Visual Prolog Version 5.x

Language Tutorial

Prolog Development Center A/S
H.J. Holst Vej 3-5C,
DK - 2605 Broendby, Copenhagen
Denmark

Copyright

Table of Contents

Using This Manual

If you have never programmed in Prolog before, you should read all of this manual. Chapters 1-10 cover Prolog fundamentals, and you should read them before attempting any serious application development. The later chapters become very important as soon as you want to do serious programming. If you program in a procedural programming language such as C, Pascal, or Basic, pay close attention to the procedural discussions. At the end of Chapter 4, you will find a procedural overview of the material covered in the first three tutorial chapters. We also provide procedural discussions of recursion in Chapter 6.

If you have programmed in other Prologs and have a good understanding of Prolog fundamentals, you won't need much review. However, Visual Prolog has several extensions and is different from interpreted Prologs. We recommend that you read the release notes and Chapters 1 as an introduction. Chapter 3 explains the structure of a Visual Prolog program and Chapter 5 introduces the declarations. We also recommend that you read Chapter 8 on Visual Prolog's facts section, and Chapter 14, on the external database.

Chapters 12 through 16 provide valuable information that you will need if you plan to do serious programming.

If you think you could use a review of Visual Prolog programming, we recommend that you read from Chapter 16 on.

This user's guide is divided into four parts: a short introduction to the Visual Prolog environment; then the first ten tutorial chapters – which teach you how to program in Visual Prolog; then six chapters – which gives an overview of the predefined features of Visual Prolog - the standard predicates, the last part gives a complete systematic overview of the language, modular programming and interfacing to other languages.

Here's a summary of each chapter in this book:

Part 1: Introduction to Visual Prolog

Chapter 1: Getting Started describes how to install Visual Prolog on your computer, how to use Visual Prolog's Visual Development Environment for running examples supplied for this book, provides a quick guide through the steps involved in creating, running, and saving your first Visual Prolog program. This chapter explains how to apply Visual Development Environment's Test Goal utility to run the Language Tutorial examples supplied with Visual Prolog.

Part 2: Tutorial Chapters 2 – 10: Learning Visual Prolog

Chapter 2: Fundamentals of Prolog provides a general introduction to Prolog from a natural language perspective and discusses how to convert natural language statements and questions into Prolog facts, rules, and queries.

Chapter 3: Visual Prolog Programs covers Visual Prolog syntax, discusses the sections of a Visual Prolog program, and introduces programming in Visual Prolog.

Chapter 4: Unification and Backtracking describes how Visual Prolog solves problems and assigns values to variables.

Chapter 5: Simple and Compound Objects discusses declaring and building structures in Visual Prolog.

Chapter 6: Repetition and Recursion explains how to write repetitive procedures using backtracking and recursion; also introduces recursive structures and trees.

Chapter 7: Lists and Recursion introduces lists and their use through recursion, as well as covers general list manipulation.

Chapter 8: Visual Prolog's Internal Fact Databases discusses using of Visual Prolog's facts sections for adding facts to your program at run time and for storing global information.

Chapter 9: Arithmetic and Comparison introduces the full range of arithmetic and comparison functions built into Visual Prolog and gives examples that demonstrate how to use them.

Chapter 10: Advanced Techniques controlling the flow analysis, using reference variables, pointers to predicates, the binary domain, term conversions, using the dynamic cut, tools and techniques for error and signal handling, and programming style for efficient programs.

Chapter 11: Classes and Objects gives a short introduction to object oriented programming and introduces the object mechanism in Visual Prolog.

Part 3: Tutorial Chapters 12 – 16: Using Visual Prolog

Chapter 12: Writing, Reading, and Files introduces I/O in Visual Prolog; covers reading and writing, and file and directory handling.

Chapter 13: String-Handling in Visual Prolog covers string manipulation, including string comparison and string conversion, plus constructing and parsing strings.

Chapter 14: The External Database System covers Visual Prolog's external database system: chained data, B+ trees, storing data (in EMS, conventional memory, and hard disk), and sorting data. Includes examples of constructing real database applications.

Chapter 15: System-Level Programming introduces the low-level control supported within Visual Prolog: system calls, BIOS, low-level memory addressing, and bit manipulation.

Chapter 16: Example Prolog Programs provides a diverse group of Prolog program examples demonstrating some of the elegant ways Prolog solves complex problems.

Part 4: Reference Chapters 17 – 18: An overview

Chapter 17 Elements of the Language gives a systematic overview of all the features in the Visual Prolog language. The chapter also introduces modular programming.

Chapter 18 Interfacing with Other Languages gives a description on how to interface with C and other languages

PART 1

Introduction to Visual Prolog

CHAPTER 1

Using Visual Prolog's Visual Development Environment

This chapter describes the basic operation of the Visual Prolog system focusing on running the examples described in this book.

We assume, that you have experience using the Graphical User Interface system, the windowing system. This might be either 16-bit Windows 3.x or 32-bit Windows (95/98 and NT/2000). You should thus know about using menus, closing and resizing windows, loading a file in the **File Open** dialog etc. If you do not have this knowledge, you should not start off trying to create an application that runs under this environment. You must first learn to use the environment.

If you are a beginner to Prolog, you don't want to mix learning the Prolog language with the complexity of creating Windows applications with event handling and all the Windows options and possibilities. The code for the examples in this book are platform independent: They can run in DOS text mode, under UNIX, or with the adding of a little extra code for the User Interface handling, in a Windowing environment like MS Windows.

We do suggest that you at an early stage work your way through the Guided Tour in the Getting Started book and try to compile some of the examples in the VPI subdirectory. This gives you an impression what can be done with Visual Prolog - just so you know what you should be able to do, when you have learned to master Visual Prolog.

However, if you are going to seriously use the Visual Prolog system, you need to learn the basic concepts of Visual Prolog properly. You will not be able to build a skyscraper without having a solid foundation. In Visual Prolog the foundation is understanding the Prolog language and the VPI layer.

What Needs to be Installed for This Book?

To run and test the examples described in this book you need during installation of Visual Prolog from CD:

- In the dialog **Compilers**. Install the Visual Development Environment (**VDE**). We recommend that you choose the Win32 version.
- In the dialog **Libraries**. Check ON libraries correspondent to the selected VDE platform.
- In the dialog **Documentation**. You must check ON installation of **Answers (Language Tutorial)** and **Examples (Language Tutorial)**. We recommend you also to switch ON installation of all other items listed in the **Documentation** dialog.
- In the dialog Final. We recommend you to check ON the Associate 32-bit VDE with Project File Extensions PRJ & VPR.

Starting the Visual Prolog's Visual Development Environment

The Visual Prolog's installation program will install a program group with an Icon, which are normally used to start Visual Prolog's Visual Development Environment (**VDE**). However, there are many ways to start an application in the GUI World, if you prefer another method, you can just start the Visual Development Environment's executable file VIP.EXE from BIN\WIN\32 (32-bit Windows version) or BIN\WIN\16 (16-bit Windows version) directories under the main Visual Prolog directory.

If the Visual Development Environment had an open project (a .PRJ or .VPR file) the last time the VDE was closed on your computer, it will automatically reopen this project next time it starts.

If while installation of Visual Prolog's from CD you had checked ON the **Associate 32-bit VDE with Project File Extensions PRJ & VPR**, then you can simply double click on a project file (file name extension .PRJ or .VPR). The Visual Development Environment will be activated and the selected project will be loaded.

To run most examples in this manual, you should use Visual Development Environment's utility **Test Goal**. The **Test Goal** utility can be activated with the menu item **Project | Test Goal** or simply by the hot key **Ctrl+G**. For correct running of these examples with the **Test Goal** utility, the VDE should use the special settings of loaded projects. We recommend you to create and always use the following special **TestGoal** project.

Creating the TestGoal Project for Running the Examples

To run with the **Test Goal** utility, Language Tutorial's examples require that some non-default settings of Visual Prolog's compiler options should be specified. These options can be specified as the project settings with the following actions:

1. Start Visual Prolog's VDE.

   If this is the first time you start the VDE, then it does not have a loaded project and you will see the picture like this (also you will be informed that the default Visual Prolog initialization file is created):

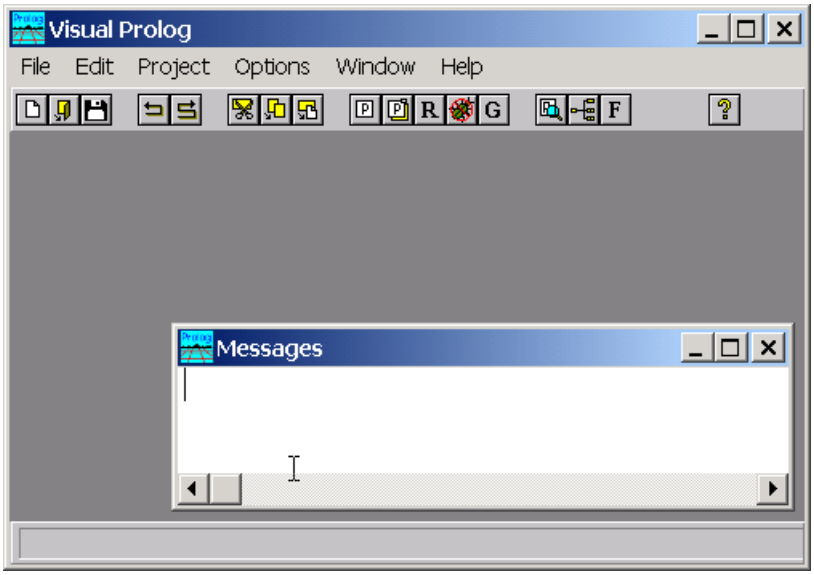


Figure 1.1: Start Visual Development Environment

2. Start creation of a new project.

   Select **Project | New Project** menu item, the **Application Expert** dialog will be activated.

3. Specify the **Base Directory** and the **Project Name**.

Suppose that while installation of Visual Prolog you had selected Visual Prolog's *root directory* C:\VIP, then the *Language Tutorial* examples should be installed into the \DOC\Examples subdirectory of this *root directory*. In this case, we recommend you to specify the following **Base Directory**:

```
C:\VIP\DOC\Examples\TestGoal
```

This choice is convenient for future loading of Prolog source files of the *Language Tutorial* examples.

In the **Project Name,** we recommend to specify "*TestGoal*".

Also check ON the **Multiprogrammer Mode** and click inside the **Name of .PRJ File** control. You will see that the project file name `TestGoal.PRJ` appears.

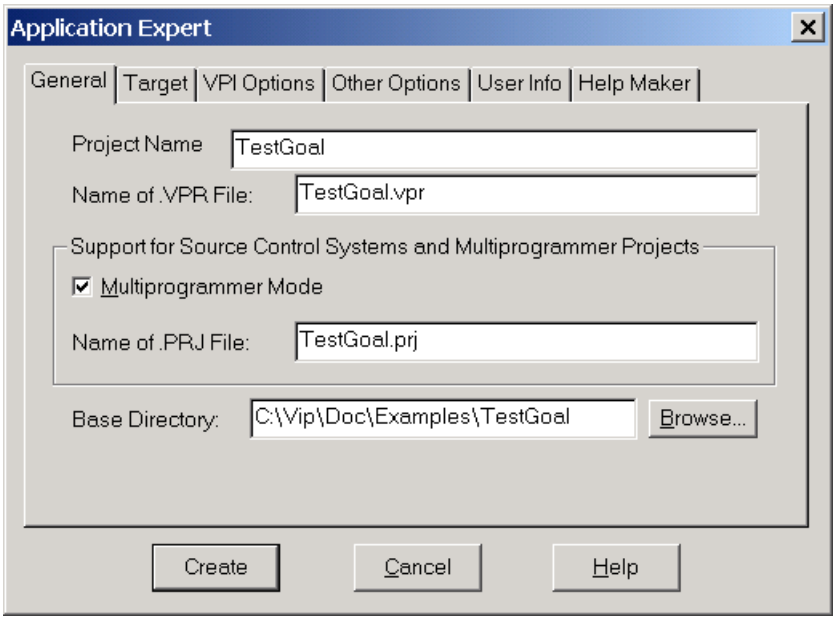


Figure 1.2: Application Expert's General Settings

Specify the **Target** settings.

In the Application Expert's Target tab, we recommend to select the following settings:

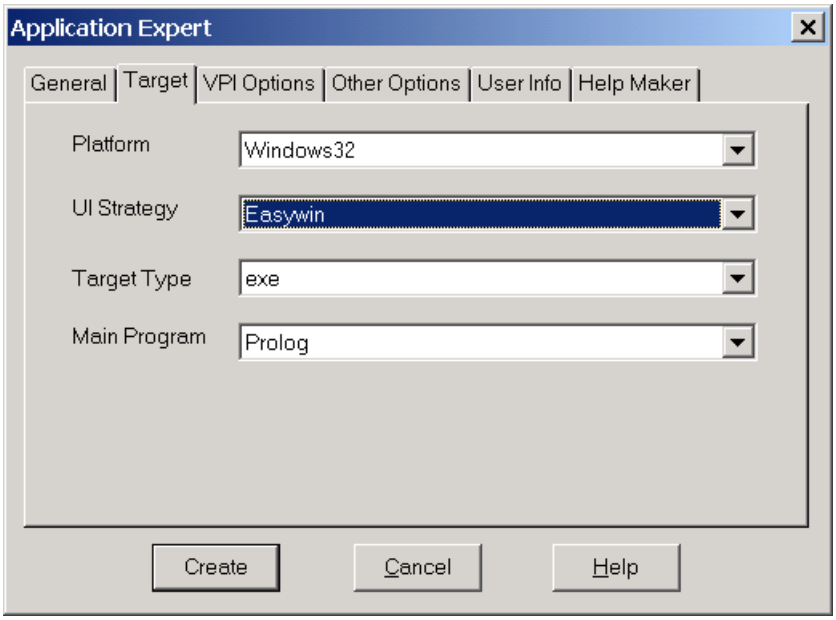


Figure 1.3: Application Expert's Target Settings

Now press the **Create** button to create the default project files.

4. Set the required Compiler Options for the created **TestGoal** project.

   Select the **Options | Project | Compiler Options** menu item, the **Compiler Options** dialog is activated. Choose the **Warnings** tab. In this tab:

   - Check ON the **Nondeterm** radio button. This enforces Visual Prolog's compiler to accept that by default all user-defined predicates are nondeterministic (can produce more than one solution).
   - Check Off: the **Non Quoted Symbols**, **Strong Type Conversion Check**, and **Check Type of Predicates**. These will suppress some possible compiler warnings that are not important for understanding of the *Language Tutorial* examples.
   - As the result the **Compiler Options** dialog will looks like the following:

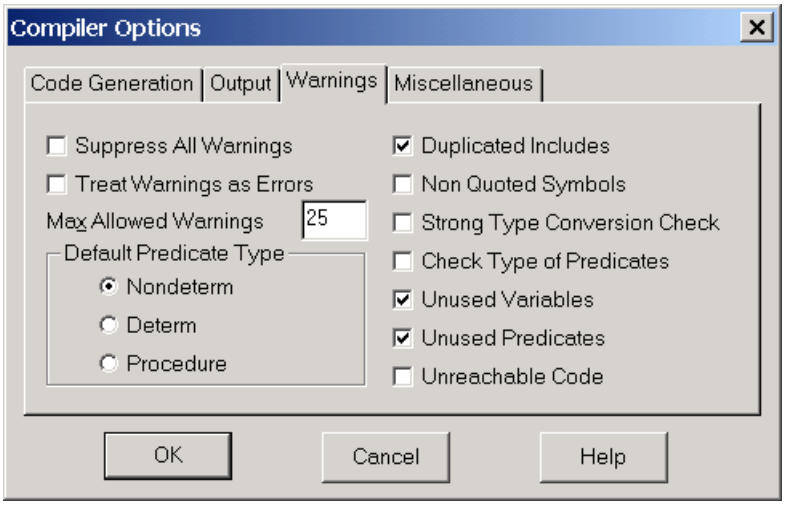


Figure 1.4: The Compiler Options Settings

Press the **OK** button to save the Compiler Options settings.

Opening an Editor Window

To create a new edit window, you can use the menu command **File | New**. This will bring up a new editor window with the title "NONAME".

The VDE's editor is a fairly standard text editor. You can use cursor keys and the mouse as you are used to in other editors. It supports **Cut**, **Copy** and **Paste, Undo** and **Redo**, which you can all activate from the **Edit** menu. Also the **Edit** menu shows the accelerator keys associated for these actions. The editor is documented in the *Visual Development Environment* manual.

Running and Testing a Program

To check, that your system is set up properly, you should try to type in the following text in the window:

```
GOAL write("Hello world"),nl.
```

This is what is called a GOAL in the Prolog terminology, and this is enough to be a program that can be executed. To execute the GOAL, you should activate the menu item **Project | Test Goal**, or just press the accelerator key **Ctrl+G**. If your system is installed properly, your screen will look like the following:

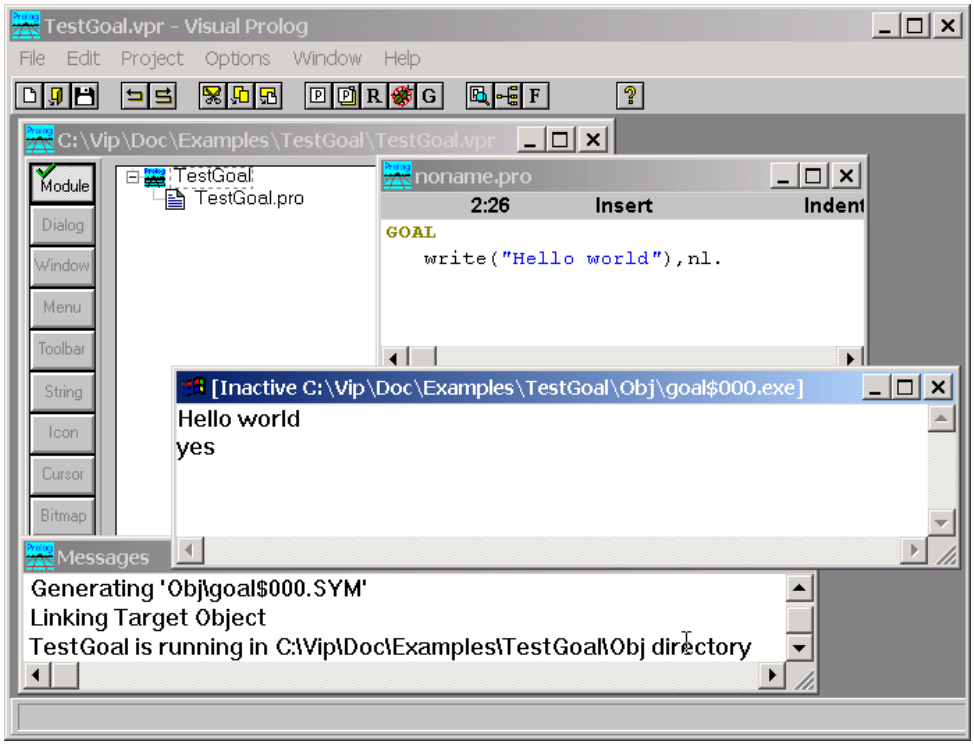


Figure 1.5: The "Hello world" test

The result of the execution will come up in a separate window (on this figure it has title: `[Inactive C:\Vip\Doc\Examples\TestGoal\Obj\goal$000.exe]`), which you must close before you can test another GOAL.

Testing Language Tutorial Examples

As we recommended, you have installed **Examples (Language Tutorial)** and **Answers (Language Tutorial)**. You can find the Language Tutorial examples in

the subdirectory: DOC\EXAMPLES, and answers to exercises in the subdirectory DOC\ANSWERS.

The examples are named after the chapter they appear in: `chCCeNN.pro`, where *CC* will be 02, 03, 04, etc. according to the chapter number, and *NN* is the example number within that chapter (01, 02, 03, etc.).

Testing Examples in Test Goal

You should now try to open one of these examples by Visual Development Environment, and test it using the Test Goal utility. These involves the following steps:

1. Start Visual Prolog's VDE.
2. Use the **Project | Open Project** menu command to open the special **TestGoal** project (see above).
3. Use the **File | Open** command to open one of `chCCeNN.pro` files.
4. Use the **Project | Test Goal** command (or press **Ctrl+G**) to test the GOAL of the loaded example.

   The **Test Goal** will finds ALL possible solutions for the GOAL and display values of ALL variables used in the GOAL.

Remarks About Properties of the Test Goal Utility

The Visual Development Environment's **Test Goal** utility treats the GOAL as a special program, which it compiles, links, generates the executable from and runs this executable. The **Test Goal** internally extends the specified code of the GOAL in the way enforcing the generated program to find ALL possible solutions and display values of ALL used variables. The **Test Goal** utility compiles this code using the Compiler Options specified to the opened project (we had specified the recommended Compiler Options for the **TestGoal** project above). Notice that the **Test Goal** utility compiles only the code specified in the active editor window (it simply ignores code in any other opened editors or the project modules, if they are). Linking the executable, the **Test Goal** uses the EASYWIN strategy (that is described in the Visual Development Environment manual). Notice that you cannot specify any linking options to the **Test Goal;** because it ignores any **Make Options** specified to the opened project. Therefore, the **Test Goal** cannot use any global predicates defined in different modules. Notice that the **Test Goal** has restriction on number of variables that can be used in the GOAL. Currently it is 12 for 32-bit Windows VDE, but this can be changed without any notification.

Testing Examples as Standalone Executables

Most examples in Language Tutorial are intended to be tested with the Test Goal utility, but some examples, for instance `ch04e05.pro`, are intended to be tested as *standalone executables*. We recommend the following procedure for testing such examples:

1. Start Visual Prolog's VDE and open the previously created **TestGoal** project (see above).
2. Open the file TestGoal.PRO for editing (simply double click on *TestGoal* item in the Project window).
3. The Prolog code in the file TestGoal.PRO starts (after the header comment) with the **include** directive:

   ```
   include "TestGoal.INC"
   ```

4. Comment this **include** directive and delete all other Prolog code. Notice that this **include** directive can be commented (ignored) for simple examples from *Language Tutorial*, but it is required for more complicated programs.
5. Include the file with the example source code, for instance `ch04e05.pro`. Notice that filenames in **include** directives should contain correct paths relative to the project root directory; therefore, we recommend using the editor's Insert -> FileName command:

   - Type:

     ```
     include
     ```

   - Activate the menu command **Edit | Insert | FileName**; the **Get & Insert FileName** dialog is activated. Browse for the required filename (`ch04e05.pro`), highlight it and click the **Open** button. Now the file TestGoal.PRO should contain the lines:

     ```
     % include "TestGoal.INC"   % Can be commented in simple
     examples
     include "C:\\VIP_SS\\DOC\\EXAMPLES\\ch04e05.pro"
     ```

6. Now you can compile the example source code (`ch04e05.pro`), create the correspondent executable file (in this case it will be the file TestGoal.EXE) and run it as a standalone executable. You can execute all these actions with one command: **Project | Run** or simply with the hot key **F9**.

Handling Errors

If you, like all programmers do, happen to make some errors in your program and try to compile the program, then the VDE will display the **Errors (Warnings)** window, which contains the list of detected errors. You can double click on one of these errors to come to the position of the error in the source text. (Beginning with Visual Prolog v. 5.3, you can click on one of these errors and press the hot key **F1** to display extended information about the selected error.) In previous versions, you can press **F1** to display Visual Prolog's on-line Help. When in the Help window, click the **Search** button and type in an error number; the help topic with extended information about the error will be displayed.

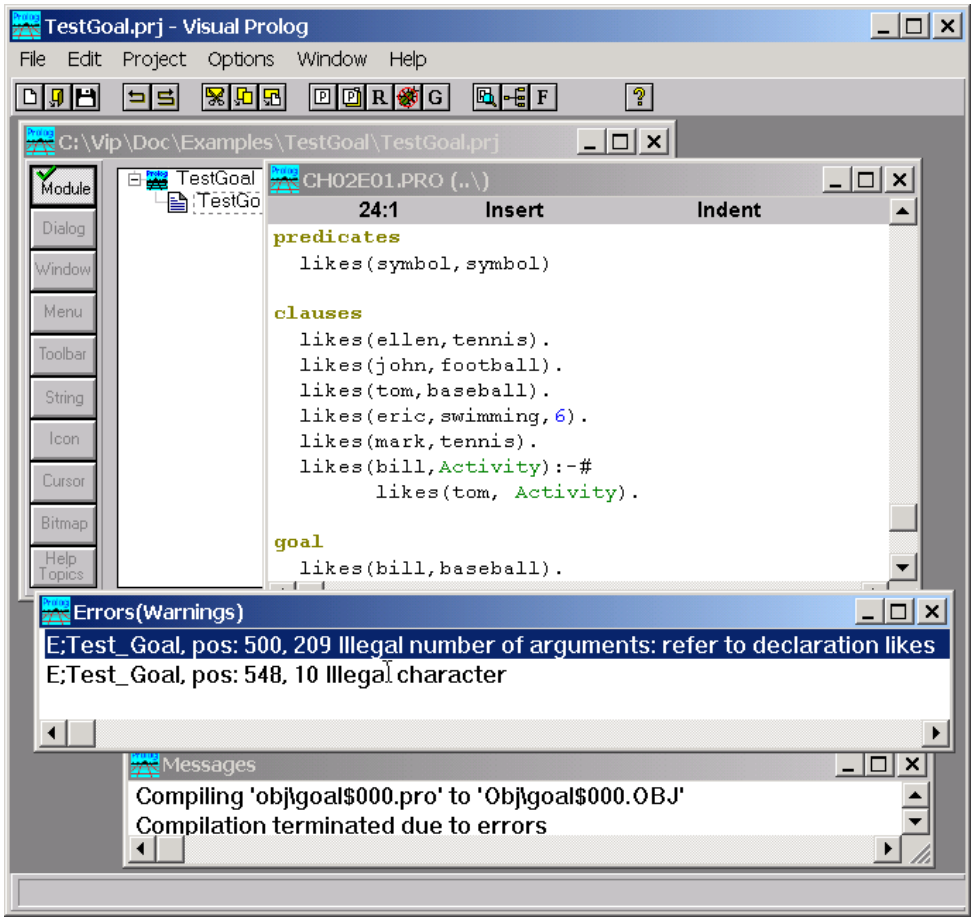


Figure 1.6: Handling Errors

PART 2

Tutorial Chapters 2 – 11: Learning Visual Prolog

CHAPTER 2

Prolog Fundamentals

This is the first in a sequence of chapters giving a step-by-step tutorial introduction to the Visual Prolog language. We begin this chapter with an introduction to programming in logic. After that, we discuss some of Prolog's basic concepts, including clauses, predicates, variables, goals, and matching.

PROgramming in LOGic

In Prolog, you arrive at solutions by logically inferring one thing from something already known. Typically, a Prolog program isn't a sequence of actions – it's a collection of facts together with rules for drawing conclusions from those facts. Prolog is therefore what is known as a declarative language.

Prolog is based on Horn clauses, which are a subset of a formal system called predicate logic. Don't let this name scare you. Predicate logic is simply a way of making it clear how reasoning is done. It's simpler than arithmetic once you get used to it.

Prolog uses a simplified variation of predicate logic syntax because it provides an easy-to-understand syntax very similar to natural language, and because computers are not as fast, large, or as inexpensive as we would like. If Prolog were to accept English statements, the compiler would need to know every possible way something could be worded in English. In many cases, it would take many times longer to translate the program into something the computer understands than it would to run the program. The computer hardware needed to run such a system would be monstrous.

Prolog includes an inference engine, which is a process for reasoning logically about information. The inference engine includes a pattern matcher, which retrieves stored (known) information by matching answers to questions. Prolog tries to infer that a hypothesis is true (in other words, answer a question) by questioning the set of information already known to be true. Prolog's known world is the finite set of facts (and rules) that are given in the program.

One important feature of Prolog is that, in addition to logically finding answers to the questions you pose, it can deal with alternatives and find all possible

solutions rather than only one. Instead of just proceeding from the beginning of the program to the end, Prolog can actually back up and look for more than one way of solving each part of the problem.

Predicate logic was developed to easily convey logic-based ideas into a written form. Prolog takes advantage of this syntax to develop a programming language based on logic. In predicate logic, you first eliminate all unnecessary words from your sentences. You then transform the sentence, placing the relationship first and grouping the objects after the relationship. The objects then become arguments that the relationship acts upon. For example, the following sentences are transformed into predicate logic syntax:

Natural Language:	**Predicate Logic:**
A car is fun. A rose is red.	`fun(car).` `red(rose).`
Bill likes a car if the car is fun.	`likes(bill, Car) if fun(Car).`

Sentences: Facts and Rules

The Prolog programmer defines *objects* and *relations*, then defines *rules* about when these relations are true. For example, the sentence

```
Bill likes dogs.
```

shows a relation between the objects *Bill* and *dogs*; the relation is *likes*. Here is a rule that defines when the sentence `Bill likes dogs` is true:

```
Bill likes dogs if the dogs are nice.
```

Facts: What Is Known

In Prolog, a relation between objects is called a *predicate*. In natural language, a relation is symbolized by a sentence. In the predicate logic that Prolog uses, a relation is summarized in a simple phrase – a fact – that consists of the relation name followed by the object or objects (enclosed in parentheses). As with a sentence, the fact ends with a period (.).

Here are some more facts expressing "likes" relations in natural language:

```
Bill likes Cindy.
Cindy likes Bill.
Bill likes dogs.
```

Here are the same facts, written in Prolog syntax:

```
likes(bill, cindy).
likes(cindy, bill).
likes(bill, dogs).
```

Facts can also express properties of objects as well as relations; in natural language "Kermit is green" and "Caitlin is a girl." Here are some Prolog facts that express these same properties:

```
green(kermit).
girl(caitlin).
```

Rules: What You Can Infer from Given Facts

Rules enable you to infer facts from other facts. Another way to say this is that a *rule*, as conclusions is a conclusion that is known to be true if one or more other conclusions or facts are found to be true. Here are some rules concerning a "likes" relation:

```
Cindy likes everything that Bill likes.
Caitlin likes everything that is green.
```

Given these rules, you can infer from the previous facts some of the things that Cindy and Caitlin like:

```
Cindy likes Cindy.
Caitlin likes Kermit.
```

To encode these same rules into Prolog, you only need to change the syntax a little, like this:

```
likes(cindy, Something):- likes(bill, Something).
likes(caitlin, Something):- green(Something).
```

The `:-` symbol is simply pronounced "if", and serves to separate the two parts of a rule: the head and the body.

You can also think of a rule as a procedure. In other words, these rules

```
likes(cindy, Something):- likes(bill, Something)
likes(caitlin, Something):- green(Something).
```

also mean "To prove that Cindy likes something, prove that Bill likes that same thing" and "To prove that Caitlin likes something, prove that it is green." In the same side effects procedural way, a rule can ask Prolog to perform actions other than proving things – such as writing something or creating a file.

Queries

Once we give Prolog a set of facts, we can proceed to ask questions concerning these facts; this is known as *querying the Prolog system*. We can ask Prolog the same type of questions that we would ask you about these relations. Based upon the known facts and rules given earlier, you can answer questions about these relations, just as Prolog can.

In natural language, we ask you:

```
Does Bill like Cindy?
```

In Prolog syntax, we ask Prolog:

```
likes(bill, cindy).
```

Given this query, Prolog would answer

```
yes
```

because Prolog has a fact that says so. As a little more complicated and general question, we could ask you in natural language:

```
What does Bill like?
```

In Prolog syntax, we ask Prolog:

```
likes(bill, What).
```

Notice that Prolog syntax does not change when you ask a question: this query looks very similar to a fact. However, it is important to notice that the second object – *What* – begins with a capital letter, while the first object – *bill* – does not. This is because *bill* is a fixed, constant object – a known value – but *What* is a variable. Variables always begin with an upper-case letter or an underscore.

Prolog always looks for an answer to a query by starting at the top of the facts. It looks at each fact until it reaches the bottom, where there are no more. Given the query about what Bill likes, Prolog will return

```
What=cindy
What=dogs
2 Solutions
```

This is because Prolog knows

```
likes(bill, cindy).
```

and

```
likes(bill, dogs).
```

We hope that you draw the same conclusion.

If we were to ask you (and Prolog):

```
What does Cindy like?

likes(cindy, What).
```

Prolog would answer

```
What = bill
What = cindy
What = dogs
3 solutions
```

This is because Prolog knows that Cindy likes Bill, and that Cindy likes what Bill likes, and that Bill likes Cindy and dogs.

We could ask Prolog other questions that we might ask a person; however, a question such as "What girl does Bill like?" will yield no solution because Prolog, in this case, knows no facts about girls, and it can't draw any conclusions based on material not known (supplied to it). In this example, we have not given Prolog any relation or property to determine if any of the objects are girls.

Putting Facts, Rules, and Queries Together

1. Suppose you have the following facts and rules:

   ```
   A fast car is fun.
   A big car is nice.
   A little car is practical.
   Bill likes a car if the car is fun.
   ```

   When you read these facts, you can deduce that Bill likes a fast car. In much the same way, Prolog will come to the same conclusion. If no fact were given about fast cars, then you would not be able to logically deduce what kind of a car Bill likes. *You* could take a guess at what kind of a car might be fun, but Prolog only knows what you tell it; Prolog does not guess.

2. Here's an example demonstrating how Prolog uses rules to answer queries. Look at the facts and rules in this portion of Program `ch02e01.pro`:

```
likes(ellen, tennis).
likes(john, football).
likes(tom, baseball).
likes(eric, swimming).
likes(mark, tennis).
likes(bill, Activity):- likes(tom, Activity).
```

The last line in Program `ch02e01.pro` is a rule:

```
likes(bill, Activity):- likes(tom, Activity).
```

This rule corresponds to the natural language statement

```
Bill likes an activity if Tom likes that activity.
```

In this rule, the head is `likes(bill, Activity)`, and the body is `likes(tom, Activity)`. Notice that there is no fact in this example about Bill liking baseball. For Prolog to discover if Bill likes baseball, you can give the query

```
likes(bill, baseball).
```

When attempting to find a solution to this query, Prolog will use the rule:

```
likes(bill, Activity):- likes(tom, Activity).
```

3. Load Program `ch02e01.pro` into the Visual Prolog's Visual Development Environment and run it with the **Test Goal** utility (see **Testing Language Tutorial Examples** on page 12).

```
PREDICATES
   likes(symbol,symbol)

CLAUSES
   likes(ellen,tennis).
   likes(john,football).
   likes(tom,baseball).
   likes(eric,swimming).
   likes(mark,tennis).

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

GOAL
   likes(bill, baseball).
```

The Test Goal replies in the application's window

```
yes
```

It has combined the rule

```
likes(bill, Activity):- likes(tom, Activity).
```

with the fact

```
likes(tom, baseball).
```

to decide that

```
likes(bill, baseball).
```

Try also this query:

```
likes(bill, tennis).
```

The Test Goal replies

```
no
```

Visual Prolog replies `no` to the latest query ("Does Bill like tennis?") because:

- There is no fact that says Bill likes tennis.
- Bill's relationship with tennis can't be inferred using the given rule and the available facts.

Of course, it may be that Bill absolutely adores tennis in real life, but Visual Prolog's response is based *only* upon the facts and the rules you have given it in the program.

Variables: General Sentences

In Prolog, *variables* enable you to write general facts and rules and ask general questions. In natural language, you use variables in sentences all the time. A typical general statement in English could be

```
Bill likes the same thing as Kim.
```

As we mentioned earlier in this chapter, to represent a variable in Prolog, the first character of the name must be an upper-case letter or an underscore. For example, in the following line, *Thing* is a variable.

```
likes(bill, Thing):- likes(kim, Thing).
```

In the preceding discussion of rules, you saw this line:

```
likes(cindy, Something):- likes(bill, Something).
```

The object *Something* begins with a capital letter because it is a variable; it must be able to match anything that Bill likes. It could equally well have been called *X* or *Zorro*.

The objects *bill* and *cindy* begin with lower-case letters because they are *not* variables – instead, they are symbols, having a constant value. Visual Prolog can also handle arbitrary text strings, much like we've been handling symbols above, if the text is surrounded by double quotes. Hence, the token `bill` could have been written as `"Bill"`, if you wanted it to begin with an upper-case letter.

Overview

1. A Prolog program is made up of two types of phrases (also known as *clauses*): facts and rules.
   - Facts are relations or properties that you, the programmer, know to be true.
   - Rules are dependent relations; they allow Prolog to infer one piece of information from another. A rule becomes true if a given set of conditions is proven to be true. Each rule depends upon proving its conditions to be true.
2. In Prolog, all rules have two parts: a head and a body separated by the special `:-` token.
   - The head is the fact that would be true if some number of conditions were true. This is also known as the conclusion or the dependent relation.
   - The body is the set of conditions that must be true so that Prolog can prove that the head of the rule is true.
3. As you may have already guessed, facts and rules are really the same, except that a fact has no explicit body. The fact simply behaves as if it had a body that was always true.
4. Once you give Prolog a set of facts and/or rules, you can proceed to ask questions concerning these; this is known as *querying the Prolog system*. Prolog always looks for a solution by starting at the top of the facts and/or rules, and keeps looking until it reaches the bottom.
5. Prolog's inference engine takes the conditions of a rule (the body of the rule) and looks through its list of known facts and rules, trying to satisfy the conditions. Once all the conditions have been met, the dependent relation (the head of the rule) is found to be true. If all the conditions can't be matched with known facts, the rule doesn't conclude anything.

Exercises

Write natural language sentences that represent what these Prolog facts might convey to a human reader. (Remember that, to the computer, these facts are simple pieces of information that can be used for matching answers to questions.)

1. `likes(jeff, painting).`
2. `male(john).`
3. `building("Empire State Building", new_york).`
4.
```
person(roslin, jeanie, "1429 East Sutter St.",
"Scotts Valley", "CA", 95066).
```

Write Visual Prolog facts that represent the following natural language statements:

1. Helen likes pizza.
2. San Francisco is in California.
3. Amy's telephone number is 476-0299.
4. Len's father is Alphonso Grenaldi.

From Natural Language to Prolog Programs

In the first section of this chapter we talked about facts and rules, relations, general sentences, and queries. Those words are all part of a discussion of logic and natural language. Now we're going to discuss the same ideas, but we're going to use more Prolog-ish words, like clauses, predicates, variables, and goals.

Clauses (Facts and Rules)

Basically, there are only two types of phrases that make up the Prolog language; a phrase can be either a *fact* or a *rule*. These phrases are known in Prolog as *clauses*. The heart of a Prolog program is made up of clauses.

More About Facts

A fact represents one single instance of either a property of an object or a relation between objects. A fact is self-standing; Prolog doesn't need to look any further for confirmation of the fact, and the fact can be used as a basis for inferences.

More About Rules

In Prolog, as in ordinary life, it is often possible to find out that something is true by inferring it from other facts. The Prolog construct that describes what you can infer from other information is a rule. A rule is a property or relation known to be true when some set of other relations is known. Syntactically, these relations are separated by commas, as we illustrate in example 1 below.

Examples of Rules

1. This first example shows a rule that can be used to conclude whether a menu item is suitable for Diane.

   ```
   Diane is a vegetarian and eats only what her doctor tells her to eat.
   ```

   Given a menu and the preceding rule, you can conclude if Diane can order a particular item on the menu. To do this, you must check to see if the item on the menu matches the constraints given.

   a. Is `Food_on_menu` a vegetable?

   b. Is `Food_on_menu` on the doctor's list?

   c. Conclusion: If both answers are yes, Diane can order `Food_on_menu`.

   In Prolog, a relationship like this must be represented by a rule because the conclusion is based on facts. Here's one way of writing the rule:

   ```
   diane_can_eat(Food_on_menu):-
      vegetable(Food_on_menu),
      on_doctor_list(Food_on_menu).
   ```

   Notice here the comma after `vegetable(Food_on_menu)`. The comma introduces a conjunction of several goals, and is simply read as "and"; both `vegetable(Food_on_menu)` **and** `on_doctor_list(Food_on_menu)` must be true, for `diane_can_eat(Food_on_menu)` to be true.

2. Suppose you want to make a Prolog fact that is true if *Person1* is the parent of *Person2*. This is easy enough; simply state the Prolog fact

   ```
   parent(paul, samantha).
   ```

   This shows that Paul is the parent of Samantha. But, suppose your Visual Prolog fact database already has facts stating father relationships. For example, "Paul is the father of Samantha":

   ```
   father(paul, samantha).
   ```

And you also have facts stating mother relationships; "Julie is the mother of Samantha":

```
mother(julie, samantha).
```

If you already had a collection of facts stating these father/mother relationships, it would be a waste of time to write parent facts into the fact database for each parent relationship.

Since you know that *Person1* is the parent of *Person2* if *Person1* is the father of *Person2* or if *Person1* is the mother of *Person2*, then why not write a rule to convey these constraints? After stating these conditions in natural language, it should be fairly simple to code this into a Prolog rule by writing a rule that states the relationships.

```
parent(Person1, Person2):- father(Person1, Person2).
parent(Person1, Person2):- mother(Person1, Person2).
```

These Prolog rules simply state that

```
Person1 is the parent of Person2 if Person1 is the father of Person2.
Person1 is the parent of Person2 if Person1 is the mother of Person2.
```

3. Here's another example:

```
A person can buy a car if the person likes the car and the car is for
sale.
```

This natural language relationship can be conveyed in Prolog with the following rule:

```
can_buy(Name, Model):-
   person(Name),
   car(Model),
   likes(Name, Model),
for_sale(Model).
```

This rule shows the following relationship:

```
Name can_buy Model if
   Name is a person and
   Model is a car and
   Name likes Model and
   Model is for sale.
```

This Prolog rule will succeed if all four conditions in the body of the rule succeed.

4. Here is a program designed to find solutions to this car-buying problem (test it as it is described in **Testing Language Tutorial Examples** on page 12):

```
/* Program ch02e02.pro */

PREDICATES
   can_buy(symbol, symbol)
   person(symbol)
   car(symbol)
   likes(symbol, symbol)
   for_sale(symbol)

CLAUSES
   can_buy(X,Y):-
   person(X),
   car(Y),
   likes(X,Y),
   for_sale(Y).

   person(kelly).
   person(judy).
   person(ellen).
   person(mark).

   car(lemon).
   car(hot_rod).

   likes(kelly, hot_rod).
   likes(judy, pizza).
   likes(ellen, tennis).
   likes(mark, tennis).

   for_sale(pizza).
   for_sale(lemon).
   for_sale(hot_rod).
```

   What can Judy and Kelly buy? Who can buy the hot rod? You can try the following goals:

```
   can_buy(Who, What).
   can_buy(judy, What).
   can_buy(kelly, What).
   can_buy(Who, hot_rod).
```

   Experiment! Add other facts and maybe even a rule or two to this Prolog program. Test the new program with queries that you make up. Does Prolog respond in a way you would expect it to?

Exercises

1. Write natural-language sentences corresponding to the following Visual Prolog rules:

```
eats(Who, What):- food(What), likes(Who, What).

pass_class(Who):- did_homework(Who), good_attendance(Who).

does_not_eat(toby, Stuff):- food(Stuff), greasy(Stuff).

owns(Who, What):- bought(Who, What).
```

2. Write Visual Prolog rules that convey the meaning of these natural-language sentences:
   a. A person is hungry if that person's stomach is empty.
   b. Everybody likes a job if it's fun and it pays well.
   c. Sally likes french fries if they're cooked.
   d. Everybody owns a car who buys one, pays for it, and keeps it.

Predicates (Relations)

The symbolic name of a relation is called the *predicate* name. The objects that it relates are called its *arguments*; in the fact `likes(bill, cindy)`, the relation *likes* is the predicate and the objects *bill* and *cindy* are the arguments.

Here are some examples of Prolog predicates with zero or more arguments:

```
pred(integer, symbol)
person(last, first, gender)
run
insert_mode
birthday(firstName, lastName, date)
```

As we've shown here, a predicate might not have any arguments at all, but the use of such a predicate is limited. You can use a query such as `person(rosemont,Name,male)` to find out Mr. Rosemont's first name. But what can you do with the zero-argument query run? You can find out whether the clause `run` is in the program, or – if `run` is the head of a rule, you can evaluate that rule. This can be useful in a few cases – for instance, you might want to make a program behave differently depending on whether the clause `insert_mode` is present.

Variables (General Clauses)

In a simple query, you can use variables to ask Prolog to find who likes tennis. For example:

```
likes(X, tennis).
```

This query uses the letter *X* as a variable to indicate an unknown person. Variable names in Visual Prolog must begin with a capital letter (or an underscore), after which any number of letters (upper-case or lower-case), digits, or underline characters (_) can be used. For example, the following are valid variable names:

```
My_first_correct_variable_name
Sales_10_11_86
```

while the next three are invalid:

```
1stattempt
second_attempt
"disaster"
```

(Careful choice of variable names makes programs more readable. For example,

```
likes(Person, tennis).
```

is better than

```
likes(X, tennis).
```

because *Person* makes more sense than *X*.) Now try the goal

```
GOAL likes(Person, tennis).
```

Visual Prolog replies

```
Person=ellen
Person=mark
2 Solutions
```

because the goal can be solved in just two ways; namely, by taking the variable *Person* and successively matching it with the values *ellen* and *mark*.

In variable names, except for the first character (which must be an upper-case letter or an underscore), Visual Prolog allows lower-case or upper-case letters in any position. One way to make variable names more readable is by using mixed upper-case and lower-case letters, as in

```
IncomeAndExpenditureAccount
```

How Variables Get Their Values

You may have noticed that Prolog has no assignment statement; this is a significant distinction between Prolog and other programming languages. ***Variables in Prolog get their values by being matched to constants in facts or rules.***

Until it gets a value, a variable is said to be *free*; when it gets a value, it becomes *bound*. But it only stays bound for the time needed to obtain one solution to the query; then Prolog unbinds it, backs up, and looks for alternative solutions.

