creative Thought*. When I read Pushkin's book in translation, I understood how difficult is to find a good heuristic function, and how much more difficult it is to translate a text from a language to another, specially if you do not know the subject treated in the text well. Natural language processing is difficult not only for computers, but for people too. For instance, Dr. Pushkin tried to explain the rules of chess, in order to use the game as a test bed for Artificial Intelligence. In his explanation, he has said that the knight moves along a Γ -shaped path, where Γ is a letter of the Russian alphabet. The translator came out with the following rendering of the original: *The knight moves along an upsidedown L-shaped path*. After that long digression into the realms of translation, let us return to heuristic search. The cities will be represented by their coordinates:

```
xy("a", 2, 4). xy("b", 5, 6).
```

Two place facts describe the one way roads that connect the cities:

```
op("a", "b" ). op("b", "m"). op("m", "f").
```

You have learned that, in the heuristic search, the queue must be sorted so that the most promising paths come forward. The sorting algorithm that you will learn is very interesting, and was invented by Tony Hoare. Each path is represented by the structure shown below.

```
t(real, real, it)
```

where the first argument gives the value of the heuristic function F for the node; the second argument gives the value of $g(N)$. The sorting algorithm has a comparing predicate as its first argument, and pushes the most promising branches to the front of the queue.

```
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.
```

The complete program is implemented below.

```
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).

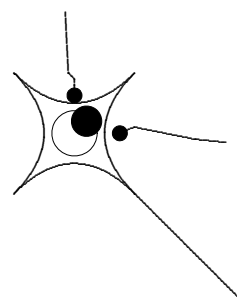
```


Chapter 24

Neural Networks

Spain is not distinguished for its science. In fact, there are families in France that have twice as many Nobel Laureates in science than the whole of Spain. In Europe, there are very few countries with less Nobel Laureates than Spain; one of them is Serbia, and one of the Serbians who did not receive the Nobel prize is Nikola Tesla, and a country that has Nikola Tesla, does not need Nobel laureates. Notwithstanding, the only two Spanish Nobel laureates are counted among the greatest scientists of all time, together with Newton, and Archimedes. In this chapter, you will learn about one of them.

The Spanish scientist Don Santiago Ramón y Cajal studied the physiology of the nervous system and concluded that it was driven by cells that he called neurons. Nowadays, scientists believe that a given neuron receives inputs from other neurons through links called synapses. By the way, synapsis is the Greek word for link. Typically, a neuron collects signals from others through a set of structures called dendrites, which means *tree shoots* in Greek; if the intensity of the incoming signals goes beyond a certain threshold, the neuron is fired, and sends out electrical stimuli through a stem known as an axon, which splits into the branch-like dendrites, whose synapses will activate other neurons. So, to sum up... at the end of each dendrite, a synapse converts neuron activity into electrical stimulus that inhibits or excites another neuron.



In 1947, Warren McCulloch and Walter Pitts proposed the first mathematical model for a neuron. Their neuron is a binary device, with a fixed threshold; it receives multiple inputs from excitatory synapses, which gives to each input source a weight that is effectively a positive integer; inhibitory inputs have an absolute veto power over any excitatory inputs; at each pro-

cessing step the neurons are synchronously updated by summing the weighted excitatory inputs and setting the output to 1 if and only if the sum is greater than or equal to the threshold.

The next major advance in the theory of neural networks was the perceptron, introduced by Frank Rosenblatt in his 1958 paper. Rosenblatt was a professor at Cornell University, that happens to be my alma mater. However, I did not have the good fortune of knowing him, since he passed away many years before my arrival at that university.

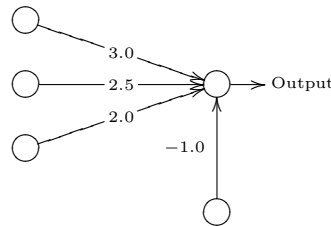
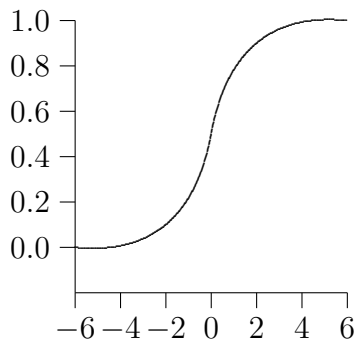


Figure 24.1: A three input perceptron



The perceptron adds up the input signals multiplied by a weight (a different weight for each input signal), and if the result is greater than a threshold, it produces a non zero output. This scheme was improved by feeding the weighted sum to a sigmoid function, and taking the value returned as the output of the perceptron. If you plot a sigmoid, it takes the shape of the Greek letter ς , that is known as final sigma; hence the name sigmoid. The sigmoid is defined as

$$\varsigma(x) = \frac{1}{1 + e^{-x}}$$

In Prolog, this definition becomes:

```
sigmoid(X)= 1.0/(1.0+math::exp(-X)).
```

Figure 24.1 above shows a perceptron with three inputs. Let the inputs be x_1 , x_2 , and x_3 . In this case, the output will be the result of the expression

$$\varsigma(3x_1 + 2.5x_2 + 2.0x_3 - 1)$$

One interesting feature of the perceptron is its learning abilities: You can train the perceptron to recognize input patterns. In fact, given enough examples, there is an algorithm that modifies the weights of the perceptron, so

that it can discriminate between two or more patterns. For instance, a two input perceptron can learn to tell when a **and-gate** will output 1, and when it will output 0.

The learning algorithm of the perceptron must have a set of training examples. Let us assume that it is learning the workings of an **and-gate**. Here are the examples:

```
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).
```

The perceptron has a predicting procedure, that uses the formula $\varsigma(\sum w_i x_i)$ to predict the output value, given a set of inputs. In Prolog, the predicting procedure can be coded as shown below.

```
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).
```

The weights are stored in a database.

```
facts
    weight:(unsigned, real).
clauses
    weight(0, 1.5). weight(1, 2.0). weight(2, 1.8).
```

The initial weights are fictitious. The true weights must be found by the learning algorithm. The procedure `getWgt` has a very simple definition:

```
getWgt(I, W) :- weight(I, W), !.
getWgt(_I, 0.0).
```

You may ask why one needs to define `getWgt` at all; it would be easier to use `weight` directly; the problem is that `weight` defines a `nondeterm` predicate, and we need a procedural `weight` retrieving predicate. By the way, one needs a procedure to retrieve the examples too:

```
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).
```

The learning algorithm is based on the calculation and correction of the error made by the predicting function. The error of a given example is calculated by the following predicate:

```
evalError(Obj, E) :- nn:predicted_out(Obj, VC),
                    nn:egratia(Obj, _I1, _I2, V),
                    E= (VC-V)*(VC-V).
```

What we really need is not the error of a given example, but the sum of the errors for each example. One can calculate this total error as shown below.

```
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.
```

The computational scheme *foreach* will pick each example from the `Exs` list, calculate the corresponding error, and add the result of the calculation to the accumulator `acc`, which is stored in a global variable.

```
class facts
    acc:real := 0.0.
```

The weights will be updated through the gradient descent method in such a way as to reduce the error to a minimum.

```
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]).
```

Gradient descent is an optimization algorithm, that finds a local minimum of a function by taking steps proportional to the negative of the gradient of the function at the current point. At each step of the gradient descent method, one updates the weight according to the following formula:

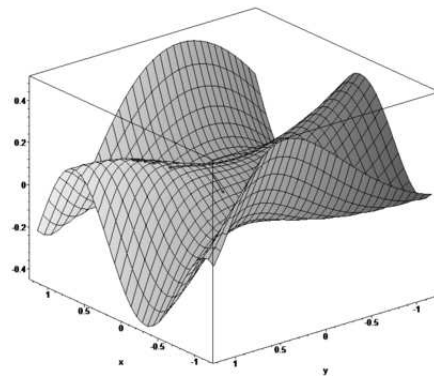
$$\omega_{n+1} = \omega_n - \gamma \nabla \mathbf{error}(\omega)$$

In this formula, if $\gamma > 0$ is a small enough number, $\mathbf{error}(\omega_{n+1}) < \mathbf{error}(\omega_n)$. If one starts with ω_0 , the sequence $\omega_0, \omega_1, \omega_2 \dots$ will converge to a minimum. The minimum will be reached when the gradient becomes zero (or close to zero, in practical situations). You can think that there is a terrain, with mountains and valleys; the position on this terrain is given by the coordinates ω_1 and ω_2 ; the heights of mountains, and the depths of valleys represent errors. The training algorithm moves the network from an initial position to a valley, i.e., to a position where the error reaches a local minimum.