This is a very important point: ***You can't store information by giving a value to a variable.*** Variables are used as part of the pattern-matching process, not as a kind of information storage.

Take a look at the following example, which uses program `ch02e03.pro` to demonstrate how and when variables get their values.

```
                    /* Program ch02e03.pro */

PREDICATES
   likes(symbol,symbol)

CLAUSES
   likes(ellen,reading).
   likes(john,computers).
   likes(john,badminton).
   likes(leonard,badminton).
   likes(eric,swimming).
   likes(eric,reading).
```

Consider this query: Is there a person who likes both reading and swimming?

```
   likes(Person, reading), likes(Person, swimming).
```

Prolog will solve the two parts of this query by searching the program's clauses from top to bottom. In the first part of the query

```
   likes(Person, reading)
```

the variable *Person* is free; its value is unknown before Prolog attempts to find a solution. On the other hand, the second argument, *reading*, is known. Prolog searches for a fact that matches the first part of the query. The first fact in the program

```
   likes(ellen, reading)
```

is a match (*reading* in the fact matches *reading* in the query), so Prolog binds the free variable *Person* to the value *ellen*, the relevant value in the fact. At the same time, Prolog places a pointer in the list of facts indicating how far down the search procedure has reached.

Next, in order for the query to be fully satisfied (find a person who likes both reading and swimming), the second part must also be fulfilled. Since *Person* is now bound to *ellen*, Prolog must search for the fact

```
likes(ellen, swimming)
```

Prolog searches for this fact from the beginning of the program, but no match occurs (because there is no such fact in the program). The second part of the query is not true when *Person* is *ellen*.

Prolog now "unbinds" *Person* and attempts another solution of the first part of the query with *Person* once again a free variable. The search for another fact that fulfills the first part of the query starts from the pointer in the list of facts. (This returning to the place last marked is known as *backtracking*, which we'll cover in chapter **4**.)

Prolog looks for the next person who likes reading and finds the fact `likes(eric, reading)`. *Person* is now bound to *eric*, and Prolog tries once again to satisfy the second part of the query, this time by looking in the program for the fact

```
likes(eric, swimming)
```

This time it finds a match (the last clause in the program), and the query is fully satisfied. Prolog (the **Test Goal**) returns

```
Person=eric
1 Solution
```

Anonymous Variables

Anonymous variables enable you to unclutter your programs. If you only need certain information from a query, you can use anonymous variables to ignore the values you don't need. In Prolog, the anonymous variable is represented by a lone underscore ("_").

The following *parents* example demonstrates how the anonymous variable is used. Load Program `ch02e04.pro`. into the TestGoal project (see page 13)

```
                    /* Program ch02e04.pro */

PREDICATES
   male(symbol)
   female(symbol)
   parent(symbol, symbol)

CLAUSES
   male(bill).
   male(joe).

   female(sue).
   female(tammy).

   parent(bill,joe).
   parent(sue,joe).
   parent(joe,tammy).
```

The anonymous variable can be used in place of any other variable. The difference is that the anonymous variable will never get set to a value.

For example, in the following query, you need to know which people are parents, but you don't need to know who their children are. Prolog realizes that each time you use the underscore symbol in the query, you don't need information about what value is represented in that variable's place.

```
GOAL
   parent(Parent, _).
```

Given this query, Prolog (the **Test Goal**) replies

```
   Parent=bill
   Parent=sue
   Parent=joe
   3 Solutions
```

In this case, because of the anonymous variable, Prolog finds and reports three parents, but it does not report the values associated with the second argument in the ***parent*** clause.

Anonymous variables can also be used in facts. The following Prolog facts

```
   owns(_, shoes).
   eats(_).
```

could be used to express the natural language statements

```
Everyone owns shoes.
Everyone eats.
```

The anonymous variable matches anything. A named variable would work equally well in most cases, but its name would serve no useful purpose.

Goals (Queries)

Up to now, we've been mixing the word *query* when talking about the questions you ask Prolog, with the more common name *goal*, which we'll use from now on. Referring to queries as goals should make sense: when you query Prolog, you are actually giving it a goal to accomplish ("Find an answer to this question, if one exists: ...").

Goals can be simple, such as these two:

```
likes(ellen, swimming).

likes(bill, What).
```

or they can be more complex. In the "Variables" section of this chapter, you saw a goal made up of two parts:

```
likes(Person, reading), likes(Person, swimming).
```

A goal made up of two or more parts is known as a *compound goal*, and each part of the compound goal is called a *subgoal*.

Often you need to know the intersection of two goals. For instance, in the previous parents example, you might also need to know which persons are male parents. You can get Prolog to search for the solutions to such a query by setting a compound goal. Load the Program `ch02e04.pro` (see page 13) and enter the following compound goal:

```
Goal parent(Person, _), male(Person).
```

Prolog will first try to solve the subgoal

```
parent(Person, _)
```

by searching the clauses for a match, then binding the variable *Person* to a value returned by ***parent*** (*Person* is a parent). The value that ***parent*** returns will then provide the second subgoal with the value on which to search (Is *Person* – now bound – a male?).

```
male(Person)
```

If you entered the goal correctly, Prolog (the Test Goal) will answer

```
Person=bill
Person=joe
2 Solutions
```

Compound Goals: Conjunctions and Disjunctions

As you have seen, you can use a compound goal to find a solution where both subgoal A *and* subgoal B are true (a *conjunction*), by separating the subgoals with a comma, but this is not all. You can also find a solution where subgoal A *or* subgoal B is true (a *disjunction*), by separating the subgoals with a semicolon. Here's an example program illustrating this idea:

```
                        /* Program ch02e05.pro */

predicates
   car(symbol,long,integer,symbol,long)
   truck(symbol,long,integer,symbol,long)
   vehicle(symbol,long,integer,symbol,long)

clauses
   car(chrysler,130000,3,red,12000).
   car(ford,90000,4,gray,25000).
   car(datsun,8000,1,red,30000).

   truck(ford,80000,6,blue,8000).
   truck(datsun,50000,5,orange,20000).
   truck(toyota,25000,2,black,25000).

   vehicle(Make,Odometer,Age,Color,Price):-
      car(Make,Odometer,Age,Color,Price)
      ;
      truck(Make,Odometer,Age,Color,Price).
```

Load this program into the TestGoal project (see page 13). Add the goal:

```
goal
   car(Make, Odometer, Years_on_road, Body, 25000).
```

This goal attempts to find a car described in the clauses that costs exactly $25,000. Prolog (the Test Goal) replies

```
Make=ford, Odometer=90000, Years_on_road=4, Body=gray
1 Solution
```

But this goal is slightly unnatural, since you'd probably rather ask a question like:

```
Is there a car listed that costs less than $25,000?
```

You can get Visual Prolog to search for a solution by setting this compound goal:

```
car(Make, Odometer, Years_on_road, Body, Cost),       /*subgoal A and*/
Cost < 25000.                                           /*subgoal B */
```

This is known as a conjunction. To fulfill this compound goal, Prolog will try to solve the subgoals in order. First, it will try to solve

```
car(Make, Odometer, Years_on_road, Body, Cost).
```

and then

```
Cost < 25000.
```

with the variable *Cost* referring to the same value in both subgoals. Try it out now with the Test Goal (see page 13).

Note: The subgoal `Cost < 25000` involves the relation *less than*, which is built into the Visual Prolog system. The *less than* relation is no different from any other relation involving two numeric objects, but it is more natural to place the symbol for it between the two objects.

Now we will try to see if the following, expressed in natural language, is true:

```
Is there a car listed that costs less than $25,000?, or is there a
truck listed that costs less than $20,000?
```

Prolog will search for a solution if you set this compound goal:

```
car(Make,Odometer,Years_on_road,Body,Cost), Cost<25000
   ; /* subgoal A or */
truck(Make,Odometer,Years_on_road,Body,Cost), Cost < 20000.
      /* subgoal B */
```

This kind of compound goal is known as a disjunction. This one sets up the two subgoals as alternatives, much as though they were two clauses for the same rule. Prolog will then find any solution that satisfies either of the subgoals.

To fulfill this compound goal, Prolog will try to solve the first subgoal ("find a car..."), which is composed of these subgoals:

```
car(Make, Odometer, Years_on_road, Body, Cost.)
```

and

```
Cost < 25000.
```

If a car is found, the goal will succeed; if not, Prolog will try to fulfill the second compound goal ("find a truck..."), made up of the subgoals

```
truck(Make, Odometer, Years_on_road, Body, Cost),
```

and

```
Cost < 20000.
```

Comments

It's good programming style to include comments in your program to explain things that might not be obvious to someone else (or to you in six months). This makes the program easy for you and others to understand. If you choose appropriate names for variables, predicates, and domains, you'll need fewer comments, since the program will be more self-explanatory.

Multiple-line comments must begin with the characters /* (slash, asterisk) and end with the characters */ (asterisk, slash). To set off single-line comments, you can use these same characters, or you can begin the comment with a percent sign (%).

```
/* This is an example of a comment */

% This is also a comment

/*************************************/
/* and so are these three lines        */
/*************************************/

/*You can also nest a Visual Prolog comment /*within a comment*/ like
this */
```

In Visual Prolog you can also use a comment after each subdomain in declarations of domains:

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

and in declarations of predicates:

```
predicates
   conv(STRING uppercase,STRING lowercase)
```

The words `title`, `author`, `name`, `uppercase` and `lowercase` will be ignored by the compiler, but makes the program much more readable.

What Is a Match?

In the previous sections of this chapter, we've talked about Prolog "matching answers to questions", "finding a match", "matching conditions with facts", "matching variables with constants", and so on. In this section we explain what we mean when we use the term "match."

There are several ways Prolog can match one thing to another. Obviously, ***identical structures match each other***;

```
parent(joe,tammy) matches parent(joe,tammy).
```

However, ***a match usually involves one or more free variables***. For example, with *X* free,

```
parent(joe,X) matches parent(joe,tammy)
```

and *X* takes on (is bound to) the value *tammy*.

If *X* is already bound, it acts exactly like a constant. Thus, if *X* is bound to the value *tammy*, then

`parent(joe,X)` matches `parent(joe,tammy)` but

`parent(joe,X)` would not match `parent(joe,millie)`

The second instance doesn't match because, once a variable becomes bound, its value can't change.

How could a variable, bindings already be bound when Prolog tries to match it with something? Remember that variables don't store values – they only stay bound for the length of time needed to find (or try to find) one solution to one goal. So the only way a variable could be bound before trying a match is that the goal involves more than one step, and the variable became bound in a previous step. For example,

```
parent(joe,X), parent(X,jenny)
```

is a legitimate goal; it means, "Find someone who is a child of Joe and a parent of Jenny." Here *X* will already be bound when the subgoal `parent(X,jenny)` is reached. If there is no solution to `parent(X,jenny)`, Prolog will unbind *X* and go back and try to find another solution to `parent(joe,X)`, then see if `parent(X,jenny)` will work with the new value of *X*.

Two free variables can even match each other. For example,

```
parent(joe,X) matches parent(joe,Y)
```

binding the variables *X* and *Y* to each other. As long as the binding lasts, *X* and *Y* are treated as a single variable, and if one of them gets a value, the other one will immediately have the same value. When free variables are bound to each other like this, they're called pointers, shared *free sharing variables*. Some really powerful programming techniques involve binding together variables that were originally separate.

In Prolog, variable bindings (values) are passed in two ways: in and out. The direction in which a value is passed is referred to as its *flow pattern*. When a variable is passed into a clause, it is an *input argument*, signified by `(i)`; when passed out of a clause, a variable is an *output argument*, signified by `(o)`.

Summary

These are the ideas we've introduced in this chapter:

1. A Prolog program is made up of *clauses*, which conceptually are two types of phrases: facts and rules.
   - Facts are relations or properties that you, the programmer, know to be true.
   - Rules are dependent relations; they allow Prolog to infer one piece of information from another.
2. *Facts* have the general form:

   ```
   property(object1, object2, ..., objectN)
   ```

   or

   ```
   relation(object1, object2, ..., objectN)
   ```

   where a property is a *property of* the objects and a relation is a *relation between* the objects. As far as Prolog programming goes, the distinction doesn't exist and we will refer to both as relations in this book.
3. Each fact given in a program consists of either a relation that affects one or more objects or a property of one or more objects. For example, in the Prolog fact

   ```
   likes(tom, baseball).
   ```

   the relation is *likes*, and the objects are *tom* and *baseball*; Tom likes baseball. Also, in the fact

   ```
   left_handed(benjamin)
   ```

the property is `left_handed` and the object is benjamin; in other words, Benjamin is left-handed.

4. *Rules* have the general form `Head:- Body`, which looks like this in a program:

```
relation(object,object,...,object):-
   relation(object,...,object),
     .
     .
   relation(object,...,object).
```

5. You are free to choose names for the relations and objects in your programs, subject to the following constraints:
   - Object names must begin with a lower-case letter, followed by any number of characters; characters are upper-case or lower-case letters, digits, and underscores.
   - Properties and relation names must start with a lower-case letter, followed by any combination of letters, digits, and underscore characters.
6. A *predicate* is the symbolic name (identifier) for a relation and a sequence of arguments. A Prolog program is a sequence of clauses and directives, and a procedure is a sequence of clauses defining a predicate. Clauses that belong to the same predicate must follow one another.
7. *Variables* enable you to write general facts and rules and ask general questions.
   - Variable names in Visual Prolog must begin with a capital letter or an underscore character (_), after which you can use any number of letters (upper-case or lower-case), digits, or underscores.
   - Variables in Prolog get their values by being matched to constants in facts or rules. Until it gets a value, a variable is said to be free; when it gets a value, it becomes bound.
   - You can't store information globally by binding a value to a variable, because a variable is only bound within a clause.
8. If you only need certain information from a query, you can use *anonymous variables* to ignore the values you don't need. In Prolog, the anonymous variable is represented by a lone underscore (_).

   The anonymous variable can be used in place of any other variable; it matches anything. The anonymous variable will never get set to a value.
9. Asking Prolog questions about the facts in your program is known as *querying the Prolog system*; the query is commonly called a *goal*. Prolog tries

to satisfy a goal (answer the query) by starting at the top of the facts, looking at each fact until it reaches the bottom.

10. A *compound goal* is a goal made up of two or more parts; each part of the compound goal is called a *subgoal*. Compound goals can be *conjunctions* (subgoal A and subgoal B) or *disjunctions* (subgoal A or subgoal B).

11. Comments make your programs easier to read; you can enclose a comment with delimiters `/* like this */` or precede a single-line comment with a percent sign, `% like this`.

12. There are several ways Prolog can match one thing to another:

    - Identical structures match each other.
    - A free variable matches a constant or a previously-bound variable (and becomes bound to that value).
    - Two free variables can match (and be bound to) each other. As long as the binding lasts, they are treated as a single variable; if one gets a value, the other will immediately have the same value.

CHAPTER 3

Visual Prolog Programs

The syntax of Visual Prolog is designed to express knowledge about properties and relationships. You've already seen the basics of how this is done; in Chapter 2 you learned about clauses (facts and rules), predicates, variables, and goals.

Unlike other versions of Prolog, Visual Prolog is a typed Prolog compiler; you declare the *types* of the objects that each predicate applies to. The type declarations allow Visual Prolog programs to be compiled right down to native machine code, giving execution speeds similar to those of compiled C and pascal.

We discuss the four basic sections of a Visual Prolog program – where you declare and define the predicates and arguments, define rules, and specify the program's goal – in the first part of this chapter. In the second part of this chapter we take a closer look at declarations and rule syntax. Then, at the end of this chapter, we briefly introduce the other sections of a Visual Prolog program, including the **facts**, **constants**, and various **global** sections, and compiler directives.

Visual Prolog's Basic Program Sections

Generally, a Visual Prolog program includes four basic program sections. These are the **clauses** section, the **predicates** section, the **domains** section, and the **goal** section.

- The **clauses** section is the heart of a Visual Prolog program; this is where you put the facts and rules that Visual Prolog will operate on when trying to satisfy the program's goal.
- The **predicates** section is where you declare your predicates and the domains (types) of the arguments to your predicates. (You don't need to declare Visual Prolog's built-in predicates.)
- The **domains** section is where you declare any domains you're using that aren't Visual Prolog's standard domains. (You don't need to declare standard domains.)

- The **goal** section is where you put the starting goal for a Visual Prolog program.

The Clauses Section

The **clauses** section is where you put all the facts and rules that make up your program. Most of the discussion in Chapter **2** was centered around the clauses (facts and rules) in your programs; what they convey, how to write them, and so on.

If you understand what facts and rules are and how to write them in Prolog, you know what goes in the **clauses** section. Clauses for a given predicate must be placed together in the **clauses** section; a sequence of clauses defining a predicate is called a *procedure*.

When attempting to satisfy a goal, Visual Prolog will start at the top of the **clauses** section, looking at each fact and rule as it searches for a match. As Visual Prolog proceeds down through the **clauses** section, it places internal pointers next to each clause that matches the current subgoal. If that clause is not part of a logical path that leads to a solution, Visual Prolog returns to the set pointer and looks for another match (this is *backtracking*, which we mentioned in Chapter **2**).

The Predicates Section

If you define your own predicate in the **clauses** section of a Visual Prolog program, you *must* declare it in a **predicates** section, or Visual Prolog won't know what you're talking about. When you declare a predicate, you tell Visual Prolog which domains the arguments of that predicate belong to.

Visual Prolog comes with a wealth of built-in predicates. You don't need to declare any of Visual Prolog's built-in predicates that you use in your program. The *Visual Prolog* on-line help gives a full explanation of the built-in predicates.

Facts and rules define predicates. The **predicates** section of the program simply lists each predicate, showing the types (domains) of its arguments. Although the **clauses** section is the heart of your program, Visual Prolog gets much of its efficiency from the fact that you also declare the types of objects (arguments) that your facts and rules refer to.

How to Declare User-Defined Predicates

A *predicate declaration* begins with the predicate name, followed by an open (left) parenthesis. After the predicate name and the open parenthesis come zero or more arguments to the predicate.

```
predicateName(argument_type1, argument_type2, ..., argument_typeN)
```

Each argument type is followed by a comma, and the last argument type is followed by the closing (right) parenthesis. Note that, unlike the clauses in the **clauses** section of your program, a predicate declaration is not followed by a period. The argument types are either standard domains or domains that you've declared in the **domains** section.

Predicate Names

The name of a predicate must begin with a letter, followed by a sequence of letters, digits, and underscores. The case of the letters does not matter, but we strongly recommend using only a lower-case letter as the first letter in the predicate name. (Other versions of Prolog don't allow predicate names to begin with upper-case letters or underscores, and future versions of Visual Prolog might not, either.) Predicate names can be up to 250 characters long.

You can't use spaces, the minus sign, asterisks, slashes, or other non-alphanumeric characters in predicate names. Valid naming characters in Visual Prolog consist of the following:

Upper-case Letters	:	`A, B, ... , Z`
Lower-case Letters	:	`a, b, ... , z`
Digits	:	`0, 1, ... , 9`
Underscore character	:	`_`

All predicate names and arguments can consist of combinations of these characters, as long as you obey the rules for forming both predicate and argument names.

Below are a few examples of legal and illegal predicate names.

Legal Predicate Names	**Illegal Predicate Names**
fact	[fact]
is_a	*is_a*
has_a	has/a
patternCheckList	pattern-Check-List

choose_Menu_Item	choose Menu Item
predicateName	predicate<Name>
first_in_10	>first_in_10

Predicate Arguments

The *arguments* to the predicates must belong to known Visual Prolog domains. A domain can be a *standard domain*, or it can be one you declare in the **domains** section.

Examples

1. If you declare a predicate `my_predicate(symbol, integer)` in the **predicates** section, like this:

   ```
   PREDICATES
      my_predicate(symbol, integer)
   ```

   you don't need to declare its arguments' domains in a **domains** section, because ***symbol*** and ***integer*** are standard domains. But if you declare a predicate `my_predicate(name, number)` in the **predicates** section, like this:

   ```
   PREDICATES
      my_predicate(name, number)
   ```

   you will need to declare suitable domains for *name* and *number*. Assuming you want these to be ***symbol*** and ***integer*** respectively, the domain declaration looks like this:

   ```
   DOMAINS
      name = symbol
      number = integer

   PREDICATES
      my_predicate(name, number)
   ```

2. This program excerpt shows some more predicate and domain declarations:

   ```
   DOMAINS
      person, activity = symbol
      car, make, color = symbol
      mileage, years_on_road, cost = integer
   ```

```
PREDICATES
   likes(person, activity)
   parent(person, person)
   can_buy(person, car)
   car(make, mileage, years_on_road, color, cost)
   green(symbol)
   ranking(symbol, integer)
```

This excerpt specifies the following information about these predicates and their arguments:

- The predicate **likes** takes two arguments (person and activity), both of which belong to unique **symbol** domains (which means that their values are names rather than numbers).
- The predicate **parent** takes two person arguments, where person is a **symbol** type.
- The predicate **can_buy** takes two arguments, person and car, which are also both **symbol** types.
- The predicate **car** takes five arguments: make and color are of unique **symbol** domains, while *mileage*, *years_on_road*, and *cost* are of unique **integer** domains.
- The predicate **green** takes one argument, a symbol: there is no need to declare the argument's type, because it's of the standard domain **symbol**.
- The predicate **ranking** takes two arguments, both of which belong to standard domains (**symbol** and **integer**), so there is no need to declare the argument types.

Chapter 5, "Simple and Compound Objects," gives more detail about domain declarations.

The Domains Section

In traditional Prolog there is only one type - **the term**. We have the same in Visual Prolog, but we are declaring what the domains of the arguments to the predicates actually are.

Domains enable you to give distinctive names to different kinds of data that would otherwise look alike. In a Visual Prolog program, objects in a relation (the arguments to a predicate) belong to domains; these can be pre-defined domains, or special domains that you specify.

The **domains** section serves two very useful purposes. First, you can give meaningful names to domains even if, internally, they are the same as domains that already exist. Second, special domain declarations are used to declare data structures that are not defined by the standard domains.

It is sometimes useful to declare a domain when you want to clarify portions of the **predicates** section. Declaring your own domains helps document the predicates that you define by giving a useful name to the argument type.

Examples

1. Here's an example to illustrate how declaring domains helps to document your predicates:

   ```
   Frank is a male who is 45 years old.
   ```

   With the pre-defined domains, you come up with the following predicate declaration:

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

   This declaration will work fine for most purposes. But suppose you want to maintain your code months after you've finished writing it. The preceding predicate declaration won't mean much to you in six months. Instead, the following declarations will help you understand what the arguments in the predicate declaration stand for:

   ```
   DOMAINS
      name, sex = symbol
      age       = integer

   PREDICATES
      person(name, sex, age)
   ```

   One of the main advantages of this declarations, is that Visual Prolog can catch type errors, like the following obvious mistake:

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

   Even though ***name*** and ***sex*** are both defined as ***symbol***, they are not equivalent to each other. This enables Visual Prolog to detect an error if you accidentally swap them. This feature is very useful when your programs get large and complex.

   You might be wondering why we don't use special domains for all argument declarations, since special domains communicate the meaning of the

argument so much better. The answer is that once an argument is typed to a specific domain, that domain can't be mixed with another domain you have declared, even if the domains are the same! So, even though ***name*** and ***sex*** are of the same domain (***symbol***), they can't be mixed. However, all user-defined domains can be matched with the pre-defined domains.

2. This next example program will yield a *type error* when run (see **Testing Language Tutorial Examples** on page 12):

```
                    /* Program ch03e01.pro */

DOMAINS
   product,sum = integer

PREDICATES
   add_em_up(sum,sum,sum)
   multiply_em(product,product,product)

CLAUSES
   add_em_up(X,Y,Sum):-
      Sum=X+Y.

   multiply_em(X,Y,Product):-
      Product=X*Y.
```

   This program does two things: It adds and it multiplies. Add the goal

```
   add_em_up(32, 54, Sum).
```

   Visual Prolog (the Test Goal) will come up with

```
   Sum=86
   1 Solution
```

   which is the sum of the two integers you supplied the program.

   On the other hand, this program will also multiply two arguments with the ***multiply_em*** predicate. Now experiment with this program. If you need to figure out what the product of 31 and 13 is, you could enter the goal:

```
   multiply_em(31, 13, Product).
```

   Visual Prolog (the Test Goal) would then respond with the correct answer.

```
   Product=403
   1 Solution
```

   But suppose you need the sum of 42 and 17; the goal for this would be

```
   add_em_up(42, 17, Sum).
```

Now you need to double the product of 31 and 17, so you write the following goal:

```
multiply_em(31, 17, Sum), add_em_up(Sum, Sum, Answer).
```

You might expect Visual Prolog (the Test Goal) to return

```
Sum=527, Answer=1054
1 Solution
```

But, instead, you get a type error. What happened is that you tried to pass the resulting value of multiply_em (that is, of domain product), into the first and second arguments in ***add_em_up***, which have domains of ***sum***. This yields a type error because ***product*** is a different domain than ***sum***. Even though both domains are really of type **integer**, they are different domains, and are treated as such.

So, if a variable is used in more than one predicate within a clause, it must be declared the same in each predicate. Be sure that you fully understand the concept behind the type error given here; knowing the concept will avoid frustrating compiler error messages. Later in this chapter we will describe the different automatic and explicit type-conversions Visual Prolog offers.

3. To further understand how you can use domain declarations to catch type errors, consider the following program example:

```
                    /* Program ch03e02.pro */

DOMAINS
   brand,color = symbol
   age = byte
   price, mileage = ulong

PREDICATES
   car(brand,mileage,age,color,price)

CLAUSES
   car(chrysler,130000,3,red,12000).
   car(ford,90000,4,gray,25000).
   car(datsun,8000,1,black,30000).
```

Here, the ***car*** predicate declared in the **predicates** section takes five arguments. One belongs to the ***age*** domain, which is of **byte** type. On the 'x86 family of CPUs, a **byte** is an 8-bit unsigned integer, which can take on values between 0 and 255, both inclusive. Similarly, the domains ***mileage*** and ***price*** are of type **ulong**, which is a 32-bit unsigned integer, and the domains ***brand*** and ***color*** are of type **symbol**.

We'll discuss the built-in domains in greater detail in a moment. For now, load this program into the TestGoal project (see page 13) and try each of the following goals in turn:

```
car(renault, 13, 40000, red, 12000).
car(ford, 90000, gray, 4, 25000).
car(1, red, 30000, 80000, datsun).
```

Each goal produces a domain error. In the first case, for example, it's because ***age*** must be a **byte**. Hence, Visual Prolog can easily detect if someone typing in this goal has reversed the ***mileage*** and ***age*** objects in predicate ***car***. In the second case, ***age*** and ***color*** have been swapped, and in the third case you get to find out for yourself where the mixups are.

The Goal Section

Essentially, the **goal** section is the same as the body of a rule: it's simply a list of subgoals. There are two differences between the **goal** section and a rule:

1. The **goal** keyword is *not* followed by :-.
2. Visual Prolog automatically executes the goal when the program runs.

It's as if Visual Prolog makes a call to **goal**, and the program runs, trying to satisfy the body of the goal rule. If the subgoals in the **goal** section all succeed, then the program terminates successfully. If, while the program is running, a subgoal in the **goal** section fails, then the program is said to have failed. (Although, from an external point of view, there isn't necessarily any difference; the program simply terminates.)

A Closer Look at Declarations and Rules

Visual Prolog has several built-in standard domains. You can use standard domains when declaring the types of a predicate's arguments. Standard domains are already known to Visual Prolog and should not be defined in the **domains** section.

We'll first look at all the integral ones, shown in Table 3.1.

Table 3.1:Integral Standard Domains

Domain	***Description and implementation***

short A small, signed, quantity.

All platforms	16 bits,2s comp	-32768 .. 32767

ushort A small, unsigned, quantity.

All platforms	16 bits	0 .. 65535

long A large signed quantity

All platforms	32 bits,2s comp	-2147483648 .. 2147483647

ulong A large, unsigned quantity

All platforms	32 bits	0 .. 4294967295

integer A signed quantity, having the natural size for the machine/platform architecture in question.

16bit platforms	16 bits,2s comp	-32768 .. 32767
32bit platforms	32 bits,2s comp	-2147483648 .. 2147483647

unsigned An unsigned quantity, having the natural size for the machine/platform architecture in question.

16bit platforms	16 bits	0 .. 65535
32bit platforms	32 bits	0 .. 4294967295

byte

All platforms	[3] 8 bits	0 .. 255

word

All platforms	16 bits	0 .. 65535

dword

All platforms	32 bits	0 .. 4294967295

Syntactically, a value belonging in one of the integral domains is written as a sequence of digits, optionally preceded by a minus-sign for the signed domains, with no white-space. There are also octal and hexadecimal syntaxes for the integral domains; these will be illustrated in chapter **9**.

The ***byte***, ***word***, and ***dword*** domains are most useful when dealing with machine-related quantities, except perhaps for the ***byte***; an 8-bit integral quantity can prove quite relevant, as we have already seen. For general use, the ***integer*** and ***unsigned*** quantities are the ones to use, augmented by the ***short*** and ***long*** (and their unsigned counterparts) for slightly more specialized applications. Generally, the most efficient code results from using what's natural for the machine; a ***short*** is not as efficient on a '386 platform as a ***long***, and a ***long*** is not as efficient on a '286 platform as a ***short***, hence the different implementations of ***integer*** and ***unsigned***.

In domain declarations, the ***signed*** and **unsigned** keywords may be used in conjunction with the ***byte***, ***word***, and ***dword*** built-in domains to construct new integral domains, as in

```
DOMAINS
      i8 = signed byte
```

creating a new integral domain having a range of -128 to +127.

The other basic domains are shown in table 3.2. Visual Prolog recognizes several other standard domains, but we cover them in other chapters, after you have a good grasp of the basics.

Table 3.3: Basic Standard Domains

Domain	Description and implementation
char	A character, implemented as an unsigned byte. Syntactically, it is written as a character surrounded by single quotation marks: 'a'.

real	A floating-point number, implemented as 8 bytes in accordance with IEEE conventions; equivalent to C's ***double***. Syntactically, a real is written with an optional sign (+ or -) followed by some digits *DDDDDDD*, then an optional decimal point (.) followed by more digits *DDDDDDD*, and an optional exponential part (e(+ or -)DDD):
	`<+|-> DDDDD <.> DDDDDDD <e <+|-> DDD>`
	Examples of real numbers:
	`42705 9999 86.72`
	`9111.929437521e238 79.83e+21`
	
	Here 79.83e+21 means 79.83 x 10^21, just as in other languages.
	The permitted number range is 1) 10^{-307} to 1) 10^{308} (1*e*-307 to 1*e*+308). Values from the integral domains are automatically converted to real numbers when necessary.

string	A sequnce of characters, implemented as a pointer to a zero-terminated byte array, as in C. Two formats are permitted for strings: 1. a sequence of letters, numbers and underscores, provided the first character is lower-case; or 2. a character sequence surrounded by a pair of double quotation marks. Examples of strings: `telephone_number "railway ticket" "Dorid Inc"` Strings that you write in the program can be up to 255 characters long. Strings that the Visual Prolog system reads from a file or builds up internally can be up to 64K characters long on 16-bit platforms, and (theoretically) up to 4G long on 32-bit platforms.
symbol	A sequence of characters, implemented as a pointer to an entry in a hashed symbol-table, containing strings. The syntax is the same as for strings.

Symbols and strings are largely interchangeable as far as your program is concerned, but Visual Prolog stores them differently. Symbols are kept in a look-up table, and their addresses, rather than the symbols themselves, are stored to represent your objects. This means that symbols can be matched very quickly,

and if a symbol occurs repeatedly in a program, it can be stored very compactly. Strings are not kept in a look-up table; Visual Prolog examines them character-by-character whenever they are to be matched. You must determine which domain will give better performance in a particular program.

The following table gives some examples of simple objects that belong to the basic standard domains.

Table 3.4: Simple Objects

`"&&", caitlin, "animal lover", b_l_t`	(symbol or string)
`-1, 3, 5, 0`	(integer)
`3.45, 0.01, -30.5, 123.4e+5`	(real)
`'a', 'b', 'c'  '/', '&'`	(char)

Typing Arguments in Predicate Declarations

Declaring the domain of an argument in the **predicates** section is called *typing* the argument. For example, suppose you have the following relationship and objects:

```
Frank is a male who is 45 years old.
```

The Prolog fact that corresponds to this natural language relation might be

```
person(frank, male, 45).
```

In order to declare ***person*** as a predicate with these three arguments, you could place the following declaration in the **predicates** section:

```
person(symbol, symbol, unsigned)
```

Here, you have used standard domains for all three arguments. Now, whenever you use the predicate ***person***, you must supply three arguments to the predicate; the first two must be of type ***symbol***, while the third argument must be an ***integer***.

If your program only uses standard domains, it does not need a **domains** section; you have seen several programs of this type already.

Or, suppose you want to define a predicate that will tell you the position of a letter in the alphabet. That is,

```
alphabet_position(Letter, Position)
```

will have *Position* = 1 if *Letter* = a, *Position* = 2 if *Letter* = b, and so on. The clauses for this predicate would look like this:

```
alphabet_position(A_character, N).
```

If standard domains are the only domains in the predicate declarations, the program does not need a **domains** section. Suppose you want to define a predicate so that the goal will be true if *A_character* is the *N*-th letter in the alphabet. The clauses for this predicate would look like this:

```
alphabet_position('a', 1).
alphabet_position('b', 2).
alphabet_position('c', 3).
...
alphabet_position('z', 26).
```

You can declare the predicate as follows:

```
PREDICATES
   alphabet_position(char, unsigned)
```

and there is no need for a **domains** section. If you put the whole program together, you get

```
PREDICATES
   alphabet_position(char, integer)

CLAUSES
   alphabet_position('a', 1).
   alphabet_position('b', 2).
   alphabet_position('c', 3).
   /* ... other letters go here ... */
   alphabet_position('z', 26).
```

Here are a few sample goals you could enter:

```
alphabet_position('a', 1).

alphabet_position(X, 3).

alphabet_position('z', What).
```

Exercises

1. Program `ch03e04.pro` is a complete Visual Prolog program that functions as a mini telephone directory. The **domains** section is not needed here, since only standard domains are used.

```
                    /* Program ch03e04.pro */

PREDICATES
   phone_number(symbol,symbol)

CLAUSES
   phone_number("Albert","EZY-3665").
   phone_number("Betty","555-5233").
   phone_number("Carol","909-1010").
   phone_number("Dorothy","438-8400").

goal
```

Add each of these goals in turn to the code of the program `ch03e04.pro`, then try them with the **Test Goal**:

a. `phone_number("Carol", Number).`

b. `phone_number(Who, "438-8400").`

c. `phone_number("Albert", Number).`

d. `phone_number(Who, Number).`

Now update the clauses. Suppose that Kim shares a condominium with Dorothy and so has the same phone number. Add this fact to the **clauses** section and try the goal

```
phone_number(Who, "438-8400").
```

You should get two solutions to this query:

```
Who=Dorothy
Who=Kim
2 Solutions
```

2. To illustrate the ***char*** domain, program `ch03e05.pro` defines ***isletter***, which, when given the goals

```
isletter('%').
isletter('Q').
```

will return No and Yes, respectively.

```
                    /* Program ch03e05.pro */

PREDICATES
   isletter(char)
```

```
CLAUSES
   /* When applied to characters, '<=' means */
   /* "alphabetically precedes or is the same as" */
   isletter(Ch):-
      'a' <= Ch,
      Ch <= 'z'.

   isletter(Ch):-
      'A' <= Ch,
      Ch <= 'Z'.
```

Load Program `ch03e05.pro` into the TestGoal project (see page 13) and try each of these goals in turn:

a. `isletter('x').`

b. `isletter('2').`

c. `isletter("hello").`

d. `isletter(a).`

e. `isletter(X).`

Goals (c) and (d) will result in a type error message, and (e) will return a `Free variable` message, because you can't test whether an unidentified object follows *a* or precedes *z*.

Multiple Arity

The *arity* of a predicate is the number of arguments that it takes. You can have two predicates with the same name but different arity. You must group different arity versions of a given predicate name together in both the **predicates** and **clauses** sections of your program; apart from this restriction, the different arities are treated as completely different predicates.

```
                        /* Program ch03e06.pro */

DOMAINS

   person = symbol

PREDICATES

   father(person)                              % This person is a father
   father(person, person)   % One person is the father of the other person
```

```
CLAUSES
   father(Man):-
      father(Man,_).
   father(adam,seth).
   father(abraham,isaac).
```

Rule Syntax

Rules are used in Prolog when a fact depends upon the success (truth) of another fact or group of facts. As we explained in Chapter **2**, a Prolog rule has two parts: the *head* and the *body*. This is the generic syntax for a Visual Prolog rule:

```
HEAD :- <Subgoal>, <Subgoal>, ..., <Subgoal>.
```

The body of the rule consists of one or more subgoals. Subgoals are separated by commas, specifying conjunction, and the last subgoal in a rule is terminated by a period.

Each subgoal is a call to another Prolog predicate, which may succeed or fail. In effect, calling another predicate amounts to evaluating its subgoals, and, depending on their success or failure, the call will succeed or fail. If the current subgoal can be satisfied (proven true), the call returns, and processing continues on to the next subgoal. Once the final subgoal in a rule succeeds, the call returns successfully; if any of the subgoals fail, the rule immediately fails.

To use a rule successfully, Prolog must satisfy all of the subgoals in it, creating a consistent set of variable bindings as it does so. If one subgoal fails, Prolog will back up and look for alternatives to earlier subgoals, then proceed forward again with different variable values. This is called *backtracking*. A full discussion of backtracking and how Prolog finds solutions is covered in Chapter **4**.

Prolog if Symbol vs. IF in Other Languages

As we have mentioned earlier, the **:-** separating the head and the body of a rule, is read "if". However, a Prolog **if** differs from the **IF** written in other languages, such as Pascal.

In Pascal, for instance, the condition contained in the **IF** statement must be met *before* the body of the statement can be executed; in other words,

"if HEAD is true, then BODY is true (or: then do BODY)"

This type of statement is known as an *if/then conditional*. Prolog, on the other hand, uses a different form of logic in its rules. The head of a Prolog rule is concluded to be true if (*after*) the body of the rule succeeds; in other words,

"HEAD is true if BODY is true (or: if BODY can be done)"

Seen in this manner, a Prolog rule is in the form of a *then/if conditional*.

Automatic Type Conversions

When Visual Prolog matches two variables, it's not always necessary that they belong to the same domain. Also, variables can sometimes be bound to constants from other domains. This (selective) mixing is allowed because Visual Prolog performs automatic type conversion (from one domain to another) in the following circumstances:

- Between **string**s and **symbol**s.
- Between all the integral domains and also **real**. When a character is converted to a numeric value, the number is the ASCII value for that character.

An argument from a domain ***my_dom*** declared in this form

```
DOMAINS
      my_dom = <base domain>       /*<base domain> is a standard domain */
```

can mix freely with arguments from that base domain and all other standard domains that are compatible with that base domain. (If the base domain is ***string***, arguments from the ***symbol*** domain are compatible; if the base domain is ***integer***, arguments from the ***real***, ***char***, ***word***, etc., domains are compatible.

These type conversions mean, for example, that you can

- call a predicate that handles **string**s with a **symbol** argument, and vice versa
- call a predicate that handles **real**s with an **integer** argument
- call a predicate that handles characters with **integer** values
- use characters in expressions and comparisons without needing to look up their ASCII values.

There are a number of rules deciding what domain the result of the expression belongs to, when different domains are mixed. These will be detailed in chapter **9**.

Other Program Sections

Now that you're reasonably familiar with the **clauses**, **predicates**, **domains**, and **goal** sections of a Visual Prolog program, we'll tell you a little bit about some

other commonly-used program sections: the **facts** section, the **constants** section, and the various **global** sections. This is just an introduction; as you work through the rest of the tutorials in this book, you'll learn more about these sections and how to use them in your programs.

The Facts Section

A Visual Prolog program is a collection of facts and rules. Sometimes, while the program is running, you might want to update (change, remove, or add) some of the facts the program operates on. In such a case, the facts constitute a *dynamic* or *internal* database of facts; it can change while the program is running. Visual Prolog includes a special section for declaring the facts in the program that are to be a part of the dynamic (or changing) database of facts; this is the **facts** section.

The keyword **facts** declares the **facts** section. It is here that you declare the facts to be included in the dynamic facts section. Visual Prolog includes a number of built-in predicates that allow easy use of the dynamic facts section. The keyword **facts** is synonymous with the obsolete keyword **database.**

Chapter **8** provides a complete discussion of the facts section and the predicates used along with it.

The Constants Section

You can declare and use symbolic constants in your Visual Prolog programs. A constant declaration section is indicated by the keyword **constants**, followed by the declarations themselves, using the following syntax:

```
<Id> =  <Macro definition>
```

<*Id*> is the name of your symbolic constant, and `<Macro definition>` is what you're assigning to that constant. Each `<Macro definition>` is terminated by a newline character, so there can only be one constant declaration per line. Constants declared in this way can then be referred to later in the program.

Consider the following program fragment:

```
CONSTANTS
   zero = 0
   one  = 1
   two  = 2
   hundred = (10*(10-1)+10)
   pi   = 3.141592653
   ega  = 3
   slash_fill = 4
   red  = 4
```

Before compiling your program, Visual Prolog will replace each constant with the actual string to which it corresponds. For instance:

```
   ...,
   A = hundred*34, delay(A),
   setfillstyle(slash_fill, red),
   Circumf = pi*Diam,
   ...
```

will be handled by the compiler in exactly the same way as

```
   ...,
   A = (10*(10-1)+10)*34, delay(A),
   setfillstyle(4, 4),
   Circumf = 3.141592653*Diam,
   ...
```

There are a few restrictions on the use of symbolic constants:

- The definition of a constant can't refer to itself. For example:

  ```
  my_number = 2*my_number/2 /* Is not allowed  */
  ```

  will generate the error message Recursion in constant definition.

- The system does not distinguish between upper-case and lower-case in a **constants** declaration. Consequently, when a **constants** identifier is used in the **clauses** section of a program, the first letter must be lower-case to avoid confusing constants with variables. So, for example, the following is a valid construction:

  ```
  CONSTANTS
     Two = 2

  GOAL
     A=two, write(A).
  ```

- There can be several **constants** declaration sections in a program, but constants must be declared before they are used.
- Declared constants are effective from their point of declaration to the end of the source file, and in any files included after the declaration. Constant identifiers can only be declared once. Multiple declarations of the same identifier will result in the error message: `This constant is already defined`.

The Global Sections

Visual Prolog allows you to declare some domains, predicates, and clauses in your program to be *global* (rather than *local*); you do this by setting aside separate **global domains**, **global predicates**, and **global facts** sections at the top of your program. These global sections are discussed in the chapter **17**.

The Compiler Directives

Visual Prolog provides several *compiler directives* you can add to your program to tell the compiler to treat your code in specified ways when compiling. You can also set most of the compiler directives from the **Options** | **Project** | **Compiler Options** menu item in the Visual Prolog system. Compiler directives are covered in detail in the chapter **17**, but you'll want to know how to use a couple of them before you get to that chapter, so we introduce the basic ones here.

The include Directive

As you get more familiar with using Visual Prolog, you'll probably find that you use certain procedures over and over again in your programs. You can use the include directive to save yourself from having to type those procedures in again and again.

Here's an example of how you could use it:

1. You create a file (such as MYSTUFF.PRO) in which you declare your frequently used predicates (using domains and predicates sections) and give the procedures defining those predicates in a clauses section.
2. You write the source text for the program that will make use of these procedures.
3. At a natural boundary in your source text, you place the line

```
include "mystuff.pro"
```

(A natural boundary is anywhere in your program that you can place a domains, facts, predicates, clauses, or goal section.)

4. When you compile your source text, Visual Prolog will compile the contents of MYSTUFF.PRO right into the final compiled product of your source text.

You can use the include directive to include practically any often-used text into your source text, and one included file can in turn include another (but a given file can only be included once in your program). The include directive can appear at any natural boundary in your source text. However, you must observe the restrictions on program structure when you include a file into your source text.

Summary

These are the ideas we've introduced in this chapter:

1. A Visual Prolog program has the following basic structure:

```
DOMAINS
   /* ...
   domain declarations
   ... */

PREDICATES
   /* ...
   predicate declarations
   ... */

CLAUSES
   /* ...
   clauses (rules and facts)
   ... */

GOAL
   /* ...
   subgoal_1,
   subgoal_2,
   etc. */
```

2. The **clauses** section is where you put the facts and rules that Visual Prolog will operate on when trying to satisfy the program's goal.
3. The **predicates** section is where you declare your predicates and the domains (types) of the arguments to your predicates. Predicate names must begin with a letter (preferably lower-case), followed by a sequence of letters, digits, and

underscores, up to 250 characters long. You can't use spaces, the minus sign, asterisks, or slashes in predicate names. Predicate declarations are of the form

```
PREDICATES
   predicateName(argumentType1, argumentType2, ..., argumentTypeN)
```

argumentType1,..., *argumentTypeN* are either standard domains or domains that you've declared in the **domains** section. Declaring the domain of an argument and defining the argument's type are the same things.

4. The **domains** section is where you declare any nonstandard domains you're using for the arguments to your predicates. Domains in Prolog are like types in other languages. Visual Prolog's basic standard domains are ***char***, ***byte***, ***short***, ***ushort***, ***word***, ***integer***, ***unsigned***, ***long***, ***ulong***, ***dword***, ***real***, ***string***, and ***symbol***; the more specialized standard domains are covered in other chapters. The basic domain declarations are of the form

   ```
   DOMAINS
      argumentType1,..., argumentTypeN     = <standardDomain>
   ```

   Compound domain declarations are of the form:

   ```
      argumentType_1,..., argumentType_N    = <compoundDomain_1>;
                                            <compoundDomain_2>;
                                            < ... >;
                                            <compoundDomain_M>;
   ```

   Compound domains haven't been covered in this chapter; you'll see them in Chapter 5.

5. The **goal** section is where you put your program's *goal* (in PDC Prolog we also used here term *internal goal*); this allows the program to be compiled, built and run as standalone executable independent of the Visual Development Environment. In standalone executables, Visual Prolog only searches for the first solution for the program *goal*, and the values to which *goal* variables are bound are not displayed.

   Some Prolog environments (for instance, the old PDC Prolog environment) support, so called *external goals* (as counterpart to term *internal goal*). When the PDC Prolog environment runs a program that does not contain an internal goal, then the environment displays the special dialog in which you can enter an *external goal* at run time. With an external goal, Prolog searches for all goal solutions, and displays the values to which goal variables are bound. Visual Prolog's Visual Development Environment does not support *external goals*. However, for simple programs (like most examples in this Language Tutorial) you can use the special Visual Development Environment's utility

Test Goal. The **Test Goal** searches for all solutions for the *goal*, and displays the values to which the *goal* variables are bound.

6. The *arity* of a predicate is the number of arguments that it takes; two predicates can have the same name but different arity. You must group a predicate's different arity versions together in both the **predicates** and **clauses** sections, but different arities are treated as completely different predicates.

7. Rules are of the form

   ```
   HEAD :- <Subgoal1>, <Subgoal2>, ..., <SubgoalN>.
   ```

   For a rule to succeed, Prolog must satisfy all of its subgoals, creating a consistent set of variable bindings. If one subgoal fails, Prolog backs up and looks for alternatives to earlier subgoals, then proceeds forward with different variable values. This is called *backtracking*.

8. The **:-** ("if") in Prolog should not be confused with the **IF** used in other languages; a Prolog rule is in the form of a *then/if* conditional, while **IF** statements in other languages are in the form of an *if/then* conditional.

CHAPTER 4

Unification and Backtracking

This chapter is divided into four main parts. In the first part, we examine in detail the process Visual Prolog uses when trying to match a call (from a subgoal) with a clause (in the **clauses** section of the program). This search process includes a procedure known as *unification*, which attempts to match up the data-structures embodied in the call with those found in a given clause. In Prolog, unification implements several of the procedures you might know from other, more traditional languages – procedures such as parameter passing, case selection, structure building, structure access, and assignment.

In the second part, we show you how Visual Prolog searches for solutions to a goal (through *backtracking*) and how to control a search. This includes techniques that make it possible for a program to carry out a task that would otherwise be impossible, either because the search would take too long (which is less likely with Visual Prolog than with other Prologs) or because the system would run out of free memory.

In the third part of this chapter, we introduce a predicate you can use to encourage backtracking, and go into more detail about how you can control backtracking. We also introduce a predicate you can use to verify that a certain constraint in your program is (or is not) met.

To shed more light on the subject, in the fourth part of this chapter we review the more important tutorial material (presented so far) from a procedural perspective. We show how you can understand the basic aspects of Prolog, a declarative language, by also looking at them as procedures.

Matching Things Up: Unification

Consider Program `ch04e01.pro` in terms of how the **Test Goal** utility (see page 12) will search for all solutions for the goal

```
written_by(X, Y).
```

When Visual Prolog tries to fulfill the goal `written_by(X, Y)`, it must test each ***written_by*** clause in the program for a match. In the attempt to match the

arguments *X* and *Y* with the arguments found in each ***written_by*** clause, Visual Prolog will search from the top of the program to the bottom. When it finds a clause that matches the goal, it binds values to free variables so that the goal and the clause are identical; the goal is said to *unify* with the clause. This matching operation is called *unification*.

```
                        /* Program ch04e01.pro */

DOMAINS
   title,author = symbol
   pages        = unsigned

PREDICATES
   book(title, pages)
   written_by(author, title)
   long_novel(title)

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

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

   long_novel(Title):-
      written_by(_, Title),
      book(Title, Length),
      Length > 300.
```

Since *X* and *Y* are free variables in the goal, and a free variable can be unified with any other argument (even another free variable), the call (goal) can be unified with the first ***written_by*** clause in the program, as shown here:

```
        written_by(   X   ,    Y    ).
                     |         |
        written_by(fleming, "DR NO").
```

Visual Prolog makes a match, *X* becomes bound to `fleming`, and *Y* becomes bound to "*DR NO*." At this point, Visual Prolog displays

```
X=fleming, Y=DR NO
```

Since the **Test Goal** (see page 12) looks for all solutions for the specified goal, the goal is also unified with the second ***written_by*** clause

```
written_by(melville,"MOBY DICK").
```

and the Test Goal executable displays the second solution:

```
X=melville, Y=MOBY DICK
2 Solutions
```

If, on the other hand, you give the program the goal

```
written_by(X, "MOBY DICK").
```

Visual Prolog will attempt a match with the first clause for ***written_by***:

```
written_by(  X ,"MOBY DICK").
             |       |
written_by(fleming,"DR NO").
```

Since "*MOBY DICK*" and "*DR NO*" do not match, the attempt at unification fails. Visual Prolog then tries the next fact in the program:

```
written_by(melville, "MOBY DICK").
```

This does unify, and *X* becomes bound to `melville`.

Consider how Visual Prolog executes the following:

```
long_novel(X).
```

When Visual Prolog tries to fulfill a goal, it investigates whether or not the call can match a fact or the head of a rule. In this case, the match is with

```
long_novel(Title)
```

Visual Prolog looks at the clause for ***long_novel***, trying to complete the match by unifying the arguments. Since *X* is not bound in the goal, the free variable *X* can be unified with any other argument. *Title* is also unbound in the head of the ***long_novel*** clause. The goal matches the head of the rule and unification is made. Visual Prolog will subsequently attempt to satisfy the subgoals to the rule.

```
long_novel(Title):-
   written_by(_, Title),
   book(Title, Length),
   Length>300.
```

In attempting to satisfy the body of the rule, Visual Prolog will call the first subgoal in the body of the rule, `written_by(_, Title)`. Notice that, since who authored the book is immaterial, the anonymous variable (_) appears in the position of the ***author*** argument. The call `written_by(_, Title)` becomes the current subgoal, and Prolog searches for a solution to this call.

Prolog searches for a match with this subgoal from the top of the program to the bottom. In doing so, it achieves unification with the first fact for ***written_by*** as follows:

```
written_by(_,      Title),
           |         |
written_by(fleming,"DR NO").
```

The variable *Title* becomes bound to "*DR NO*" and the next subgoal, `book(Title, Length)`, is called with this binding.

Visual Prolog now begins its next search, trying to find a match with the call to ***book***. Since *Title* is bound to "*DR NO*", the actual call resembles `book("DR NO", Length)`. Again, the search starts from the top of the program. Notice that the first attempt to match with the clause `book("MOBY DICK", 250)` will fail, and Visual Prolog will go on to the second clause of ***book*** in search of a match. Here, the book title matches the subgoal and Visual Prolog binds the variable *Length* with the value 310.

The third clause in the body of ***long_novel*** now becomes the current subgoal:

```
Length > 300.
```

Visual Prolog makes the comparison and succeeds; 310 is greater than 300. At this point, all the subgoals in the body of the rule have succeeded and therefore the call `long_novel(X)` succeeds. Since the *X* in the call was unified with the variable *Title* in the rule, the value to which *Title* is bound when the rule succeeds is returned to the call and unified with the variable *X*. *Title* has the value "*DR NO*" when the rule succeeds, so Visual Prolog will output:

```
X=DR NO
1 Solution
```

In the following chapters, we will show several advanced examples of unification. However, there are still a few basics that need to be introduced first, such as complex structures. In the next section of this chapter, we'll discuss how Prolog searches for its solutions.

Backtracking

Often, when solving real problems, you must pursue a path to its logical conclusion. If this conclusion does not give the answer you were looking for, you must choose an alternate path. For instance, you might have played maze games when you were a child. One sure way to find the end of the maze was to turn left

at every fork in the maze until you hit a dead end. At that point you would back up to the last fork, and try the right-hand path, once again turning left at each branch encountered. By methodically trying each alternate path, you would eventually find the right path and win the game.

Visual Prolog uses this same backing-up-and-trying-again method, called *backtracking*, to find a solution to a given problem. As Visual Prolog begins to look for a solution to a problem (or goal), it might have to decide between two possible cases. It sets a marker at the branching spot (known as a *backtracking point*) and selects the first subgoal to pursue. If that subgoal fails (equivalent to reaching a dead end), Visual Prolog will backtrack to the *backtracking* point and try an alternate subgoal.

Here is a simple example (use the TestGoal see page 13 to run this example):

```
                        /* Program ch04e02.pro */

PREDICATES
   likes(symbol,symbol)
   tastes(symbol,symbol)
   food(symbol)

CLAUSES
   likes(bill,X):-
      food(X),
      tastes(X,good).

   tastes(pizza,good).
   tastes(brussels_sprouts,bad).

   food(brussels_sprouts).
   food(pizza).
```

This small program is made up of two sets of facts and one rule. The rule, represented by the relationship ***likes***, simply states that Bill likes good-tasting food.

To see how backtracking works, give the program the following goal to solve:

```
likes(bill, What).
```

When Prolog begins an attempt to satisfy a goal, it starts at the top of the program in search of a match.

In this case, it will begin the search for a solution by looking from the top for a match to the subgoal `likes(bill, What)`.

It finds a match with the first clause in the program, and the variable *What* is unified with the variable *X*. Matching with the head of the rule causes Visual Prolog to attempt to satisfy that rule. In doing so, it moves on to the body of the rule, and calls the first subgoal located there: `food(X)`.

When a new call is made, a search for a match to that call also begins at the top of the program.

In the search to satisfy the first subgoal, Visual Prolog starts at the top, attempting a match with each fact or head of a rule encountered as processing goes down into the program.

It finds a match with the call at the first fact representing the ***food*** relationship. Here, the variable *X* is bound to the value *brussels_sprouts*. Since there is more than one possible answer to the call `food(X)`, Visual Prolog sets a backtracking point next to the fact `food(brussels_sprouts)`. This backtracking point keeps track of where Prolog will start searching for the next possible match for `food(X)`.

When a call has found a successful match, the call is said to succeed, and the next subgoal in turn may be tried.

With *X* bound to *brussels_sprouts*, the next call made is

```
tastes(brussels_sprouts, good)
```

and Visual Prolog begins a search to attempt to satisfy this call, again starting from the top of the program. Since no clause is found to match, the call fails and Visual Prolog kicks in its automatic backtracking mechanism. When backtracking begins, Prolog retreats to the last backtracking point set. In this case, Prolog returns to the fact `food(brussels_sprouts)`.

Once a variable has been bound in a clause, the only way to free that binding is through backtracking.

When Prolog retreats to a backtracking point, it frees all the variables set after that point, and sets out to find another solution to the original call.

The call was `food(X)`, so the binding of `brussels_sprouts` for *X* is released. Prolog now tries to resolve this call, beginning from the place where it left off. It finds a match with the fact `food(pizza)]` and returns, this time with the variable *X* bound to the value *pizza*.

Prolog now moves on to the next subgoal in the rule, with the new variable binding. A new call is made, `tastes(pizza, good)`, and the search begins at the top of the program. This time, a match is found and the goal returns successfully.

Since the variable *What* in the goal is unified with the variable *X* in the ***likes*** rule, and the variable *X* is bound to the value *pizza*, the variable *What* is now bound to the value *pizza* and Visual Prolog reports the solution

```
What=pizza
1 Solution
```

Visual Prolog's Relentless Search for Solutions

As we've described earlier, with the aid of backtracking, Visual Prolog will not only find the first solution to a problem, but is actually capable of finding all possible solutions.

Consider Program `ch04e03.pro`, which contains facts about the names and ages of some players in a racquet club.

```
                        /* Program ch04e03.pro */

DOMAINS
   child = symbol
   age   = integer

PREDICATES
   player(child, age)

CLAUSES
   player(peter,9).
   player(paul,10).
   player(chris,9).
   player(susan,9).
```

You'll use Visual Prolog to arrange a ping-pong tournament between the nine-year-olds in a racquet club. There will be two games for each pair of club players. Your aim is to find all possible pairs of club players who are nine years old. This can be achieved with the compound goal:

```
goal
   player(Person1, 9),
   player(Person2, 9),
   Person1 <> Person2.
```

In natural language: Find *Person1* (age 9) and *Person2* (age 9) so that *Person1* is different from *Person2*.

1. Visual Prolog will try to find a solution to the first subgoal `player(Person1, 9)` and continue to the next subgoal only after the first subgoal is reached. The first subgoal is satisfied by matching *Person1* with *peter*. Now Visual Prolog can attempt to satisfy the next subgoal:

   ```
   player(Person2, 9)
   ```

   by also matching *Person2* with *peter*. Now Prolog comes to the third and final subgoal

   ```
   Person1 <> Person2
   ```

2. Since *Person1* and *Person2* are both bound to *peter*, this subgoal fails. Because of this, Visual Prolog backtracks to the previous subgoal, and searches for another solution to the second subgoal:

   ```
   player(Person2, 9)
   ```

   This subgoal is fulfilled by matching *Person2* with *chris*.

3. Now, the third subgoal:

   ```
   Person1 <> Person2
   ```

   can succeed, since *peter* and *chris* are different. Here, the entire goal is satisfied by creating a tournament between the two players, *chris* and *peter*.

4. However, since Visual Prolog must find all possible solutions to a goal, it backtracks to the previous goal – hoping to succeed again. Since

   ```
   player(Person2, 9)
   ```

   can also be satisfied by taking *Person2* to be *susan*, Visual Prolog tries the third subgoal once again. It succeeds (since *peter* and *susan* are different), so another solution to the entire goal has been found.

5. Searching for more solutions, Visual Prolog once again backtracks to the second subgoal, but all possibilities for this subgoal have been exhausted. Because of this, backtracking now continues back to the first subgoal. This can be satisfied again by matching *Person1* with *chris*. The second subgoal now succeeds by matching *Person2* with *peter*, so the third subgoal is satisfied, again fulfilling the entire goal. Here, another tournament has been scheduled, this time between *chris* and *peter*.

6. Searching for yet another solution to the goal, Visual Prolog backtracks to the second subgoal in the rule. Here, *Person2* is matched to *chris* and again the third subgoal is tried with these bindings. The third subgoal fails, since *Person1* and *Person2* are equal, so backtracking regresses to the second subgoal in search of another solution. *Person2* is now matched with *susan*,

and the third subgoal succeeds, providing another tournament for the racket club (*chris* vs. *susan*).

7. Once again, searching for all solutions, Prolog backtracks to the second subgoal, but this time without success. When the second subgoal fails, backtracking goes back to the first subgoal, this time finding a match for *Person1* with *susan*. In an attempt to fulfill the second subgoal, Prolog matches *Person2* with *peter*, and subsequently the third subgoal succeeds with these bindings. A fifth tournament has been scheduled for the players.
8. Backtracking again goes to the second subgoal, where *Person2* is matched with *chris*. A sixth solution is found for the racquet club, producing a full set of tournaments.
9. The final solution tried is with both *Person1* and *Person2* bound to *susan*. Since this causes the final subgoal to fail, Visual Prolog must backtrack to the second subgoal, but there are no new possibilities. Visual Prolog then backtracks to the first subgoal, but the possibilities for *Person1* have also been exhausted. No more solutions can be found for the goal, so the program terminates.

Type in this compound goal for the program:

```
player(Person1, 9),
player(Person2, 9),
Person1 <> Person2.
```

Verify that Visual Prolog (see how to use the Test Goal utility on page 13) responds with

```
Person1=peter, Person2=chris
Person1=peter, Person2=susan
Person1=chris, Person2=peter
Person1=chris, Person2=susan
Person1=susan, Person2=peter
Person1=susan, Person2=chris
6 Solutions
```

Notice how backtracking might cause Visual Prolog to come up with redundant solutions. In this example, Visual Prolog does not distinguish that `Person1 = peter` is the same thing as `Person2 = peter`. We will show you later in this chapter how to control the search Visual Prolog generates.

Exercise in Backtracking

Using Program `ch04e04.pro`, decide what Visual Prolog will reply to the following goal:

```
player(Person1, 9), player(Person2, 10).
```

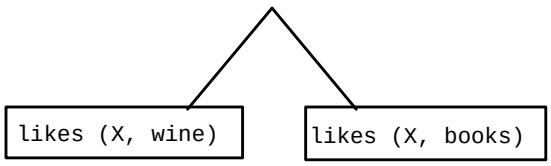
Check your answer by typing in the exercise and the given goal when you run the program.

A Detailed Look at Backtracking

With this simple example under your belt, you can take a more detailed look at how Visual Prolog's backtracking mechanism works. Start by looking at Program `ch04e04.pro` in light of the following goal, which consists of two subgoals:

```
likes(X, wine) , likes(X, books)
```

When evaluating the goal, Visual Prolog notes which subgoals have been satisfied and which have not. This search can be represented by a goal tree:



Before the goal evaluation begins, the goal tree consists of two unsatisfied subgoals. In the following goal tree diagrams, a subgoal satisfied in the goal tree is marked with an underline, and the corresponding clause is shown beneath that subgoal.

```
                        /* Program ch04e04.pro */

domains
   name,thing = symbol

predicates
   likes(name, thing)
   reads(name)
   is_inquisitive(name)
```

```
clauses
   likes(john,wine).
   likes(lance,skiing).
   likes(lance,books).
   likes(lance,films).
   likes(Z,books):-
      reads(Z),
      is_inquisitive(Z).

   reads(john).

   is_inquisitive(john).

goal
   likes(X,wine),likes(X,books).
```

The Four Basic Principles of Backtracking

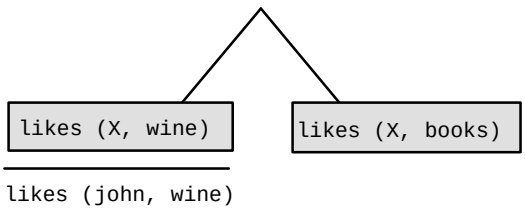
In this example, the goal tree shows that two subgoals must be satisfied. To do so, Visual Prolog follows the first basic principle of backtracking:

Subgoals must be satisfied in order, from top to bottom.

Visual Prolog determines which subgoal it will use when trying to satisfy the clause according to the second basic principle of backtracking:

Predicate clauses are tested in the order they appear in the program, from top to bottom.

When executing Program `ch04e04.pro`, Visual Prolog finds a matching clause with the first fact defining the ***likes*** predicate. Take a look at the goal tree now.



The subgoal `likes(X, wine)` matches the fact `likes(john, wine)` and binds *X* to the value *john*. Visual Prolog tries to satisfy the next subgoal to the right.

The call to the second subgoal begins a completely new search with the binding *X = john*. The first clause

```
likes(john, wine)
```

does not match the subgoal

```
likes(X, books)
```

since *wine* is not the same as *books*. Visual Prolog must therefore try the next clause, but *lance* does not match the value *X* (because, in this case, *X* is bound to *john*), so the search continues to the third clause defining the predicate ***likes***:

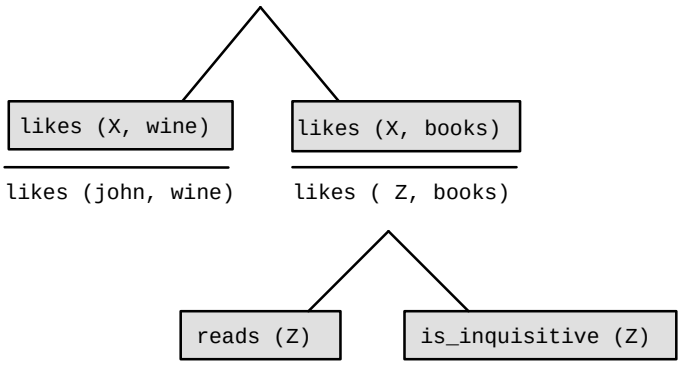
```
likes(Z, books):- reads(Z), is_inquisitive(Z).
```

The argument *Z* is a variable, so it is able to match with *X*. The second arguments agree, so the call matches the head of the rule. When *X* matches *Z*, the arguments are *unified*. With the arguments unified, Visual Prolog will equate the value *X* has (which is *john*) with the variable *Z*. Because of this, now the variable *Z* also has the value *john*.

The subgoal now matches the left side (head) of a rule. Continued searching is determined by the third basic principle of backtracking:

When a subgoal matches the head of a rule, the body of that rule must be satisfied next. The body of the rule then constitutes a new set of subgoals to be satisfied.

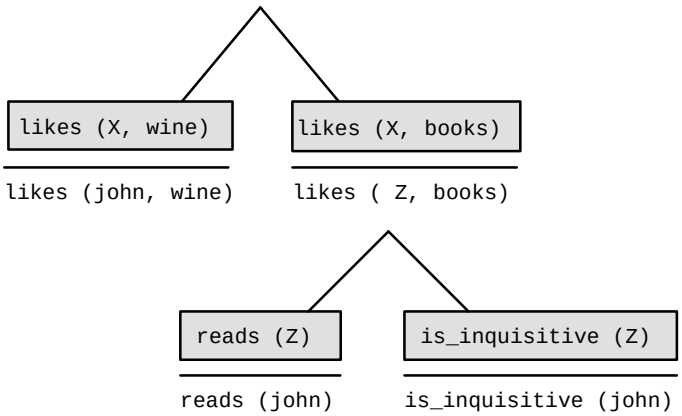
This yields the following goal tree:



The goal tree now includes the subgoals

```
reads(Z) and is_inquisitive(Z)
```

where *Z* is bound to the value *john*. Visual Prolog will now search for facts that match both subgoals. This is the resulting final goal tree:



According to the fourth basic principle of backtracking:

> ***A goal has been satisfied when a matching fact is found for each of the extremities (leaves) of the goal tree.***

So now the initial goal is satisfied.

Visual Prolog uses the result of the search procedure in different ways, depending on how the search was initiated. If the goal is a call from a subgoal in the body of a rule, Visual Prolog attempts to satisfy the next subgoal in the rule after the call has returned. If the goal is a query from the user, Visual Prolog (see page 13 how to use the Test Goal utility) replies directly:

```
X=john
1 Solution
```

As you saw in Program `ch04e04.pro`, having once satisfied the goal, Visual Prolog's Test Goal utility backtracks to find all alternate solutions. It also backtracks if a subgoal fails, hoping to re-satisfy a previous subgoal in such a way that the failed subgoal is satisfied by other clauses.

To fulfill a subgoal, Visual Prolog begins a search with the first clause that defines the predicate. One of two things can then happen:

1. It finds a matching clause, in which case the following occurs:
   a. If there is another clause that can possibly re-satisfy the subgoal, Visual Prolog places a pointer (to indicate a backtracking point) next to the matching clause.
   b. All free variables in the subgoal that match values in the clause are bound to the corresponding values.
   c. If the matching clause is the head of a rule, that rule's body is then evaluated; the body's subgoals must succeed for the call to succeed.
2. It can't find a matching clause, so the goal fails. Visual Prolog backtracks as it attempts to re-satisfy a previous subgoal. When processing reaches the last backtracking point, Visual Prolog frees all variables that had been assigned new values since the backtracking point was set, then attempts to re-satisfy the original call.

Visual Prolog begins a search from the top of the program. When it backtracks to a call, the new search begins from the last backtracking point set. If the search is unsuccessful, it backtracks again. If backtracking exhausts all clauses for all subgoals, the goal fails.

Backtracking in Standalone Executables

Here is another, slightly more complex, example, illustrating how in Visual Prolog backtracking works to find the *goal* solution, when the program is compiled and run as a standalone executable (see **Testing Examples as Standalone Executables** on page 14).

```
                        /* Program ch04e05.pro */

predicates
   type(symbol, symbol)
   is_a(symbol, symbol)
   lives(symbol, symbol)
   can_swim(symbol)

clauses
   type(ungulate,animal).
   type(fish,animal).

   is_a(zebra,ungulate).
   is_a(herring,fish).
   is_a(shark,fish).
```

```
   lives(zebra,on_land).
   lives(frog,on_land).
   lives(frog,in_water).
   lives(shark,in_water).

   can_swim(Y):-
      type(X,animal),
      is_a(Y,X),
      lives(Y,in_water).

goal
   can_swim(What),
   write("A ",What," can swim\n"),
   readchar(_).
```

When the program is compiled and runs as a standalone executable (for example, using the menu command **Project | Run**), Visual Prolog will automatically begin executing the goal, attempting to satisfy all the subgoals in the **goal** section.

1. Visual Prolog calls the ***can_swim*** predicate with a free variable *What*. In trying to solve this call, Visual Prolog searches the program looking for a match. It finds a match with the clause defining ***can_swim***, and the variable *What* is unified with the variable *Y*.
2. Next, Visual Prolog attempts to satisfy the body of the rule. In doing so, Visual Prolog calls the first subgoal in the body of the rule, `type(X, animal)`, and searches for a match to this call. It finds a match with the first fact defining the ***type*** relationship.
3. At this point, *X* is bound to *ungulate*. Since there is more than one possible solution, Visual Prolog sets a backtracking point at the fact `type(ungulate, animal)`.
4. With *X* bound to *ungulate*, Visual Prolog makes a call to the second subgoal in the rule `(is_a(Y, ungulate))`, and again searches for a match. It finds one with the first fact, `is_a(zebra, ungulate)`. *Y* is bound to *zebra* and Prolog sets a backtracking point at `is_a(zebra, ungulate)`.
5. Now, with *X* bound to *ungulate* and *Y* bound to *zebra*, Prolog tries to satisfy the last subgoal, `lives(zebra, in_water)`. Prolog tries each ***lives*** clause, but there is no `lives(zebra, in_water)` clause in the program, so the call fails and Prolog begins to backtrack in search of another solution.
6. When Visual Prolog backtracks, processing returns to the last point where a backtracking point was placed. In this case, the last backtracking point was placed at the second subgoal in the rule, on the fact `is_a(zebra, ungulate)`.

7. When Visual Prolog reaches a backtracking point, it frees the variables that were assigned new values after the last backtracking point and attempts to find another solution to the call it made at that time. In this case, the call was `is_a(Y, ungulate)`.
8. Visual Prolog continues down into the clauses in search of another clause that will match with this one, starting from the point where it previously left off. Since there are no other clauses in the program that can match this one, the call fails and Visual Prolog backtracks again in an attempt to solve the original goal.
9. From this position, the last backtracking point was set at `type(ungulate, animal)`.
10. Visual Prolog frees the variables set in the original call and tries to find another solution to the call `type(X, animal)`. The search begins after the backtracking point. Visual Prolog finds a match with the next ***type*** fact in the program `(type(fish, animal))`; *X* is bound to *fish*, and a new backtracking point is set at that fact.
11. Visual Prolog now moves down to the next subgoal in the rule; since this is a new call, the search begins at the top of the program with `is_a(Y, fish)`.
12. Visual Prolog finds a match to this call and *Y* is bound to *herring*.
13. Since *Y* is now bound to *herring*, the next subgoal called is `lives(herring, in_water)`. Again, this is a new call, and the search begins from the top of the program.
14. Visual Prolog tries each ***lives*** fact, but fails to find a match and the subgoal fails.
15. Visual Prolog now returns to the last backtracking point, `is_a(herring, fish)`.
16. The variables that were bound by this matching are now freed. Starting at the point where it last left off, Visual Prolog now searches for a new solution to the call `is_a(Y, fish)`.
17. Visual Prolog finds a match with the next ***is_a*** clause, and *Y* becomes bound to the symbol *shark*.
18. Visual Prolog tries the last subgoal again, with the variable *Y* bound to *shark*. It calls `lives(shark, in_water)`; the search begins at the top of the program, since this is a new call. It finds a match and the last subgoal to the rule succeeds.

19. At this point, the body of the `can_swim(Y)` rule is satisfied. Visual Prolog returns *Y* to the call `can_swim(What)`. Since *What* is bound to *Y*, and *Y* is bound to *shark*, *What* is now bound to *shark* in the goal.

20. Visual Prolog continues processing where it left off in the **goal** section, and calls the second subgoal in the goal.

21. Visual Prolog completes the program by outputting

```
A shark can swim.
```

and the program terminates successfully.

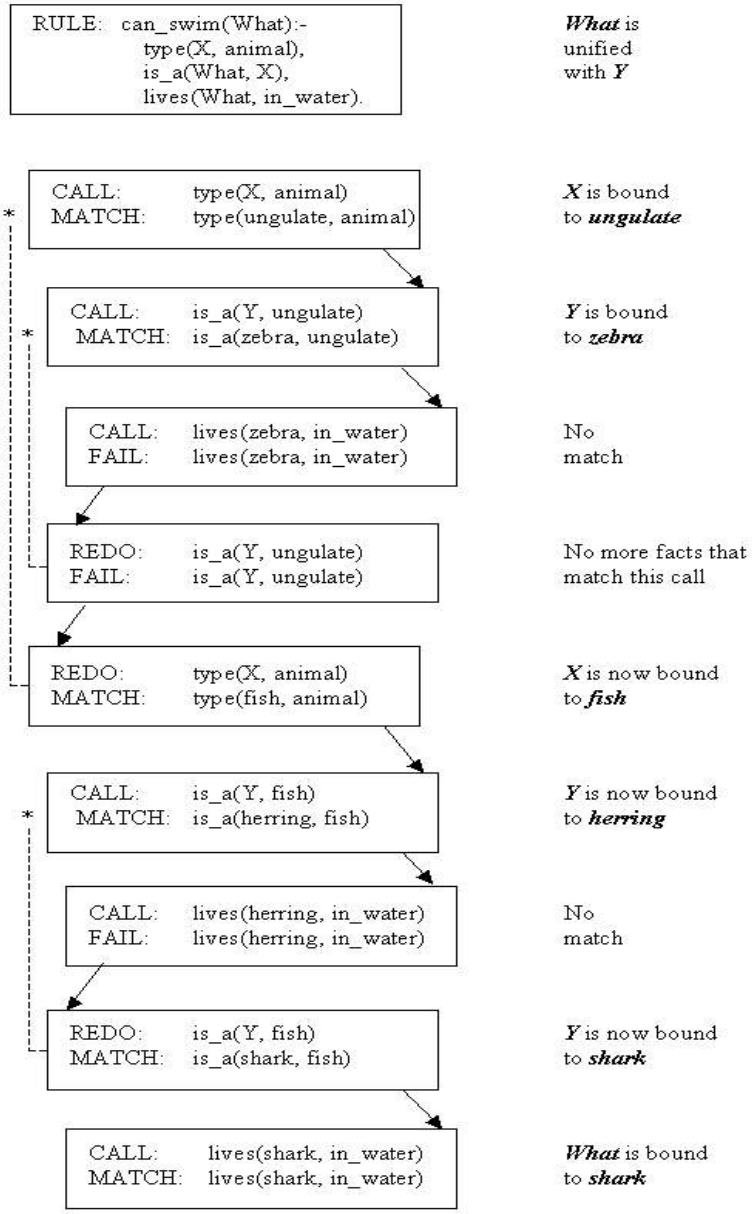


Figure 4.1: How the can_swim Program Works

Try to follow these steps using the Visual Prolog Debugger. Run the Debugger from the VDE with the **Project | Debug** command. When the Debugger window appears, choose the Debugger's menu command **View | Local Variables**, and use the **Run | Trace Into** command (or **F7** hot key) to trace the program execution and to inspect instantiation of variables. (For more instructions, see the chapter *Debugging Prolog Programs* in the *Getting Started* and the chapter *The Debugger* in the *Visual Development Environment* manuals.)

Controlling the Search for Solutions

Prolog's built-in backtracking mechanism can result in unnecessary searching; because of this, inefficiencies can arise. For instance, there may be times when you want to find unique solutions to a given question. In other cases, it may be necessary to force Visual Prolog to continue looking for additional solutions even though a particular goal has been satisfied. In cases such as these, you must control the backtracking process. In this section, we'll show you some techniques you can use to control Visual Prolog's search for the solutions to your goals.

Visual Prolog provides two tools that allow you to control the backtracking mechanism: the ***fail*** predicate, which is used to *force* backtracking, and the ***cut*** (signified by ***!***), which is used to *prevent* backtracking.

Using the *fail* Predicate

Visual Prolog begins backtracking when a call fails. In certain situations, it's necessary to force backtracking in order to find alternate solutions. Visual Prolog provides a special predicate, ***fail***, to force failure and thereby encourage backtracking. The effect of the ***fail*** predicate corresponds to the effect of the comparison `2 = 3` or any other impossible subgoal. Program `ch04e06.pro` illustrates the use of this special predicate.

```
                        /* Program ch04e06.pro */

DOMAINS
   name = symbol

PREDICATES
   father(name, name)
   everybody

CLAUSES
   father(leonard,katherine).
   father(carl,jason).
   father(carl,marilyn).
```

```
everybody:-
   father(X,Y),
   write(X," is ",Y,"'s father\n"),
   fail.
```

Let one wish to find all solutions to `father(X,Y)`. If he uses the **Test Goal** utility, then he can simply use the goal:

```
goal
   father(X,Y).
```

The Test Goal utility will find ALL solutions to `father(X,Y)` and display values of all variables in the following manner:

```
X=leonard, Y=katherine
X=carl, Y=jason
X=carl, Y=marilyn

3 Solutions
```

But if you compile this program and run the obtained executable, then Visual Prolog will find only the first matched solution for `father(X,Y)`. In built executables, once a *goal* specified in the program **goal** section has completely succeeded, there is nothing that tells Visual Prolog to backtrack. Because of this, an internal call to ***father*** will come up with only one solution and does not display any variables at all. This definitely is not what you need. However, the predicate ***everybody*** in Program `ch04e06.pro` uses ***fail*** to force backtracking, and therefore finds all possible solutions.

The object of the predicate ***everybody*** is to find ALL solutions to ***father*** and to produce a cleaner response from program runs. Compare the above answers of the Test Goal utility to the goal `father(X,Y)` and the answers to the goal:

```
goal
   everybody.
```

displayed by the generated executable:

```
leonard is katherine's father
carl is jason's father
carl is marilyn's father
```

The predicate ***everybody*** uses backtracking to generate more solutions for `father(X, Y)` by forcing Prolog to backtrack through the body of the ***everybody*** rule:

```
father(X, Y), write(X," is ",Y,"'s father\n"), fail.
```

fail can never be satisfied (it always fails), so Visual Prolog is forced to backtrack. When backtracking takes place, Prolog backtracks to the last call that can produce multiple solutions. Such a call is labeled *non-deterministic*. ***A non-deterministic call contrasts with a call that can produce only one solution, which is a deterministic call.***

The ***write*** predicate can't be re-satisfied (it can't offer new solutions), so Visual Prolog must backtrack again, this time to the first subgoal in the rule.

Notice that it's useless to place a subgoal after ***fail*** in the body of a rule. Since the predicate ***fail*** always fails, there would be no way of reaching a subgoal located after ***fail***.

Exercises

1. Load and run Program `ch04e06.pro` and evaluate the following goals:

   a. `father(X, Y).`

   b. `everybody.`

2. Edit the body of the rule defining ***everybody*** so that the rule ends with the call to the ***write*** predicate (delete the call to ***fail***). Now compile and run the program, giving ***everybody*** as the goal. Why doesn't Visual Prolog's Test Goal find all the solutions as it does with the query `father(X, Y)`?

3. Repair the call to ***fail*** at the end of the ***everybody*** rule. Again, give the query `everybody` as the goal and start the Test Goal. Why are the solutions to ***everybody*** terminated by `no`? For a clue, append ***everybody*** as a second clause to the definition of predicate ***everybody*** and re-evaluate the goal.

Preventing Backtracking: The Cut

Visual Prolog contains the cut, which is used to prevent backtracking; it's written as an exclamation mark (!). The effect of the cut is simple: It is impossible to backtrack across a cut.

You place the cut in your program the same way you place a subgoal in the body of a rule. When processing comes across the cut, the call to ***cut*** immediately succeeds, and the next subgoal (if there is one) is called. Once a cut has been passed, it is not possible to backtrack to subgoals placed before the cut in the clause being processed, and it is not possible to backtrack to other predicates defining the predicate currently in process (the predicate containing the cut).

There are two main uses of the cut:

1. When you know in advance that certain possibilities will never give rise to meaningful solutions, it's a waste of time and storage space to look for alternate solutions. If you use a cut in this situation, your resulting program will run quicker and use less memory. This is called a *green cut.*
2. When the logic of a program demands the cut, to prevent consideration of alternate subgoals. This is a *red cut*.

How to Use the Cut

In this section, we give examples that show how you can use the cut in your programs. In these examples, we use several schematic Visual Prolog rules (*r1*, *r2*, and *r3*), which all describe the same predicate ***r***, plus several subgoals (*a*, *b*, *c*, etc.).

Prevent Backtracking to a Previous Subgoal in a Rule

```
r1 :- a, b, !, c.
```

This is a way of telling Visual Prolog that you are satisfied with the first solution it finds to the subgoals *a* and *b*. Although Visual Prolog is able to find multiple solutions to the call to *c* through backtracking, it is not allowed to backtrack across the cut to find an alternate solution to the calls *a* or *b*. It is also not allowed to backtrack to another clause that defines the predicate ***r1***.

As a concrete example, consider Program `ch04e07.pro`.

```
                        /* Program ch04e07.pro */

PREDICATES
   buy_car(symbol,symbol)
   car(symbol,symbol,integer)
   colors(symbol,symbol)

CLAUSES
   buy_car(Model,Color):-
      car(Model,Color,Price),
      colors(Color,sexy),!,
      Price < 25000.

   car(maserati,green,25000).
   car(corvette,black,24000).
   car(corvette,red,26000).
   car(porsche,red,24000).
```

```
colors(red,sexy).
colors(black,mean).
colors(green,preppy).
```

In this example, the goal is to find a Corvette with a sexy color and a price that's ostensibly affordable. The cut in the ***buy_car*** rule means that, since there is only one Corvette with a sexy color in the known facts, if its price is too high there's no need to search for another car.

Given the goal

```
buy_car(corvette, Y)
```

1. Visual Prolog calls ***car***, the first subgoal to the ***buy_car*** predicate.
2. It makes a test on the first car, the Maserati, which fails.
3. It then tests the next ***car*** clauses and finds a match, binding the variable *Color* with the value *black*.
4. It proceeds to the next call and tests to see whether the car chosen has a sexy color. Black is not a sexy color in the program, so the test fails.
5. Visual Prolog backtracks to the call to ***car*** and once again looks for a Corvette to meet the criteria.
6. It finds a match and again tests the color. This time the color is sexy, and Visual Prolog proceeds to the next subgoal in the rule: the cut. The cut immediately succeeds and effectively "freezes into place" the variable bindings previously made in this clause.
7. Visual Prolog now proceeds to the next (and final) subgoal in the rule: the comparison

   ```
   Price < 25000.
   ```

8. This test fails, and Visual Prolog attempts to backtrack in order to find another car to test. Since the cut prevents backtracking, there is no other way to solve the final subgoal, and the goal terminates in failure.

Prevent Backtracking to the Next Clause

The cut can be used as a way to tell Visual Prolog that it has chosen the correct clause for a particular predicate. For example, consider the following code:

```
r(1):- ! , a , b , c.
r(2):- ! , d.
r(3):- ! , c.
r(_):- write("This is a catchall clause.").
```

Using the cut makes the predicate ***r*** deterministic. Here, Visual Prolog calls ***r*** with a single integer argument. Assume that the call is `r(1)`. Visual Prolog searches the program, looking for a match to the call; it finds one with the first clause defining ***r***. Since there is more than one possible solution to the call, Visual Prolog places a backtracking point next to this clause.

Now the rule fires and Visual Prolog begins to process the body of the rule. The first thing that happens is that it passes the cut; doing so eliminates the possibility of backtracking to another ***r*** clause. This eliminates backtracking points, increasing the run-time efficiency. It also ensures that the error-trapping clause is executed only if none of the other conditions match the call to ***r***.

Note that this type of structure is much like a "case" structure written in other programming languages. Also notice that the test condition is coded into the head of the rules. You could just as easily write the clauses like this:

```
r(X) :- X = 1 , ! , a , b , c.
r(X) :- X = 2 , ! , d.
r(X) :- X = 3 , ! , c.
r(_) :- write("This is a catchall clause.").
```

However, you should place the testing condition in the head of the rule as much as possible, as doing this adds efficiency to the program and makes for easier reading.

As another example, consider the following program. Run this program with the Test Goal.

```
                        /* Program ch04e08.pro */

PREDICATES
   friend(symbol,symbol)
   girl(symbol)
   likes(symbol,symbol)
```

```
CLAUSES
   friend(bill,jane):-
      girl(jane),
      likes(bill,jane),
      !.
   friend(bill,jim):-
      likes(jim,baseball),
      !.
   friend(bill,sue):-
      girl(sue).

   girl(mary).
   girl(jane).
   girl(sue).

   likes(jim,baseball).
   likes(bill,sue).

goal
   friend(bill, Who).
```

Without cuts in the program, Visual Prolog would come up with two solutions: Bill is a friend of both Jim and Sue. However, the cut in the first clause defining ***friend*** tells Visual Prolog that, if this clause is satisfied, it has found a friend of Bill and there's no need to continue searching for more friends. A cut of this type says, in effect, that you are satisfied with the solution found and that there is no reason to continue searching for another friend.

Backtracking can take place inside the clauses, in an attempt to satisfy the call, but once a solution is found, Visual Prolog passes a cut. The ***friend*** clauses, written as such, will return one and only one friend of Bill's (given that a friend can be found).

Determinism and the Cut

If the ***friend*** predicate (defined in the previous program) were coded without the cuts, it would be a non-deterministic predicate (one capable of generating multiple solutions through backtracking). In many implementations of Prolog, programmers must take special care with non-deterministic clauses because of the attendant demands made on memory resources at run time. However, Visual Prolog makes internal checks for non-deterministic clauses, reducing the burden on you, the programmer.

However, for debugging (and other) purposes, it can still be necessary for you to intercede; the `check_determ` compiler directive is provided for this reason. If `check_determ` is inserted at the very beginning of a program, Visual Prolog will

display a warning if it encounters any non-deterministic clauses during compilation.

You can make non-deterministic clauses into deterministic clauses by inserting cuts into the body of the rules defining the predicate. For example, placing cuts in the clauses defining the ***friend*** predicate causes that predicate to be deterministic because, with the cuts in place, a call to ***friend*** can return one, and only one, solution.

The *not* Predicate

This program demonstrates how you can use the ***not*** predicate to identify an honor student: one whose grade point average (GPA) is at least 3.5 and who is not on probation.

```
                        /* Program ch04e10.pro */

DOMAINS
   name = symbol
   gpa  = real

PREDICATES
   honor_student(name)
   student(name, gpa)
   probation(name)

CLAUSES
   honor_student(Name):-
      student(Name, GPA),
      GPA>=3.5,
      not(probation(Name)).

   student("Betty Blue", 3.5).
   student("David Smith", 2.0).
   student("John Johnson", 3.7).

   probation("Betty Blue").
   probation("David Smith").

goal
   honor_student(X).
```

There is one thing to note when using ***not***: ***The not predicate succeeds when the subgoal can't be proven true.*** This results in a situation that prevents unbound variables from being bound within a ***not***. When a subgoal with free variables is called from within ***not***, Visual Prolog will return the error message `Free variables not allowed in 'not' or 'retractall'`. This happens because, for

Prolog to bind the free variables in a subgoal, that subgoal must unify with some other clause and the subgoal must succeed. The correct way to handle unbound variables within a ***not*** subgoal is with anonymous variables.

Here are some examples of correct clauses and incorrect clauses.

```
likes(bill, Anyone):-              /* 'Anyone' is an output argument */
   likes(sue, Anyone),
   not(hates(bill, Anyone).
```

In this example, *Anyone* is bound by `likes(sue, Anyone)` before Visual Prolog finds out that `hates(bill, Anyone)` is not true. This clause works just as it should.

If you rewrite this so that it calls ***not*** first, you will get an error message to the effect that free variables are not allowed in ***not***.

```
likes(bill, Anyone):-                      /* This won't work right */
   not(hates(bill, Anyone)),
   likes(sue, Anyone).
```

Even if you correct this (by replacing *Anyone* in `not(hates(bill, Anyone))` with an anonymous variable) so that the clause does not return the error, it will still return the wrong result.

```
likes(bill, Anyone):-                      /* This won't work right */
   not(hates(bill, _)),
   likes(sue, Anyone).
```

This clause states that Bill likes *Anyone* if nothing that Bill hates is known and if Sue likes *Anyone*. The original clause stated that Bill likes *Anyone* if there is some *Anyone* that Sue likes and that Bill does not hate.

Example

Always be sure that you think twice when using the ***not*** predicate. Incorrect use will result in an error message or errors in your program's logic. The following is an example of the proper way to use the ***not*** predicate.

```
                        /* Program ch04e11.pro */

PREDICATES
   likes_shopping(symbol)
   has_credit_card(symbol,symbol)
   bottomed_out(symbol,symbol)
```

```
CLAUSES
   likes_shopping(Who):-
      has_credit_card(Who,Card),
      not(bottomed_out(Who,Card)),
      write(Who," can shop with the ",Card, " credit card.\n").

   has_credit_card(chris,visa).
   has_credit_card(chris,diners).
   has_credit_card(joe,shell).
   has_credit_card(sam,mastercard).
   has_credit_card(sam,citibank).

   bottomed_out(chris,diners).
   bottomed_out(sam,mastercard).
   bottomed_out(chris,visa).

goal
   likes_shopping(Who).
```

Exercises

1. Suppose an average taxpayer in the USA is a married US citizen with two children who earns no less than $500 a month and no more than $2,000 per month. Define a ***special_taxpayer*** predicate that, given the goal `special_taxpayer(fred)`, will succeed only if *fred* fails one of the conditions for an average taxpayer. Use the cut to ensure that there is no unnecessary backtracking.