The ideal situation happens when the network reaches a global minimum. The Americans who serve in counter-espionage like to say that they are the best of the best. That is what one wants for the solution of any optimization problem: The best of the best. Notwithstanding, gradient methods more often than not become trapped in local minimum. This happens because when the network reaches a minimum, it doesn't matter whether the minimum is local or global, its gradient comes close to zero; in fact, the algorithm knows that it reached a minimum, because the gradient reaches a value close to zero. The problem is that this criterium does not say whether the minimum is local or global. The dictionary says that a *gradient* gives the inclination of a road or railway. At the top of a mountain, or at the bottom of a valley, the gradient is zero. If one uses a level at the top of a mountain, or at the bottom of a valley, s/he will not detect any inclination at all.

A neural network training algorithm updates the weights of a neuron, until the error stays in some small neighbourhood; then it stops.

```
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 Neuron description

A neuron like the one shown in figure 24.2 can be described by the following equation:

$$\omega_1 x_1 + \omega_2 x_2 + \omega_0 = 0$$

This is the equation of a straight line. If one uses more entries, s/he will get straight lines in high dimensions, but they will remain straight lines nevertheless. Let us plot a logical function known as *or*-gate. It has two inputs, x_1 , and x_2 .

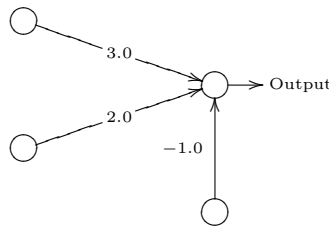
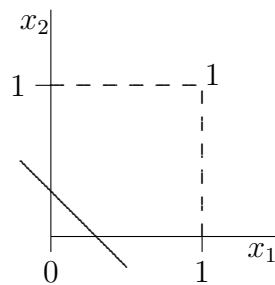


Figure 24.2: A three input perceptron



The *or*-gate function produces 1, if either $x_1 = 1$, or $x_2 = 1$; it also produces 1, if both $x_1 = 1$, and $x_2 = 1$. The graphics of the figure shows the behavior of the *or*-gate. What the neural network of figure 24.2 does is to draw a straight line separating the points where *or*-gate is 1 from the point where it is 0, as you can see from the figure. Then, all that a perceptron can do is to learn how to draw straight

lines in a state space, in order to separate clusters of points with a given property. The problem is that one often cannot draw such straight lines in a multidimensional space.

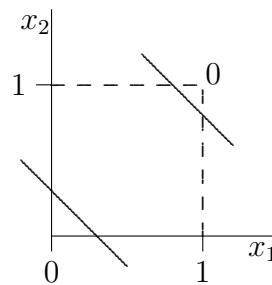
The *xor*-gate is much more useful than the *or*-gate for two reasons. First, the two inputs of the *xor*-gate can represent logic propositions like:

$x_1 = \text{the machine is on}; x_2 = \text{the machine is off}$
 $x_1 = \text{the door is open}; x_2 = \text{the door is closed}$

Since the machine cannot be on, and off at the same time, $x_1 = 1$ **and** $x_2 = 1$ cannot be true; $x_1 = 0$ **and** $x_2 = 0$ cannot be true either. To describe logical gates, one often uses truth tables, that give the output value for all values of the input variables; below, you can see a truth table for the *xor*-gate.

x_1	x_2	output
1	1	0
1	0	1
0	1	1
0	0	0

You can see from the figure that a single straight line cannot discriminate between the values of a *xor*-gate. However, two straight lines can separate the values that happen to be zero, from values that happen to be 1, as you can see in the figure. Likewise, a single perceptron cannot learn how to calculate the output values of a *xor*-gate, but a network with more than one perceptron, like the one that you see



in figure 24.3, can perform the task. In the next paragraphs, you will learn how to use objects to implement a multi-layer neuron network in Prolog.

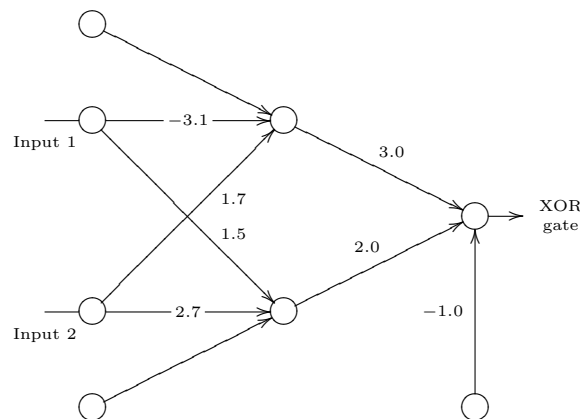


Figure 24.3: A neural network that can learn how to calculate a xor-gate

24.2 Implementation of a multi-layer network

- Create a new console project.