2. Players in a certain squash club are divided into three leagues, and players may only challenge members in their own league or the league below (if there is one).

   Write a Visual Prolog program that will display all possible matches between club players in the form:

   ```
   tom versus bill
   marjory versus annette
   ```

   Use the cut to ensure, for example, that

   ```
   tom versus bill
   ```

   and

   ```
   bill versus tom
   ```

   are not both displayed.

3. This is an exercise in backtracking, not a test of your ability to solve murder mysteries. Load and run with the Test Goal the following program.

   (**Note:** Bert is guilty because he has a motive and is smeared in the same stuff as the victim.)

```
                    /* Program ch04e12.pro */

DOMAINS
   name,sex,occupation,object,vice,substance = symbol
   age=integer

PREDICATES
   person(name, age, sex, occupation)
   had_affair(name, name)
   killed_with(name, object)
   killed(name)
   killer(name)
   motive(vice)
   smeared_in(name, substance)
   owns(name, object)
   operates_identically(object, object)
   owns_probably(name, object)
   suspect(name)

/* * * Facts about the murder * * */
CLAUSES
   person(bert,55,m,carpenter).
   person(allan,25,m,football_player).
   person(allan,25,m,butcher).
   person(john,25,m,pickpocket).

   had_affair(barbara,john).
   had_affair(barbara,bert).
   had_affair(susan,john).

   killed_with(susan,club).
   killed(susan).

   motive(money).
   motive(jealousy).
   motive(righteousness).

   smeared_in(bert, blood).
   smeared_in(susan, blood).
   smeared_in(allan, mud).
   smeared_in(john, chocolate).
   smeared_in(barbara,chocolate).
```

```
   owns(bert,wooden_leg).
   owns(john,pistol).

/* * * Background knowledge * * */
   operates_identically(wooden_leg, club).
   operates_identically(bar, club).
   operates_identically(pair_of_scissors, knife).
   operates_identically(football_boot, club).

   owns_probably(X,football_boot):-
      person(X,_,_,football_player).
   owns_probably(X,pair_of_scissors):-
      person(X,_,_,hairdresser).
   owns_probably(X,Object):-
      owns(X,Object).

/* * * * * * * * * * * * * * * * * * * * *
 * Suspect all those who own a weapon with   *
 * which Susan could have been killed.       *
 * * * * * * * * * * * * * * * * * * * * * */

   suspect(X):-
      killed_with(susan,Weapon) ,
      operates_identically(Object,Weapon) ,
      owns_probably(X,Object).

/* * * * * * * * * * * * * * * * * * * * * * * *
 * Suspect men who have had an affair with Susan.  *
 * * * * * * * * * * * * * * * * * * * * * * * * */

   suspect(X):-
      motive(jealousy),
      person(X,_,m,_),
      had_affair(susan,X).

/* * * * * * * * * * * * * * * * * * *
 * Suspect females who have had an       *
 * affair with someone that Susan knew.  *
 * * * * * * * * * * * * * * * * * * * */

   suspect(X):-
      motive(jealousy),
      person(X,_,f,_),
      had_affair(X,Man),
      had_affair(susan,Man).
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Suspect pickpockets whose motive could be money.  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * */
   suspect(X):-
      motive(money),
      person(X,_,_,pickpocket).

   killer(Killer):-
      person(Killer,_,_,_),
      killed(Killed),
      Killed <> Killer, /* It is not a suicide */
      suspect(Killer),
      smeared_in(Killer,Goo),
      smeared_in(Killed,Goo).

goal
   killer(X).
```

Prolog from a Procedural Perspective

Now that you've read chapters **2**, **3**, and the first three parts of this chapter, you should have a pretty good understanding of the basics of Prolog programming and using Visual Prolog. Remember, Prolog is a declarative language, which means that you describe a problem in terms of facts and rules and let the computer figure out how to find a solution. Other programming languages – such as Pascal, BASIC, and C – are procedural, which means that you must write subroutines and functions that tell the computer exactly what steps to go through in order to solve the problem.

We're going to back up now and review of some of the material you've just learned about Prolog, but this time we're going to present it from a procedural perspective.

How Rules and Facts Are Like Procedures

It's easy to think of a Prolog rule as a procedure definition. For instance, the rule

```
likes(bill,Something):- likes(cindy,Something).
```

means,

"To prove that Bill likes something, prove that Cindy likes it."

With this in mind, you can see how procedures like

```
say_hello:- write("Hello"), nl.
```

and

```
greet:-
   write("Hello, Earthlings!"),
   nl.
```

correspond to subroutines and functions in other programming languages.

You can even think of Prolog facts of as procedures; for instance, the fact

```
likes(bill, pasta).
```

means

> "To prove that Bill likes pasta, do nothing – and by the way, if the arguments *Who* and *What* in your query `likes(Who, What)` are free variables, you can bind them to *bill* and *pasta*, respectively."

Some programming procedures that you might be familiar with from other languages are *case statements*, *boolean tests*, *GoTo statements*, and *computational returns*. In the next sections, by reiterating what we've already covered from a different (procedural) point of view, we'll show you how Prolog rules can perform these same functions.

Using Rules Like Case Statements

One big difference between rules in Prolog and procedures in other languages is that Prolog allows you to give multiple alternative definitions of the same procedure. This came up with the "parent" program earlier on page 34; a person can be a parent by being a father or by being a mother, so the definition of "parent" is made up of two rules.

You can use multiple definitions like you use a Pascal **case** statement by writing a different definition for each argument value (or set of argument values). Prolog will try one rule after another until it finds a rule that matches, then perform the actions that rule specifies, as in Program `ch04e13.pro`.

```
                        /* Program ch04e13.pro */

PREDICATES
   action(integer)
```

```
CLAUSES
   action(1):-
      nl,
      write("You typed 1."),nl.
   action(2):-
      nl,
      write("You typed two."),nl.
   action(3):-
      nl,
      write("Three was what you typed."),nl.
   action(N):-
      nl,
      N<>1, N<>2, N<>3,
      write("I don't know that number!"),nl.

GOAL
      write("Type a number from 1 to 3: "),
      readint(Choice),
      action(Choice).
```

If the user types 1, 2, or 3, ***action*** will be called with its argument bound to the appropriate value, and it will match only one of the first three rules.

Performing Tests within the Rule

Look more closely at the fourth clause for ***action***. It will match whatever argument it's called with, binding *X* to that value. So you have to make sure that it doesn't print `I don't know that number` unless the number is indeed out of range. That's the purpose of the subgoals

```
X<>1, X<>2, X<>3
```

where <> means *not equal*. In order to print `I don't know that number`, Prolog must first prove that *X* is not 1, 2, or 3. If any of these subgoals fail, Prolog will try to back up and find alternatives – but there aren't any alternatives, so the rest of the clause will never be executed.

Notice that ***action*** relies on *Choice* being bound. If you call ***action*** with a free variable as an argument, the compiler would raise an error.

The *cut* as a GoTo

Program `ch04e13.pro` is somewhat wasteful because, after choosing and executing the correct rule, Prolog still keeps looking for alternatives and has to find out the hard way that the last rule doesn't apply.

It would save time and memory if you could tell Prolog to stop looking for alternatives. And you can, by using the **cut**, which means,

> "If you get this far, don't do any backtracking within this rule, and don't look for any alternatives to this rule."

In other words, "Burn your bridges behind you." Backtracking is still possible, but only at a higher level. If the current rule was called by another rule, and the higher rule has alternatives, they can still be tried. But the **cut** rules out alternatives within, and alternatives to, the present rule.

Using cuts, the program can be rewritten as follows:

```
                        /* Program ch04e14.pro */

PREDICATES
   action(integer)

CLAUSES
   action(1):-!,
      nl,
      write("You typed 1.").
   action(2):-!,
      nl,
      write("You typed two.").
   action(3):-!,
      nl,
      write("Three was what you typed.").
   action(_):-
      write("I don't know that number!").

GOAL
      write("Type a number from 1 to 3: "),
      readint(Num),
      action(Num),nl.
```

The ***cut*** has no effect unless it is actually executed. That is, in order to perform a cut, Prolog must actually get into the rule containing the ***cut*** and reach the point where the ***cut*** is located.

The ***cut*** can be preceded by other tests, like this:

```
   action(X) :- X>3, !, write("Too high.").
```

In this rule, the ***cut*** won't have any effect unless the subgoal `X>3` succeeds first.

Notice that the order of the rules is now significant. In `ch04e13.pro`, you could have written the rules in any order; only one of them will match any particular

number. But in Program `ch04e14.pro` you must make sure that the computer doesn't even try the rule that prints `I don't know that number` unless all of the preceding rules have been tried (and have not executed their cuts).

The cuts in `ch04e14.pro` are what some people call *red cuts* – cuts that change the logic of the program. If you had kept the tests `X<>1`, `X<>2`, and `X<>3`, changing the program only by inserting a ***cut*** in each clause, you would have been using *green cuts* – cuts that save time in a program that would be equally correct without them. The efficiency gained is not as great, but there is less risk of making an error in the program.

The ***cut*** is a powerful, but messy, Prolog operation. In this respect it resembles the **GoTo** statement in other programming languages – you can do many things with it, but it can make your program really hard to understand.

Returning Computed Values

As we have seen, a Prolog rule or fact can return information to the goal that called it. This is done by binding arguments that were previously unbound. The fact

```
likes(bill, cindy).
```

returns information to the goal

```
likes(bill, Who).
```

by binding *Who* to *cindy*.

A rule can return the results of a computation the same way. Here's a simple example:

```
                        /* Program ch04e15.pro */

PREDICATES
   classify(integer,symbol)

CLAUSES
   classify(0,zero).
   classify(X,negative):-
      X < 0.
   classify(X,positive):-
      X > 0.
```

The first argument of ***classify*** must always be either a constant or a bound variable. The second argument can be either bound or unbound; it gets matched

with the symbol *zero*, *negative*, or *positive*, depending on the value of the first argument.

Here are some examples of how rules can return values:

1. You can ask (using the Test Goal) whether 45 is positive by giving the goal:

```
Goal classify(45, positive).

yes
```

   Because 45 is greater than 0, only the third clause of ***classify*** can succeed. In doing so, it matches the second argument with *positive*. But the second argument is already *positive*, so the match succeeds, and you get the answer `yes`.

2. Conversely, if the match fails, you get `no`:

```
Goal classify(45, negative).

no
```

   What happens is this:

- Prolog tries the first clause, but the first argument won't match 0 (nor does the second argument match zero).
- Then it tries the second clause, binding X to 45, but the test X<0 fails.
- So it backs out and tries the third clause, but this time the second arguments don't match.

3. To get an actual answer, rather than just `yes` or `no`, you must call ***classify*** with the second argument free:

```
Goal classify(45, What).

What=positive
1 Solution
```

   Here's what really takes place in this case:

   a. The goal `classify(45, What)` won't match the head of the first clause, `classify(0, zero)`, because 45 doesn't match 0. So the first clause can't be used.

   b. Again, the goal `classify(45, What)` matches the head of the second clause, `classify(X, negative)`, binding *X* to 45 and *negative* to *What*. But then the subgoal `X<0,` fails, because *X* is 45 and it is not true that

45<0. So Prolog backs out of this clause, undoing the variable bindings just created.

c. Finally, `classify(45, What)` matches `classify(X, positive)`, binding *X* to `45` and *What* to `positive`. The test `X>0` succeeds. Since this is a successful solution, Prolog doesn't backtrack; it returns to the calling procedure (which in this case is the goal that you typed). And since the variable *X* belongs to the calling procedure, that procedure can use its binding – in this case, to print out the value automatically.

Summary

In this chapter we've introduced unification, backtracking, determinism, the predicates ***not*** and ***fail***, and the ***cut*** (!), and we've reviewed the important parts of the tutorial information up to this point from a procedural perspective.

1. Prolog facts and rules receive information by being called with arguments that are constants or bound variables; they return information to the calling procedure by binding variable arguments that were unbound.
2. *Unification* is the process of matching two predicates and assigning free variables to make the predicates identical. This mechanism is necessary so Prolog can identify which clauses to call and bind values to variables. These are the major points about matching (unification) presented in this chapter:
   a. When Prolog begins an attempt to satisfy a goal, it starts at the top of the program in search of a match.
   b. When a new call is made, a search for a match to that call also begins at the top of the program.
   c. When a call has found a successful match, the call is said to *return*, and the next subgoal in turn can be tried.
   d. Once a variable has been bound in a clause, the only way to free that binding is through backtracking.
3. *Backtracking* is the mechanism that instructs Prolog where to go to look for solutions to the program. This process gives Prolog the ability to search through all known facts and rules for a solution. These are the four basic principles of backtracking given in this chapter:
   a. Subgoals must be satisfied in order, from top to bottom.
   b. Predicate clauses are tested in the order they appear in the program, from top to bottom.

c. When a subgoal matches the head of a rule, the body of that rule must be satisfied next. The body of the rule then constitutes a new set of subgoals to be satisfied.

d. A goal has been satisfied when a matching fact is found for each of the extremities (leaves) of the goal tree.

4. A call that can produce multiple solutions is *non-deterministic*, while a call that can produce one and only one solution is *deterministic*.

5. Visual Prolog provides three tools for controlling the course of your program's logical search for solutions: these are the two predicates ***fail*** and ***not***, and the ***cut***.

- The **fail** predicate always fails; it forces backtracking in order to find alternate solutions.
- The **not** predicate succeeds when its associated subgoal can't be proven true.
- The **cut** prevents backtracking.

6. It's easy to think of a Prolog rule as a procedure definition. From a procedural perspective, rules can function as **case** statements, perform boolean tests, act like **GoTo** statements (using the **cut**), and return computed values.

CHAPTER 5

Simple and Compound Objects

So far, we've only shown you a few kinds of Visual Prolog data objects, such as numbers, symbols, and strings. In this chapter we discuss the whole range of data objects that Visual Prolog can create, from simple to compound objects.

We also show the different types of data structures and data objects that a Visual Prolog program can contain. Because the standard domains do not cover some of the compound data structures, we explain how to declare these compound data structures in both the **domains** section and the **predicates** section of your programs.

Simple Data Objects

A simple data object is either a variable or a constant. Don't confuse this use of the word "constant" with the symbolic constants you define in the **constants** section of a program. What we mean here by a constant, is anything identifying an object not subject to variation, such as a character (a ***char***), a number (an integral value or a ***real***), or an atom (a ***symbol*** or ***string***).

Variables as Data Objects

Variables, which we've discussed in chapter **2**, must begin with an upper-case letter (*A*-*Z*) or an underscore (_). A single underscore represents an anonymous variable, which stands for a "don't care what it is" situation. In Prolog, a variable can bind with any legal Prolog argument or data object.

Prolog variables are local, not global. That is, if two clauses each contain a variable called *X*, these *X*s are two distinct variables. They may get bound to each other if they happen to be brought together during unification, but ordinarily they have no effect on each other.

Constants as Data Objects

Constants include characters, numbers, and atoms. Again, don't confuse constants in this context with the symbolic constants defined in the **constants**

section of a program. A constant's value is its name. That is, the constant 2 can only stand for the number 2, and the constant `abracadabra` can only stand for the symbol *abracadabra*.

Characters

Characters are ***char*** type. The printable characters (ASCII 32-127) are the digits 0-9, upper-case letters *A-Z*, lower-case letters *a-z*, and the punctuation and familiar TTY characters. Characters outside this range may not be portable between different platforms; in particular, characters less than ASCII 32 (space) are control characters, traditionally used by terminals and communication equipment.

A character constant is simply written as the character you want, enclosed by single quotes:

```
'a'          '3'
'*'          '{'
'W'          'A'
```

If, however, you want to specify a backslash or a single quote itself as the character, precede it by a backslash (\):

`'\\'` backslash `'\''` single quote.

There are a few characters that perform a special function, when preceded by the escape character:

`'\n'`	Newline (linefeed)
`'\r'`	Carriage return.
`'\t'`	Tab (horizontal)

Character constants can also be written as their ASCII codes, preceded by the escape character, like this:

```
'\225'       ß
'\3'         %]
```

but the exact character displayed by more exotic ASCII values will vary depending on your video-card/terminal.

Numbers

Numbers are either from one of the integral domains (see Table 3.1 on page 51), or the ***real*** domain. Real numbers are stored in the IEEE standard format and range from 1*e*-308 to 1*e*308 (10^{-308} to 10^{+308}). Examples are:

Integers	*Real Numbers*
3	3.
-77	34.96
32034	-32769
-10	4e27
0	-7.4e-296

Atoms

An atom is either a ***symbol*** or a ***string***. The distinction between these is largely a question about machine-representation and implementation, and is generally not syntactically visible. When an atom is used as an argument in a predicate call, it is the declaration for the predicate that determines if that argument should be implemented as a ***string*** or a ***symbol***.

Visual Prolog performs an automatic type conversion between the ***string*** domain and the ***symbol*** domain, so you can use ***symbol*** atoms for ***string*** domains and ***string*** atoms for the ***symbol*** domains. However, there is a loose convention stating that anything in double quotes should be considered a ***string***, while anything not needing to be quoted to be syntactically valid is a ***symbol***:

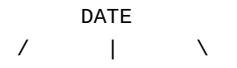
- **Symbol** atoms are names starting with a lower-case letter, and containing only letters, digits, and underscores.
- **String** atoms are bound within double quotes and can contain any combination of characters, except ASCII NULL (0, binary zero), which marks the end of the string.

Symbol Atoms	**String Atoms**
`food`	`"Jesse James"`
`rick_Jones_2nd`	`"123 Pike street"`
`fred_Flintstone_1000_Bc_Rd_Bedrock`	`"jon"`
`a`	`"a"`
`new_york`	`"New York"`
`pdcProlog`	`"Visual Prolog, by Prolog Development Center"`

As far as the ***string***/***symbol*** domain interchangeability goes, this distinction is not important. However, things such as predicate names and functors for compound objects (introduced below) must follow the syntactic conventions for ***symbol***s.

Compound Data Objects and Functors

Compound data objects allow you to treat several pieces of information as a single item in such a way that you can easily pick them apart again. Consider, for instance, the date April 2, 1988. It consists of three pieces of information – the month, day, and year – but it's useful to treat the whole thing as a single object with a treelike structure:



You can do this by declaring a domain containing the compound object ***date***:

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

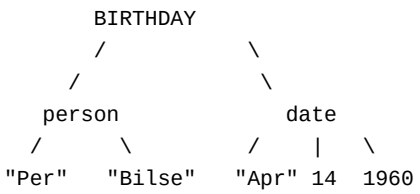
and then simply writing e.g.

```
   ..., D = date("October",15,1991), ...
```

This looks like a Prolog fact, but it isn't here – it's just a data object, which you can handle in much the same way as a symbol or number. It begins with a name, usually called a *functor* (in this case ***date***), followed by three arguments.

Note carefully that a functor in Visual Prolog has nothing to do with a function in other programming languages. ***A functor does not stand for some computation to be performed.*** It's just a name that identifies a kind of compound data object and holds its arguments together.

The arguments of a compound data object can themselves be compound. For instance, you might think of someone's birthday as an information structure like this:



In Prolog you would write this as:

```
   birthday(person("Per","Bilse"),date("Apr",14,1960))
```

In this example, there are two parts to the compound object ***birthday***: the object `person("Per", "Bilse")` and the object `date("Apr", 14, 1960)`. The functors of these data objects are ***person*** and ***date***.

Unification of Compound Objects

A compound object can unify either with a simple variable or with a compound object that matches it (perhaps containing variables as parts of its internal structure). This means you can use a compound object to pass a whole collection of items as a single object, and then use unification to pick them apart. For example,

```
date("April",14,1960)
```

matches *X* and binds *X* to `date("April",14,1960)`.

Also

```
date("April",14,1960)
```

matches `date(Mo,Da,Yr)` and binds *Mo* to `"April"`, *Da* to `14`, and *Yr* to `1960`.

Some examples of programming with compound objects follow in the next sections.

Using the Equal Sign to Unify Compound Objects

Visual Prolog performs unification in two places. The first is when a call or goal matches the head of a clause. The second is the across the ***equal*** (=) sign, which is actually an *infix predicate* (a predicate that is located *between* its arguments rather than *before* them).

Visual Prolog will make the necessary bindings to unify the objects on both sides of the ***equal*** sign. This is useful for finding the values of arguments within a compound object. For example, the following code excerpt tests if two people have the same last name, then gives the second person the same address as the first.

```
                        /* Program ch05e01.pro */

DOMAINS
   person                 = person(name,address)
   name                   = name(first,last)
   address                = addr(street,city,state)
   street                 = street(number,street_name)
   city,state,street_name = string
   first,last             = string
   number                 = integer

GOAL
      P1 = person(name(jim,mos),addr(street(5,"1st st"),igo,"CA")),
      P1 = person(name(_,mos),Address),
      P2 = person(name(jane,mos),Address),
      write("P1=",P1),nl,
      write("P2=",P2),nl.
```

Treating Several Items as One

Compound objects can be regarded and treated as single objects in your Prolog clauses, which greatly simplifies programming. Consider, for example, the fact

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

in which you state that John owns the book *From Here to Eternity*, written by James Jones. Likewise, you could write

```
owns(john, horse(blacky)).
```

which can be interpreted as

```
John owns a horse named blacky.
```

The compound objects in these two examples are

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

and

```
horse(blacky)
```

If you had instead written two facts:

```
owns(john, "From Here to Eternity").
owns(john, blacky ).
```

you would not have been able to decide whether `blacky` was the title of a book or the name of a horse. On the other hand, you can use the first component of a compound object – the functor – to distinguish between different objects. This example used the functors ***book*** and ***horse*** to indicate the difference between the objects.

Remember: Compound objects consist of a functor and the objects belonging to that functor, as follows:

```
functor(object1, object2, ..., objectN)
```

An Example Using Compound Objects

An important feature of compound objects allows you to easily pass a group of values as one argument. Consider a case where you are keeping a telephone database. In your database, you want to include your friends' and family members' birthdays. Here is a section of code you might have come up with:

```
PREDICATES
   phone_list(symbol, symbol, symbol, symbol, integer, integer)
         /* ( First,   Last,  Phone,  Month,     Day,    Year) */

CLAUSES
   phone_list(ed, willis, 422-0208, aug, 3, 1955).
   phone_list(chris, grahm, 433-9906, may, 12, 1962).
```

Examine the data, noticing the six arguments in the fact ***phone_list***; five of these arguments can be broken down into two compound objects, like this:

```
          person                    birthday
         /     \                  /     |     \
   First Name  Last Name         Month  Day  Year
```

It might be more useful to represent your facts so that they reflect these compound data objects. Going back a step, you can see that ***person*** is a relationship, and the first and last names are the objects. Also, ***birthday*** is a relationship with three arguments: month, day, and year. The Prolog representation of these relationships is

```
person(First_name, Last_name)
birthday(Month, Day, Year)
```

You can now rewrite your small database to include these compound objects as part of your database.

```
DOMAINS
   name = person(symbol, symbol)                        /* (First, Last) */
   birthday = b_date(symbol, integer, integer)    /* (Month, Day, Year) */
   ph_num = symbol                                       /* Phone_number */

PREDICATES
   phone_list(name, ph_num, birthday)

CLAUSES
   phone_list(person(ed, willis), "422-0208", b_date(aug, 3, 1955)).
   phone_list(person(chris, grahm), "433-9906", b_date(may, 12, 1962)).
```

In this program, two compound **domains** declarations were introduced. We go into more detail about these compound data structures later in this chapter. For now, we'll concentrate on the benefits of using such compound objects.

The ***phone_list*** predicate now contains three arguments, as opposed to the previous six. Sometimes breaking up your data into compound objects will clarify your program and might help process the data.

Now add some rules to your small program. Suppose you want to create a list of people whose birthdays are in the current month. Here's the program code to accomplish this task; this program uses the standard predicate ***date*** to get the current date from the computer's internal clock. The ***date*** predicate is discussed later in chapter **15**. For now, all you need to know is that it will return the current year, month, and day from your computer's clock.

```
                        /* Program ch05e03.pro */

DOMAINS
   name = person(symbol,symbol)                         /* (First, Last) */
   birthday = b_date(symbol,integer,integer)      /* (Month, Day, Year) */
   ph_num = symbol                                       /* Phone_number */

PREDICATES
   phone_list(name,ph_num,birthday)
   get_months_birthdays()
   convert_month(symbol,integer)
   check_birthday_month(integer,birthday)
   write_person(name)
```

```
CLAUSES
   get_months_birthdays:-
      write("************ This Month's Birthday List *************"),nl,
      write(" First name\t\t Last Name\n"),
      write("************************************************"),nl,
      date(_, This_month, _),            /* Get month from system clock */
      phone_list(Person, _, Date),
      check_birthday_month(This_month, Date),
      write_person(Person),
      fail.

   get_months_birthdays:-
      write("\n\n Press any key to continue: "),nl,
      readchar(_).

   write_person(person(First_name,Last_name)):-
      write("  ",First_name,"\t\t   ",Last_name),nl.

   check_birthday_month(Mon,b_date(Month,_,_)):-
      convert_month(Month,Month1),
      Mon = Month1.

   phone_list(person(ed, willis), "767-8463", b_date(jan, 3, 1955)).
   phone_list(person(benjamin, thomas), "438-8400", b_date(feb, 5, 1985)).
   phone_list(person(ray, william), "555-5653", b_date(mar, 3, 1935)).
   phone_list(person(thomas, alfred), "767-2223", b_date(apr, 29, 1951)).
   phone_list(person(chris, grahm), "555-1212", b_date(may, 12, 1962)).
   phone_list(person(dustin, robert), "438-8400", b_date(jun, 17, 1980)).
   phone_list(person(anna, friend), "767-8463", b_date(jun, 20, 1986)).
   phone_list(person(brandy, rae), "555-5653", b_date(jul, 16, 1981)).
   phone_list(person(naomi, friend), "767-2223", b_date(aug, 10, 1981)).
   phone_list(person(christina, lynn), "438-8400", b_date(sep, 25, 1981)).
   phone_list(person(kathy, ann), "438-8400", b_date(oct, 20, 1952)).
   phone_list(person(elizabeth, ann), "555-1212", b_date(nov, 9, 1984)).
   phone_list(person(aaron, friend), "767-2223", b_date(nov, 15, 1987)).
   phone_list(person(jennifer, caitlin), "438-8400", b_date(dec, 31,
              1981)).
```

```
   convert_month(jan, 1).
   convert_month(feb, 2).
   convert_month(mar, 3).
   convert_month(apr, 4).
   convert_month(may, 5).
   convert_month(jun, 6).
   convert_month(jul, 7).
   convert_month(aug, 8).
   convert_month(sep, 9).
   convert_month(oct, 10).
   convert_month(nov, 11).
   convert_month(dec, 12).

GOAL
   get_months_birthdays().
```

Load and run the Test Goal with this program.

How do compound data objects help in this program? This should be easy to see when you examine the code. Most of the processing goes on in the ***get_months_birthdays*** predicate.

1. First, the program makes a window to display the results.
2. After this, it writes a header in the window to help interpret the results.
3. Next, in ***get_months_birthdays***, the program uses the built-in predicate ***date*** to obtain the current month.
4. After this, the program is all set to search the database and list the people who were born in the current month. The first thing to do is find the first person in the database. The call `phone_list(Person, _, Date)` binds the person's first and last names to the variable *Person* by binding the entire functor ***person*** to *Person*. It also binds the person's birthday to the variable *Date*.

   Notice that you only need to use one variable to store a person's complete name, and one variable to hold the birthday. This is the power of using compound data objects.
5. Your program can now pass around a person's birthday simply by passing on the variable *Date*. This happens in the next subgoal, where the program passes the current month (represented by an integer) and the birthday (of the person it's processing) to the predicate ***check_birthday_month***.
6. Look closely at what happens. Visual Prolog calls the predicate ***check_birthday_month*** with two variables: The first variable is bound to an integer, and the second is bound to a ***birthday*** term. In the head of the rule

that defines ***check_birthday_month***, the first argument, *This_month*, is matched with the variable *Mon*. The second argument, *Date*, is matched against `b_date(Month, _,_)`.

Since all you're concerned with is the month of a person's birthday, you have used the anonymous variable for both the day and the year of birth.

7. The predicate ***check_birthday_month*** first converts the symbol for the month into an integer value. Once this is done, Visual Prolog can compare the value of the current month with the value of the person's birthday month. If this comparison succeeds, then the subgoal ***check_birthday_month*** succeeds, and processing can continue. If the comparison fails (the person currently being processed was not born in the current month), Visual Prolog begins to backtrack to look for another solution to the problem.
8. The next subgoal to process is ***write_person***. The person currently being processed has a birthday this month, so it's OK to print that person's name in the report. After printing the information, the clause fails, which forces backtracking.
9. ***Backtracking always goes up to the most recent non-deterministic call and tries to re-satisfy that call.*** In this program, the last non-deterministic call processed is the call to ***phone_list***. It is here that the program looks up another person to be processed. If there are no more people in the database to process, the current clause fails; Visual Prolog then attempts to satisfy this call by looking further down in the database. Since there is another clause that defines ***get_months_birthdays***, Visual Prolog tries to satisfy the call to ***get_months_birthdays*** by satisfying the subgoals to this other clause.

Exercise

Modify the previous program so that it will also print the birth dates of the people listed. Next, add telephone numbers to the report.

Declaring Domains of Compound Objects

In this section, we show you how domains for compound objects are defined. After compiling a program that contains the following relationships:

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

and

```
owns(john, horse(blacky)).
```

you could query the system with this goal:

```
owns(john, X)
```

The variable *X* can be bound to different types of objects: a book, a horse, or perhaps other objects you define. Because of your definition of the ***owns*** predicate, you can no longer employ the old predicate declaration of ***owns***:

```
owns(symbol, symbol)
```

The second argument no longer refers to objects belonging to the domain ***symbol***. Instead, you must formulate a new declaration to the predicate, such as

```
owns(name, articles)
```

You can describe the ***articles*** domain in the **domains** section as shown here:

```
DOMAINS
   articles = book(title,author); horse(name)
                                        /* Articles are books or horses */
   title, author, name = symbol
```

The semicolon is read as *or*. In this case, two alternatives are possible: A book can be identified by its title and author, or a horse can be identified by its name. The domains ***title***, ***author***, and ***name*** are all of the standard domain **symbol**.

More alternatives can easily be added to the **domains** declaration. For example, ***articles*** could also include a boat, a house, or a bankbook. For a boat, you can make do with a functor that has no arguments attached to it. On the other hand, you might want to give a bank balance as a figure within the bankbook. The **domains** declaration of ***articles*** is therefore extended to:

```
   articles       = book(title, author) ; horse(name) ;
                    boat ; bankbook(balance)
   title, author, name = symbol
   balance      = real
```

Here is a full program that shows how compound objects from the domain ***articles*** can be used in facts that define the predicate ***owns***.

```
                        /* Program ch05e04.pro */

DOMAINS
   articles             = book(title, author) ;
                          horse(name) ; boat ;
                          bankbook(balance)
   title, author, name  = symbol
   balance              = real

PREDICATES
   owns(name,articles)

CLAUSES
   owns(john, book("A friend of the family", "Irwin Shaw")).
   owns(john, horse(blacky)).
   owns(john, boat).
   owns(john, bankbook(1000)).

goal
   owns(john, Thing).
```

Now load the program into Visual Development Environment and run the Test Goal.

Visual Prolog (the Test Goal) responds with:

```
   Thing=book("A friend of the family","Irwin Shaw")
   Thing=horse("blacky")
   Thing=boat
   Thing=bankbook(1000)

   4 Solutions
```

Writing Domain Declarations: a Summary

This is a generic representation of how to write domain declarations for compound objects:

```
   domain =alternative1(D, D, ...);
          alternative2(D, D, ...);
          ...
```

Here, ***alternative1*** and ***alternative2*** are arbitrary (but different) functors. The notation `(D, D, ...)` represents a list of domain names that are either declared elsewhere or are one of the standard domain types (such as ***symbol***, ***integer***, ***real***, etc).

Note:

1. The alternatives are separated by semicolons.
2. Every alternative consists of a functor and, possibly, a list of domains for the corresponding arguments.
3. If the functor has no arguments, you can write it as `alternativeN` or `alternativeN( )` in your programs. In this book, we use the former syntax.

Multi-Level Compound Objects

Visual Prolog allows you to construct compound objects on several levels. For example, in

```
book("The Ugly Duckling", "Andersen")
```

instead of using the author's last name, you could use a new structure that describes the author in more detail, including both the author's first and last names. By calling the functor for the resulting new compound object ***author***, you can change the description of the book to

```
book("The Ugly Duckling", author("Hans Christian", "Andersen"))
```

In the old domain declaration

```
book(title, author)
```

the second argument of the ***book*** functor is *author*. But the old declaration

```
author = symbol
```

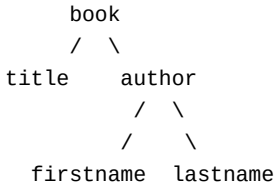
can only include a single name, so it's no longer sufficient. You must now specify that an author is also a compound object made up of the author's first and last name. You do this with the domain statement:

```
author = author(first_name, last_name)
```

which leads to the following declarations:

```
DOMAINS
   articles              = book(title, author); ..       /* First level */
   author                = author(first_name, last_name)/* Second level */
   title, first_name, last_name = symbol                 /* Third level */
```

When using compound objects on different levels in this way, it's often helpful to draw a "tree":



A domain declaration describes only one level of the tree at a time, and not the whole tree. For instance, a book can't be defined with the following domain declaration:

```
book = book(title,author(first_name,last_name))       /* Not allowed */
```

An Example That Illustrates Sentence Structure

As another example, consider how to represent the grammatical structure of the sentence

```
ellen owns the book.
```

using a compound object. The most simple sentence structure consists of a noun and a verb phrase:

```
sentence = sentence(noun, verbphrase)
```

A noun is just a simple word:

```
noun = noun(word)
```

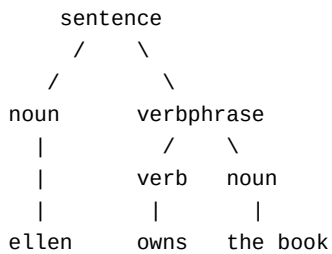
A verb phrase consists of either a verb with a noun phrase or a single verb.

```
verbphrase = verbphrase(verb, noun); verb(word)
verb     = verb(word)
```

Using these domain declarations (*sentence*, *noun*, *verbphrase*, and *verb*), the sentence `ellen owns the book.` becomes

```
sentence(noun(ellen), verbphrase(verb(owns), noun(book)))
```

The corresponding tree is



A data structure like this might be the output of a parser, which is a program that determines the grammatical structure of a sentence. Parsing is not built into Visual Prolog, but we have included a parser implementing simple sentence analysis with your Visual Prolog package. (Try to run the project VPI\PROGRAMS\SEN_AN when you're ready to tackle this subject.)

Exercises

1. Write a suitable **domains** declaration using compound objects that could be used in a Visual Prolog catalog of musical shows. A typical entry in the catalog might be

   ```
   Show: West Side Story
   Lyrics: Stephen Sondheim
   Music: Leonard Bernstein
   ```

2. Using compound objects wherever possible, write a Visual Prolog program to keep a database of United States senators. Entries should include the senator's first and last name, affiliation (state and party), size of constituency, date of election, and voting record on ten bills. Or, if you're not familiar with United States senators, use any political (or other) organization that you're familiar with.

Compound Mixed-Domain Declarations

In this section, we discuss three different types of domain declarations you can add to your programs. These declarations allow you to use predicates that

1. take an argument, more than one type of more than one possible type
2. take a variable number of arguments, each of a specified type
3. take a variable number of arguments, some of which might be of more than one possible type

Multiple-Type Arguments

To allow a Visual Prolog predicate to accept an argument that gives information of different types, you must add a functor declaration. In the following example, the ***your_age*** clause will accept an argument of type ***age***, which can be a ***string***, a ***real***, or an ***integer***.

```
domains
   age = i(integer); r(real); s(string)

predicates
   your_age(age)

clauses
   your_age(i(Age)) :- write(Age).
   your_age(r(Age)) :- write(Age).
   your_age(s(Age)) :- write(Age).
```

Visual Prolog does not allow the following domain declaration:

```
domains
   age = integer; real; string                        /* Not permitted. */
```

Lists

Suppose you are keeping track of the different classes a professor might teach. You might produce the following code:

```
PREDICATES
   teacher(symbol First_name, symbol Last_name, symbol Class)

CLAUSES
   teacher(ed, willis, english1).
   teacher(ed, willis, math1).
   teacher(ed, willis, history1).
   teacher(mary, maker, history2).
   teacher(mary, maker, math2).
   teacher(chris, grahm, geometry).
```

Here, you need to repeat the teacher's name for each class he or she teaches. For each class, you need to add another fact to the database. Although this is perfectly OK in this situation, you might find a school where there are hundreds of classes; this type of data structure would get a little tedious. Here, it would be helpful if you could create an argument to a predicate that could take on one *or more* values.

A list in Prolog does just that. In the following code, the argument *class* is declared to be of a *list* type. We show here how a list is represented in Prolog, but list-handling predicates are covered in chapter **7**.

```
DOMAINS
   classes = symbol*                          /* declare a list domain */

PREDICATES
   teacher(symbol First, symbol Last, classes Classes)

CLAUSES
   teacher(ed, willis, [english1, math1, history1]).
   teacher(mary, maker, [history2, math2]).
   teacher(chris, grahm, [geometry]).
```

In this example, the code is more concise and easier to read than in the preceding one. Notice the **domains** declaration:

```
DOMAINS
   classes = symbol*
```

The asterisk (*) means that *classes* is a list of symbols. You can just as easily declare a list of integers:

```
DOMAINS
   integer_list = integer*
```

Once you declare a domain, it's easy to use it; just place it as an argument to a predicate declared in the **predicates** section. Here's an example of using an integer list:

```
DOMAINS
   integer_list = integer*

PREDICATES
   test_scores(symbol First, symbol Last, integer_list Test_Scores)

CLAUSES
   test_scores(lisa, lavender, [86, 91, 75]).
   test_scores(libby, dazzner, [79, 75]).
   test_scores(jeff, zheutlin, []).
```

In the case of *Jeff Zheutlin*, notice that a list doesn't need to contain any elements at all.

Lists are discussed in greater detail in chapter **7**.

Summary

These are the important points covered in this chapter:

1. A Visual Prolog program can contain many types of data objects: simple and compound, standard and user-defined. A simple data object is one of the following:
   - a variable; such as X, MyVariable, _another_variable, or a single underscore (_) for an anonymous variable
   - a constant; a **char**, an **integer** or **real** number, or a **symbol** or **string** atom
2. *Compound data objects* allow you to treat several pieces of information as a single item. A compound data object consists of a name (known as a *functor*) and one or more arguments. You can define a domain with several alternative functors.
3. A *functor* in Visual Prolog is not the same thing as a function in other programming languages. ***A functor does not stand for some computation to be performed.*** It's just a name that identifies a kind of compound data object and holds its arguments together.
4. Compound objects can be regarded and treated as single objects; you use the functor to distinguish between different objects. Visual Prolog allows you to construct compound objects on several levels; the arguments of a compound data object can also be compound objects. With compound mixed domain declarations, you can use predicates that:
   - take an argument of more than one possible type (functor declaration).
   - take a variable number of arguments, each of a specified type (list declaration).
   - take a variable number of arguments, some of which might be of more than one possible type.

CHAPTER 6

Repetition and Recursion

Much of the usefulness of computers comes from the fact that they are good at doing the same thing over and over again. Prolog can express repetition both in its procedures and in its data structures. The idea of a repetitive data structure may sound strange, but Prolog allows you to create data structures whose ultimate size is not known at the time you create them. In this chapter, we discuss repetitive processes first (as loops and recursive procedures), then cover recursive data structures.

Repetitive Processes

Pascal, BASIC, or C programmers who start using Visual Prolog are often dismayed to find that the language has no FOR, WHILE, or REPEAT statements. There is no direct way to express iteration. Prolog allows only two kinds of repetition – backtracking, in which it searches for multiple solutions in a single query, and *recursion*, in which a procedure calls itself.

As it turns out, this lack doesn't restrict the power of the Prolog language. In fact, Visual Prolog recognizes a special case of recursion – called tail recursion – and compiles it into an iterative loop in machine language. This means that although the program logic is expressed recursively, the compiled code is as efficient as it would be in Pascal or BASIC.

In this section, we explore the art of writing repetitive processes in Prolog. As you'll see, recursion is – in most cases – clearer, more logical, and less error-prone than the loops that conventional languages use. Before delving into recursion, however, take another look at backtracking.

Backtracking Revisited

When a procedure backtracks, it looks for another solution to a goal that has already been satisfied. It does this by retreating to the most recent subgoal that has an untried alternative, using that alternative, then moving forward again. You can exploit backtracking as a way to perform repetitive processes.

Example

Program `ch06e01.pro` demonstrates how to use backtracking to perform repetitive processes – it prints all solutions to a query.

```
                        /* Program ch06e01.pro */

PREDICATES
   country(symbol)
   print_countries

CLAUSES
   country("England").
   country("France").
   country("Germany").
   country("Denmark").

   print_countries:-
      country(X),
      write(X),                                  /* write the value of X */
      nl,                                            /* start a new line */
      fail.
   print_countries.

goal
   print_countries.
```

The predicate ***country*** simply lists the names of various countries, so that a goal such as

```
   country(X)
```

has multiple solutions. The predicate ***print_countries*** then prints out all of these solutions. It is defined as follows:

```
   print_countries :-
      country(X), write(X), nl, fail.

   print_countries.
```

The first clause says:

> "To print countries, find a solution to `country(X)`, then write *X* and start a new line, then fail."

In this case, "fail" means:

> "assume that a solution to the original goal has not been reached, so back up and look for an alternative."

The built-in predicate ***fail*** always fails, but you could equally well force backtracking by using any other goal that would always fail, such as `5=2+2` or `country(shangri_la)`.

The first time through, *X* is bound to *england*, which is printed. Then, when it hits ***fail***, the computer backs up. There are no alternative ways to satisfy `nl` or `write(X)`, so the computer looks for a different solution to `country(X)`.

The last time `country(X)` was executed, it bound a value to the previously free variable *X*. So, before retrying this step, the computer unbinds *X* (frees it). Then it can look for an alternative solution for `country(X)` and bind *X* to a different value. If it succeeds, processing goes forward again and the name of another country is printed.

Eventually, the first clause runs out of alternatives. The only hope then is to try another clause for the same predicate. Sure enough, execution falls through to the second clause, which succeeds without doing anything further. In this way the goal `print_countries` terminates with success. Its complete output is

```
england
france
germany
denmark

yes
```

If the second clause were not there, the `print_countries` goal would terminate with failure, and the final message would be `no`. Apart from that, the output would be the same.

Exercise

Modify `ch06e01.pro` so that

1. ***country*** has two arguments, *name* and *population*, and
2. only those countries with populations greater than 10 million (1e+7) are printed

Pre- and Post-Actions

Typically, a program that retrieves all the solutions to a goal will also want to do something beforehand and afterward. For instance, your program could

1. Print `Some delightful places to live are....`
2. Print all solutions to `country(X)`

3. Close by printing `And maybe others.`

Note that ***print_countries***, as defined in the preceding example, already includes clauses that print all solutions to `country(X)` and close by (potentially) printing a final message.

The first clause for ***print_countries*** corresponds to step 2 and prints all the solutions; its second clause corresponds to step 3 and simply terminates the goal successfully (because the first clause always fails).

You could change the second clause in `ch06e01.pro` to

```
print_countries :- write("And maybe others."), nl.
```

which would implement step 3 as specified.

What about step 1? There's no reason why ***print_countries*** should have only two clauses. It can have three, like this:

```
print_countries :-
   write("Some delightful places to live are"),nl,
   fail.
print_countries :-
   country(X),
   write(X),nl,
   fail.
print_countries :-
   write("And maybe others."), nl.
```

The ***fail*** in the first clause is important – it ensures that, after executing the first clause, the computer backs up and tries the second clause. It's also important that the predicates ***write*** and ***nl*** do not generate alternatives; strictly speaking, the first clause tries all possible solutions before failing.

This three-clause structure is more of a trick than an established programming technique. A more fastidious programmer might try to do things this way:

```
print_countries_with_captions :-
   write("Some delightful places to live are"),nl,
   print_countries,
   write("And maybe others."),nl.
print_countries :-
   country(X),
   write(X),nl,
   fail.
```

There's nothing essentially wrong here, but this hypothetical fastidious programmer has made a mistake.

Exercise

Don't look ahead – figure out what's wrong with this program, and fix it!

You're right – the problem is that, as written in the latest example, ***print_countries*** will always fail, and ***print_countries_with_captions*** will never get to execute any of the subgoals that follow it. As a result, `And maybe others.` will never be printed.

To fix this, all you need to do is restore the original second clause for ***print_countries***.

```
   print_countries.
```

to its original position. If you want the goal ***print_countries_with_captions*** to succeed, it must have at least one clause that does not contain ***fail***.

Implementing Backtracking with Loops

Backtracking is a good way to get all the alternative solutions to a goal. But even if your goal doesn't have multiple solutions, you can still use backtracking to introduce repetition. Simply define the two-clause predicate

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

This tricks Prolog's control structure into thinking it has an infinite number of different solutions. (Never mind how – after reading about tail recursion, you'll see how this works.) The purpose of ***repeat*** is to allow backtracking *ad infinitum*.

```
                        /* Program ch06e02.pro */

   /* Uses repeat to keep accepting characters and printing them
      until the user presses Enter. */

PREDICATES
   repeat
   typewriter

CLAUSES
   repeat.
   repeat:-repeat.
```

```
   typewriter:-
      repeat,
      readchar(C),                          /* Read a char, bind C to it */
      write(C),
   C = '\r',!.                  /* Is it a carriage return? fail if not */

goal
   typewriter(),nl.
```

Program `ch06e02.pro` shows how ***repeat*** works. The rule `typewriter :- ...` describes a procedure that accepts characters from the keyboard and prints them on the screen until the user presses the **Enter** (**Return**) key.

typewriter works as follows:

1. Execute ***repeat*** (which does nothing).
2. Then read a character into the variable *C*.
3. Then write *C*.
4. Then check if *C* is a carriage return.
5. If so, you're finished. If not, backtrack and look for alternatives. Neither ***write*** nor ***readchar*** generates alternative solutions, so backtrack all the way to ***repeat***, which always has alternative solutions.
6. Now processing can go forward again, reading another character, printing it, and checking whether it's a carriage return.

Note, by the way, that *C* looses its binding when you backtrack past `readchar(C)`, which bound it. This kind of unbinding is vital when you use backtracking to obtain alternative solutions to a goal, but it makes it hard to use backtracking for any other purpose. The reason is that, although a backtracking process can repeat operations any number of times, it can't "remember" anything from one repetition to the next. ***All variables loose their values when execution backtracks over the steps that established those values.*** There is no simple way for a repeat loop to keep a counter, a total, or any other record of its progress.

Exercises

1. Modify 2 so that, if the user types lower-case letters, they will be displayed as upper-case.
2. If you'd like to play with file I/O now, look up the appropriate built-in predicates and write a program that uses a repeat loop to copy a file character-by-character. (Refer to chapter **12**.)

Recursive Procedures

The other way to express repetition is through recursion. A recursive procedure is one that calls itself. Recursive procedures have no trouble keeping records of their progress because counters, totals, and intermediate results can be passed from each iteration to the next as arguments.

The logic of recursion is easy to follow if you forget, for the moment, how computers work. (Prolog is so different from machine language that ignorance of computers is often an asset to the Prolog programmer.) Forget for the moment that the computer is trekking through memory addresses one by one, and imagine a machine that can follow recipes like this one:

```
To find the factorial of a number N:

   If N is 1, the factorial is 1.

   Otherwise, find the factorial of N-1, then multiply it by N.
```

This recipe says: To find the factorial of 3, you must find the factorial of 2, and, to find the factorial of 2, you must find the factorial of 1. Fortunately, you can find the factorial of 1 without referring to any other factorials, so the repetition doesn't go on forever. When you have the factorial of 1, you multiply it by 2 to get the factorial of 2, then multiply that by 3 to get the factorial of 3, and you're done.

In Visual Prolog:

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

factorial(X, FactX) :-
   Y = X-1,
   factorial(Y, FactY),
   FactX = X*FactY.
```

A complete program is as follows:

```
                        /* Program ch06e03.pro */

   /* Recursive program to compute factorials.
      Ordinary recursion, not tail recursion. */

PREDICATES
   factorial(unsigned,real)

CLAUSES
   factorial(1,1):-!.
```

```
   factorial(X,FactX):-
      Y=X-1,
      factorial(Y,FactY),
      FactX = X*FactY.

goal
   X=3,
   factorial(X,FactX).
```

What the Computer is Really Doing

But wait a minute, you say. How does the computer execute ***factorial*** while it's in the middle of executing ***factorial***? If you call ***factorial*** with *X*=3, ***factorial*** will then call itself with *X*=2. Will *X* then have two values, or will the second value just wipe out the first, or what?

The answer is that the computer creates a new copy of ***factorial*** so that ***factorial*** can call itself as if it were a completely separate procedure. The executable code doesn't have to be duplicated, of course, but the arguments and internal variables do.

This information is stored in an area called a *stack frame*, which is created every time a rule is called. When the rule terminates, the stack is reset (unless it was a non-deterministic return) and execution continues in the stack frame for the parent.

Advantages of Recursion

Recursion has three main advantages:

- It can express algorithms that can't conveniently be expressed any other way.
- It is logically simpler than iteration.
- It is used extensively in list processing.

Recursion is the natural way to describe any problem that contains within itself another problem of the same kind. Examples include tree search (a tree is made up of smaller trees) and recursive sorting (to sort a list, partition it, sort the parts, and then put them together).

Logically, recursive algorithms have the structure of an inductive mathematical proof. The preceding recursive factorial algorithm, in Program `ch06e02.pro`, describes an infinite number of different computations by means of just two clauses. This makes it easy to see that the clauses are correct. Further, the correctness of each clause can be judged independently of the other.

Tail Recursion Optimization

Recursion has one big drawback: It eats memory. Whenever one procedure calls another, the calling procedure's state of execution must be saved so that it (the calling procedure) can resume where it left off after the called procedure has finished. This means that, if a procedure calls itself 100 times, 100 different states of execution must be stored at once. (The saved state of execution is known as a *stack frame*.) The maximum stack size on 16bit platforms, such as the IBM PC running DOS, is 64K, which will accommodate, at most, 3000 or 4000 stack frames. On 32bit platforms, the stack may theoretically grow to several GigaBytes; here, other system limitations will set in before the stack overflows. Anyway, what can be done to avoid using so much stack space?

It turns out that there's a special case in which a procedure can call itself without storing its state of execution. What if the calling procedure isn't going to resume after the called procedure finishes?

Suppose the calling procedure calls a procedure as its very last step. When the called procedure finishes, the calling procedure won't have anything else to do. This means the calling procedure doesn't need to save its state of execution, because that information isn't needed any more. As soon as the called procedure finishes, control can go directly to wherever it would have gone when the calling procedure finished.

For example, suppose that procedure ***A*** calls procedure ***B***, and ***B*** calls procedure ***C*** as its very last step. When ***B*** calls ***C***, ***B*** isn't going to do anything more. So, instead of storing the current state of execution for ***C*** under ***B***, you can replace ***B***'s old stored state (which isn't needed any more) with ***C***'s current state, making appropriate changes in the stored information. When ***C*** finishes, it thinks it was called by ***A*** directly.

Now suppose that, instead of calling ***C***, procedure ***B*** calls *itself* as its very last step. The recipe says that, when ***B*** calls ***B***, the stack frame for the calling ***B*** should be replaced by a stack frame for the called ***B***. This is a particularly simple operation; only the arguments need to be set to new values, and then processing jumps back to the beginning of the procedure. So, from a procedural point of view, what happens is very similar to updating the control variables in a loop.

This is called *tail recursion optimization*, or *last-call optimization*. Note that for technical reasons, recursive functions (predicates returning a value, described in chapter **10**) cannot be tail recursive.

Making Tail Recursion Work

What does it mean to say that one procedure calls another "as its very last step?" In Prolog, this means that

1. The call is the very last subgoal of the clause.
2. There are no backtracking points earlier in the clause.

Here's an example that satisfies both conditions:

```
count(N) :-
   write(N), nl,
   NewN = N+1,
   count(NewN).
```

This procedure is tail recursive; it calls itself without allocating a new stack frame, so it never runs out of memory. As program `ch06e04.pro` shows, if you give it the goal

```
count(0).
```

count will print integers starting with 0 and never ending. Eventually, rounding errors will make it print inaccurate numbers, but it will never stop.

```
                        /* Program ch06e04.pro */

   /* Tail recursive program that never runs out of memory */

PREDICATES
   count(ulong)

CLAUSES
   count(N):-
      write('\r',N),
      NewN = N+1,
      count(NewN).

GOAL
   nl,
   count(0).
```

Exercise

Without looking ahead, modify `ch06e04.pro` so that it is no longer tail recursive. How many iterations can it execute before running out of memory? Try it and see. (On 32-bit platforms, this will take a considerable length of time, and the program will most likely not run out of stack space; it, or the system, will run out

of memory in general. On 16-bit platforms, the number of possible iterations is directly related to the stack size.

How Not to Do Tail Recursion

Now that you've seen how to do tail recursion right, program `ch06e05.pro` shows you three ways to do it wrong.

1. ***If the recursive call isn't the very last step, the procedure isn't tail recursive.*** For example:

```
badcount1(X) :-
   write('\r',X),
   NewX = X+1,
   badcount1(NewX),
   nl.
```

   Every time ***badcount1*** calls itself, a stack frame has to be saved so that control can return to the calling procedure, which has yet to execute its final ***nl***. So only a few thousand recursive calls can take place before the program runs out of memory.

2. ***Another way to lose tail recursion is to leave an alternative untried at the time the recursive call is made.*** Then a stack frame must be saved so that, if the recursive call fails, the calling procedure can go back and try the alternative. For example:

```
badcount2(X) :-
   write('\r',X),
   NewX = X+1,
   badcount2(NewX).
badcount2(X) :-
X < 0,
   write("X is negative.").
```

   Here, the first clause of ***badcount2*** calls itself before the second clause has been tried. Again, the program runs out of memory after a certain number of calls.

3. The untried alternative doesn't need to be a separate clause for the recursive procedure itself. It can equally well be an alternative in some other clause that it calls. For example:

```
badcount3(X) :-
   write('\r',X),
   NewX = X+1,
   check(NewX),
   badcount3(NewX).

check(Z) :- Z >= 0.
check(Z) :- Z < 0.
```

Suppose *X* is positive, as it normally is. Then, when ***badcount3*** calls itself, the first clause of ***check*** has succeeded, but the second clause of ***check*** has not yet been tried. So ***badcount3*** has to preserve a copy of its stack frame in order to go back and try the other clause of ***check*** if the recursive call fails.

```
                        /* Program ch06e05.pro */

/* In 32bit memory architectures, the examples here
   will run for a considerable length of time, occupying large amounts
   of memory and possibly reducing system performance significantly.
*/
PREDICATES
   badcount1(long)
   badcount2(long)
   badcount3(long)
   check(long)

CLAUSES
/* badcount1:
   The recursive call is not the last step. */

   badcount1(X):-
      write('\r',X),
      NewX = X+1,
      badcount1(NewX),
      nl.

/* badcount2:
   There is a clause that has not been tried
   at the time the recursive call is made. */

   badcount2(X):-
      write('\r',X),
      NewX = X+1,
      badcount2(NewX).

   badcount2(X):-
      X < 0,
      write("X is negative.").
```

```
/* badcount3:
   There is an untried alternative in a
   predicate called before the recursive call. */

   badcount3(X):-
      write('\r',X),
      NewX = X+1,
      check(NewX),
      badcount3(NewX).

   check(Z):-
      Z >= 0.
   check(Z):-
      Z < 0.
```

Cuts to the Rescue

By now, you may think it's impossible to guarantee that a procedure is tail recursive. After all, it's easy enough to put the recursive call in the last subgoal of the last clause, but how do you guarantee there are no alternatives in any of the other procedures that it calls?

Fortunately, you don't have to. The cut `(!)` allows you to discard whatever alternatives may exist. You'll need to use the `check_determ` compiler directive to guide you through setting the cuts. (Compiler directives are described in the chapter **17**.)

You can fix up ***badcount3*** as follows (changing its name in the process):

```
   cutcount3(X) :-
      write('\r',X),
      NewX = X+1,
      check(NewX),
      !,
      cutcount3(NewX).
```

leaving ***check*** as it was.

The cut means "burn your bridges behind you" or, more precisely, "once you reach this point, disregard alternative clauses for this predicate and alternative solutions to earlier subgoals within this clause." That's precisely what you need. Because alternatives are ruled out, no stack frame is needed and the recursive call can go inexorably ahead.

A cut is equally effective in ***badcount2***, by negating and moving the test from the second clause to the first:

```
cutcount2(X) :-
   X >= 0, !,
   write('\r',X),
   NewX = X+1,
   cutcount2(NewX).

cutcount2(X) :-
   write("X is negative.").
```

A cut is really all about making up ones mind. You set a cut whenever you can look at non-deterministic code, and say "Yes! Go ahead!" – whenever it's obvious that alternatives are of no interest. In the original version of the above example, which tries to illustrate a situation where you have to decide something about *X* (the test `X < 0` in the second clause), the second clause had to remain an option as the code in the first clause didn't test *X*. By moving the test to the first clause and negating it, a decision can be reached already there and a cut set in accordance: "Now I know I don't want to write that *X* is negative."

The same applies to ***cutcount3***. The predicate ***check*** illustrates a situation where you want to do some additional processing of *X*, based on its sign. However, the code for ***check*** is, in this case for illustration, non-deterministic, and the cut after the call to it is all about you having made up your mind. After the call to ***check***, you can say "Yes! Go ahead!" However, the above is slightly artificial – it would probably be more correct for ***check*** to be deterministic:

```
check(Z) :- Z >= 0, !, ... % processing using Z
check(Z) :- Z < 0, ... %processing using Z
```

And, since the test in the second clause of ***check*** is the perfect negation of the test in the first, ***check*** can be further rewritten as:

```
check(Z) :- Z >= 0, !, % processing using Z
check(Z) :- ... % processing using Z
```

When a cut is executed, the computer assumes there are no untried alternatives and does not create a stack frame. Program `ch06e06.pro` contains modified versions of ***badcount2*** and ***badcount3***:

```
                        /* Program ch06e06.pro */

/* Shows how badcount2 and badcount3 can be fixed by adding cuts to
   rule out the untried clauses. These versions are tail recursive. */

PREDICATES
   cutcount2(long)
   cutcount3(long)
   check(long)
```

```
CLAUSES
/* cutcount2:
   There is a clause that has not been tried
   at the time the recursive call is made. */

   cutcount2(X):-
      X>=0,
      !,
      write('\r',X),
      NewX = X + 1,
      cutcount2(NewX).
   cutcount2(_):-
      write("X is negative.").

/* cutcount3:
   There is an untried alternative in a
   clause called before the recursive call. */

   cutcount3(X):-
      write('\r',X),
      NewX = X+1,
      check(NewX),
      !,
      cutcount3(NewX).

   check(Z):-Z >= 0.
   check(Z):-Z < 0.
```

Unfortunately, cuts won't help with ***badcount1***, whose need for stack frames has nothing to do with untried alternatives. The only way to improve ***badcount1*** would be to rearrange the computation so that the recursive call comes at the end of the clause.

Using Arguments as Loop Variables

Now that you've mastered tail recursion, what can you do about loop variables and counters? To answer that question, we'll do a bit of Pascal-to-Prolog translation, assuming that you're familiar with Pascal. Generally, the results of direct translations between two languages, whether natural or programming, are poor. The following isn't too bad and serves as a reasonable illustration of strictly imperative programming in Prolog, but you should never write Prolog programs by blind translation from another language. Prolog is a very powerful and expressive language, and properly written Prolog programs will display a programming style and problem focus quite different from what programs in other languages do.

In the "Recursion" section, we developed a recursive procedure to compute factorials; in this section we'll develop an iterative one. In Pascal, this would be:

```
P := 1;
for I := 1 to N do P := P*I;
FactN := P;
```

If you're unfamiliar with Pascal, the `:-` is the assignment, read as "becomes". There are four variables here. *N* is the number whose factorial will be calculated; *FactN* is the result of the calculation; *I* is the loop variable, counting from 1 to *N*; and *P* is the variable in which the product accumulates. A more efficient Pascal programmer might combine *FactN* and *P*, but in Prolog it pays to be fastidiously tidy.

The first step in translating this into Prolog is to replace **for** with a simpler **loop** statement, making what happens to *I* in each step more explicit. Here is the algorithm recast as a **while** loop:

```
P := 1;                                        /* Initialize P and I */
I := 1;
while I <= N do                                        /* Loop test */
   begin
      P := P*I;                                    /* Update P and I */
      I := I+1
   end;
FactN := P;                                      /* Return result */
```

shows the Prolog translation constructed from this Pascal **while** loop.

```
                        /* Program ch06e07.pro */

PREDICATES
   factorial(unsigned,long)
   factorial_aux(unsigned,long,unsigned,long)

   /* Numbers likely to become large are declared as longs. */

CLAUSES
   factorial(N, FactN):-
      factorial_aux(N,FactN,1,1).
```

```
factorial_aux(N,FactN,I,P):-
   I <= N,!,
   NewP = P * I,
   NewI = I + 1,
   factorial_aux(N, FactN, NewI, NewP).
factorial_aux(N, FactN, I, P) :-
   I > N,
   FactN = P.
```

Let's look at this in greater detail.

The factorial clause has only *N* and *FactN* as arguments; they are its input and output, from the viewpoint of someone who is using it to find a factorial. A second clause, `factorial_aux(N, FactN, I, P)`, actually performs the recursion; its four arguments are the four variables that need to be passed along from each step to the next. So ***factorial*** simply invokes ***factorial_aux***, passing to it *N* and *FactN*, along with the initial values for *I* and *P*, like so:

```
factorial(N, FactN) :-
   factorial_aux(N, FactN, 1, 1).
```

That's how *I* and *P* get initialized.

How can ***factorial*** "pass along" *FactN*? It doesn't even have a value yet! The answer is that, conceptually, all Visual Prolog is doing here is unifying a variable called *FactN* in one clause with a variable called *FactN* in another clause. The same thing will happen whenever ***factorial_aux*** passes *FactN* to itself as an argument in a recursive call. Eventually, the last *FactN* will get a value, and, when this happens, all the other *FactN*-s, having been unified with it, will get the same value. We said "conceptually" above, because in reality there is only one *FactN*. Visual Prolog can determine from the source code that *FactN* is never really used before the second clause for ***factorial_aux***, and just shuffles the same *FactN* around all the time.

Now for ***factorial_aux***. Ordinarily, this predicate will check that *I* is less than or equal to *N* – the condition for continuing the loop – and then call itself recursively with new values for *I* and *P*. Here another peculiarity of Prolog asserts itself. In Prolog there is no assignment statement such as

```
P = P + 1
```

which is found in most other programming languages. ***You can't change the value of a Prolog variable.*** In Prolog, the above is as absurd as in algebra, and will fail. Instead, you have to create a new variable and say something like

```
NewP = P + 1
```

So here's the first clause:

```
factorial_aux(N, FactN, I, P) :-
   I <= N, !,
   NewP = P*I,
   NewI = I+1,
   factorial_aux(N, FactN, NewI, NewP).
```

As in ***cutcount2***, the cut enables last-call optimization to take effect, even though the clause isn't the last in the predicate.

Eventually *I* will exceed *N*. When it does, processing should unify the current value of *P* with *FactN* and stop the recursion. This is done in the second clause, which will be reached when the test *I* <= *N* in the first clause fails:

```
factorial_aux(N, FactN, I, P) :-
   I > N,
   FactN = P.
```

But there is no need for *FactN* = *P* to be a separate step; the unification can be performed in the argument list. Putting the same variable name in the positions occupied by *FactN* and *P* requires the arguments in these positions to be matched with each other. Moreover, the test *I* > *N* is redundant since the opposite has been tested for in the first clause. This gives the final clause:

```
factorial_aux(_, FactN, _, FactN).
```

Exercises

1. The following is a more elegant version of ***factorial***.

```
                        /* Program ch06e08.pro */

PREDICATES
   factorial(unsigned,long)
   factorial(unsigned,long,unsigned,long)
   /* Numbers likely to become large are declared as longs. */

CLAUSES
   factorial(N,FactN):-
      factorial(N,FactN,1,1).
```

```
factorial(N,FactN,N,FactN):-
   !.
factorial(N,FactN,I,P):-
   NewI = I+1,
   NewP = P*NewI,
   factorial(N, FactN, NewI, NewP).
```

Load and run this program. Carefully look at the code in the second clause of ***factorial/4***. It takes advantage of the fact that the first time it's called the counter variable *I* always has the value 1. This allows the multiplication step to be carried out with the incremented counter variable *NewI* rather than *I*, saving one recursion/iteration. This is reflected in the first clause.

2. Write a tail recursive program that behaves like 2 but doesn't use backtracking.
3. Write a tail recursive program that prints a table of powers of 2, like this:

```
N        2^N
--     -----
1          2
2          4
3          8
4         16
...      ...
10      1024
```

   Make it stop at 10 as shown here.

4. Write a tail recursive program that accepts a number as input and can end in either of two ways. It will start multiplying the number by itself over and over until it either reaches 81 or reaches a number greater than 100. If it reaches 81, it will print yes; if it exceeds 100, it will print no.

Recursive Data Structures

Not only can rules be recursive; so can data structures. Prolog is the only widely used programming language that allows you to define recursive data types. A data type is recursive if it allows structures to contain other structures like themselves.

The most basic recursive data type is the list, although it doesn't immediately look recursively constructed. A lot of list-processing power is built into Prolog,

but we won't discuss it here; lists are such an important part of Prolog that there is a whole chapter devoted to them, chapter 7.

In this chapter, we invent a recursive data type, implement it, and use it to write a very fast sorting program. The structure of this invented recursive data type is a tree (Figure 6.1). Crucially, each branch of the tree is itself a tree; that's why the structure is recursive.

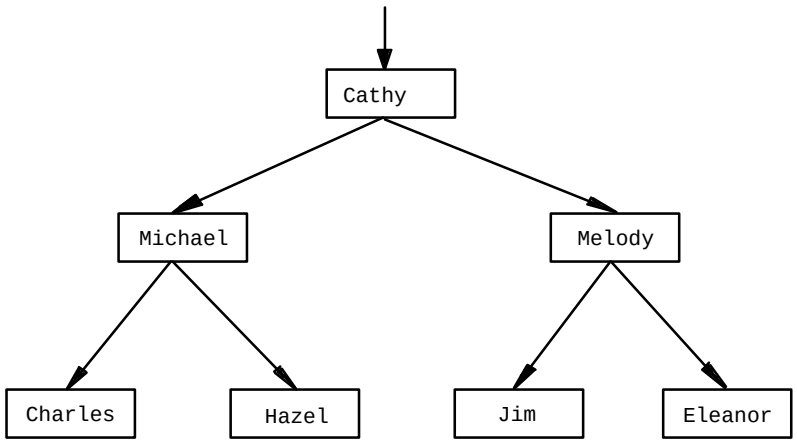


Figure 6.1: Part of a Family Tree

Trees as a Data Type

Recursive types were popularized by Niklaus Wirth in *Algorithms + Data Structures = Programs*. Wirth derived Pascal from ALGOL60 and published this work in the early 70's. He didn't implement recursive types in Pascal, but he did discuss what it would be like to have them. If Pascal had recursive types, you would be able to define a tree as something like this:

```
tree = record                                   /* Not correct Pascal! */
       name: string[80];
       left, right: tree
       end;
```

This code, translated into natural language, means: "A tree consists of a name, which is a string, and the left and right subtrees, which are trees."

The nearest approach to this in Pascal is to use pointers and say

```
treeptr = ^tree;

tree = record
       name: string[80];
       left, right: treeptr
       end;
```

But notice a subtle difference: This code deals with the *memory representation* of a tree, not the *structure* of the tree itself. It treats the tree as consisting of cells, each containing some data plus pointers to two more cells.

Visual Prolog allows truly recursive type definitions in which the pointers are created and maintained automatically. For example, you can define a tree as follows:

```
DOMAINS
   treetype = tree(string, treetype, treetype)
```

This declaration says that a tree will be written as the functor, ***tree***, whose arguments are a string and two more trees.

But this isn't quite right; it provides no way to end the recursion, and, in real life, the tree does not go on forever. Some cells don't have links to further trees. In Pascal, you could express this by setting some pointers equal to the special value nil, but pointers are an implementation issue that ordinarily doesn't surface in Prolog source code. Rather, in Prolog we define two kinds of trees: ordinary ones and empty ones. This is done by allowing a tree to have either of two functors: ***tree***, with three arguments, or ***empty***, with no arguments.

```
DOMAINS
   treetype = tree(string, treetype, treetype) ; empty
```

Notice that the names ***tree*** (a functor that takes three arguments) and ***empty*** (a functor taking no arguments) are created by the programmer; neither of them has any pre-defined meaning in Prolog. You could equally well have used ***xxx*** and ***yyy***.

This is how the tree in Figure 6.1 could appear in a Prolog program:

```
tree("Cathy",
     tree("Michael"
          tree("Charles", empty, empty)
          tree("Hazel", empty, empty))
     tree("Melody"
          tree("Jim", empty, empty)
          tree("Eleanor", empty, empty)))
```

This is indented here for readability, but Prolog does not require indentation, nor are trees indented when you print them out normally. Another way of setting up this same data structure is:

```
tree("Cathy"
   tree("Michael", tree("Charles", empty, empty), tree("Hazel", empty,
empty))
   tree("Melody", tree("Jim", empty, empty), tree("Eleanor", empty,
empty)))
```

Note that this is not a Prolog clause; it is just a complex data structure.

Traversing a Tree

Before going on to the discussion of how to create trees, first consider what you'll do with a tree once you have it. One of the most frequent tree operations is to examine all the cells and process them in some way, either searching for a particular value or collecting all the values. This is known as *traversing* the tree. One basic algorithm for doing so is the following:

1. If the tree is empty, do nothing.
2. Otherwise, process the current node, then traverse the left subtree, then traverse the right subtree.

Like the tree itself, the algorithm is recursive: it treats the left and right subtrees exactly like the original tree. Prolog expresses it with two clauses, one for empty and one for nonempty trees:

```
traverse(empty).                                    /* do nothing */

traverse(tree(X, Y, Z)) :-
   do something with X,
   traverse(Y),
   traverse(Z).
```

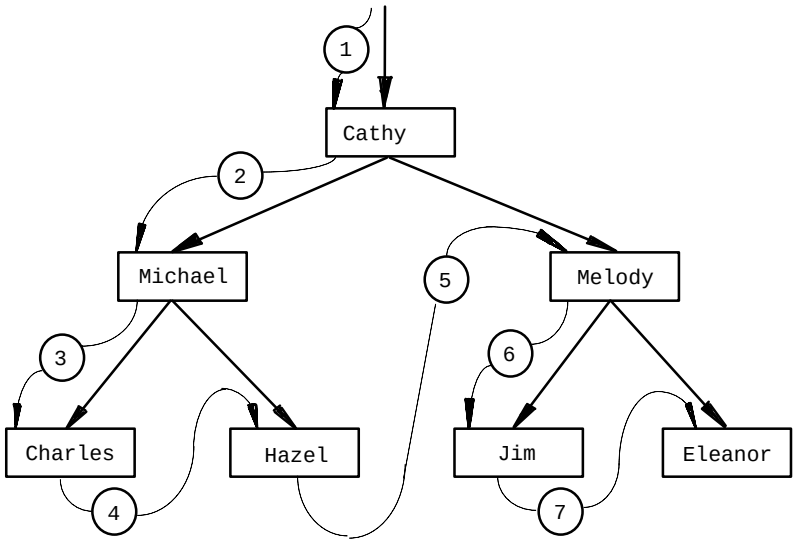


Figure 6.2: Depth-First Traversal of the Tree in Figure 6.1

This tree traversal algorithm is known as *depth-first search* because it goes as far as possible down each branch before backing up and trying another branch (Figure 6.2). To see it in action, look at program `ch06e09.pro`, which traverses a tree and prints all the elements as it encounters them. Given the tree in Figures 6.1 and 6.2, `ch06e09.pro` prints

```
Cathy
Michael
Charles
Hazel
Melody
Jim
Eleanor
```

Of course, you could easily adapt the program to perform some other operation on the elements, rather than printing them.

```
                        /* Program ch06e09.pro */

   /* Traversing a tree by depth-first search
      and printing each element as it is encountered */

DOMAINS
   treetype = tree(string, treetype, treetype) ; empty()
```

```
PREDICATES
   traverse(treetype)

CLAUSES
   traverse(empty).

   traverse(tree(Name,Left,Right)):-
      write(Name,'\n'),
      traverse(Left),
      traverse(Right).

GOAL
   traverse(tree("Cathy",
            tree("Michael",
            tree("Charles", empty, empty),
            tree("Hazel", empty, empty)),
            tree("Melody",
            tree("Jim", empty, empty),
            tree("Eleanor", empty, empty)))).
```

Depth-first search is strikingly similar to the way Prolog searches a knowledge base, arranging the clauses into a tree and pursuing each branch until a query fails. If you wanted to, you could describe the tree by means of a set of Prolog clauses such as:

```
father_of("Cathy", "Michael").
mother_of("Cathy", "Melody").
father_of("Michael", "Charles").
mother_of("Michael", "Hazel").
...
```

This is preferable if the only purpose of the tree is to express relationships between individuals. But this kind of description makes it impossible to treat the whole tree as a single complex data structure; as you'll see, complex data structures are very useful because they simplify difficult computational tasks.

Creating a Tree

One way to create a tree is to write down a nested structure of functors and arguments, as in the preceding example (Program `ch06e09.pro`). Ordinarily, however, Prolog creates trees by computation. In each step, an empty subtree is replaced by a nonempty one through Prolog's process of unification (argument matching).

Creating a one-cell tree from an ordinary data item is trivial:

```
create_tree(N, tree(N, empty, empty)).
```

This says: "If *N* is a data item, then `tree(N, empty, empty)` is a one-cell tree containing it."

Building a tree structure is almost as easy. The following procedure takes three trees as arguments. It inserts the first tree as the left subtree of the second tree, giving the third tree as the result:

```
insert_left(X, tree(A, _, B), tree(A, X, B)).
```

Notice that this rule has no body – there are no explicit steps in executing it. All the computer has to do is match the arguments with each other in the proper positions, and the work is done.

Suppose, for example, you want to insert `tree("Michael", empty, empty)` as the left subtree of `tree("Cathy", empty, empty)`. To do this, just execute the goal

```
insert_left(tree("Michael", empty, empty),
            tree("Cathy", empty, empty),
            T)
```

and *T* immediately takes on the value

```
tree("Cathy", tree("Michael", empty, empty), empty).
```

This gives a way to build up trees step-by-step. Program `ch06e10.pro` demonstrates this technique. In real life, the items to be inserted into the tree could come from external input.

```
                        /* Program ch06e10.pro */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Simple tree-building procedures                                       *
 * create_tree(A, B) puts A in the data field of a one-cell tree         *
 * giving B insert_left(A, B, C) inserts A as left subtree of B          *
 * giving C insert_right(A, B, C) inserts A as right subtree of B        *
 * giving C                                                              *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

DOMAINS
   treetype = tree(string,treetype,treetype) ; empty()

PREDICATES
   create_tree(string,treetype)
   insert_left(treetype,treetype,treetype)
   insert_right(treetype, treetype, treetype)
   run
```

```
CLAUSES
   create_tree(A,tree(A,empty,empty)).
   insert_left(X,tree(A,_,B),tree(A,X,B)).
   insert_right(X,tree(A,B,_),tree(A,B,X)).

   run:-
      % First create some one-cell trees
      create_tree("Charles",Ch),
      create_tree("Hazel",H),
      create_tree("Michael",Mi),
      create_tree("Jim",J),
      create_tree("Eleanor",E),
      create_tree("Melody",Me),
      create_tree("Cathy",Ca),

      %.then link them up
      insert_left(Ch, Mi, Mi2),
      insert_right(H, Mi2, Mi3),
      insert_left(J, Me, Me2),
      insert_right(E, Me2, Me3),
      insert_left(Mi3, Ca, Ca2),
      insert_right(Me3, Ca2, Ca3),

      %.and print the result
      write(Ca3,'\n').

GOAL
   run.
```

Notice that there is no way to change the value of a Prolog variable once it is bound. That is why `ch06e10.pro` uses so many variable names; every time you create a new value, you need a new variable. The large number of variable names here is unusual; more commonly, repetitive procedures obtain new variables by invoking themselves recursively, since each invocation has a distinct set of variables. Notice that the Test Goal utility has restriction on the number of variables used in the goal (<12), this is why the wrapping predicate ***run*** should be used.

Binary Search Trees

So far, we have been using the tree to represent relationships between its elements. Of course, this is not the best use for trees, since Prolog clauses can do the same job. But trees have other uses.

Trees provide a good way to store data items so that they can be found quickly. A tree built for this purpose is called a *search tree*; from the user's point of view, the tree structure carries no information – the tree is merely a faster alternative to a list or array. Recall that, to traverse an ordinary tree, you look at the current cell and then at both of its subtrees. To find a particular item, you might have to look at every cell in the whole tree.

The time taken to search an ordinary tree with N elements is, on the average, proportional to N.

A *binary search tree* is constructed so that you can predict, upon looking at any cell, which of its subtrees a given item will be in. This is done by defining an ordering relation on the data items, such as alphabetical or numerical order. Items in the left subtree precede the item in the current cell and, in the right subtree, they follow it. Figure 6.3 shows an example. Note that the same names, added in a different order, would produce a somewhat different tree. Notice also that, although there are ten names in the tree, you can find any of them in – at most – five steps.

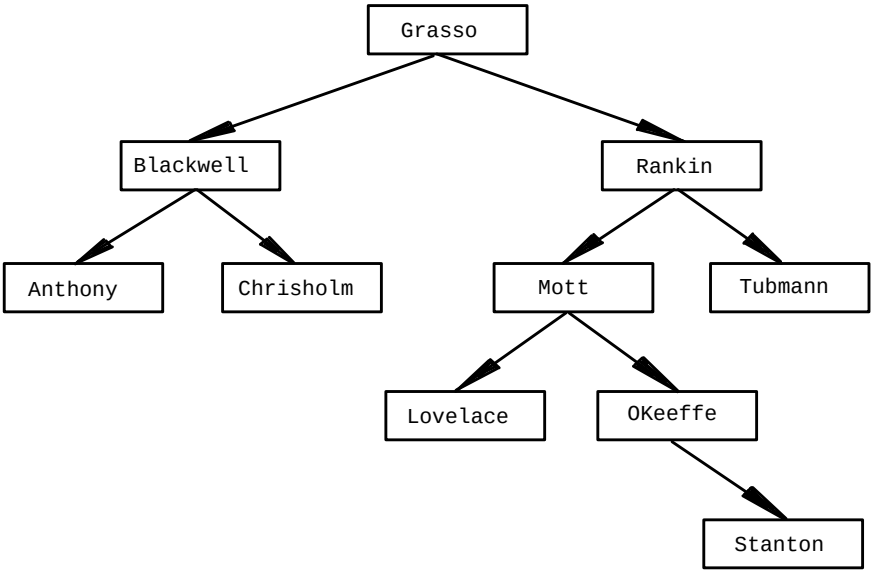


Figure 6.3: Binary Search Tree

Every time you look at a cell in a binary search tree during a search, you eliminate half the remaining cells from consideration, and the search proceeds very quickly. If the size of the tree were doubled, then, typically, only one extra step would be needed to search it.

> ***The time taken to find an item in a binary search tree is, on the average, proportional to log2 N (or, in fact, proportional to log N with logarithms to any base).***

To build the tree, you start with an empty tree and add items one by one. The procedure for adding an item is the same as for finding one: you simply search for the place where it ought to be, and insert it there. The algorithm is as follows:

If the current node is an empty tree, insert the item there.

Otherwise, compare the item to be inserted and the item stored in the current node. Insert the item into the left subtree or the right subtree, depending on the result of the comparison.

In Prolog, this requires three clauses, one for each situation. The first clause is

```
insert(NewItem, empty, tree(NewItem, empty, empty) :- !.
```

Translated to natural language, this code says "The result of inserting *NewItem* into *empty* is `tree(NewItem, empty, empty)`." The cut ensures that, if this clause can be used successfully, no other clauses will be tried.

The second and third clauses take care of insertion into nonempty trees:

```
insert(NewItem, tree(Element, Left, Right),
   tree(Element, NewLeft, Right) :-
      NewItem < Element,
      !,
      insert(NewItem, Left, NewLeft).

insert(NewItem, tree(Element, Left, Right),
   tree(Element, Left, NewRight) :-
      insert(NewItem, Right, NewRight).
```

If *NewItem* < *Element*, you insert it into the left subtree; otherwise, you insert it into the right subtree. Notice that, because of the cuts, you get to the third clause only if neither of the preceding clauses has succeeded. Also notice how much of the work is done by matching arguments in the head of the rule.

Tree-Based Sorting

Once you have built the tree, it is easy to retrieve all the items in alphabetical order. The algorithm is again a variant of depth-first search:

1. If the tree is empty, do nothing.
2. Otherwise, retrieve all the items in the left subtree, then the current element, then all the items in the right subtree.

Or, in Prolog:

```
retrieve_all(empty).                                    /* Do nothing */

retrieve_all(tree(Item, Left, Right)) :-
   retrieve_all(Left),
   do_something_to(Item),
   retrieve_all(Right).
```

You can sort a sequence of items by inserting them into a tree and then retrieving them in order. For *N* items, this takes time proportional to *N* log *N*, because both insertion and retrieval take time proportional to log *N*, and each of them has to be done *N* times. This is the fastest known sorting algorithm.

Example

Program `ch06e11.pro` uses this technique to alphabetize character input. In this example we use some of Visual Prolog's standard predicates we haven't introduced before. These predicates will be discussed in detail in later chapters.

```
                        /* Program ch06e11.pro */

DOMAINS
   chartree = tree(char, chartree, chartree); end

PREDICATES
   nondeterm do(chartree)
   action(char, chartree, chartree)
   create_tree(chartree, chartree)
   insert(char, chartree, chartree)
   write_tree(chartree)
   nondeterm repeat
```

```
CLAUSES
   do(Tree):-
      repeat,nl,
      write("*************************************************"),nl,
      write("Enter 1 to update tree\n"),
      write("Enter 2 to show tree\n"),
      write("Enter 7 to exit\n"),
      write("*************************************************"),nl,
      write("Enter number - "),
      readchar(X),nl,
      action(X, Tree, NewTree),
      do(NewTree).

   action('1',Tree,NewTree):-
      write("Enter characters or # to end: "),
      create_Tree(Tree, NewTree).
   action('2',Tree,Tree):-
      write_Tree(Tree),
      write("\nPress a key to continue"),
      readchar(_),nl.
   action('7', _, end):-
      exit.

   create_Tree(Tree, NewTree):-
      readchar(C),
      C<>'#',
      !,
      write(C, " "),
      insert(C, Tree, TempTree),
      create_Tree(TempTree, NewTree).
      create_Tree(Tree, Tree).

   insert(New,end,tree(New,end,end)):-
      !.
   insert(New,tree(Element,Left,Right),tree(Element,NewLeft,Right)):-
      New<Element,
      !,
      insert(New,Left,NewLeft).
   insert(New,tree(Element,Left,Right),tree(Element,Left,NewRight)):-
      insert(New,Right,NewRight).

   write_Tree(end).
   write_Tree(tree(Item,Left,Right)):-
      write_Tree(Left),
      write(Item, " "),
      write_Tree(Right).
```

```
   repeat.
   repeat:-repeat.

GOAL
   write("*************** Character tree sort ******************"),nl,
   do(end).
```

Load and run Program `ch06e11.pro` and watch how Visual Prolog does tree-based sorting on a sequence of characters.

Exercises

1. Program `ch06e12.pro` is similar to `ch06e11.pro`, but more complex. It uses the same sorting technique to alphabetize any standard text file, line by line. Typically it's more than five times faster than "SORT.EXE", the sort program provided by DOS, but it's beaten by the highly optimized "sort" on UNIX. Nevertheless, tree-based sorting is remarkably efficient.

   In this example we use some of the predicates from Visual Prolog's file system, to give you a taste of file redirection. To redirect input or output to a file, you must tell the system about the file; you use ***openread*** to read from the file or ***openwrite*** to write to it. Once files are open, you can switch I/O between an open file and the screen with ***writedevice***, and between an open file and the keyboard with ***readdevice***. These predicates are discussed in detail later in chapter **12**.

   Load and run Program `ch06e12.pro`. When it prompts File to read type in the name of an existing text file; the program will then alphabetize that file, line by line.

```
                        /* Program ch06e12.pro */

DOMAINS
   treetype = tree(string, treetype, treetype) ; empty
   file     = infile ; outfile

PREDICATES
   main
   read_input(treetype)
   read_input_aux(treetype, treetype)
   insert(string, treetype, treetype)
   write_output(treetype)
```

```
CLAUSES
   main :-
      write("PDC Prolog Treesort"),nl,
      write("File to read:  "),
      readln(In),nl,
      openread(infile, In),      /* open the specified file for reading */
      write("File to write: "),
      readln(Out),nl,
      openwrite(outfile, Out),
      readdevice(infile),
                   /* redirect all read operations to the opened file */
      read_input(Tree),
      writedevice(outfile),
                  /* redirect all write operations to the opened file */
      write_output(Tree),
      closefile(infile),           /* close the file opened for reading */
      closefile(outfile).

   /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * read_input(Tree)                                                *
    *   reads lines from the current input device until EOF, then *
    *   instantiates Tree to the binary search tree built         *
    *   therefrom                                                   *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

   read_input(Tree):-
      read_input_aux(empty,Tree).

   /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * read_input_aux(Tree, NewTree)                                    *
    *  reads a line, inserts it into Tree giving NewTree,         *
    *  and calls itself recursively unless at EOF.                *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

   read_input_aux(Tree, NewTree):-
      readln(S),
      !,
      insert(S, Tree, Tree1),
      read_input_aux(Tree1, NewTree).

   read_input_aux(Tree, Tree). /* The first clause fails at EOF. */

   /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *  insert(Element, Tree, NewTree)                                *
    *     inserts Element into Tree giving NewTree.                *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

   insert(NewItem, empty, tree(NewItem,empty,empty)):-!.
```

```
insert(NewItem,tree(Element,Left,Right),tree(Element,NewLeft, Right)):-
   NewItem < Element,
   !,
   insert(NewItem, Left, NewLeft).

insert(NewItem,tree(Element,Left,Right),tree(Element,Left,NewRight)):-
   insert(NewItem, Right, NewRight).

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * write_output(Tree)                                               *
 *    writes out the elements of Tree in alphabetical order.       *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

write_output(empty). /* Do nothing */

write_output(tree(Item,Left,Right)):-
   write_output(Left),
   write(Item), nl,
   write_output(Right).

GOAL
   main,nl.
```

2. Use recursive data structures to implement *hypertext*. A hypertext is a structure in which each entry, made up of several lines of text, is accompanied by pointers to several other entries. Any entry can be connected to any other entry; for instance, you could get to an entry about Abraham Lincoln either from "Presidents" or from "Civil War."

   To keep things simple, use one-line entries (strings) and let each of them contain a pointer to only one other entry.

   Hint: Start with

   ```
   DOMAINS
      entrytype = empty() ; entry(string, entry)
   ```

   Build a linked structure in which most of the entries have a nonempty second argument.

3. Now, take your hypertext implementation and redo it using Prolog clauses. That is, use clauses (rather than recursive data structures) to record which entry follows which.

Summary

These are the major points covered in this chapter:

1. In Prolog there are two ways to repeat the same clause: through *backtracking* and *recursion*. By failing, Prolog will backtrack to find a new piece of data and repeat the clause until there are no more options. Recursion is the process of a clause calling itself.
2. Backtracking is very powerful and memory efficient, but variables are freed after each iteration, so their values are lost. Recursion allows variables to be incremented, but it is not memory efficient.
3. However, Visual Prolog does tail recursion elimination, which relieves the memory demands of recursion. For Visual Prolog to achieve tail recursion elimination, the recursive call must be the last subgoal in the clause body.

CHAPTER 7

Lists and Recursion

List processing – handling objects that contain an arbitrary number of elements – is a powerful technique in Prolog. In this chapter, we explain what lists are and how to declare them, then give several examples that show how you might use list processing in your own applications. We also define two well-known Prolog predicates – ***member*** and ***append*** – while looking at list processing from both a recursive and a procedural standpoint.

After that, we introduce ***findall***, a Visual Prolog standard predicate that enables you to find and collect all solutions to a single goal. We round out this chapter with a discussion of compound lists – combinations of different types of elements – and an example of parsing by difference lists.

What Is a List?

In Prolog, a *list* is an object that contains an arbitrary number of other objects within it. Lists correspond roughly to arrays in other languages, but, unlike an array, a list does not require you to declare how big it will be before you use it.

There are other ways to combine several objects into one, of course. If the number of objects to be combined is known in advance, you can make them the arguments of a single compound data structure. And even if the number of objects is unpredictable, you can use a recursive compound data structure, such as a tree. But lists are usually easier to use because the language provides a concise notation for them.

A list that contains the numbers 1, 2, and 3 is written as

```
[ 1, 2, 3 ]
```

Each item contained in the list is known as an *element*. To form a list data structure, you separate the elements of a list with commas and then enclose them in square brackets. Here are some examples:

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

Declaring Lists

To declare the domain for a list of integers, you use the **domains** declaration, like this:

```
DOMAINS
   integerlist = integer*
```

The asterisk means "list of"; that is, **integer*** means "list of integers."

Note that the word *list* has no special meaning in Visual Prolog. You could equally well have called your list domain *zanzibar*. It's the asterisk, not the name, that signifies a list domain.

The elements in a list can be anything, including other lists. However, all elements in a list must belong to the same domain, and in addition to the declaration of the list domain there must be a **domains** declaration for the elements:

```
DOMAINS
   elementlist = elements*
   elements    = ....
```

Here *elements* must be equated to a single domain type (for example: ***integer***, ***real***, or ***symbol***) or to a set of alternatives marked with different functors. Visual Prolog does not allow you to mix standard types in a list. For example, the following declarations would not properly indicate a list made up of ***integers***, ***reals***, and ***symbols***:

```
elementlist = elements*
elements =  integer; real; symbol                        /* Incorrect */
```

The way to declare a list made up of ***integers***, ***reals***, and ***symbols*** is to define a single domain comprising all three types, with functors to show which type a particular element belongs to. For example:

```
elementlist = elements*
elements = i(integer); r(real); s(symbol)
                                     /* the functors are i, r, and s */
```

(For more information about this, refer to "Compound Lists" later in this chapter.)

Heads and Tails

A list is really a recursive compound object. It consists of two parts: the head, of list which is the first element, and the tail, which is a list comprising all the

subsequent elements. ***The tail of a list is always a list; the head of a list is an element.*** For example,

the head of `[a, b, c]` is `a`

the tail of `[a, b, c]` is `[b, c]`

What happens when you get down to a one-element list? The answer is that

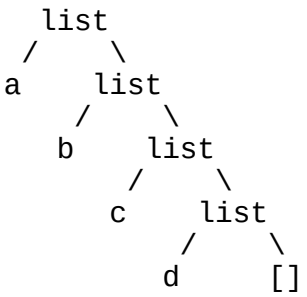
the head of `[c]` is `c`

the tail of `[c]` is `[]`

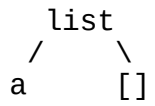
If you take the first element from the tail of a list enough times, you'll eventually get down to the empty list (`[ ]`).

The empty list can't be broken into head and tail.

This means that, conceptually, lists have a tree structure just like other compound objects. The tree structure of `[a, b, c, d]` is



Further, a one-element list such as `[a]` is not the same as the element that it contains because, simple as it looks, `[a]` is really the compound data structure shown here:



List Processing

Prolog provides a way to make the head and tail of a list explicit. Instead of separating elements with commas, you can separate the head and tail with a vertical bar (|). For instance,

`[a, b, c]` is equivalent to `[a|[b, c]]`

and, continuing the process,

`[a|[b, c]]` is equivalent to `[a|[b|[c]]]`

which is equivalent to `[a|[b|[c|[]]]]`

You can even use both kinds of separators in the same list, provided the vertical bar is the last separator. So, if you really want to, you can write `[a, b, c, d]` as `[a, b|[c, d]]`. Table 7.1 gives more examples.

Table 7.1: Heads and Tails of Lists

List	Head	Tail
`['a', 'b', 'c']`	`'a'`	`['b', 'c']`
`[ 'a' ]`	`'a'`	`[] /* an empty list */`
`[ ]`	`undefined`	`undefined`
`[[1, 2, 3], [2, 3, 4], []]`	`[1, 2, 3]`	`[[2, 3, 4], []]`

Table 7.2 gives several examples of list unification.

Table 7.2: Unification of Lists

List 1	List 2	Variable Binding
`[X, Y, Z]`	`[egbert, eats, icecream]`	`X=egbert, Y=eats, Z=icecream`
`[7]`	`[X | Y]`	`X=7, Y=[]`
`[1, 2, 3, 4]`	`[X, Y | Z]`	`X=1, Y=2, Z=[3,4]`
`[1, 2]`	`[3 | X]`	`fail`

Using Lists

Because a list is really a recursive compound data structure, you need recursive algorithms to process it. The most basic way to process a list is to work through it, doing something to each element until you reach the end.

An algorithm of this kind usually needs two clauses. One of them says what to do with an ordinary list (one that can be divided into a head and a tail). The other says what to do with an empty list.

Writing Lists

For example, if you just want to print out the elements of the list, here's what you do:

```
                        /* Program ch07e01.pro */

DOMAINS
   list = integer*                 /* or whatever type you wish to use */

PREDICATES
   write_a_list(list)

CLAUSES
   write_a_list([]).          /* If the list is empty, do nothing more. */

   write_a_list([H|T]):-
                     /* Match the head to H and the tail to T, then... */
      write(H),nl,
      write_a_list(T).