```
Project Name: neurons
UI Strategy:  console
```

- Select the root of the *Project Tree*. Choose the option *File/New* from the VDE Task Menu. This will open the *Create Project Item* dialog. Select the *Class* tag from the left hand side pane; create a **network** class, that *Creates Objects*. Press the *Create*-button. Add the class specification to `network.cl`, `network.i`, and `network.pro`.

```
class network : network %File: network.cl
  predicates
    sigmoid:(real) -> real.
end class network
```

Add the class interface to `network.i`:

```
%File: network.i
interface network
  open core
domains
  example= e(real, real, real).
  examples= example*.
predicates
  setWgt:(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
```

Add the listing below to `network.pro`. Build the application.

```
%File 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

```

Objects of the *network* class can play the role of a neuron. In the main module, you will find the training program, and a test `run()` predicate for the **and**-gate, and also for the **xor**-gate. Build the application, and *Run in Window*.

```

%File: 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 Running the two layer neural network

After building the `neurons` application, and running it, you will get the following result:

```

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. . .
```

As you can see, the learning algorithm converged to a behavior that is very close to the desired behavior of the **xor**-gate.

24.4 A Historical Perspective

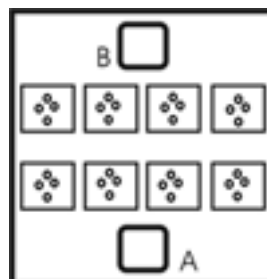
There are people who claim that, *when Rosenblatt introduced the concept of perceptron, Minsky crusaded against him on a very nasty and personal level, including contacting every group who funded Rosenblatt's research to denounce him as a charlatan, hoping to ruin Rosenblatt professionally and to cut off all funding for his research in neural nets.* As I told you, I was not lucky enough to know Rosenblatt, though I know Minsky well (I interviewed him when I used to double as a newspaper reporter), and I am sure that such slanders against Minsky simply aren't true. What Minsky and his collaborator Papert did was to point out the weakness of the theory that the intelligence can emerge purely from the learning activities of a neural network. In fact, Minsky practically demonstrated a theorem that neural network cannot learn how to do certain things; this limitation is not exclusive to artificial neural networks; there are things that even our natural neural network cannot learn: In the cover of Minsky's book (*Perceptrons*), there is a labyrinth like drawing; the reader should discover whether the labyrinth interlaced walls are connected or not. Minsky claims that it is impossible to train a neural network to perform such a task, and it does not matter whether the neural network is natural, or artificial.

Chapter 25

Alpha Beta Pruning

Mancala is a family of board games that are very popular in Africa and Asia; it is a count and capture game. The game of this family best known in the Western world is Kalah. Mancala games play a role in many African and some Asian societies comparable to that of chess in the West. It is also popular in India, specially among the Tamil women.

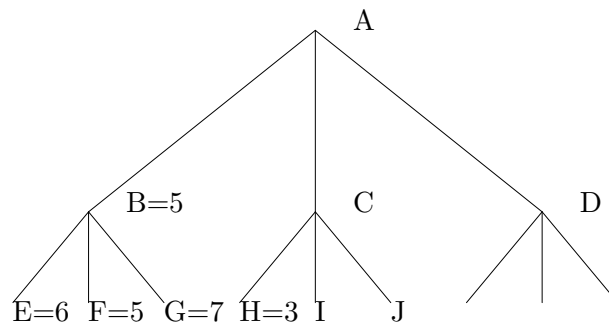
In the variant of Mancala that we are going to learn, the board has a series of holes arranged in two rows, one for each player. The holes may be referred to as *pits*, *pots*, or *houses*. Two large holes at two ends of the board, called stores, are used for holding captured pieces. Playing pieces are seeds, beans, or stones, that are placed in and transferred between the holes during play. At the beginning of a player's turn, s/he selects a hole from his/her side, and sows its seeds around the board.



In the process of sowing, all the seeds from a hole are dropped one-by-one into subsequent holes in a motion wrapping around the board. Depending on the contents of each hole sown in a lap, a player may capture seeds from the board. If the hole contains less than 3 seeds, the player can capture its contents, and also the seed that s/he would drop in that hole. The captured seeds go to the players store. The game ends if one player captures at least 10 seeds more than his/her opponent. The game also ends if one player has no seeds on his side of the board. In any case, the player that captures more seeds wins the game.