GOAL
   write_a_list([1, 2, 3]).
```

Here are the two ***write_a_list*** clauses described in natural language:

```
   To write an empty list, do nothing.

   Otherwise, to write a list, write its head (which
   is a single element), then write its tail (a list).
```

The first time through, the goal is:

```
   write_a_list([1, 2, 3]).
```

This matches the second clause, with `H=1` and `T=[2, 3]`; this writes 1 and then calls ***write_a_list*** recursively with the tail of the list:

```
   write_a_list([2, 3]).                    /* This is write_a_list(T). */
```

This recursive call matches the second clause, this time with `H=2` and `T=[3]`, so it writes 2 and again calls ***write_a_list*** recursively:

```
   write_a_list([3]).
```

Now, which clause will *this* goal match? Recall that, even though the list `[3]` has only one element, it *does* have a head and tail; the head is 3 and the tail is `[]`. So again the goal matches the second clause, with `H=3` and `T=[]`. Hence, 3 is written and ***write_a_list*** is called recursively like this:

```
write_a_list([]).
```

Now you see why this program needs the first clause. The second clause won't match this goal because `[]` can't be divided into head and tail. So, if the first clause weren't there, the goal would fail. As it is, the first clause matches and the goal succeeds without doing anything further.

Exercise

Is ***write_a_list*** tail-recursive? Would it be if the two clauses were written in the opposite order?

Counting List Elements

Now consider how you might find out how many elements are in a list. What *is* the length of a list, anyway? Here's a simple logical definition:

The length of [] is 0.

The length of any other list is 1 plus the length of its tail.

Can you implement this? In Prolog it's very easy. It takes just two clauses:

```
                        /* Program ch07e02.pro */

DOMAINS
   list = integer*                  /* or whatever type you want to use */

PREDICATES
   length_of(list,integer)

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

Take a look at the second clause first. Crucially, `[_|T]` will match any nonempty list, binding *T* to the tail of the list. The value of the head is unimportant; as long as it exists, it can be counted it as one element.

So the goal:

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

will match the second clause, with `T=[2, 3]`. The next step is to compute the length of *T*. When this is done (never mind how), *TailLength* will get the value `2`, and the computer can then add `1` to it and bind *L* to `3`.

So how is the middle step executed? That step was to find the length of [2, 3] by satisfying the goal

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

In other words, ***length_of*** calls itself recursively. This goal matches the second clause, binding

- `[3]` in the goal to `T` in the clause and
- `TailLength` in the goal to `L` in the clause.

Recall that *TailLength* in the goal will not interfere with *TailLength* in the clause, because ***each recursive invocation of a clause has its own set of variables***. If this is unclear, review the section on recursion in chapter **6**.

So now the problem is to find the length of `[3]`, which will be `1`, and then add `1` to that to get the length of `[2, 3]`, which will be `2`. So far, so good.

Likewise, ***length_of*** will call itself recursively again to get the length of `[3]`. The tail of `[3]` is `[]`, so *T* is bound to `[]`, and the problem is to get the length of `[]`, then add 1 to it, giving the length of `[3]`.

This time it's easy. The goal

```
length_of([], TailLength)
```

matches the *first* clause, binding *TailLength* to `0`. So now the computer can add 1 to that, giving the length of `[3]`, and return to the calling clause. This, in turn, will add 1 again, giving the length of `[2, 3]`, and return to the clause that called it; this original clause will add 1 again, giving the length of `[1, 2, 3]`.

Confused yet? We hope not. In the following brief illustration we'll summarize the calls. We've used subscripts to indicate that similarly-named variables in different clauses – or different invocations of the same clause – are distinct.

```
length_of([1, 2, 3], L1).
  length_of([2, 3], L2).
    length_of([3], L3).
      length_of([], 0).
```

```
L3 =  0+1 = 1.
L2 = L3+1 = 2.
L1 = L2+1 = 3.
```

Exercises

1. What happens when you satisfy the following goal?

   ```
   length_of(X, 3), !.
   ```

   Does the goal succeed, and if so, what is bound to *X*? Why? (Work through carefully by hand to see how this works.)

2. Write a predicate called ***sum_of*** that works exactly like ***length_of***, except that it takes a list of numbers and adds them up. For example, the goal:

   ```
   sum_of([1, 2, 3, 4], S).
   ```

   should bind *S* to 10.

3. What happens if you execute this goal?

   ```
   sum_of(List, 10).
   ```

   This goal says, "Give me a list whose elements add up to 10." Can Visual Prolog do this? If not, why not? (Hint: It's not possible to do arithmetic on unbound variables in Prolog.)

Tail Recursion Revisited

You probably noticed that ***length_of*** is not, and can't be, tail-recursive, because the recursive call is not the last step in its clause. Can you create a tail-recursive list-length predicate? Yes, but it will take some effort.

The problem with ***length_of*** is that you can't compute the length of a list until you've already computed the length of the tail. It turns out there's a way around this. You'll need a list-length predicate with three arguments.

- One is the list, which the computer will whittle away on each call until it eventually becomes empty, just as before.
- Another is a free argument that will ultimately contain the result (the length).
- The third is a counter that starts out as 0 and increments on each call.

When the list is finally empty, you'll unify the counter with the (up to then) unbound result.

```
                        /* Program ch07e03.pro */

DOMAINS
   list = integer*                  /* or whatever type you want to use */

PREDICATES
   length_of(list,integer,integer)

CLAUSES
   length_of([], Result, Result).
   length_of([_|T],Result,Counter):-
      NewCounter = Counter + 1,
      length_of(T, Result, NewCounter).

GOAL
      length_of([1, 2, 3], L, 0),             /* start with Counter = 0 */
      write("L=",L), nl.
```

This version of the ***length_of*** predicate is more complicated, and in many ways less logical, than the previous one. We've presented it merely to show you that, by devious means, ***you can often find a tail-recursive algorithm for a problem that seems to demand a different type of recursion***.

Exercises

1. Try both versions of ***length_of*** on enormous lists (lists with perhaps 200 to 500 elements). What happens? On long lists, how do they compare in speed?
2. What happens with the new version of ***length_of*** if you give the following goal?

   ```
   length_of(MyList, 5, 0).
   ```

   Hint: You are discovering a very important property of Prolog called *interchangeability of unknowns*. Not all Prolog predicates have it.
3. Rewrite ***sum_of*** to work like the new version of ***length_of***.

Another Example – Modifying the List

Sometimes you want to take a list and create another list from it. You do this by working through the list element by element, replacing each element with a computed value. For example, here is a program that takes a list of numbers and adds 1 to each of them:

```
                        /* Program ch07e04.pro */

DOMAINS
   list = integer*

PREDICATES
   add1(list,list)

CLAUSES
   add1([], []). /* boundary condition */
   add1([Head|Tail],[Head1|Tail1]):-              /* separate the head */
                                          /* from the rest of the list */
      Head1= Head+1,                     /* add 1 to the first element */
      add1(Tail,Tail1).       /* call element with the rest of the list */

goal
   add1([1,2,3,4], NewList).
```

To paraphrase this in natural language:

```
   To add 1 to all the elements of the empty list,
      just produce another empty list.

   To add 1 to all the elements of any other list,
      add 1 to the head and make it the head of the result, and then
      add 1 to each element of the tail and make that the tail of the
      result.
```

Load the program, and run the Test Goal with the specified goal `add1([1,2,3,4], NewList)`.

The Test Goal will return

```
   NewList=[2,3,4,5]
   1 Solution
```

Tail Recursion Again

Is ***add1*** tail-recursive? If you're accustomed to using Lisp or Pascal, you might think it isn't, because you think of it as performing the following operations:

1. Split the list into *Head* and *Tail*.
2. Add 1 to *Head*, giving *Head1*.
3. Recursively add 1 to all the elements of *Tail*, giving *Tail1*.
4. Combine *Head1* and *Tail1*, giving the resulting list.

This isn't tail-recursive, because the recursive call is not the last step.

But – and this is important – ***that is not how Prolog does it***. In Visual Prolog, ***add1*** *is* tail-recursive, because its steps are really the following:

1. Bind the head and tail of the original list to *Head* and *Tail*.
2. Bind the head and tail of the result to *Head1* and *Tail1*. (*Head1* and *Tail1* do not have values yet.)
3. Add 1 to *Head*, giving *Head1*.
4. Recursively add 1 to all the elements of *Tail*, giving *Tail1*.

When this is done, *Head1* and *Tail1* are *already* the head and tail of the result; there is no separate operation of combining them. So the recursive call really is the last step.

More on Modifying Lists

Of course, you don't actually need to put in a replacement for every element. Here's a program that scans a list of numbers and copies it, leaving out the negative numbers:

```
                        /* Program ch07e05.pro */

DOMAINS
   list = integer*

PREDICATES
   discard_negatives(list, list)

CLAUSES
   discard_negatives([], []).

   discard_negatives([H|T],ProcessedTail):-
      H < 0,                          /* If H is negative, just skip it */
      !,
      discard_negatives(T, ProcessedTail).

   discard_negatives([H|T],[H|ProcessedTail]):-
      discard_negatives(T, ProcessedTail).
```

For example, the goal

```
   discard_negatives([2, -45, 3, 468], X)
```

gives `X=[2, 3, 468]`.

And here's a predicate that copies the elements of a list, making each element occur twice:

```
doubletalk([], []).

doubletalk([H|T], [H, H|DoubledTail]) :-
   doubletalk(T, DoubledTail).
```

List Membership

Suppose you have a list with the names *John*, *Leonard*, *Eric*, and *Frank* and would like to use Visual Prolog to investigate if a given name is in this list. In other words, you must express the relation "membership" between two arguments: a name and a list of names. This corresponds to the predicate

```
member(name, namelist).          /* "name" is a member of "namelist" */
```

In Program `ch07e06.pro`, the first clause investigates the head of the list. If the head of the list is equal to the name you're searching for, then you can conclude that *Name* is a member of the list. Since the tail of the list is of no interest, it is indicated by the anonymous variable. Thanks to this first clause, the goal

```
member(john, [john, leonard, eric, frank])
```

is satisfied.

```
                        /* Program ch07e06.pro */

DOMAINS
   namelist = name*
   name = symbol

PREDICATES
   member(name, namelist)

CLAUSES
   member(Name, [Name|_]).
   member(Name, [_|Tail]):-
      member(Name,Tail).
```

If the head of the list is not equal to *Name*, you need to investigate whether *Name* can be found in the tail of the list.

In English:

```
Name is a member of the list if Name is the first element
of the list, or
Name is a member of the list if Name is a member of the tail.
```

The second clause of ***member*** relates to this relationship. In Visual Prolog:

```
member(Name, [_|Tail]) :- member(Name, Tail).
```

Exercises

1. Load Program `ch07e06.pro` and try the Test Goal the following goal:

   ```
   member(susan, [ian, susan, john]).
   ```

2. Add **domain** and **predicate** statements so you can use ***member*** to investigate if a number is a member of a list of numbers. Try several goals, including

   ```
   member(X, [1, 2, 3, 4]).
   ```

   to test your new program.

3. Does the order of the two clauses for the ***member*** predicate have any significance? Test the behavior of the program when the two rules are swapped. The difference appears if you test the goal

   ```
   member(X, [1, 2, 3, 4, 5])
   ```

   in both situations.

Appending One List to Another: Declarative and Procedural Programming

As given, the ***member*** predicate of Program `ch07e06.pro` works in two ways. Consider its clauses once again:

```
member(Name, [Name|_]).
member(Name, [_|Tail]) :- member(Name, Tail).
```

You can look at these clauses from two different points of view: declarative and procedural.

- From a declarative viewpoint, the clauses say

  Name `is a member of a list if the head is equal to` *Name*`;`
  `if not,` *Name* `is a member of the list if it is a member of the tail.`

- From a procedural viewpoint, the two clauses could be interpreted as saying:

  ```
  To find a member of a list, find its head;
     otherwise, find a member of its tail.
  ```

These two points of view correspond to the goals

```
member(2, [1, 2, 3, 4]).
```

and

```
member(X, [1, 2, 3, 4]).
```

In effect, the first goal asks Visual Prolog to check whether something is true; the second asks Visual Prolog to find all members of the list `[1,2,3,4]`. Don't be confused by this. The ***member*** predicate is the same in both cases, but its behavior may be viewed from different angles.

Recursion from a Procedural Viewpoint

The beauty of Prolog is that, often, when you construct the clauses for a predicate from one point of view, they'll work from the other. To see this duality, in this next example you'll construct a predicate to append one list to another. You'll define the predicate ***append*** with three arguments:

```
append(List1, List2, List3)
```

This combines *List1* and *List2* to form *List3*. Once again you are using recursion (this time from a procedural point of view).

If *List1* is empty, the result of appending *List1* and *List2* will be the same as *List2*. In Prolog:

```
append([], List2, List2).
```

If *List1* is not empty, you can combine *List1* and *List2* to form *List3* by making the head of *List1* the head of *List3*. (In the following code, the variable *H* is used as the head of both *List1* and *List3*.) The tail of *List3* is *L3*, which is composed of the rest of *List1* (namely, *L1*) and all of *List2*. In Prolog:

```
append([H|L1], List2, [H|L3]) :-
   append(L1, List2, L3).
```

The ***append*** predicate operates as follows: While *List1* is not empty, the recursive rule transfers one element at a time to *List3*. When *List1* is empty, the first clause ensures that *List2* hooks onto the back of *List3*.

Exercise

The predicate ***append*** is defined in Program `ch07e07.pro`. Load the program.

```
                    /* Program ch07e07.pro */

DOMAINS
   integerlist = integer*

PREDICATES
   append(integerlist,integerlist,integerlist)

CLAUSES
   append([],List,List).
   append([H|L1],List2,[H|L3]):-
      append(L1,List2,L3).
```

Now run it with the following goal:

```
append([1, 2, 3], [5, 6], L).
```

Now try this goal:

```
append([1, 2], [3], L), append(L, L, LL).
```

One Predicate Can Have Different Uses

Looking at ***append*** from a declarative point of view, you have defined a relation between three lists. This relation also holds if *List1* and *List3* are known but *List2* isn't. However, it also holds true if only *List3* is known. For example, to find which two lists could be appended to form a known list, you could use a goal of the form

```
append(L1, L2, [1, 2, 4]).
```

With this goal, Visual Prolog will find these solutions:

```
L1=[], L2=[1,2,4]
L1=[1], L2=[2,4]
L1=[1,2], L2=[4]
L1=[1,2,4], L2=[]
4 Solutions
```

You can also use ***append*** to find which list you could append to `[3,4]` to form the list `[1,2,3,4]`. Try giving the goal

```
append(L1, [3,4], [1,2,3,4]).
```

Visual Prolog finds the solution

```
L1=[1,2].
```

This ***append*** predicate has defined a relation between an *input set* and an *output set* in such a way that the relation applies both ways. Given that relation, you can ask

```
Which output corresponds to this given input?
```

or

```
Which input corresponds to this given output?
```

The status of the arguments to a given predicate when you call that predicate is referred to as a *flow pattern*. An argument that is bound or instantiated at the time of the call is an input argument, signified by (i); a free argument is an output argument, signified by (o).

The ***append*** predicate has the ability to handle any flow pattern you provide. However, not all predicates have the capability of being called with different flow patterns. When a Prolog clause is able to handle multiple flow patterns, it is known as an *invertible clause*. When writing your own Visual Prolog clauses, keep in mind that an invertible clause has this extra advantage and that creating invertible clauses adds power to the predicates you write.

Exercise

Amend the clauses defining ***member*** in Program `ch07e06.pro` and construct the clauses for a predicate ***even_member*** that will succeed if you give the goal

```
even_member(2, [1, 2, 3, 4, 5, 6]).
```

The predicate should also display the following result:

```
X=2
X=4
X=6
3 Solutions
```

given the goal

```
even_member(X, [1, 2, 3, 4, 5, 6]).
```

Finding All the Solutions at Once

In chapter **6**, we compared backtracking and recursion as ways to perform repetitive processes. Recursion won out because, unlike backtracking, it can pass

information (through arguments) from one recursive call to the next. Because of this, a recursive procedure can keep track of partial results or counters as it goes along.

But there's one thing backtracking can do that recursion *can't* do – namely, find all the alternative solutions to a goal. So you may find yourself in a quandary: You need all the solutions to a goal, but you need them all at once, as part of a single compound data structure. What do you do?

Fortunately, Visual Prolog provides a way out of this impasse. The built-in predicate ***findall*** takes a goal as one of its arguments and collects all of the solutions to that goal into a single list. ***findall*** takes three arguments:

- The first argument, VarName, specifies which argument in the specified predicate is to be collected into a list.
- The second, **mypredicate**, indicates the predicate from which the values will be collected.
- The third argument, ListParam, is a variable that holds the list of values collected through backtracking. Note that there must be a user-defined domain to which the values of ListParam belong.

Program `ch07e08.pro` uses ***findall*** to print the average age of a group of people.

```
                        /* Program ch07e08.pro */

DOMAINS
   name,address = string
   age = integer
   list = age*

PREDICATES
   person(name, address, age)
   sumlist(list, age, integer)

CLAUSES
   sumlist([],0,0).
   sumlist([H|T],Sum,N):-
      sumlist(T,S1,N1),
      Sum=H+S1, N=1+N1.

   person("Sherlock Holmes", "22B Baker Street", 42).
   person("Pete Spiers", "Apt. 22, 21st Street", 36).
   person("Mary Darrow", "Suite 2, Omega Home", 51).
```

```
GOAL
      findall(Age,person(_, _, Age),L),
      sumlist(L,Sum,N),
      Ave = Sum/N,
      write("Average=", Ave),nl.
```

The ***findall*** clause in this program creates a list *L*, which is a collection of all the ages obtained from the predicate ***person***. If you wanted to collect a list of all the people who are 42 years old, you could give the following subgoal:

```
findall(Who, person(Who, _, 42), List)
```

Before trying this, please note that it requires the program to contain a domain declaration for the resulting list:

```
slist = string*
```

Compound Lists

A list of integers can be simply declared as

```
integerlist = integer*
```

The same is true for a list of real numbers, a list of symbols, or a list of strings.

However, it is often valuable to store a combination of different types of elements within a list, such as:

```
[2, 3, 5.12, ["food", "goo"], "new"] /* Not correct Visual Prolog*/
```

Compound lists are lists that contain more than one type of element. You need special declarations to handle lists of multiple-type elements, because ***Visual Prolog requires that all elements in a list belong to the same domain.*** The way to create a list in Prolog that stores these different types of elements is to use functors, because ***a domain can contain more than one data type as arguments to functors***.

The following is an example of a domain declaration for a list that can contain an integer, a character, a string, or a list of any of these:

```
DOMAINS                              /* the functors are l, i, c, and s */
   llist = l(list); i(integer); c(char); s(string)
   list = llist*
```

The list

```
[ 2, 9, ["food", "goo"], "new" ]        /* Not correct Visual Prolog */
```

would be written in Visual Prolog as

```
[i(2), i(9), l([s("food"), s("goo")]), s("new")]
                                              /* Correct Visual Prolog */
```

The following example of ***append*** shows how to use this domain declaration in a typical list-manipulation program.

```
                        /* Program ch07e09.pro */

DOMAINS
   llist = l(list); i(integer); c(char); s(string)
   list = llist*

PREDICATES
   append(list,list,list)

CLAUSES
   append([],L,L).
   append([X|L1],L2,[X|L3]):-
      append(L1, L2, L3).

GOAL
      append([s(likes), l([s(bill), s(mary)])],[s(bill), s(sue)],Ans),
      write("FIRST  LIST: ", Ans,"\n\n"),
      append([l([s("This"),s("is"),s("a"),s("list")]),s(bee)],
             [c('c')],Ans2),
      write("SECOND LIST: ", Ans2, '\n', '\n').
```

Exercises

1. Write a predicate, ***oddlist***, that takes two arguments. The first argument is a list of integers, while the second argument returns a list of all the odd numbers found in the first list.
2. Write a predicate, ***real_average***, that calculates the average value of all the elements in a list of ***reals***.
3. Write a predicate that takes a compound list as its first argument and returns a second argument that is the list with all the sub-lists removed. This predicate is commonly known as ***flatten***, as it flattens a list of lists into a single list. For example, the call

   ```
   flatten([s(ed), i(3), l([r(3.9), l([s(sally)])])], r(4.21), X)
   ```

   returns

```
X = [s(ed), i(3), r(3.9), s(sally), r(4.21)]
1 Solution
```

which is the original list, flattened.

Parsing by Difference Lists

Program `ch07e10.pro` demonstrates parsing by *difference lists*. The process of parsing by difference lists works by reducing the problem; in this example we transform a string of input into a Prolog structure that can be used or evaluated later.

The parser in this example is for a very primitive computer language. Although this example is very advanced for this point in the tutorial, we decided to put it here because parsing is one of the areas where Visual Prolog is very powerful. If you do not feel ready for this topic, you can skip this example and continue reading the tutorial without any loss of continuity.

```
                        /* Program ch07e10.pro */

DOMAINS
   toklist  = string*

PREDICATES
   tokl(string,toklist)

CLAUSES
   tokl(Str,[H|T]):-
      fronttoken(Str,H,Str1),!,
      tokl(Str1,T).
   tokl(_,[]).

/* * * * * * * * * * * * * * * * * * * * * * * * * * *
 * This second part of the program is the parser *
 * * * * * * * * * * * * * * * * * * * * * * * * * * */

DOMAINS
   program   = program(statementlist)
   statementlist = statement*

/* * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Definition of what constitutes a statement  *
 * * * * * * * * * * * * * * * * * * * * * * * * * * */

   statement = if_Then_Else(exp,statement,statement);
               if_Then(exp,statement);
               while(exp,statement);
               assign(id,exp)
```

```
/* * * * * * * * * * * * * * *
 * Definition of expression  *
 * * * * * * * * * * * * * * */

   exp       = plus(exp,exp);
               minus(exp,exp);
               var(id);
               int(integer)

   id        = string

PREDICATES
   s_program(toklist,program)
   s_statement(toklist,toklist,statement)
   s_statementlist(toklist,toklist,statementlist)
   s_exp(toklist,toklist,exp)
   s_exp1(toklist,toklist,exp,exp)
   s_exp2(toklist,toklist,exp)

CLAUSES
   s_program(List1,program(StatementList)):-
      s_statementlist(List1,List2,StatementList),
      List2=[].

   s_statementlist([],[],[]):-!.
   s_statementlist(List1,List4,[Statement|Program]):-
      s_statement(List1,List2,Statement),
      List2=[";"|List3],
      s_statementlist(List3,List4,Program).
```

```
s_statement(["if"|List1],List7,if_then_else(Exp,Statement1,
            Statement2)):-
   s_exp(List1,List2,Exp),
   List2=["then"|List3],
   s_statement(List3,List4,Statement1),
   List4=["else"|List5],!,
   s_statement(List5,List6,Statement2),
   List6=["fi"|List7].
s_statement(["if"|List1],List5,if_then(Exp,Statement)):-!,
   s_exp(List1,List2,Exp),
   List2=["then"|List3],
   s_statement(List3,List4,Statement),
   List4=["fi"|List5].
s_statement(["do"|List1],List4,while(Exp,Statement)):-!,
   s_statement(List1,List2,Statement),
   List2=["while"|List3],
   s_exp(List3,List4,Exp).
s_statement([ID|List1],List3,assign(Id,Exp)):-
   isname(ID),
   List1=["="|List2],
   s_exp(List2,List3,Exp).

s_exp(LIST1,List3,Exp):-
   s_exp2(List1,List2,Exp1),
   s_exp1(List2,List3,Exp1,Exp).

s_exp1(["+"|List1],List3,Exp1,Exp):-!,
   s_exp2(List1,List2,Exp2),
   s_exp1(List2,List3,plus(Exp1,Exp2),Exp).
s_exp1(["-"|List1],List3,Exp1,Exp):-!,
   s_exp2(List1,List2,Exp2),
   s_exp1(List2,List3,minus(Exp1,Exp2),Exp).
s_exp1(List,List,Exp,Exp).

s_exp2([Int|Rest],Rest,int(I)):-
   str_int(Int,I),!.
s_exp2([Id|Rest],Rest,var(Id)):-
   isname(Id).
```

Load and run this program, then enter the following goal:

```
Goal tokl("b=2; if b then a=1 else a=2 fi; do a=a-1 while a;",Ans),
   s_program(Ans,Res).
```

Visual Prolog will return the program structure:

```
Ans=["b","=","2",";","if","b","then","a","=","1",
   "else","a","=","2","fi",";","do","a","=","a",
   "-","1","while","a",";"
   ],
Res=program([assign("b",int(2)),
             if_then_else(var("b"),assign("a",int(1)),
               assign("a",int(2))),
             while(var("a"),assign("a",minus(var("a"),int(1))))
             ])
1 Solution
```

The transformation in this example is divided into two stages: scanning and parsing. The ***tokl*** predicate is the scanner; it accepts a string and converts it into a list of tokens. All the predicates with names beginning in ***s_*** are parser predicates. In this example the input text is a Pascal-like program made up of Pascal-like statements. This programming language only understands certain statements: IF THEN ELSE, IF THEN, DO WHILE, and ASSIGNMENT. Statements are made up of expressions and other statements. Expressions are addition, subtraction, variables, and integers.

Here's how this example works:

1. The first scanner clause, ***s_program***, takes a list of tokens and tests if it can be transformed into a list of statements.
2. The predicate ***s_statementlist*** takes this same list of tokens and tests if the tokens can be divided up into individual statements, each ending with a semicolon.
3. The predicate ***s_statement*** tests if the first tokens of the token list make up a legal statement. If so, the statement is returned in a structure and the remaining tokens are returned back to ***s_statementlist***.
    a. The four clauses of the ***s_statement*** correspond to the four types of statements the parser understands. If the first ***s_statement*** clause is unable to transform the list of tokens into an IF THEN ELSE statement, the clause fails and backtracks to the next ***s_statement*** clause, which tries to transform the list of tokens into an IF THEN statement. If that clause fails, the next one tries to transform the list of tokens into a DO WHILE statement.
    b. If the first three ***s_statement*** clauses fail, the last clause for that predicate tests if the statement does assignment. This clause tests for assignment by testing if the first term is a symbol, the second term is "=", and the next terms make up a simple math expression.

4. The ***s_exp***, ***s_exp1***, and ***s_exp2*** predicates work the same way, by testing if the first terms are expressions and – if so – returning the remainder of the terms and an expression structure back to ***s_statement***.

See the Sentence Analyzer VPI\PROGRAMS\SEN_AN program on your disk for a more detailed example of parsing natural-language.

Summary

These are the important points covered in this chapter:

1. *Lists* are objects that can contain an arbitrary number of elements; you declare them by adding an asterisk at the end of a previously defined domain.
2. A list is a recursive compound object that consists of a head and a tail. The head is the first element and the tail is the rest of the list (without the first element). The tail of a list is always a list; the head of a list is an element. A list can contain zero or more elements; the empty list is written `[]`.
3. The elements in a list can be anything, including other lists; all elements in a list must belong to the same domain. The domain declaration for the elements must be of this form:

   ```
   DOMAINS
      elementlist = elements*
      elements    = ....
   ```

   where elements = one of the standard domains (**integer**, **real**, etc.) or a set of alternatives marked with different functors (`int(integer)`; `rl(real)`; `smb(symbol)`; etc.). You can only mix types in a list in Visual Prolog by enclosing them in compound objects/functors.
4. You can use separators (commas, [, and |) to make the head and tail of a list explicit; for example, the list

   ```
   [a, b, c, d]
   ```

   can be written as

   ```
   [a|[b, c, d]] or
   [a, b|[c, d]] or
   [a, b, c|[d]] or
   [a|[b|[c, d]]] or
   [a|[b|[c|[d]]]] or even
   [a|[b|[c|[d|[]]]]]
   ```

5. List processing consists of recursively removing the head of the list (and usually doing something with it) until the list is an empty list.
6. The classic Prolog list-handling predicates ***member*** and ***append*** enable you to check if an element is in a list and check if one list is in another (or append one list to another), respectively.
7. A predicate's *flow pattern* is the status of its arguments when you call it; they can be *input parameters* (i) – which are bound or instantiated – or *output parameters* (o), which are free.
8. Visual Prolog provides a built-in predicate, ***findall***, which takes a goal as one of its arguments and collects all of the solutions to that goal into a single list.
9. Because Visual Prolog requires that all elements in a list belong to the same domain, you use functors to create a list that stores different types of elements.
10. The process of *parsing by difference lists* works by reducing the problem; the example in this chapter transforms a string of input into a Prolog structure that can be used or evaluated later.

CHAPTER 8

Visual Prolog's Internal Fact Databases

In this chapter, we describe how you declare *internal fact databases* and how you can modify the contents of your fact databases.

An *internal fact database* is composed of facts that you can add directly into and remove from your Visual Prolog program at run time. You declare the predicates describing the fact databases in the **facts** sections of your program, and you use these predicates the same way you use the ones declared in the **predicates** section.

In Visual Prolog, you use the predicates ***assert, asserta, assertz*** to add new facts to the fact databases, and the predicates ***retract*** and ***retractall*** to remove existing facts. You can modify the contents of your fact databases by first retracting a fact and then asserting the new version of that fact (or a different fact altogether). The predicates ***consult/1*** and ***consult/2*** read facts from a file and asserts them into internal fact databases, and predicates ***save/1*** and ***save/2*** save the contents of internal fact databases to a file.

Visual Prolog treats facts belonging to fact databases differently from the way it treats normal predicates. Facts for the fact database predicates are kept in tables, which are easy to modify, while the normal predicates are compiled to binary code for maximum speed.

Declaring the Fact Databases

The keyword **facts** (it is synonymous to the obsolete keyword **database**) marks the beginning of the **facts** section declaration. A **facts** section consists of a sequence of declarations for predicates describing the correspondent internal fact database. You can add facts – but not rules – to a fact databases from the keyboard at run time with ***asserta*** and ***assertz***. Or, by calling the standard predicates ***consult***, you can retrieve the added facts from a disk file. The **facts** section looks something like in the following example.

```
DOMAINS
   name, address = string
   age = integer
   gender = male ; female

FACTS
   person(name, address, age, gender)

PREDICATES
   male(name, address, age)
   female(name, address, age)
   child(name, age, gender)

CLAUSES
   male(Name, Address, Age) :-
      person(Name, Address, Age, male).
   ...
```

In this example, you can use the predicate ***person*** the same way you use the other predicates, (***male***, ***female***, ***child***); the only difference is that you can insert and remove facts for ***person*** while the program is running.

There are two restrictions on using predicates declared in facts sections:

1. You can add them into fact databases as facts only – not as rules.
2. Facts in fact databases may not have free variables.

It is possible to declare several **facts** sections, but in order to do this, you must explicitly name each **facts** section.

```
FACTS - mydatabase
   myFirstRelation(integer)
   mySecondRelation(real, string)
   myThirdRelation(string)
   /* etc. */
```

This declaration of a facts section with the name ***mydatabase*** creates the ***mydatabase*** internal fact database. If you don't supply a name for a fact database, it defaults to the standard name ***dbasedom***. Notice that a program can contain the *local unnamed facts section* only if the program consists of the single compilation module, which is not declared to be a part of a project. (See *Modular Programming* on page 252). The Visual Development Environment executes a program file as a single compilation module only with the **Test Goal** utility. Otherwise, the unnamed facts section has to be declared *global*. This is done by preceding the keyword **facts** (or **database**) with the keyword **global**.

The names of predicates in a facts section must be unique within a module (source file); you cannot use the same predicate name in two different facts sections or in a facts section and in a predicates section. However, the predicates in the local named facts sections are private to the module in which they are declared, and do not interfere with local predicates in other modules.

Using the Fact Databases

Because Visual Prolog represents a relational database as a collection of facts, you can use it as a powerful query language for internal databases. Visual Prolog's unification algorithm automatically selects facts with the correct values for the known arguments and assigns values to any unknown arguments, while its backtracking algorithm can give all the solutions to a given query.

Accessing the Fact Databases

Predicates belonging to a fact database are accessed in exactly the same way as other predicates. The only visible difference in your program is that the declarations for the predicates are in a ***facts*** section instead of a ***predicates*** section. Given for instance the following:

```
DOMAINS
   name = string
   sex = char

FACTS
   person(name,sex)

CLAUSES
   person("Helen",'F').
   person("Maggie",'F').
   person("Suzanne",'F').
   person("Per",'M').
```

you can call **person** with the goal `person(Name,'F')` to find all women, or `person("Maggie",'F')` to verify that there is a woman called Maggie in your fact database.

You should be aware that, by their very nature, predicates in facts sections are always nondeterministic. Because facts can be added anytime at run time, the compiler must always assume that it's possible to find alternative solutions during backtracking. If you have a predicate in a facts section for which you'll never have more than one fact, you can override this by prefacing the declaration with

the keyword **determ** (or keyword **single** if the predicate must always have one and only one fact) to the declaration:

```
FACTS
   determ daylight_saving(integer)
```

You will get an error if you try to add a fact for a deterministic database predicate which already has a fact.

Updating the Fact Databases

Facts for database predicates can be specified at compile time in the **clauses** section, as in the example above. At run time, facts can be added and removed by using the predicates described below. Note that facts specified at compile time in the **clauses** section can be removed too, they are not different from facts added at run time.

Visual Prolog's standard database predicates ***assert***, ***asserta***, ***assertz***, ***retract***, ***retractall***, ***consult***, and ***save*** will all take one or two arguments. The optional second argument is the name of a facts section. We describe these predicates in the following pages. The notation ***"/1"*** and ***"/2"*** after each predicate name refers to the number of arguments that arity version of the predicate takes. The comments after the formats (such as `/* (i) */` and `/* (o,i) */` show the flow pattern(s) for that predicate.

Adding Facts at Run Time

At run time, facts can be added to the fact databases with the predicates: ***assert***, ***asserta*** and ***assertz***, or by loading facts from a file with ***consult***.

There are three predicates to add a single fact at runtime:

```
asserta(<the fact>)                                  /* (i) */
asserta(<the fact>, facts_sectionName)             /* (i, i) */

assertz(<the fact>)                                  /* (i) */
assertz(<the fact>, facts_sectionName)             /* (i, i) */

assert(<the fact>)                                   /* (i) */
assert(<the fact>, facts_sectionName)              /* (i, i) */
```

asserta asserts a new fact into the fact database before the existing facts for the given predicate, while ***assertz*** asserts a new fact after the existing facts for that predicate. ***assert*** behaves like ***assertz***.

The assertion predicates always know which fact database to insert the fact in, because the names of the database predicates are unique within a program (for global facts sections) or module (for local facts sections). However, you can use the optional second argument for type-checking purposes in order to ensure that you are working on the correct fact database.

The first of the following goals inserts a fact about Suzanne for the ***person*** predicate, after all the facts for ***person*** currently stored in the fact database. The second inserts a fact about Michael before all the currently stored facts for ***person***. The third inserts a fact about John after all the other ***likes*** facts in the fact database *likesDatabase*, while the fourth inserts a fact about Shannon in the same facts section, before all the other ***likes*** facts.

```
assertz(person("Suzanne", "New Haven", 35)).
asserta(person("Michael", "New York", 26)).
assertz(likes("John", "money"), likesDatabase).
asserta(likes("Shannon", "hard work"), likesDatabase).
```

After these calls the fact databases look as if you'd started with the following facts:

```
/* Internal fact database – dbasedom */
person("Michael", "New York", 26).
/* ... other person facts ... */
person("Suzanne", "New Haven", 35).

/* Internal fact database – likesDatabase */
likes("Shannon", "hard work").
/* ... other likes facts ... */
likes("John", "money").
```

Be careful that you don't accidentally write code asserting the same fact twice. The fact databases do not impose any kind of uniqueness, and the same fact may appear many times in a fact database. However, a uniqueness-testing version of ***assert*** is very easy to write:

```
FACTS - people
   person(string,string)

PREDICATES
   uassert(people)
```

```
CLAUSES
   uassert(person(Name,Address)):-
      person(Name,Address),
      !
      ;                                  % OR
      assert(person(Name,Address)).
```

Loading Facts from a File at Run Time

consult reads in a file, *fileName*, containing facts declared in a **facts** section and asserts them at the end of the appropriate fact database. ***consult*** takes one or two arguments:

```
consult(fileName) /* (i) */
consult(fileName, databaseName) /* (i, i) */
```

Unlike ***assertz***, if you call ***consult*** with only one argument (no facts section name), it will only read facts that were declared in the default (unnamed) ***dbasedom*** facts section.

If you call ***consult*** with two arguments (the file name and a facts section name), it will only consult facts from that named facts section. If the file contains anything other than facts belonging to the specified fact database, ***consult*** will return an error when it reaches that part of the file.

Keep in mind that the ***consult*** predicate reads one fact at a time; if the file has ten facts, and the seventh fact has some syntax error, ***consult*** will insert the first six facts into the facts section – then issue an error message.

Note that ***consult*** is only able to read a file in exactly the same format as ***save*** generates (in order to insert facts as fast as possible). There can be

- no upper-case characters except in double-quoted strings
- no spaces except in double-quoted strings
- no comments
- no empty lines
- no symbols without double quotes

You should be careful when modifying or creating such a file of facts with an editor.

Removing Facts at Run Time

retract unifies facts and removes them from the fact databases. It's of the following form:

```
retract(<the fact>[, databaseName])                    /* (i, i) */
```

retract will remove the first fact in your fact database that matches *<the fact>*, instantiating any free variables in *<the fact>* in the process. Retracting facts from a fact database is exactly like accessing it, with the side effect that the matched fact is removed. Unless the database predicate accessed by ***retract*** was declared to be deterministic, ***retract*** is nondeterministic and will, during backtracking, remove and return the remaining matching facts, one at a time. When all matching facts have been removed, ***retract*** fails.

Suppose you have the following **facts** sections in your program:

```
DATABASE
   person(string, string, integer)

FACTS - likesDatabase
   likes(string, string)
   dislikes(string, string)

CLAUSES
   person("Fred", "Capitola", 35).
   person("Fred", "Omaha", 37).
   person("Michael", "Brooklyn", 26).

   likes("John", "money").
   likes("Jane", "money").
   likes("Chris", "chocolate").
   likes("John", "broccoli").

   dislikes("Fred", "broccoli").
   dislikes("Michael", "beer").
```

Armed with these **facts** sections, you give Visual Prolog the following subgoals:

```
   retract(person("Fred", _, _)),                      /* 1 */
   retract(likes(_, "broccoli")),                      /* 2 */
   retract(likes(_, "money"), likesDatabase),          /* 3 */
   retract(person("Fred", _, _), likesDatabase).       /* 4 */
```

The first subgoal retracts the first fact for ***person*** about *Fred* from the default ***dbasedom*** fact database. The second subgoal retracts the first fact matching `likes(X, "broccoli")` from the fact database ***likesDatabase***. With both of these subgoals, Visual Prolog knows which fact database to retract from because the

names of the database predicates are unique: ***person*** is only in the default fact database, and ***likes*** is only in the fact database ***likesDatabase***.

The third and fourth subgoals illustrate how you can use the optional second argument for type checking. The third subgoal succeeds, retracting the first fact that matches `likes(_, "money")` from ***likesDatabase***, but the fourth cannot be compiled because there are no (and cannot be) ***person*** facts in the fact database ***likesDatabase***. The error message given by the compiler is:

```
506 Type error: The functor does not belong to the domain.
```

The following goal illustrates how you can obtain values from ***retract***:

```
GOAL
   retract(person(Name, Age)),
   write(Name, ", ", Age),nl,
   fail.
```

If you supply the name of a fact database as the second argument to ***retract***, you don't have to specify the name of the database predicate you're retracting from. In this case, ***retract*** will find and remove all facts in the specified fact database. Here is an example:

```
GOAL
   retract(X, mydatabase),
   write(X),
   fail.
```

Removing Several Facts at Once

retractall removes all facts that match <*the fact*> from your facts section, and is of the following form:

```
retractall(<the fact>[, databaseName])
```

retractall behaves as if defined by

```
retractall(X):- retract(X), fail.
retractall(_).
```

but it's considerably faster than the above.

As you can gather, ***retractall*** always succeeds exactly once, and you can't obtain output values from ***retractall***. This means that, as with ***not***, you must use underscores for free variables.

As with ***assert*** and ***retract***, you can use the optional second argument for type checking. And, as with ***retract***, if you call ***retractall*** with an underscore, it can remove all the facts from a given fact database.

The following goal removes all the facts about males from the database of ***person*** facts:

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

The next goal removes all the facts from the fact database ***mydatabase***.

```
retractall(_, mydatabase).
```

Saving a database of facts at runtime

save saves facts from a given **facts** section to a file. ***save*** takes one or two arguments:

```
save(fileName)                                    /* (i) */
save(fileName, databaseName)                    /* (i, i) */
```

If you call ***save*** with only one argument (no facts section name), it will save the facts from the default ***dbasedom*** database to the file *fileName*.

If you call ***save*** with two arguments (the file name and a facts section name), it will save all facts existing in the fact database ***databaseName*** to the named file.

Keywords Determining Fact Properties

Facts section declarations can use the following optional keywords:

```
facts [ - <databasename>]
   [nocopy] [{ nondeterm | determ | single }]
      dbPredicate ['(' [Domain [ArgumentName]]* ')']
```

The optional keywords **nondeterm**, **determ** or **single** declares the determinism mode of the declared database predicate ***dbPredicate***. Only one of them can be used. If the determinism mode for a database predicate is not declared explicitly, then the default **nondeterm** is accepted. Notice that the setting for **Default Predicate Mode** (specified in the VDE's dialog **Compiler Options**) does not effect onto the **nondeterm** default for database predicates.

nondeterm

Determines that the fact database can contain any number of facts for the database predicate *dbPredicate*. This is the default mode.

determ

Determines that the fact database at any time can contain no more than one fact of the database predicate ***dbPredicate***.

single

Determines that the fact database always contains one and only one fact of the database predicate ***dbPredicate***.

nocopy

Normally, when a database predicate is called to bind a variable to a string or a compound object, then the referenced data are copiedb from the heap to the Visual Prolog global stack (GStack). The **nocopy** declares that the data will not be copied and variables will reference directly to the fact's data stored in heap. This can considerably increase efficiency, but should be used carefully. If a copy was not made, the variable would point to garbage after the fact retraction was made.

global

Determines that the facts section is *global*. (See **Modular Programming** on page 252.) Notice that safe programming techniques require that you do not use global facts. Instead you can use global predicates handling local facts.

Facts declared with the keyword *nondeterm*.

The keyword **nondeterm** is the default mode for facts (database predicates) declared in facts sections. If none of the keywords **determ** or **single** are used in a fact declaration, the compiler applies **nondeterm** mode. Normally, by their very nature, database predicates are non-deterministic. Because facts can be added at any moment at runtime, the compiler must normally assume that it is possible to find alternative solutions during backtracking.

Facts declared with the keyword *determ*.

The keyword **determ** declares that the fact database can contain no more than one fact for the database predicate declared with this keyword. So if a program tries to assert a second such fact into the fact database, then Prolog will generate a runtime error. Therefore, the programmer should take special care asserting deterministic facts.

Preceding a fact with **determ** enables the compiler to produce better code, and you will not get non-deterministic warnings for calling such a predicate. This is useful for flags, counters, and other things that are essentially global characteristics.

Particularly note that when retracting a fact that is declared to be **determ**, the call to non-deterministic predicates ***retract/1*** and ***retract/2*** will be deterministic. So if you know that at any moment the fact database contains no more then one fact `counter` then you can write:

```
FACTS
   determ counter(integer CounterValue)

GOAL
   ...
   retract(counter(CurrentCount)),
   /* here Prolog will not set backtracking point */
   Count= CurrentCount + 1,
   assert(counter(Count)),
```

instead of

```
FACTS
   counter(integer CounterValue)

PREDICATES
   determ retract_d(dbasedom)

CLAUSES
   retract_d(X): - retract(X),!. % deterministic predicate

GOAL
   ...
   retract_d(counter(CurrentCount)),
   /* here Prolog will not set backtracking point */
   Count= CurrentCount + 1,
   asserta(counter(Count)),
```

Facts declared with the keyword *single*.

The keyword **single** declares that the fact database will always contain one and only one fact for the database predicate declared with the keyword **single**.

Since single facts must be already known when the program calls Goal; therefore, single facts must be initialized in clauses sections in a program source code. For example:

```
facts - properties
   single numberWindows_s(integer)

CLAUSES
   numberWindows_s(0).
```

Single facts cannot be retracted. If you try to retract a single fact then the compiler will generates an error. (In most cases the compiler can detect retracting of a single fact while the compile time.)

Since one instance of a single fact always exists, a single fact never fails if it is called with free arguments. For example, a following call

```
numberWindows_s(Num),
```

never fails if *Num* is a free variable. Therefore, it is convenient to use single facts in predicates declared with determinism mode **procedure**.

Predicates ***assert***, ***asserta***, ***assertz***, and ***consult*** applied to a **single** fact act similarly to couples of ***retract*** and ***assert*** predicates. That is, ***assert*** (***consult***) predicates change an existing instance of a fact to the specified one.

Preceding a fact with **single** enables the compiler to produce optimized code for accessing and updating of a fact. For example, for ***assert*** predicates applied to a **single** fact the compiler generates a code that works more effectively than a couple of ***retract*** and ***assert*** predicates applied to a **determ** fact (and all the more so than ***retract*** and ***assert*** predicates applied to a **nondeterm** fact).

Initialization of single facts with some domains (when you do not have the default value to use) are not trivial. The following information can be useful:

1. Notice that **binary** domain data can be initialized using text format of binary data. For example:

```
global domains
   font = binary

facts – properties
   single my_font(font)

clauses
   my_font($[00]).
```

2. Other important special case is initialization of single facts carrying the standard **ref** domain. The origin of **ref** domain is the domain for *database reference numbers* in Visual Prolog *external databases* (see *The External Database System* on page 369), but **ref** is also used in many predefined domains declared in tools and packages supplied with Visual Prolog. For instance, the fundamental VPI domain **window** is declared:

```
domains
   WINDOW = REF
```

Notice that for initialization of **ref** values you can use unsigned numbers preceded by the tilde `'~'` character. For example, you can write:

```
facts
   single mywin(WINDOW)

clauses
   mywin(~0).
```

Examples

1. This is a simple example of how to write a classification expert system using the fact database. The important advantage of using the fact database in this example is that you can add knowledge to (and delete it from) the program at run time.

```
                        /* Program ch08e01.pro */

DOMAINS
   thing = string
   conds = cond*
   cond  = string

FACTS – knowledgeBase
   is_a(thing, thing, conds)
   type_of(thing, thing, conds)
   false(cond)

PREDICATES
   run(thing)
   ask(conds)
   update

CLAUSES
   run(Item):-
      is_a(X, Item, List),
      ask(List),
      type_of(ANS, X, List2),
      ask(List2),
      write("The ", Item," you need is a/an ", Ans),nl.

   run(_):-
      write("This program does not have enough "),
      write("data to draw any conclusions."),
      nl.
```

```
ask([]).
ask([H|T]):-
   not(false(H)),
   write("Does this thing help you to "),
   write(H," (enter y/n)"),
   readchar(Ans), nl, Ans='y',
   ask(T).

ask([H|_]):-
   assertz(false(H)), fail.

is_a(language, tool, ["communicate"]).
is_a(hammer, tool, ["build a house", "fix a fender", "crack a nut"]).
is_a(sewing_machine, tool, ["make clothing", "repair sails"]).
is_a(plow, tool, ["prepare fields", "farm"]).

type_of(english, language, ["communicate with people"]).
type_of(prolog, language, ["communicate with a computer"]).

update:-
   retractall(type_of(prolog, language, ["communicate with a
   computer"])),
   asserta(type_of("Visual Prolog", language,
                           ["communicate with a personal computer"])),
   asserta(type_of(prolog, language,
                          ["communicate with a mainframe computer"])).
```

The following database facts could have been asserted using ***asserta*** or ***assertz***, or consulted from a file using ***consult***. In this example, however, they're listed in the **clauses** section.

```
   is_a(language, tool, ["communicate"]).
   is_a(hammer, tool, ["build a house", "fix a fender", "crack a nut"]).
   is_a(sewing_machine, tool, ["make clothing", "repair sails"]).
   is_a(plow, tool, ["prepare fields", "farm"]).

   type_of(english, language, ["communicate with people"]).
   type_of(prolog, language, ["communicate with a computer"]).
```

As the goal enter:

```
goal
   run(tool).
```

Respond to each question as if you were looking for some tool to communicate with a personal computer.

Now enter the following goal:

```
update, run(tool).
```

The **update** predicate is included in the source code for the program, to save you a lot of typing, and will remove the fact

```
type_of(prolog, language, ["communicate with a computer"])
```

from the fact database `knowledgeBase` and add two new facts into it:

```
type_of(prolog, language,
       ["communicate with a mainframe computer"]).
type_of("Visual Prolog", language,
["communicate with a personal computer"]).
```

Now respond to each question once again as if you were looking for some tool to communicate with a personal computer.

You can save the whole fact database `knowledgeBase` in a text file by calling the predicate ***save/2*** with names of a text file and a facts section as its arguments. For example, after the call to

```
save("mydata.dba", knowledgeBase),
```

the file `mydata.dba` will resemble the **clauses** section of an ordinary Visual Prolog program, with a fact on each line. You can read this file into memory later using the ***consult*** predicate:

```
consult("mydata.dba", knowledgeBase)
```

2. You can manipulate facts describing database predicates (facts declared in **facts** sections) as though they were terms.

   When you declare a facts section, Visual Prolog's compiler will internally generate a domain definition corresponding to the facts declaration. As an example, consider the declarations

```
FACTS - dba1           /* dba1 is the domain for these predicates */
   person(name, telno)
   city(cno, cname)
```

   Given these declarations, Visual Prolog's compiler internally generates the corresponding ***dba1*** domain:

```
DOMAINS
   dba1 = person(name, telno); city(cno, cname)
```

   This ***dba1*** domain can be used like any other predefined domain. For example, you could use the standard predicate ***readterm*** (which is covered in chapter **12**) to construct a predicate ***my_consult***, similar to the standard predicate ***consult***.

Program `ch08e02` illustrates one practical way you might use the facts section in an application. This example supposes that you have a screen handler predicate, which places text on the screen in predefined locations. A screen layout for the current screen display can be stored in the ***field*** and ***txtfield*** facts that are defined in the ***screen*** *facts section*. Several screen names can be stored in the correspondent ***screens*** fact database. At run time, the ***shiftscreen*** predicate can copy one of these stored screens to the ***screen*** fact database by first retracting all current data from the ***screen*** fact database, calling the ***screen*** predicate to get the layout information for the upcoming screen, then asserting the new screen's form into the ***screen*** facts section.

```
                    /* Program ch08e02.pro */

DOMAINS
   screenname, fname, type = symbol
   row,col,len = integer

FACTS - screenDescription
   field(fname, type, row, col, len)  /* Definitions of field on screen */
   txtfield(row, col, len, string)               /* Showing textfields */
   windowsize(row,col)

FACTS - screens
   screen(symbol,screenDescription)        /* Storing different screens */

PREDICATES
   shiftscreen(symbol)

CLAUSES
   shiftscreen(_):-
      retract(field(_,_,_,_,_)),
      fail.

   shiftscreen(_):-
      retract(txtfield(_,_,_,_)),
      fail.

   shiftscreen(_):-
      retract(windowsize(_,_)),
      fail.

   shiftscreen(Name):-
      screen(Name,Term),
      assert(Term),
      fail.

   shiftscreen(_).
```

```
GOAL
   shiftscreen(person).
```

Summary

1. Visual Prolog's *internal fact databases* are composed of the facts in your program that are grouped into *facts sections*. In facts sections you declare the user-defined database predicates used in these fact databases. The facts section declaration is started with the keyword **facts**.
2. You can name facts sections (which creates a corresponding internal domain); the default domain for (unnamed) facts sections is ***dbasedom***. Your program can have multiple facts sections, but each one must have a unique name. You can declare a given database predicate in only one **facts** section.
3. With the standard predicates ***assert***, ***asserta***, ***assertz***, and ***consult***, you can add facts to the fact databases at run time. You can remove such facts at run time with the standard predicates ***retract*** and ***retractall***.
4. The ***save*** predicates save facts from a fact database to a file (in a specific format). You can create or edit such a fact file with an editor, then insert facts from the file into the correspondent fact database of your running program with ***consult***.
5. Your program can call fact database predicates just as it calls any other predicates.
6. You can handle facts as terms when using domains internally generated for names of **facts** sections.

CHAPTER 9

Arithmetic and Comparison

Visual Prolog's arithmetic and comparison capabilities are similar to those provided in programming languages such as BASIC, C, and Pascal. Visual Prolog includes a full range of arithmetic functions; you have already seen some simple examples of Visual Prolog's arithmetic capabilities.

In this chapter we summarize Visual Prolog's built-in predicates and functions for performing arithmetic and comparisons, as well as a two-arity versions of a standard predicate used for random number generation. We'll also discuss comparison of strings and characters.

Arithmetic Expressions

Arithmetic expressions consist of *operands* (numbers and variables), *operators* (+, -, *, /, ***div***, and ***mod***), and parentheses. The symbols on the right side of the equal sign (which is the = predicate) in the following make up an arithmetic expression.

```
A = 1 + 6 / (11 + 3) * Z
```

Leading "0x" or "0o" signify hexadecimal and octal numbers, respectively, e.g.

```
0xFFF = 4095
   86 = 0o112 + 12
```

The value of an expression can only be calculated if all variables are bound at the time of evaluation. The calculation then occurs in a certain order, determined by the priority of the arithmetic operators; operators with the highest priority are evaluated first.

Operations

Visual Prolog can perform all four basic arithmetic operations (addition, subtraction, multiplication, and division) between integral and real values; the type of the result is determined according to Table **9**.1.

Table 9.1 Arithmetic Operations

Operand 1	Operator	Operand 2	Result
integral	+, -, *	integral	integral
real	+, -, *	integral	real
integral	+, -, *	real	real
real	+, -, *	real	real
integral or real	/	integral or real	real
integral	***div***	integral	integral
integral	***mod***	integral	integral

In case of mixed integral arithmetic, involving both signed and unsigned quantities, the result is signed. The size of the result will be that of the larger of the two operands. Hence, if an **ushort** and a **long** are involved the result is **long**; if an **ushort** and an **ulong** are involved the result is **ulong**.

Order of Evaluation

Arithmetic expressions are evaluated in this order:

1. If the expression contains sub-expressions in parentheses, the sub-expressions are evaluated first.
2. If the expression contains multiplication (*) or division (/, ***div*** or ***mod***), these operations are carried out next, working from left to right through the expression.
3. Finally, addition (+) and subtraction (-) are carried out, again working from left to right.

Hence, these are the operator precedence:

Table 9.2 Operator Precedence

Operator	Priority
`+ -`	1
`* / mod div`	2

- + (unary)	3

In the expression `A = 1 + 6/(11+3)*Z`, assume that *Z* has the value 4, since variables must be bound before evaluation.

1. (11 + 3) is the first sub-expression evaluated, because it's in parentheses; it evaluates to 14.
2. Then 6/14 is evaluated, because / and * are evaluated left to right; this gives 0.428571.
3. Next, 0.428571 * 4 gives 1.714285.
4. Finally, evaluating 1 + 1.714285 gives the value of the expression as 2.714285.

A will then be bound to 2.714285 that makes it a real value.

However, you should exercise some care when handling floating-point (***real***) quantities. In most cases they are not represented accurately and small errors can accumulate, giving unpredictable results. An example follows later in the chapter.

Functions and Predicates

Visual Prolog has a full range of built-in mathematical functions and predicates that operate on integral and real values. The complete list is given in Table 9.3

Table 9.4: Visual Prolog Arithmetic Predicates and Functions

Name	**Description**
X mod Y	Returns the remainder (modulo) of X divided by Y.
X div Y	Returns the quotient of X divided by Y.
abs(X)	If ***X*** is bound to a positive value *val*, `abs(X)` returns that value; otherwise, it returns `-1 * val`.
cos(X)	The trigonometric functions require that X be bound to
sin(X)	a value representing an angle in radians.
tan(X)	Returns the tangent of its argument.

arctan(X)	Returns the arc tangent of the real value to which X is bound.
exp(X)	e raised to the value to which X is bound.
ln(X)	Logarithm of X, base e.
log(X)	Logarithm of X, base 10.
sqrt(X)	Square root of X.
random(X)	Binds X to a random real; 0 <= X < 1.
random(X, Y)	Binds Y to a random integer; 0 <= Y < X.
round(X)	Returns the rounded value of X. The result still being a real
trunc(X)	Truncates X. The result still being a real
val(domain, X)	Explicit conversion between numeric domains.

Generating Random Numbers

Visual Prolog provides two standard predicates for generating random numbers. One returns a random ***real*** between 0 and 1, while the other returns a random ***integer*** between 0 and a given integer. Additionally, the random numbering sequence may be re-initialized.

random/1

This version of ***random*** returns a random real number that satisfies the constraints

```
0 <= RandomReal < 1.
```

random/1 takes this format:

```
random(RandomReal)                                    /* (o) */
```

random/2

This version of ***random*** takes two arguments, in this format:

```
random(MaxValue, RandomInt)                          /* (i, o) */
```

It binds *RandomInt* to a random integer that satisfies the constraints

```
0 <= RandomInt < MaxValue
```

random/2 is much faster than ***random/1*** because ***random/2*** only uses integer arithmetic.

randominit/1

randominit will initialize the random number generator and is of the following form:

```
randominit(Seed)                                        /* (i) */
```

The default random number seed value is generated as function from system time, and the *Seed* argument to **randominit** sets this seed value. The main use for ***randominit*** is to provide repeatable sequences of pseudo random numbers for statistical testing. Note that both the integer and floating point versions of **random** use the same seed and basic number generator.

Example

Program `ch9e01.pro` uses ***random/1*** to select three names from five at random.

```
                        /* Program ch9e01.pro */

PREDICATES
   person(integer, symbol)
   rand_int_1_5(integer)
   rand_person(integer)

CLAUSES
   person(1,fred).
   person(2,tom).
   person(3,mary).
   person(4,dick).
   person(5,george).

   rand_int_1_5(X):-
      random(Y),
      X=Y*4+1.
```

```
   rand_person(0):-!.
   rand_person(Count):-
      rand_int_1_5(N),
      person(N,Name),
      write(Name),nl,
      NewCount=Count-1,
      rand_person(NewCount).

GOAL
   rand_person(3).
```

Integer and Real Arithmetic

Visual Prolog provides predicates and functions for: modular arithmetic, integer division, square roots and absolute values, trigonometry, transcendental functions, rounding (up or down), and truncation. They are summarized in Table 9.3 and are explained on the following pages.

mod/2

mod performs the function *X modulo Y* (where *X* and *Y* are integers).

```
   X mod Y                                                    /* (i, i) */
```

The expression `Z = X mod Y` binds *Z* to the result. For example,

```
   Z = 7 mod 4                                        /* Z will equal 3 */
   Y = 4 mod 7                                        /* Y will equal 4 */
```

div/2

div performs the integer division *X/Y* (where *X* and *Y* are integers).

```
   X div Y                                                    /* (i, i) */
```

The expression `Z = X` div Y binds *Z* to the integer part of the result. For example,

```
   Z = 7 div 4                                        /* Z will equal 1 */
   Y = 4 div 7                                        /* Y will equal 0 */
```

abs/1

abs returns the absolute value of its argument.

```
   abs(X)                                                      /* (i) */
```

The expression `Z = abs(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Z = abs(-7)                                          /* Z will equal 7 */
```

cos/1

cos returns the cosine of its argument.

```
cos(X)                                                          /* (i) */
```

The expression `Z = cos(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Pi = 3.141592653,
Z = cos(Pi)                                        /* Z will equal -1 */
```

sin/1

sin returns the sine of its argument.

```
sin(X)                                                          /* (i) */
```

The expression `Z = sin(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example:

```
Pi = 3.141592653,
Z = sin(Pi)                                  /* Z will almost equal 0 */
```

tan/1

tan returns the tangent of its argument.

```
tan(X)                                                          /* (i) */
```

The expression `Z = tan(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Pi = 3.141592653,
Z = tan(Pi)                                  /* Z will almost equal 0 */
```

arctan/1

arctan returns the arc tangent of the real value to which *X* is bound.

```
arctan(X)                                                       /* (i) */
```

The expression `Z = arctan(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Pi = 3.141592653,
Z = arctan(Pi)                          /* Z will equal 1.2626272556 */
```

exp/1

exp returns *e* raised to the value to which *X* is bound.

```
exp(X)                                                      /* (i) */
```

The expression `Z = exp(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Z = exp(2.5)                            /* Z will equal 12.182493961 */
```

ln/1

ln returns the natural logarithm of *X* (base *e*).

```
ln(X                                                        /* (i) */
```

The expression `Z = ln(X)` binds *Z* (if it's free) to the result, or succeeds/fails if Z is already bound. For example,

```
Z = ln(12.182493961)                              /* Z will equal 2.5 */
```

log/1

log returns the base 10 logarithm of *X*.

```
log(X)                                                      /* (i) */
```

The expression `Z = log(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Z = log(2.5)                           /* Z will equal 0.39794000867 */
```

sqrt/1

sqrt returns the positive square root of *X*.

```
sqrt(X)                                                     /* (i) */
```

The expression `Z = sqrt(X)` binds *Z* (if it's free) to the result, or succeeds/fails if *Z* is already bound. For example,

```
Z = sqrt(25)                                    /* Z will equal 5 */
```

round/1

round returns the rounded value of *X*.

```
round(X)                                                    /* (i) */
```

round rounds *X* up or down to the nearest integral value of *X*, but performs no type conversion. For example,

```
Z1 = round(4.51)                              /* Z1 will equal 5.0 */
Z2 = round(3.40)                              /* Z2 will equal 3.0 */
```

Both *Z1* and *Z2* are floating point values following the above; only the fractional parts of the arguments to ***round*** have been rounded up or down.

trunc/1

trunc truncates *X* to the right of the decimal point, discarding any fractional part. Just like ***round***, ***trunc*** performs no type conversion.

```
trunc(X)                                                    /* (i) */
```

For example,

```
Z = trunc(4.7)                                 /* Z will equal 4.0 */
```

Again, *Z* is a floating-point number.

val/2

val provides general purpose conversion between the numeric domains, in cases where you want full control over the result of the operation. ***val*** observes any possible overflow condition. The format is

```
Result = val(returnDomain,Expr)
```

where *Expr* will be evaluated (if it's an expression), the result converted to *returnDomain* and unified with *Result*. Visual Prolog also has a ***cast*** function that will convert uncritically between any domains; this is described in chapter **10**.

Exercise

Use the trigonometric functions in Visual Prolog to display a table of sine, cosine, and tangent values on the screen. The left column of the table should contain angle values in degrees, starting at 0 degrees and continuing to 360 degrees in steps of 15 degrees.

Note: Because the trigonometric functions take values expressed in radians, you must convert radians to angles to obtain entries for the left column.

```
Degrees = Radians * 180/3.14159265...
```

Comparisons

Visual Prolog can compare arithmetic expressions as well as characters, strings, and symbols. The following statement is the Visual Prolog equivalent of "The total of *X* and 4 is less than 9 minus *Y*."

```
X + 4 < 9 - Y
```

The *less than* (<) relational operator indicates the relation between the two expressions, `X + 4` and `9 - Y`.

Visual Prolog uses *infix notation*, which means that operators are placed *between* the operands (like this: `X+4`) instead of preceding them (like this: `+(X,4)`).

The complete range of relational operators allowed in Visual Prolog is shown in Table 9.4.

Table 9.5: Relational Operators

Symbol	Relation
<	less than
<=	less than or equal to
=	equal
>	greater than
>=	greater than or equal to
<> or ><	not equal

Equality and the equal (=) Predicate

In Visual Prolog, statements like `N = N1 - 2` indicate a relation between three objects (*N*, *N*1, and 2), or a relation between two objects (*N* and the value of *N*1 - 2). If *N* is still free, the statement can be satisfied by binding *N* to the value of the expression `N1 - 2`. This corresponds roughly to what other programming languages call an *assignment* statement. Note that *N*1 must always be bound to a value, since it is part of an expression to be evaluated.

When using the equal predicate (=) to compare real values, you must take care to ensure that the necessarily approximate representation of real numbers does not lead to unexpected results. For example, the goal

```
7/3 * 3 = 7
```

will frequently fail (the exact outcome depends on the accuracy of the floating point calculations in use on your particular platform). Program `ch9e02.pro` illustrates another example:

```
                        /* Program ch9e02.pro */

PREDICATES
   test(real,real)

CLAUSES
   test(X,X):-!,
      write("ok\n").
   test(X,Y):-
      Diff = X-Y,
      write(X,"<>",Y,"\nX-Y = ",Diff,'\n').

GOAL
      X=47,
      Y=4.7*10,
      test(X,Y).
```

Except when running Prolog on the UNIX platform, where it behaves as one might expect, this prints:

```
47<>47
X-Y = 7.1054273576E-15
```

Therefore, when comparing two real values for equality you should always check that the two are within a certain range of one another.

Example

Program `ch9e03.pro` shows how to handle approximate equality; this is an iterative procedure for finding the square root in order to calculate the solutions to the quadratic equation:

```
A*X*X + B*X + C = 0
```

The existence of solutions depends on the value of the discriminant *D*, defined as follows:

```
D = B*B - 4*A*C.
```

- D > 0 implies that there are two unique solutions.
- D = 0 implies there is only one solution.
- D < 0 implies that there are no solutions if X is to take a real value (there can be one or two complex solutions).

```
                        /* Program ch9e03.pro */

PREDICATES
   solve(real, real, real)
   reply(real, real, real)
   mysqrt(real, real, real)
   equal(real, real)

CLAUSES
   solve(A,B,C):-
      D=B*B-4*A*C,
      reply(A, B, D), nl.

   reply(_,_,D):-
      D < 0,
      write("No solution"),
      !.
   reply(A,B,D):-
      D=0,
      X=-B/(2*A),write("x=", X),
      !.
   reply(A,B,D):-
      mysqrt(D,D,SqrtD),
      X1=(-B+SqrtD)/(2*A),
      X2 = (-B - SqrtD)/(2*A),
      write("x1 = ", X1," and x2 = ", X2).
```

```
mysqrt(X,Guess,Root):-
   NewGuess = Guess-(Guess*Guess-X)/2/Guess,
   not(equal(NewGuess,Guess)),
   !,
   mysqrt(X,NewGuess,Root).

   mysqrt(_,Guess,Guess).

equal(X,Y):-
   X/Y >0.99999,
   X/Y < 1.00001.
```

To solve the quadratic equation, this program calculates the square root of the discriminant, *D*. The program calculates square roots with an iterative formula where a better guess (*NewGuess*) for the square root of *X* can be obtained from the previous guess (*Guess*):

```
NewGuess = Guess-(Guess*Guess-X)/2/Guess
```

Each iteration gets a little closer to the square root of *X*. Once the condition `equal(X, Y)` is satisfied, no further progress can be made, and the calculation stops. Once this calculation stops, the program can solve the quadratic using the values *X1* and *X2*, where

```
X1 = (-B + sqrtD)/(2*A)
X2 = (-B - sqrtD)/(2*A)
```

Exercises

1. Load Program ch**9**e03.pro and try the Test Goal with the following goals:

   ```
   solve(1, 2, 1).
   solve(1, 1, 4).
   solve(1, -3, 2).
   ```

   The solutions should be

   ```
   x = -1
   No solution
   x1 = 2 and x2 = 1
   ```

   respectively.

2. The object of this exercise is to experiment with the ***mysqrt*** predicate in Program ch**9**e03.pro. To ensure that temporary calculations are monitored, add the following as the first subgoal in the first ***mysqrt*** clause:

   ```
   write(Guess).
   ```

To see the effect of this amendment, try this goal:

```
mysqrt(8, 1, Result).
```

Next, replace the equal clause with this fact:

```
equal(X, X).
```

and retry the goal. Experiment a little more with the properties of equal. For instance, try

```
equal(X, Y) :-
   X/Y < 1.1 , X/Y > 0.9.
```

Visual Prolog has a built-in square root function, **sqrt**. For example,

```
X = sqrt(D)
```

will bind *X* to the square root of the value to which *D* is bound. Rewrite Program `ch9e03.pro` using ***sqrt*** and compare the answers with those from the original version.

Comparing Characters, Strings, and Symbols

Besides numeric expressions, you can also compare single characters, strings and symbols. Consider the following comparisons:

```
'a' < 'b'                                   /* Characters */
"antony" > "antonia"                           /* Strings */
P1 = peter, P2 = sally, P1 > P2                /* Symbols */
```

Characters

Visual Prolog converts the `'a' < 'b'` to the equivalent arithmetic expression 97 < 98, using the corresponding ASCII code value for each character. You should be aware that only 7 bit ASCII comparisons should be relied upon (i.e. upper and lower case letters a-z, digits, etc.). 8 bit characters, used for a number of national characters, are not necessarily portable between the different platforms.

Strings

When two strings or symbols are compared, the outcome depends on a character-by-character comparison of the corresponding positions. The result is the same as you'd get from comparing the initial characters, unless those two characters are the same. If they are, Visual Prolog compares the next corresponding pair of characters and returns that result, unless those characters are also equal, in which case it examines a third pair, and so on. Comparison stops when two differing

characters are found or the end of one of the strings is reached. If the end is reached without finding a differing pair of characters, the shorter string is considered smaller.

The comparison `"antony" > "antonia"` evaluates to true, since the two symbols first differ at the position where one contains the letter *y* (ASCII value 79) and the other the letter *i* (ASCII value 69). In the same vein, the character comparison `"aa" > "a"` is true.

Similarly, the expression `"peter" > "sally"` would be false – as determined by comparing the ASCII values for the characters that make up *peter* and *sally*, respectively. The character *p* comes before *s* in the alphabet, so *p* has the lower ASCII value. Because of this, the expression evaluates to false.

Symbols

Symbols can't be compared directly because of syntax. In the preceding `P1 = peter, P2 ...` example, the symbol *peter* can't be compared directly to the symbol *sally*; they must be bound to variables to be compared, or written as strings.

CHAPTER 10

Advanced Topics

This is an advanced chapter; we expect that you have been working with the various examples earlier in this book and are now beginning to be an experienced Visual Prolog user. In this chapter, we illustrate how you can control the flow analysis by using the standard predicates ***free*** and ***bound***, reference domains, how to use them and how to separate them from the other domains. We also discuss more advanced topics about domains, including the ***binary*** domain, pointers to predicates and functions, and return values from functions. Finally, we look at error-handling, dynamic cutting, free type conversions and discuss some programming style issues that will improve your programs' efficiency.

The Flow Analysis

In a given predicate call, the known arguments are called *input* arguments (i), and the unknown arguments are called *output* arguments (o). The pattern of the input and output arguments in a given predicate call is called the *flow pattern*.

For example, if a predicate is to be called with two arguments, there are four possibilities for its flow pattern:

```
(i, i)   (i, o)   (o, i)   (o, o)
```

When compiling programs, Visual Prolog carries out a global flow analysis of the predicates. It starts with the main goal and then performs a pseudo-evaluation of the entire program, where it binds flow patterns to all predicate calls in the program.

The flow analysis is quite simple; you are actually carrying it out yourself when you write your program. Here are some examples:

```
GOAL
   cursor(R, C), R1 = R+1, cursor(R1, C).
```

In the first call to the ***cursor,*** the two variables *R* and *C* are free; this means that the ***cursor*** predicate will be called with the flow pattern `cursor(o,o)`. You know that the variables are free because this is the *first* time they've been encountered.

In the expression `R1=R+1`, the flow analyzer knows that the variable *R* is bound because it comes from the ***cursor*** predicate. If it were free, an error message would have been issued. *R1* will be a known argument after this call.

In the last call to ***cursor***, both of the variables *R1* and *C* have been encountered before, so they will be treated as input arguments; the call will have the flow pattern `cursor(i,i)`.

For each flow pattern that a user-defined predicate is called with, the flow analyzer goes through that predicate's clauses with the variables from the head set to either input or output (depending on the flow pattern being analyzed).

Here's an example illustrating this:

```
% To run this example you should in the VDE's Application Expert
% set Target settings to "DOS" and "Textmode
% and use Project | Run command

predicates
   changeattrib(Integer, Integer)

clauses
   changeattrib(NewAttrib, OldAttrib) :-
      attribute(OldAttrib), attribute(NewAttrib).

goal
      changeattrib(112, Old), write("Hello"),
      attribute(Old), write(" there"),
      readchar(_).
```

In the **goal** section, the first call to the predicate ***changeattrib*** is made with the flow pattern `changeattrib(i, o)` (because `112` is known, and `Old` is not). This implies that, in the clause for ***changeattrib***, the variable *NewAttrib* will be an input argument, and *OldAttrib* will be an output argument. Therefore, when the flow analyzer encounters the first subgoal `attribute(OldAttrib)`, the predicate ***attribute*** will be called with the flow pattern `attribute(o)`, while the second call to ***attribute*** will have the flow pattern `attribute(i)`. Finally, the call to ***attribute*** in the goal will have an input flow pattern, because *Old* came out of ***changeattrib***.

Compound Flow

If a predicate argument is a compound term it's also possible to have a compound flow pattern, where the same argument has both input and output flow. Suppose for instance that you have a database of information about countries. To enable

easy expansion with new data, it may well be desirable to contain each piece of information in its own domain alternative:

```
                        /* Program ch10e01.pro */

diagnostics

DOMAINS
   cinfo = area(string,ulong);
         population(string,ulong);
         capital(string,string)

PREDICATES
   country(cinfo)

CLAUSES
   country(area("Denmark",16633)).
   country(population("Denmark",5097000)).
   country(capital("Denmark","Copenhagen")).
   country(area("Singapore",224)).
   country(population("Singapore",2584000)).
   country(capital("Singapore","Singapore")).
```

The following depicts some of the different flow patterns *country* can be called with:

```
goal
   country(C),                                    % (o)
   country(area(Name,Area)),                      % area(o,o)
   country(population("Denmark",Pop)),            % population(i,o)
   country(capital("Singapore","Singapore")).      % (i)
```

Note that because in the last call all elements of the term are known, the flow pattern defaults to plain input `(i)`.

Load ch10e01.pro and try the goal example above with the Test Goal utility (see **Testing Language Tutorial Examples** on page 12). When you look at the VDE's **Messages** window, you will see the diagnostics output referencing the specified above flow variants in the table like this:

```
Predicate Name       Type     Determ   Size  Domains -- flowpattern
------ ---------------- -------- -------- ----- ---------- -----------
goal$000$country$1   local    nondtm   168    cinfo  -- o
goal$000$country$2   local    nondtm    72    cinfo  -- area(o,o)
goal$000$country$3   local    nondtm   108    cinfo  -- population(i,o)
goal$000$country$4   local    nondtm   416    cinfo  -- i
```

When the domains involved in a compound flow pattern are reference domains, the distinction between known and unknown arguments becomes blurred. We'll return to this example in the reference domain section later.

Specifying Flow Patterns for Predicates

It is sometimes convenient to specify flow patterns for your predicates. If you know, that your predicates will only be valid for special flow patterns, it is a good idea to specify flow patterns for your predicates because the flow analyzer will then catch any wrong usage of these predicates. After specifying the domains, a dash and the possible flow patterns can be given like in:

```
PREDICATES
   frame_text_mask(STRING,STRING,SLIST) - (i,o,o)(o,i,o)
```

Controlling the Flow Analysis

When the flow analyzer recognizes that a standard predicate is called with a nonexistent flow pattern, it issues an error message. This can help you identify meaningless flow patterns when you're creating user-defined predicates that call standard predicates.

For example, if you use:

```
   Z = X + Y
```

where the variable *X* or *Y* is not bound, the flow analyzer will give an error message saying that the flow pattern doesn't exist for that predicate. To control this situation, you can use the standard predicates ***free*** and ***bound***.

Suppose you want to create a predicate for addition, ***plus***, which can be called with all possible flow patterns. Program ch10e02.pro gives the code for such a predicate.

```
                        /* Program ch10e02.pro */

diagnostics

PREDICATES
   plus(integer, integer, integer)
   num(integer)
```

```
CLAUSES
   plus(X,Y,Z):-
      bound(X),bound(Y),Z=X+Y. /* (i,i,o) */
   plus(X,Y,Z):-
      bound(Y),bound(Z),X=Z-Y. /* (o,i,i) */
   plus(X,Y,Z):-
      bound(X),bound(Z),Y=Z-X. /* (i,o,i) */
   plus(X,Y,Z):-
      free(X),free(Y),bound(Z),num(X),Y=Z-X. /* (o,o,i) */
   plus(X,Y,Z):-
      free(X),free(Z),bound(Y),num(X),Z=X+Y. /* (o,i,o) */
   plus(X,Y,Z):-
      free(Y),free(Z),bound(X),num(Y),Z=X+Y. /* (i,o,o) */
   plus(X,Y,Z):-
      free(X),free(Y),free(Z),num(X),num(Y),Z=X+Y. /* (o,o,o) */

   % Generator of numbers starting from 0
   num(0).
   num(X):-
      num(A),
      X = A+1.
```

Reference Variables

When the flow analyzer has been through a clause, it checks that all output variables in the clause head have been bound in the clause body. ***If a variable is not bound in a clause, it needs to be treated as a reference variable.*** Here is an example demonstrating this dilemma:

```
predicates
   p(integer)

clauses
   p(X):- !.

goal
   p(V), V = 99, write(V).
```

In the Goal, the predicate *p* is called with an output pattern, but in the clause for *p*, the argument *X* is not bound. When the flow analyzer recognizes this, it will take a look at the domain corresponding to the variable. If the domain is already declared as a reference domain, there is no problem; if it is not, Visual Prolog tries to re-declare it internally as a reference domain. When it is possible (see the list of reference domains in the VDE's **Messages** window), the compiler

generates a warning. When it is impossible, for example, in programs containing several source modules, the error message is generated.

Note

Beginning with Visual Prolog v.5.2 the compiler, by default, generates an error on attempts to use Visual Prolog standard domains as reference. This example uses the basic domain **integer** as reference. Therefore, an attempt to invoke the **Test Goal** (with the default Visual Development Environment parameters) on this example will generate the error like "Basic domain becomes reference domain" (on **integer** domain). Consequently, to run the Test Goal on this example, you must explicitly specify to the compiler, that it should "*allow basic domains become reference domains*". You can do this with the command line compiler option:

```
-R+
```

To pass this option from the VDE into the command line compiler calls (while the **Test Goal),** you can specify `-R+` in the **Predefined Constants** edit control in the **Compiler Options** dialog. In this case the compiler will not generate the error.

However, we strongly recommend always explicitly declare all reference domains in **domains** sections.

When a variable is not bound in a clause, the clause cannot return a value. Instead, it will return a pointer to a *reference record* where the actual value can be inserted at a later time. This requires that the whole domain be treated equally; instead of just passing the values directly for some of the variables of that type, pointers to records will be passed through arguments belonging to the reference domain. When a compound domain becomes a reference domain, all of its subdomains must also become reference domains, because they must also be capable of containing free variables. If you just declare a compound domain to be a reference domain, the compiler will automatically know that all the subdomains are also reference domains.

Declaring Domains as Reference

When the flow analyzer encounters an unbound variable, it will only give a warning if the variable is not bound on return from a clause. If you ignore this warning, the compiler will treat the domain as a reference domain. The compiler will also try to declare all its subdomains as reference domains.

Because the code is treated the same for the whole domain, it is usually not a good idea to treat the basic domains as reference domains. Instead, you should declare a domain as being a reference domain to the desired base domain. For

instance, in the following code excerpt, the user-defined domain **refinteger** is declared to be a reference domain to the **integer** domain. All occurrences of **refinteger** types will be handled as reference domains, but any occurrence of other **integers** will still be treated as **integers**.

```
DOMAINS
   refinteger = reference integer

PREDICATES
   p(refinteger)

CLAUSES
   p(_).
```

Notice that if a base domain, for example **integer**, is treated as reference, then variables belonging to this base domain are treated as references on the base domain values. For example, integer variables will contain not integer values but references to integers. If predicates in some module (for example in a third-party C library) do not know that integer arguments are not ordinary integer values (but pointers to integer values), then calls to these predicates from other modules can be incorrect. As the result of this misunderstanding, such predicates may return wrong values or generate run-time errors. Therefore, by default, the compiler issues error message on attempts to use base domains as reference. This compiler checking can be switched OFF with the command line option `"-R"` (see the *Admonition* above). Therefore, you should never turn this checking OFF in projects calling external global functions that use basic domains in arguments, for example, if your project calls C functions from external libraries. (For instance, attempts to treat **integer** domain as reference domain in VPI based projects, will generally lead to run-time errors.

You should always explicitly declare the domains intended to be reference domains in the **domains** section. This is directly required in projects containing several modules - when global domains should handle unbound values, the compiler will not allow automatic conversion of these domains to reference domains. (Global domains and predicates are covered later in this chapter in the section *Modular Programming* on page 252.)

Notice that the following special basic domains are not allowed to become reference domains: ***file***, ***reg***, ***db_selector***, ***bt_selector***, and ***place***.

Reference Domains and the Trail Array

Because coercion’s and some extra unification are needed, reference domains will in general give a reduction in execution speed. However, some problems can

be solved far more elegant and efficiently when you use reference domains, and Visual Prolog has facilities to limit their effect.

When you use reference domains, Visual Prolog uses the trail array. The trail array is used to remember when reference variables become instantiated. This is necessary because if you backtrack to a point between the creation and the instantiation of a reference variable, it must be uninstantiated. This problem doesn't exist with ordinary variables, as their points of creation and instantiation are the same. Each instantiation recorded in the trail uses 4 bytes (the size of a 32-bit pointer). However, the trail usage is heavily optimized and no record will be placed there if there are no backtrack points between the variable's creation and instantiation.

The trail is automatically increased in size when necessary. The maximum size is 64K in the 16-bit versions of Visual Prolog, and practically unbounded in the 32-bit versions.

Using Reference Domains

The right way to use reference domains is to use them only in the few places where they are needed and to use non-reference domains for all the rest. Visual Prolog allows you to convert reference domains to non-reference domains whenever needed. For example, you can create a predicate that converts a reference integer to a non-reference integer with a single fact:

```
DOMAINS
   refint = reference integer

PREDICATES
   conv(refint,integer)

CLAUSES
   conv(X, X).
```

Visual Prolog does the conversion automatically when the same variable is used with both a reference domain and a non-reference domain, as it does in the clause when converting *X* from a ***refint*** to an **integer**. The above is only an explicit example; you don't need to write any special code to convert from reference to non-reference domains. Note that the reference variable needs to be instantiated to a value before it can be converted to the non-reference value. In the same way, if you try to convert a variable from one reference domain to another (such as from reference integers to reference characters), you should make sure the value is bound. Otherwise, Visual Prolog will issue an error message to the effect that free variables are not allowed in the context.

Pay attention to the automatic type conversions when you are creating a new free reference variable through a call to ***free*** predicate or creating a free variable with the ***equal*** predicate (=). Notice that when Visual Prolog's compiler creates a new unbound variable it needs to know to which domain this variable belongs. Otherwise, the compiler treats this variable as belonging to the first found reference domain; otherwise, if none reference domain is declared the compiler generates an error. (Notice that this behavior is subject to change in different Visual Prolog versions without any announcement.) That is, you can write:

```
predicates
   p(refinteger) – (o)

clauses
   p(X):-
      Y = X, bind_integer(X), ...
```

creating a new free reference variable with the ***equal*** predicate (=), but you should not create an unbound variable through a call to ***free*** with unknown domain like in this example:

```
goal
   free(X), ..., bind_integer(X), ...
```

With reference domains you can return variables that will receive values at a later point. You can also create structures where some places are left uninstantiated until later.

Example

To get a feel for how reference domains work, you should try some goals with the well-known predicates ***member*** and ***append***:

```
                        /* Program ch10e03.pro */

diagnostics

DOMAINS
   refinteger = integer
   reflist = reference refinteger*

PREDICATES
   member(refinteger, reflist)
   append(reflist, reflist, reflist)
```

```
CLAUSES
   member(X,[X|_]).
   member(X,[_|L]):-
      member(X,L).

   append([],L,L).
   append([X|L1],L2,[X|L3]):-
      append(L1, L2, L3).
```

Load this example program, and try the Test Goal with the following goals:

```
   member(1,L).                      % Give all lists where 1 is a member
   member(X,L), X=1.                                  % Same as before
   member(1,L), member(2,L).    % Lists starting with 1 and containing 2
   X=Y,member(X,L),member(Y,L), X=3.  % starting with X and containing Y
   member(1,L), append(L,[2,3],L1).
                       % Lists starting with X and closing with [... 2,3]
   append(L,L,L1), member(1,L).         % lists containing 1 as less twice
```

You will discover that the answers are what you logically expect them to be.

Flow Patterns Revisited

A reference variable may well be unbound and yet exist at the time it's used in a predicate call. In example ch10e01.pro, this will happen if for instance you want to find all countries having the same name as their capital, using e.g.

```
   samecaps:- country(capital(C,C)), write(C,'\n'), fail.
```

Here the variable *C* is used twice with output flow, but what the code really says is that the two variables in *capital* should share the same value once one of them becomes instantiated. Therefore, both variables are created and unified before the call. In order to do this their domain is converted to a reference domain, and both variables are in effect known at the time of call, giving a straight input flow pattern.

Note that, as said before, it's dangerous practice to let the standard domains become reference domains. If you want to use the above call, you should declare a suitable reference domain. However, this would create an overhead in all accesses of the *country* predicate, and it would probably be more efficient to use backtracking to find the special case where a country name and a capital are identical, by using.

```
   country(capital(Co,Ca)), Co = Ca, !, ...
```

Whether this is true or not depends on the size of the database, how many times you perform the call, how many other calls you have, how the arguments are used after the calls, etc.

Using Binary Trees with Reference Domains

In chapter **6**, you saw how binary trees could be used for fast and efficient sorting. However, sorting can actually be done in a more elegant fashion with reference domains. Because there is no way to change the leaves of a tree when they get new values, a lot of node copying occurs when the tree is created. When you are sorting large amounts of data, this copying can result in a memory overflow error. A reference domain can handle this by letting the leaves of the tree remain as free variables (where the subtrees will later be inserted). By using a reference domain this way, you don't need to copy the tree above the place where the new node is to be inserted.

Consider the predicate ***insert*** during the evaluation of the goal in ch10e04.pro. In this program, the ***insert*** predicate creates a binary tree using the reference domain ***tree***.

```
                        /* Program ch10e04.pro */

diagnostics

DOMAINS
   tree = reference t(val, tree, tree)
   val  = string

PREDICATES
   insert(val, tree)

CLAUSES
   insert(ID,t(ID,_,_)):-!.
   insert(ID,t(ID1,Tree,_)):-
      ID<ID1,
      !,
      insert(ID,Tree).
   insert(ID,t(_,_,Tree)):-
      insert(ID,Tree).

GOAL
      insert("tom",Tree),
      insert("dick",Tree),
      insert("harry",Tree),
      write("Tree=",Tree),
      nl, readchar(_).
```

The first subgoal, `insert("tom",Tree)`, will match with the first rule, and the compound object to which *Tree* is bound takes this form:

```
t("tom", _, _)
```

Even though the last two arguments in *t* are not bound, *t* carried is forward to the next subgoal evaluation:

```
insert("dick", Tree)
```

This, in turn, binds *Tree* to

```
t("tom", t("dick", _, _), _)
```

Finally, the subgoal

```
insert("harry", Tree)
```

binds *Tree* to

```
t("tom", t("dick", _, t("harry", _, _)), _)
```

which is the result returned by the goal.