25.1 Move generation

The most cherished games are those where an oponent is pitted against the other, in a combat of brains. However, I would like to point out that even in this kind o game, Erast Fandorin always win, as you can check by reading *The Jack of Spades*. However, computers are not as lucky as the state counsellor; therefore programmers need a tool to analyze the game; this tool is the minmax tree (see figure 10.1). In such a diagram, a player is called the Maximizer, while the other is the Minimizer. In the analysis, the Maximizer is supposed to achieve the maximum value for an evaluation function, hence the name; the minimizer tries to force the evaluation function to a minimum value. To tell the whole story, one does not use a tree, like the one in figure 10.1. To understand why, let us suppose that the maximizer is analyzing the move B, of the tree below. She knows that the Minimizer will play F, since F is the counter-move of least value. Therefore she concludes that B is worth 5 from her point of view; now she proceeds to the analysis of C; when she discovers that H=3, she can suspend the search, because a move to C will be worth at most 3, and she already has 5 in B. This process of skipping branches like I and J is called Alpha-pruning. When reasoning from the point of view of the Minimizer, all one needs to do is to invert the signal of the evaluation function, and apply the same algorithm that was used in the case of the Maximizer; in this case, skipping branches is called Beta-pruning. Finally, if the node is a leaf, evaluation is performed by a static function, i.e., a function that does not descend to the branches.



Implementing a game of strategy is a difficult task. In order to accomplish it, we are going to keep the move generation as a separate class, so the complexities of the game itself will not interfere with our strategic reasoning. Create a new console project.

```

Project Name:  kalahGame
UI Strategy:   console
  
```

Build the application in order to insert the main class into the *Project Tree*. Then create a class `mv` by selecting the option **New in New Package** from the IDE task menu.

```
%File: 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

%File: 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

```

Now you are ready to implement the main class, that will search the game-tree for the best move.

```

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, BestVal).

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).

```

Like Erast Fandorin, I don't like games; besides this, I don't have his luck, which means that I always loose. All the same, I pitted my wits against the computer, and inevitably lost. Here is the beginning of the game:

```

Present situation:
Move: -1
    6    6    6    6
  0      0      0      0      a
    6    6    6    6
0
Move: 0
    7    7    7    0
  0      0      0      0      b
    7    7    7    6
Present situation:
Move: 7
    8    8    8    1
  0      0      0      0      a
    8    8    7    0
1
Move: 1
    9    9    1    0
  2      0      0      0      b
    9    9    8    1
Present situation:
Move: 4
    10   10    0    1
  2      4      4      4      a
    1    11    9    0

```

Index

- `:-`, 49
- Accumulator parameters, 63
- and, 48
- animation, 163
- append, 112
- average, 60
- B+Tree, 201
 - handling, 205
- background
 - transparent, 90
- backtrack, 52
- Blinking animation, 159
- Botany, 151
- brush, 89
 - setBrush(B), 89
- Building, 15
- Class
 - creating, 39
 - interface, 135, 157
 - with objects, 156
- class
 - open, 52
 - userControl, 167
- class facts, 51
- class predicates, 51
- Code Expert, 18, 35
- Coelho
 - How to solve it, 109
 - Prolog by Example, 109
- Colmerauer, 118, 143
- color, 89
- Compiling a program, 15
- conjunction, 48
- console
 - Run in Window, 58
 - testing, 58
- console applications, 57
- cut, 57
- days in a year, 118
- determ, 52
- domains
 - list declaration, 104
- doPaint
 - Java, 28
- draw
 - GDIObject, 88
- Drawing predicates, 53
 - pictClose/1, 159
 - pictDraw/4, 159
 - pictOpen/1, 159
 - ropSrcCopy, 160
- drawing predicates
 - drawEllipse/2, 53
 - getVPIWindow(), 53
- Edit Control
 - getText(), 201
- Edit control
 - fact variable, 201
- Edit field
 - accessing, 39
 - contents, 39
- editor, 167
 - editorControl, 167

- loading, 168
- saving, 168
- Enable File/New, 17
- Euclid, 79
- event
 - onPainting, 87
- Executing a program, 15
- factorial, 57
- facts
 - class facts, 51
- file, 132
 - readString, 132
 - writeString, 132
- file (class), 132
- Flickering
 - elimination, 159
- Flickering animation, 159
- Form
 - create a new one, 36
 - creating, 17
 - prototype, 17
- form
 - new(Window), 87
 - onPainting, 88
 - show(), 87
- Format, 102
- Forms, 17
- four colors, 119
 - Appel and Haken, 119
- Functions
 - definition, 41
- game
 - minimax algorithm, 125
 - move generation, 123
 - Nim, 121
- Games, 155
- GDIOject, 88
- generate and test, 119
- Grammar
 - books, 209
- grammar
 - English parser, 81
 - German parser, 80
- Grammars, 79
- Graphics
 - turtle, 139
- GUI
 - Object-oriented, 33
- Hoare, 115
- Horn clauses, 49
 - facts, 49
 - head, 49
 - tail, 49
- implication, 49
- Interlingua, 144
- International language, 146
- iteration, 62
- Java
 - doPaint, 28
- junctions, 48
- L-Systems, 151
- Laplace, 115
- Latino Sine Flexione, 143
- Linaeus, 151
- Lindenmayer, 151
 - L-Systems, 151
- List
 - domains, 104
 - of string, 68
- list
 - append, 112
 - average, 60
 - domains, 60
 - intersection, 111
 - member, 110
 - reverse, 113
 - sort, 115

- Lists, 58
 - iterative reduction, 64
 - recursive reduction, 64
 - reduction, 64
 - schemes, 64
 - zip, 64
 - ZIP scheme, 66
- lists
 - [X:Xs], 59
 - head, 59
 - patterns, 59
 - tail, 59
- logical and, 48
- logical if, 49
- logical implication, 49
- logical junctors, 48
- Logo, 140
- Lope de Vega, 155
- LsF
 - Latino sine Flexione, 147
- map
 - four colors, 119
- mask, 90
- member, 110
- Menu
 - Enable option, 17
 - task, 37
- Mouse, 27
 - add code, 27
 - event, 27
 - MouseDown, 27
- multi, 52
- multi-solutions, 46
- Natural Language Processing, 143
- Newton, 115
- Nim, 121
- nondeterm, 52
- Object
 - interface, 157
- onPaiting, 88
- open class, 52
- painting
 - event handling, 87
- paiting
 - drawPolygon, 88
- Papert, 140
- pattern style, 89
- Peano, 139
 - curve, 143
 - Latino Sine Flexione, 143
 - recursion postulate, 142
- Pereira
 - Luis Moniz, 119
- Piaget, 140
- pictDraw, 90
- pictLoad, 90
- picture
 - mask, 90
 - pictDraw, 89
 - pictLoad, 89
 - transparent, 90
- polymorphism, 110
- predicates
 - class predicates, 51
 - declarations, 51
 - definition of, 42
 - determ, 52
 - multi, 52
 - nondeterm, 52
 - procedure, 52
 - type/flow patterns, 51
- procedure, 52
- Project
 - building, 15
 - Create item, 36
 - creating, 13
 - settings, 14
- Project Tree, 34
- Project tree

- add code, 18
- Prolog
 - How to solve it, 109
- Pushkin, 155
- quicksort, 115
 - Hoare, 115
- readString, 132
- real*, 60
- Recursion
 - postulate, 142
- recursion
 - iteration, 62
- Roussel, 118
- setBrush, 89
- solutions, 42
 - Multi solutions, 46
- sort, 115
- String, 68, 102
 - frontToken, 68
- string
 - format, 102
 - from file, 132
- Sundback
 - Gideon, 66
- systems
 - Incremental development, 121
- Task menu, 37
- timer
 - handling, 166
- transparent background, 90
- Tratactus, 79
- Turtle graphics, 139
- Type, 102
 - Char, 102
 - Int, 102
 - Real, 102
 - String, 102
- user control, 167
 - editorControl, 167
- Van Emden, 115
- variables, 43
- Warren, 118
- window
 - GDIObject, 88
- Wittgenstein, 79
- writeString, 132

Bibliography

- [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 Carroll] Lewis Carroll. The Game of Logic. Hard Press, 2006.
- [Sterling and Shapiro] Sterling and Shapiro. "The Art of Prolog". MIT Press / January 1986.
- [Coelho/Cotta] Coelho, H. and Cotta, J. "Prolog by Example". (1988) Berlin: Springer-Verlag.
- [HowToSolveItWithProlog] 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 Hughes] John Hughes. Why Functional Programming Matters. *Computer Journal*, vol. 32, number 2, pages 98-107, 1989.
- [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.