Try to view details of this process using the Visual Prolog Debugger. Run the Debugger from the VDE with the **Project | Debug** command. When the Debugger window appears, choose the Debugger's menu command **View | Local Variables**, and use the **Run | Trace Into** command to inspect variables instantiation while the program execution. (For more instructions see the chapter *Debugging Prolog Programs* in the *Getting Started* and the chapter *The Debugger* in the *Visual Development Environment* manuals.) Notice that in the VDE in the Application Expert's **Target** tab you should select the following settings: **Platform** = `Windows32`, **UI Strategy** = `Textmode`.

Sorting with Reference Domains

In this section, we add onto the preceding binary tree example (ch10e04.pro) to show how you can isolate the use of reference domains and convert between reference and non-reference domains. The next example defines a predicate that is able to sort a list of values.

```
                        /* Program ch10e05.pro */

diagnostics

DOMAINS
   tree = reference t(val, tree, tree)
   val  = integer
   list = integer*

PREDICATES
   insert(integer,tree)
   instree(list,tree)
   nondeterm treemembers(integer,tree)
   sort(list,list)

CLAUSES
   insert(Val,t(Val,_,_)):-!.
   insert(Val,t(Val1,Tree,_)):-
      Val<Val1,!,
      insert(Val,Tree).
   insert(Val,t(_,_,Tree)):-
      insert(Val,Tree).

   instree([],_).
   instree([H|T],Tree):-
      insert(H,Tree),
      instree(T,Tree).

   treemembers(_,T):-
      free(T),!,fail.
   treemembers(X,t(_,L,_)):-
      treemembers(X,L).
   treemembers(X,t(Refstr,_,_)):-
      X = Refstr.
   treemembers(X,t(_,_,R)):-
      treemembers(X,R).

   sort(L,L1):-
      instree(L,Tree),
      findall(X,treemembers(X,Tree),L1).

GOAL
      sort([3,6,1,4,5],L),
      write("L=",L),nl.
```

In this example, note that reference domains are only used in the tree. All other arguments use non-reference domains. You can see this diagnostic in the VDE's **Messages** window like the following:

```
REFERENCE DOMAINS
   tree
   val
```

Functions and Return Values

Visual Prolog includes syntax for letting predicates be considered functions having a return value, rather than plain predicates using an output argument. The difference is a bit more than syntactic, however. Because return values are stored in registers, Prolog functions can return values to, and get return values from, foreign languages, but that is an issue covered in the chapter *Interfacing with Other Languages* on page 503.

A function declaration looks like an ordinary predicate declaration, except that the function name is prefixed by the domain it is returning:

```
predicates
   unsigned triple(unsigned)
```

However, the clauses for a function should have an extra last argument, to be unified with the return value upon success:

```
clauses
   triple(N,Tpl):- Tpl = N*3.

goal
      TVal = triple(6), write(TVal).
```

The return value need not be one of the standard domains; it can be any domain.

If you declare a function that does not take any arguments, you must supply an empty pair of brackets when calling it, in order to distinguish it from a string symbol. Given for instance a function to return the hour of the day

```
PREDICATES
   unsigned hour()

CLAUSES
   hour(H):- time(H,_,_,_).
```

you must call it like this:

```
   ..., Hour = hour(), ...
```

and not like this

```
      ..., Hour = hour, ...
```

as this will simply consider *hour* to be the text string "hour", following which the compiler will complain about type errors once you try to use *Hour*.

It is also recommended to supply an empty pair of brackets in the declaration of functions and predicates having no arguments. If not, confusing syntax errors may result from misunderstandings between predicate names and domain names, if they clash. If for instance you have a domain named *key* and you also have a predicate named *key*, then the declaration:

```
PREDICATES
   key
   mypred
```

can be interpreted in two ways: 1) a predicate named *key* and a predicate named *mypred*, 2) a predicate name *mypred* returning a *key*. If instead you write:

```
PREDICATES
   key()
   mypred()
```

all ambiguity is resolved.

Note that when a predicate is declared as a function, having a return value, it cannot be called as an ordinary Prolog predicate using the extra argument as an output argument; it must be called as a function. The reason for this is that functions store return values in registers, meaning that the code compiled before and in particular after a function call is different from the code around a call of an ordinary predicate. For the same reason, functions calling themselves are currently not tail recursive but this may change in future versions of Visual Prolog.

For instance, if you write a function *neg* to negate each element in a list, like this:

```
DOMAINS
   ilist = integer*

PREDICATES
   ilist neg(ilist)

CLAUSES
   neg([],[]).
   neg([Head|Tail],[NHead|NTail]):-
      NHead = -Head,
      NTail = neg(Tail).
```

it is not tail-recursive, while *neg* as a predicate:

```
DOMAINS
   ilist = integer*

PREDICATES
   neg(ilist,ilist)

CLAUSES
   neg([],[]).
   neg([Head|Tail],[NHead|NTail]):-
      NHead = -Head,
      neg(Tail,NTail).
```

is tail-recursive. Therefore, don't overdo the use of functions. Their primary aim is to enable you to get returned values from, and return values to, foreign language routines.

As a final note, you should be aware that functions with arithmetic return values must be deterministic if they take part in arithmetic expressions.

Determinism Monitoring in Visual Prolog

Most programming languages are deterministic in nature. That is, any set of input values leads to a single set of instructions used to produce output values. Furthermore in most languages, for example in C, a called function can produce only a single set of output values. On the contrary, Visual Prolog naturally supports non-deterministic inference based on non-deterministic predicates.

The object behind the determinism monitoring is to save run-time storage space. In fact, when a deterministic clause succeeds, the corresponding run-time stack space can be dispensed with at once, thus freeing the storage it occupied. There are a number of reasons why determinism should also concern programmers, most of them involving programming optimization.

Visual Prolog has a strongly typed determinism system. Visual Prolog's determinism checking system enforces the programmer to declare the following two behavior aspects of predicates (and facts):

1. Whether a call to a predicate can fail;
2. Number of solutions a predicate can produce.

In more Prolog program execution terms determinism mode defines the following properties of predicate behavior:

1. Can the predicate fail? (***F**ail* - ***F***)
2. Can the predicate succeed? (***S**ucceed* - ***S***)
3. Whether Visual Prolog will set a *Backtracking Point* to call of this predicate. (***B**acktrack **P**oint* - ***BP***)

According to these aspects of determinism the following *determinism modes of predicates* (rules) are supported in Visual Prolog:

Table 10.1: Determinism Modes of Predicates

	Number of Solutions can be produced		
	0	1	> 1
Cannot fail:	**erroneous** {}	**procedure** {***S***}	**multi** {***S***, ***BP***}
Can fail:|	**failure** {***F***}	**determ** {***F***, ***S***}	**nondeterm** {***F***, ***S***, ***BP***}

Using keywords from the above table in declarations of predicates and predicate domains the programmer can declare the six different determinism modes of predicates.

multi

{***S**ucceed*, ***B**acktrack**P**oint*}

The keyword **multi** defines non-deterministic predicates that *can backtrack* and generate multiple solutions. Predicates declared with the keyword **multi** *cannot fail* and therefore always produce at least one solution.

nondeterm

{***F**ail*, ***S**ucceed*, ***B**acktrack**P**oint*}

The keyword **nondeterm** defines non-deterministic predicates that *can backtrack* and generate multiple solutions. Predicates declared with the keyword nondeterm *can fail*.

procedure

{***S**ucceed*}

The keyword **procedure** defines predicates called *procedures*. Procedures always *succeed* (*cannot fail*) and do not produce *backtrack points*. That is procedures always have one and only one solution. (But run-time errors are possible.)

The compiler always checks and gives warnings for non-deterministic clauses in procedures.

The compiler (by default) checks and gives an error if it cannot guarantee that a procedure never fails.

determ

{***F**ail, **S**ucceed*}

The keyword **determ** defines deterministic predicates that can *succeed* or *fail*, but never *backtracks*. That is, predicates declared with the keyword **determ** have no more then one solution. When a predicate is declared with the keyword **determ**, the compiler always checks and gives a warning for non-deterministic clauses in the predicate. The keyword **determ** is also used in declarations of database predicates in facts sections.

erroneous

{}

A predicate declared with the keyword **erroneous** should not *succeed* (produce a solution) and should *never fail*.

Visual Prolog supplies the following **erroneous** built-in predicates: ***exit/0***, ***exit/1***, ***errorexit/0***, and ***errorexit/1***. These predicates have a run-time error effect. That is, if a call of such predicate is surrounded by the ***trap*** predicate, then calling of the predicate will jump back to this ***trap***. (Notice that in VPI (**V**isual **P**rogramming **I**nterface) every event handler is surrounded by an internal ***trap***.)

failure

{***F**ail*}

A predicate declared with the keyword **failure** should not produce a solution but it *can fail*.

The most common example of **failure** predicates is built-in predicate ***fail***.

When a predicate is declared with the keyword **failure**, the compiler by default checks and gives a warning for possible non-deterministic clauses in the predicate.

Calling of a **failure** predicate enforces a program to backtrack to the nearest backtracking point or interrupt the program with an effect identical to a run-time error. The following example demonstrates difference between **failure** and **erroneous** predicates:

```
predicates
   failure failure_1(INTEGER) - (i)
   erroneous erroneous_0()

clauses
   erroneous_0():- exit(). % This predicate cannot fail

   failure_1(0) :-                          %This predicate can fail
      erroneous_0().
   failure_1(_) :-
      fail.
```

Notice that all Visual Prolog's standard predicates have internal definition of determinism mode as **nondeterm**, **multi**, **determ**, **procedure**, **failure** or **erroneous**.

Applying this classification to declarations of database predicates that can be declared in **facts** sections we obtain the following table:

Table 10.2: Determinism Modes of Facts

	Number of Solutions can be produced		
	0	1	> 1
Cannot fail:		**single** {***S***}	
Can fail:|		**determ** {***F***, ***S***}	**nondeterm** {***F***, ***S***, ***BP***}

Using keywords from the above table in declarations of facts the programmer can declare three different determinism modes of facts (database predicates).

nondeterm

{***F****ail*, ***S****ucceed*, ***B****acktrack****P****oint*}

Determines that the fact database can contain any number of facts for the database predicate. This is the default determinism mode for database predicates.

determ

{***F**ail*, ***S**ucceed*}

Determines that the fact database at each moment can contain no more than one fact for the database predicate declared with the keyword **determ**.

single

{***S**ucceed*}

Determines that the fact database will always contain one and only one fact for the database predicate declared with the keyword **single**.

In this table term "*Cannot fail*" related to **single** facts means that called with free arguments a **single** database predicate always gives a solution (succeeds).

Visual Prologs Determinism Checking System

Visual Prolog offers unique determinism monitoring facilities based on declarations of types of predicates and facts. All Visual Prolog's standard predicates are internally defined as **nondeterm**, **multi**, **determ**, **procedure**, **failure** or **erroneous**.

By default, the compiler checks clauses of predicates and calculates determinism modes of all user-defined predicates; the compiler gives errors/warnings if it cannot guarantee that a predicate corresponds to a declared determinism mode:

1. By default, the compiler checks user-defined predicates declared with the keywords **determ**, **procedure**, **failure** or **erroneous**, and gives warnings for clauses that can result in a non-deterministic predicate. There are two kinds of non-deterministic clauses:

   a. If a clause does not contain a cut, and there are one or more clauses that can match the same input arguments for that flow pattern.

   b. If a clause calls a non-deterministic predicate, and that predicate call is not followed by a cut.

   Because of the second reason above, non-determinism has a tendency to spread like wildfire throughout a program unless (literally) cut off by one or more cuts.

2. By default, the compiler checks user-defined predicates declared with the keywords **procedure**, **multi**, and **erroneous** and gives warning/errors if it cannot guarantee that a predicate *never fails*.

   Take into account that the compiler is able to verify only necessary conditions for fail (not necessary and sufficient). Therefore, the compiler can sometimes

generate warnings/errors for predicates (declared with the keywords **multi**, **procedure** or **erroneous**) that, in fact, will never fail. For example,

```
domains
   charlist = char*

predicates
   procedure str_chrlist(STRING,CHARLIST) - (i,o)

clauses
   str_chrlist("",[]):-!.
   str_chrlist(Str,[H|T]):-
      frontchar(Str,H,Str1),
      str_chrlist(Str1,T).
```

The ***frontchar*** predicate can fail if the first parameter *Str* is an empty string. The compiler is not sophisticated enough to detect that *Str* in the second clause of ***str_chrlist*** cannot be empty string. For this example the compiler will generate a warning like "Possibility for failure in a predicate declared as **procedure**, **multi** or **erroneous**".

Checking of determinism modes of user-defined predicates can be switched OFF by unchecking the **Check Type of Predicates** (in VDE's **Compiler Options** dialog) or with the command-line compiler option `-upro-`, but it is a dangerous programming style. Instead you should modify the code to avoid these warnings. For instance, in this example you can reorder the clauses like:

```
str_chrlist(Str,[H|T]):-
   frontchar(Str,H,Str1),
   !,
   str_chrlist(Str1,T).
str_chrlist(_,[]):-!.
```

The declaration of procedures catches many small mistakes, like forgetting a catchall clause.

There are two rules that you must use when writing predicates declared with the keyword multi, procedure or erroneous:

- If any clause of a predicate can fail than the final catchall clause must be defined in the predicate (see the **str_chrlist** example above).
- For any possible (according to declared domains) set of input arguments, a clause, having a head, which matches this set, must exist. Otherwise, the compiler will generate a warning.

For instance, in the following example the third clause for the predicate ***p*** can be missed if the predicate is declared without the **procedure** keyword, but the compiler will detect this if the predicate is declared as **procedure**.

```
DOMAINS
   BOOLEAN = INTEGER % b_True = 1, b_False = 0

PREDICATES
   procedure p(BOOLEAN)

CLAUSES
   p(b_False):- !, ... .
   p(b_True): - !, ... .
   p(_): - dlg_error("An illegal argument value").
```

Notice that the compiler handles **erroneous** predicates in a special way providing possibility to use them in the final catchall clauses (for handling error situations) in predicates of other types. For instance, the catchall clause in the previous example can be rewritten as the following:

```
p(_): - errorexit(error_vpi_package_bad_data).
```

Predicates as Arguments

So far we have only seen predicate calls of a static nature. That is, the predicates being called as subgoals are specified statically in the source code. However, in many cases it may be desirable to call different predicates, depending on previous events and evaluations, from the same place, to avoid large-scale duplication of code. To this end Visual Prolog supports a notion of predicate values, you can declare a predicate domain, and pass *predicate values* (pointers to predicates) of that domain as variables.

The main usage of this feature in Visual Prolog is to pass event handler predicates to the VPI layer.

Predicate Values

Predicate values are predicates that can be treated as values in the sense that:

- They can be passed as parameters and returned from predicates and functions.
- They can be stored in facts.

- They can be held in variables.
- They can be compared for identity.

Of course, like "plain" predicates the predicate values can be called with appropriate arguments.

The predicate values are declared as instances of *predicate domains*.

If you have declared a predicate domain (for example, **pred_Dom**) in a domains section:

```
domains
   pred_dom = procedure (integer, integer) – (i,o)
```

then you can declare one or more predicate values (functions) as belonging to this predicate domain. The syntax for declarations of predicate values is:

```
predicates
   predValue_1: pred_Dom
   predValue_2: pred_Dom
```

Here ***predValue_1***, ***predValue_2*** are names of predicate values and **pred_Dom** is the predicate domain declared in the domains section. This predicate domain **pred_Dom** can then be specified as domain for arguments to other predicates. For example:

```
predicates
   variant_process(pred_Dom PredName, integer InVar, integer OutVar)
```

Then the programmer can pass these predicates (predicate values) ***predValue_1***, ***predValue_2*** as values of the *PredName* argument to predicate ***variant_process***. Predicate ***variant_process*** will hence be able to make a vectored call.

Predicate values may be used like almost any other entities in a program. In particular, they can appear as parts of compound terms, creating object oriented possibilities where each object carries with it a series of routines for its own management.

Predicate values do however differ from most other Visual Prolog values in the following respects:

- There exist no literals for predicate values.
- Predicate values have no persistent representation. (The text representation of a predicate value is simply a hexadecimal number (i.e. the value of a pointer to a memory address)).

You should take note, that predicate values are a fairly low-level mechanism. The actual value of such a predicate value is simply a code-address, and therefore it is valid only in the particular program where it was created. Hence, although you can store and retrieve predicate values via the fact databases, highly unexpected and quite possibly disastrous results will occur if you try to use a predicate value not originating in the current program.

Predicate values have many usages. One of the most important is for callbacks.

A callback is a predicate that is used to call back from some used entity to the user of this entity. For example:

- A call back from a server to a client, or
- A call back from a service to the service user, or
- A call back from a routine to a routine user.

Callbacks are normally used for one or both of the following purposes:

- To handle asynchronous events;
- To provide advanced/dynamic parameterization.

When dealing with asynchronous events a program registers a callback with some event source. Then this event source invokes the callback whenever an event occurs. "Data ready" in asynchronous communication is a typical example of such an asynchronous event. Another very typical example is a Windows event handler.

As an example of advanced/dynamic parameterization assume a tool that can create a certain kind of window. This window has the ability to change the shape of the cursor (mouse pointer) when it enters certain parts of the window. The window is however intended to work in many different situations, and therefore it cannot know which cursor to use in which parts of the window. In fact, the choice of cursor might depend on numerous things of which the window has no knowledge at all. Subsequently the window simply leaves the choice of cursor to the program that uses the window. And the way the window does this is by invoking a callback predicate. Via this callback predicate the window asks the surrounding program - which cursor to use, when the mouse enters a certain part of the window. Since the window makes such a callback each time the mouse enters a certain part it need not receive the same cursor each time, the choice of cursor can dynamically depend on things external to the window.

Predicate Domains

The declaration for a predicate domain is of the form:

```
[global] domains
   PredDom = DetermMode [ReturnDom] (ArgList) [- [FlowPattern]] [Language]
```

Here:

PredDom

Declares the name of the predicate domain.

DetermMode

Specifies the determinism mode with one of the following keywords:

```
{procedure | determ | nondeterm | failure | erroneous | multi}
```

Remember that the determinism mode must be specified in predicate domain declarations.

ReturnDom

Defines the domain for the return value, if you are declaring a predicate domain for functions.

ArgList

Defines domains for arguments in the form:

```
[ arg_1 [ , arg_2 ]* ]
```

Here ***arg_N*** is of the form:

```
Domain_Name [ Argument_Name ]
```

Here ***Domain_Name*** can be any standard or user-defined domain. The compiler just ignores the ***Argument_Name*** (if specified).

Attention: Brackets surrounding the argument list ***ArgList*** should always be given, even when the ***ArgList*** is empty.

FlowPattern

is of the form:

```
(flow [ , flow ]*)
```

where ***flow*** is `{ i | o | functor FlowPattern | listflow }`

and ***listflow*** is `'[' flow [ , flow ]* [ '|' { i | o | listflow } ] ']'`

The char `'-'` is obligatory before the ***FlowPattern*** (if ***FlowPattern*** is specified).

The flow pattern ***FlowPattern*** specifies how each argument is to be used. It must be the letter `'i'` for an argument with input flow or the letter `'o'` for one

with output flow. A *functor* and a flow pattern must be given for a compound term (e.g. `(i, o, myfunc(i,i), o)`) or a `listflow` (e.g. `[i, myfunc(i,o), o]`).

Attention: Only one flow pattern can be specified. If it is not specified explicitly, then the default flow pattern with all input arguments is accepted implicitly.

Notice that this implicit flow pattern can be the reason of error messages like "This flow pattern does not exist".

Language

is of the form:

```
language { pascal | stdcall | asm | c | syscall | prolog}
```

The ***Language*** specification tells the compiler, which kind of calling conventions to use. It is only required when predicate values, which are declared as instances of this predicate domain, will be passed to routines written in foreign languages. (For instance, in C, Delphi, etc.)

The default calling convention is **pascal**. Notice the difference with declarations of predicates, where the default is **prolog**.

Restriction. Predicate values having calling convention **prolog** cannot be called from variables. That is, if such predicate value is passed in a variable and the received predicate value is called on appropriate arguments, then the compiler generates an error.

Here we use:

- Square brackets to indicate optional items, and braces (curly brackets) to indicate that one of the items delimited by the symbols `'|'` must be used.
- Pair of single quotes to indicate that the character surrounded by them (namely `'|'`, `'['` and `']'`) is a part of the Visual Prolog language syntax.
- Asterisk symbol `'*'` to indicate arbitrary quantity of the immediately preceding item (zero or more times).

Comparison with declaration of predicates

In contradistinction to predicate declarations, in declarations of predicate domains:

1. Only one flow pattern can be specified.
2. If a flow pattern is not specified explicitly, then the default one with all input arguments is accepted implicitly.

3. The determinism mode ***DetermMode*** should always be specified before the argument list ***ArgList*** or before the return domain ***ReturnDom*** (in declarations of predicate domains for functions). The determinism mode cannot be (re-)declared before the flow pattern.
4. Brackets of the argument list should always be given (even when the list is empty).
5. The default calling convention for predicate domains is **pascal** while for predicates the default is **prolog**.

Examples

Hence, the declaration of a predicate domain for deterministic predicates (functions) taking an **integer** as input argument and returning an **integer** return value, would be

```
DOMAINS
   list_process = determ integer (integer) - (i)
```

This predicate domain is now known as *list_process*. To declare a function ***square*** as belonging to this predicate domain, the syntax is:

```
PREDICATES
   square: list_process
```

The clause for ***square*** is just like an ordinary clause, but as it's declared as a function it needs a return argument:

```
CLAUSES
   square(E,ES):- ES = E*E.
```

Elaborating on the above, declarations of the predicate domain *ilist_p* for deterministic predicates taking an integer list (*ilist*) and a predicate value (of *list_process* predicate domain) as input arguments, and an integer list as output argument, would hence be:

```
DOMAINS
   ilist = integer*
   list_process = determ integer (integer) - (i)
   ilist_p = determ (ilist,list_process,ilist) - (i,i,o)
```

Now look at the following program:

```
                        /* Program ch10e06.pro */

DOMAINS
   ilist = integer*
   list_process = determ integer (integer) - (i)
   ilist_p = determ (ilist,list_process,ilist) - (i,i,o)

PREDICATES
   list_square: list_process
   list_cube: list_process
   il_process: ilist_p

CLAUSES
   list_square(E,ES):- ES = E*E.
   list_cube(E,EC):- EC = E*E*E.

   il_process([],_,[]).
   il_process([Head|Tail],L_Process,[P_Head|P_Tail]):-
      P_Head = L_Process(Head),
      il_process(Tail,L_Process,P_Tail).

GOAL
      List = [-12,6,24,14,-3],
      il_process(List,list_square,P_List1),
      write("P_List1 = ",P_List1,'\n'),
      il_process(List,list_cube,P_List2),
      write("P_List2 = ",P_List2,'\n').
```

This declares two functions: ***list_square*** and ***list_cube***, belonging to the *list_process* predicate domain, and a predicate ***il_process*** creating a new integer list by applying the list element-processing predicate (which is passed as a predicate value in the *L_Process* argument) to each element of the original list. Note that the domain declaration *ilist_p* is only included for illustration; ***il_process*** could equally well been declared using:

```
      PREDICATES
         il_process(ilist,list_process,ilist)
```

since it is not referred to as a variable.

With the goal shown, ***il_process*** is called twice, first creating a list of squares by applying the ***list_square*** function, and then a list of cubes by applying the ***list_cube*** function. Compile and run this program, and you will get:

```
P_List1 = [144,36,576,196,9]
P_List2 = [-1728,216,13824,2744,-27]
```

Make sure you understand the complexities of this, and, when you do, make sure you don't abuse it. It's all too easy to create totally unreadable programs. Program ch10e07, which is a somewhat elaborated version of ch10e06, illustrates the concept taken to a reasonable limit:

```
                        /* Program ch10e07.pro */

DOMAINS
   ilist = integer*
   list_process = determ integer (integer) - (i)
   list_p_list = list_process*
   elem_process = determ (integer,integer,integer) - (i,i,o)
   elem_p_list = elem_process*

PREDICATES
   list_same: list_process
   list_square: list_process
   list_cube: list_process

   elem_add: elem_process
   elem_max: elem_process
   elem_min: elem_process

   il_process(ilist,list_process,ilist)
   il_post_process(ilist,elem_process,integer)

   apply_elemprocess(ilist,elem_p_list)
   apply_listprocess(ilist,list_p_list,elem_p_list)

   string lpname(list_process)
   string epname(elem_process)

CLAUSES
   lpname(list_same,list_same).        % Map predicate values to predicate
   functors
   lpname(list_square,list_square).
   lpname(list_cube,list_cube).

   epname(elem_add,elem_add).
   epname(elem_min,elem_min).
   epname(elem_max,elem_max).

   elem_add(E1,E2,E3):- E3 = E1+E2.
   elem_max(E1,E2,E1):- E1 >= E2, !.
   elem_max(_,E2,E2).
   elem_min(E1,E2,E1):- E1 <= E2, !.
   elem_min(_,E2,E2).
```

```
   list_same(E,E).
   list_square(E,ES):- ES = E*E.
   list_cube(E,EC):- EC = E*E*E.

   il_process([],_,[]).
   il_process([Head|Tail],E_Process,[P_Head|P_Tail]):-
      P_Head = E_Process(Head),
      il_process(Tail,E_Process,P_Tail).

   il_post_process([E],_,E):-!.
   il_post_process([H|T],L_Process,Result):-
      il_post_process(T,L_Process,R1),
      L_Process(H,R1,Result).

   apply_elemprocess(_,[]).
   apply_elemprocess(P_List,[E_Process|E_Tail]):-
      il_post_process(P_List,E_Process,PostProcess),
      NE_Process = epname(E_Process),
      write(NE_Process,": Result = ",PostProcess,'\n'),
      apply_elemprocess(P_List,E_Tail).

   apply_listprocess(_,[],_).
   apply_listprocess(I_List,[L_Process|L_Tail],E_List):-
      il_process(I_List,L_Process,P_List),
      NL_Process = lpname(L_Process),
      write('\n',NL_Process,":\nProcessed list = ",P_List,
         "\nPost-processing with:\n"),
      apply_elemprocess(P_List,E_List),
      apply_listprocess(I_List,L_Tail,E_List).

GOAL
      List = [-12,6,24,14,-3],
      write("Processing ",List," using:\n"),nl,
      apply_listprocess(List,[list_same,list_square,list_cube],
         [elem_add,elem_max,elem_min]).
```

Among other things, this program illustrates the use of lists of predicate values. If you run it, you'll get the following output:

```
Processing [-12,6,24,14,-3] using:

list_same:
Processed list = [-12,6,24,14,-3]
Post-processing with:
elem_add: Result = 29
elem_max: Result = 24
elem_min: Result = -12
```

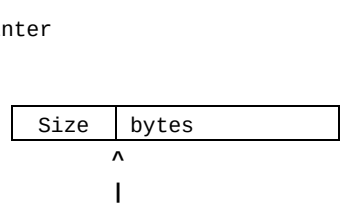
```
list_square:
Processed list = [144,36,576,196,9]
Post-processing with:
elem_add: Result = 961
elem_max: Result = 576
elem_min: Result = 9

list_cube:
Processed list = [-1728,216,13824,2744,-27]
Post-processing with:
elem_add: Result = 15029
elem_max: Result = 13824
elem_min: Result = -1728
```

The Binary Domain

Visual Prolog has a special ***binary*** domain for holding binary data, as well as special predicates for accessing individual elements of binary terms. The main use for binary terms is to hold data that has no reasonable representation otherwise, such as screen bitmaps and other arbitrary memory blocks. There are separate predicates for reading binary terms from, and writing them to, files. These will be discussed in chapter **12** "*Writing, Reading, and Files*". With the help of the built-in conversion predicate ***term_bin***, conversion from things such as binary file-headers to Prolog terms is a snap, and binary items going into or coming out of foreign language routines are easily handled. Finally arrays may also be implemented easily and efficiently.

Binary terms is a low-level mechanism, whose primary aim is to allow easy and efficient interfacing to other, non-logical, objects, and foreign languages. To this end, binary terms do not behave like other Prolog terms with respect to backtracking. Binary terms will be released if you backtrack to a point previous to their creation, but if you don't backtrack that far any changes done to the term will not be undone. We will illustrate this in the example program at the end of this section.

Implementation of binary terms



A binary term is simply a sequence of bytes, preceded by a word (16-bit platforms) or dword (32-bit platforms), holding its size.

When interfacing to other languages, you should be aware that a term of binary type (the variable passed in the foreign language function call) points to the actual contents, not the size. The *Size* field includes the space taken up by the field itself. Binary terms are subject to the usual 64K size restriction on 16-bit platforms.

Text syntax of Binary Terms

Binary terms can be read and written in text format, and also specified in source form in Visual Prolog source code. The syntax is:

```
$[b1,b2,...,bn]
```

where *b1*, *b2*, etc. are the individual bytes of the term. When a binary term is specified in source form in a program, the bytes may be written using any suitable unsigned integral format: decimal, hexadecimal, octal, or as a character. However, the text-representation of binary terms created and converted at run-time is fixed hexadecimal, with no leading "0x" on the individual bytes. Program `ch10e08.pro` illustrates this:

```
                        /* Program ch10e08.pro */

GOAL
   write("Text form of binary term: ",$['B',105,0o154,0x73,'e',0],'\n').
```

Load and run this program, and Visual Prolog will respond

```
Text form of binary term: $[42,69,6C,73,65,00]
```

You should hence be careful if you use e.g. ***readterm*** to read a binary term at runtime.

Creating Binary Terms

Below we discuss the standard predicates Visual Prolog includes, for creation of binary terms.

makebinary/1

makebinary creates and returns a binary term with the number of bytes specified, and sets its contents to binary zero.

```
..., Bin = makebinary(10), ...
```

The number of bytes should be the net size, excluding the size of the size field.

makebinary/2

makebinary is also available in a two-arity version, allowing specification of an element size.

```
..., USize = sizeof(unsigned), Bin = makebinary(10,USize), ...
```

This creates a binary term with a size given by the number of elements (10 in the above example) multiplied by the element-size (`sizeof(unsigned)` in the above), and sets its contents to zero.

composebinary/2

composebinary creates a binary term from an existing pointer and a length. It's useful in converting pointers to arbitrary blocks of memory returned by foreign language functions. The ***composebinary*** takes two arguments, and returns a binary.

```
..., Bin = composebinary(StringVar,Size), ...
```

composebinary takes a copy of the *StringVar* given as input, so changes to the created binary term *Bin* will not affect *StringVar*, and vice versa.

getbinarysize/1

getbinarysize returns the net size (in bytes) of the binary term, excluding the size field in front of the data.

```
..., Size = getbinarysize(Bin), ...
```

Accessing Binary Terms

There are eight predicates for accessing binary terms, four for setting entries and four for getting entries. Both groups perform range checking based on the size of the binary term, the index specified, and the size of the desired item (***byte***, ***word***, ***dword***, or ***real***). It is an error to try to get or set entries outside the range of the binary term.

Take special note that indices (element numbers) are 0-relative; the first element of a binary term has index 0, and the last element of an *N*-element binary term has index *N-1*.

get*entry/2

get*entry is either ***getbyteentry***, ***getwordentry***, ***getdwordentry***, or ***getrealentry***, accessing and returning the specified entry as a ***byte***, ***word***, ***dword***, or ***real***, respectively.

```
..., SomeByte = getbyteentry(Bin,3), ...
```

set*entry/3

set*entry is the counterpart to ***get*entry***, setting the specified ***byte***, ***word***, ***dword***, or ***real*** entry.

```
..., setbyteentry(Bin,3,SomeByte), ...
```

Unifying Binary Terms

Binary terms may be unified just like any other term, in clause heads or using the ***equal*** predicate `'='`:

```
..., Bin1 = Bin2, ...
```

If either of the terms is free at the time of unification, they will be unified and point to the same binary object. If both are bound at the time of unification, they will be compared for equality.

Comparing Binary Terms

The result of comparing two binary terms is as follows:

If they are of different sizes, the bigger is considered larger; otherwise, they're compared byte by byte, as unsigned values; comparison stops when two

differing bytes are found, and the result of their comparison is also the result of the comparison of the binary terms.

For instance, `$[1,2]` is bigger than `$[100]`, and smaller than `$[1,3]`.

Example

Program `ch10e09.pro` demonstrates a number of aspects of binary terms.

```
                        /* Program ch10e09.pro */

predicates
   comp_unify_bin
   comp_unify(binary,binary)
   access(binary)
   error_handler(integer ErrorCode, unsigned Index, binary)

clauses
   comp_unify_bin:-
      Bin = makebinary(5),
      comp_unify(Bin,_),
      comp_unify($[1,2],$[100]),
      comp_unify($[0],Bin),
      comp_unify($[1,2,3],$[1,2,4]).

   comp_unify(B,B):-!,
      write(B," = ",B,'\n').
   comp_unify(B1,B2):-
      B1 > B2,!,
      write(B1," > ",B2,'\n').
   comp_unify(B1,B2):-
      write(B1," < ",B2,'\n').

   access(Bin):-
      setwordentry(Bin,3,255),
      fail.                    % Changes are not undone when backtracking!
   access(Bin):-
      Size = getbinarysize(Bin),
      X = getwordentry(Bin,3),
      write("\nSize=",Size," X=",X," Bin=",Bin,'\n').

   error_handler(ErrorCode, Index, Bin):-
      write("Error ",ErrorCode," setting word index ",Index," of ",Bin,
         '\n', "Press any char to terminate execution\n"),
      readchar(_).
```

```
goal
   % Illustrate comparison and unification of binary terms
      comp_unify_bin,

   % Allocate a binary chunk of 4 words
      WordSize = sizeof(word),
      Bin = makebinary(4,WordSize),
      access(Bin),

   % Illustrate range checking; element numbers are 0-relative
      write("Run-time error due to wrong index:\n"),
      Index = 4,
      trap(setwordentry(Bin,Index,0),E, error_handler(E,Index,Bin)).
```

This example uses the ***trap*** predicate, which will be discussed in the section about error handling below.

Converting Terms to Binary Terms

A compound term may have its arguments scattered all over memory, depending on what domains they belong to. Simple types are stored directly in the term record itself, while complex types (those accessed via a pointer, and allocated separately on the global stack) will not necessarily be anywhere near the term they appear in. This is a problem if a term has to be sent out of a program, so to speak, as there is no way make an explicit copy of its contents. Unifying a term variable with another variable will only take a copy of the pointer to the term.

Using ***term_str*** (discussed in chapter **13**), it is possible to convert the term to a string and back again, but this is rather inefficient when all that's needed is a copy of the term's contents.

***term_bin* solves this problem.**

term_bin/3

term_bin will convert between a term of any domain and a block of binary data, holding the term's contents as well as pointer fixup information. The pointer fixup information will be applied to the binary data when converted back to a term, allowing recreation of any pointers to complex terms the term contains.

term_bin looks like this:

```
term_bin(domain,Term,Bin)                          /* (i,i,o) (i,_,i) */
```

The *domain* is the domain the *Term* belongs, or should belong, to, and *Bin* is a binary term holding the *Term*'s contents.

Example

Program `ch10e11.pro` demonstrates conversion between a term and its binary representation. The domains and alignment have been explicitly chosen to ease description, as they would otherwise differ between 16-bit and 32-bit platforms. Alignment of terms is usually only relevant when interfacing to foreign languages, and is fully described in the chapter **18**.

```
                        /* Program ch10e11.pro */

DOMAINS
   dom = align dword cmp(string,short)

GOAL
   T = cmp("Bilse",31),
   term_bin(dom,T,B),
   write("Binary form of ",T,":\n",B),
   term_bin(dom,T1,B),
   write("\nConverted back: ",T1,'\n').
```

If you run this, you'll get:

```
Binary form of cmp("Bilse",31):
$[01,00,00,00,0A,00,00,00,1F,00,42,69,6C,73,65,00,04,00,00,00,01,00,00,00]
Converted back: cmp("Bilse",31)
```

You shouldn't be too concerned about the actual format of this, in particular as we're dealing with implementation details, which may change. Nevertheless, we'll briefly describe the contents of the binary information:

```
$[01,00,00,00,0A,00,00,00,1F,00,42,69,6C,73,65,00,04,00,00,00,01,00,00,00]
  |           |_________| |___| |_______________| |_________| |_________|
  |                |        |           |             |           |
functor            |       31       "Bilse"\0      offset of    # of ptrs
                   |                                ptr to fix    in fixup
             0-relative                            (array, but
             ptr to string                          only one element here)
```

The *offset of ptr to fix* array will be 16-bit quantities on 16-bit platforms, as will the *# of ptrs in fixup*.

If the term contains elements from the ***symbol*** domain, the binary term will contain additional information to insert the symbols in the symbol table when the term is re-created.

Visual Prolog uses ***term_bin*** itself when storing things in the internal fact database system and when sending terms over a message pipe to another

program. If several programs share external databases or communicate over pipes, it's hence crucial that the domains involved use the same alignment.

Modular Programming

A Visual Prolog program can be broken up into modules. You can write, edit, and compile the modules separately, and then link them together to create a single executable program. If you need to change the program, you only need to edit and recompile individual modules, not the entire program – a feature you will appreciate when you write large programs. Also, modular programming allows you to take advantage of the fact that, by default, all predicate and domain names are local. This means different modules can use the same name in different ways. Visual Prolog uses two concepts to manage modular programming: *global declarations* and *projects*.

Global Declarations

By default, all names used in a module are local. Visual Prolog programs communicate across module boundaries using the predicates defined in the **global predicates** sections and in classes. The domains used in global predicates must be also defined as global domains or else they must be standard Visual Prolog domains.

Beginning with version 5.2 Visual Prolog provides enhanced handling of global declarations. In short:

1. The main project module (with the **goal**) must contain declarations of all global domains (and global facts sections) declared in all project submodules.
2. Any other project module may contain declarations of only those global domains, which are used in this module.
3. Global declarations can be placed after local declarations.
4. If any global declaration is changed, only modules including this declaration must be recompiled.

Global Domains

You make a domain global by writing it in a **global domains** section. In all other respects, global domains are the same as ordinary (local) domains.

Visual Prolog v. 5.2 provides enhanced handling of global domains. Now it is not required that all modules contain identical declarations of all global domains in exactly the same order (the special CHKDOMS.EXE utility were used to check this identity in PDC Prolog and in Visual Prolog versions previous to v. 5.2). Now you should obey only the following 2 much less strict rules:

1. Only the main project module (containing the **goal** section) must include declarations of all global domains (and global facts sections) declared in all project submodules.
2. Any other project module may contain declarations of only those global domains, which are used in this module.

This gives the following principal benefits:

- It is possible to create and use pre-compiled libraries (using global domains).
- When a global domain is changed, recompilation time is reduced, because only the modules including this domain declaration have to be recompiled.
- Your program can use more domains, since a module can include only those global domains, which are really used in this module.

According to these rules, the PDC Linker (while linking-time) checks whether the main project module includes declarations of all global domains declared in all project modules. If the PDC Linker detects a global domain *DomainName* that is declared in a submodule *FileName* and is not declared in the main module, then it generates an error message like this:

```
FileName - undefined name:$global$dom$DomainName
```

Notice that the PDC Linker compares only global domain names and it does not guarantee that a global domain has the same declarations in different project modules; this consistency is only the programmer responsibility. If you mix this up, all sorts of strange run-time problems can happen, such as a computer hanging under DOS or a protection violation on 32-bit platforms.

The easiest way to ensure that this is correct is by placing (including) all global domain declarations in a single file (for instance **ProjectName.inc**), which you can then include in every relevant module with an **include** directive like:

```
include "ProjectName.inc"
```

Visual Prolog VDE provides flexible automatic engine for handling inclusion of global domain declarations into project modules. The core of this engine is the **File Inclusion for Module** dialog, which is activated when you create a new module. (See the *Options of File Inclusion for Module dialog* in the *Visual Development Environment* manual.) For small projects you can use the simplified

strategy providing inclusion of all global domains in each project module. To guarantee this you need:

1. In the **File Inclusion for Module** dialog:
   - Check ON the "**Create <ModuleName>.DOM**" for each source module, which may introduce new global domains.
   - Check ON the "**Include <ModuleName>.DOM**", to specify that the **include** statement for <**ModuleName**>**.DOM** must be generated in the **<ProjectName>.INC** file.
2. The programmer has to place declarations of all global domains exported from a module into the correspondent **<ModuleName>.DOM** file.

Because the include directive for **<ProjectName>.INC** file is placed in all project modules, all modules will contain the same declarations of global domains.

In larger projects, you can implement more flexible "where-used" strategy for including of global domains. Instead of including <**ModuleName**>**.DOM** into the **<ProjectName>.INC** file, you can selectively include <**ModuleName**>**.DOM** files only into modules really importing global domains declared in these files. When you follow VDE's file naming and inclusion philosophy, the VDE's **Make** facility will automatically detect changes of files containing global declarations and enforce recompilation of all necessary modules, based on the file time stamps.

Global Facts Sections

You make a facts section global to a project by preceding the keyword **facts** (the obsolete keyword **database** is also possible) with the keyword **global**.

You can give initializing clauses for global facts only after the goal section in the main module.

Since Visual Prolog automatically generates the global domain correspondent to the name of each global facts section, then all rules discussed for handling of global domains should be applied to global fact sections.

Notice that the safe programming techniques require that you should not use global facts. Instead you can use global predicates operating with local facts.

Global Predicates

Global predicate declarations differ from ordinary (local) predicate declarations because they must contain a description of the flow pattern(s) by which each

given predicate can be called. If such one is not specified, all arguments will be input.

The keyword **global** specified before the keyword **predicates** specifies that all predicates and predicate values declared in the section are global to the project.

The syntax for global predicate declarations is:

```
global predicates
   [DeterminismMode] [ReturnDomain] PredicateName [(ArgList)]
         [- [FlowPatterns]] [Language] [ObjNameSpec]
```

The syntax for declarations of global predicate values is:

```
   predicateName : PredicateDomain [ObjNameSpec]
```

Here:

PredicateName

Defines the name of a declared predicate or a global predicate value.

PredicateDomain

Is a global predicate domain. It must be declared previously in a **global domains** section.

DeterminismMode

Specifies the predicate determinism mode with one of the following keywords:

```
{procedure | determ | nondeterm | failure | erroneous | multi}
```

This determinism mode is used for all specified predicate flow patterns if different determinism modes are not specified before separate flow patterns.

If the predicate determinism mode is not specified here explicitly, then the default determinism mode (as it is specified by the **Default Predicate Type** in the VDE's dialog **Compiler Options** or by the `-z[Value]` command line compiler option) is accepted. The default is **determ**.

ReturnDomain

Defines the domain for the return value, if you are declaring a function.

ArgList

Defines domains for predicate arguments in the form:

```
[ arg_1 [ , arg_2 ]* ]
```

Here ***arg_N*** is of the form:

```
Domain_Name [ Argument_Name ]
```

Domain_Name can be any standard or user-defined domain.

An optional *Argument_Name* can specify a mnemonic argument name. It must be a correct Visual Prolog name. The Visual Prolog supports this possibility to improve code readability and treats *Argument_Name* just as a comment.

Brackets surrounding the argument list ***ArgList*** can be omitted when the ***ArgList*** is empty.

FlowPatterns

Is of the form:

```
[[FlowDetermMode] FlowPattern] [[,] [FlowDetermMode] FlowPattern]*
```

FlowPattern

Is of the form:

```
(flow [ , flow ]*)
```

where ***flow*** is `{ i | o | functor FlowPattern | listflow }`

and ***listflow*** is `'[' flow [ , flow ]* [ '|' { i | o | listflow } ] ']'`

The char `'-'` is obligatory if any ***FlowPatterns*** is specified.

The flow pattern ***FlowPattern*** specifies how each argument is to be used. It must be `'i'` for an input argument and `'o'` for output arguments. A *functor* and flow pattern must be given for a compound term (e.g. `(i,o,myfunc(i,o),o)`), or a `listflow` (e.g. `[i, myfunc(i,o), o]`).

Caution: Several flow patterns can be explicitly specified for a global predicate. If none is specified explicitly, then the default flow pattern with all input arguments is accepted implicitly.

Notice that this implicit flow pattern can be the reason of error messages like "This flow pattern does not exist" when you re-declare working local redicates to global.

FlowDetermMode

One of the following keywords:

```
{procedure | determ | nondeterm | failure | erroneous | multi}
```

Optional possibility to declare separate determinism modes for flow patters. If the ***FlowDetermMode*** is specified before a flow pattern then for this flow pattern it overrides the predicate determinism mode specified before the predicate name (or the default determinism mode specified in the **Compiler Options**).

Language

Is of the form:

```
language { pascal | stdcall | asm | c | syscall | prolog}
```

The ***Language*** specification tells the compiler, which kind of calling conventions to use. It is only required when a predicate will be used from routines written in other languages (C, Delphi, etc.).

The default calling convention is **prolog**. Notice the difference with declarations of predicate domains, where the default is **pascal**.

ObjNameSpec

Is of the form

```
as "ObjectName"
```

The ***ObjNameSpec*** may be used to specify the public object-code name "*ObjectName*", overriding the default naming used by Visual Prolog compiler. The main use of this is when you are linking in modules written in other languages. (For more information see the chapter *Interfacing with Other Languages* on page 503.)

Here we use:

- Square brackets to indicate optional items, and braces (curly brackets) to indicate that one of the items delimited by the symbols `'|'` must be used.
- Pair of single quotes to indicate that the character surrounded by them (namely `'|'`, `'['` and `']'`) is a part of the Visual Prolog language syntax.
- Asterisk symbol `'*'` to indicate arbitrary quantity of the immediately preceding item (zero or more times).

Examples

In the following global predicate declaration, name and home are of type string, and age is of type integer; the arguments to ***first_pred*** can either be all bound `(i, i, i)` or all free `(o, o, o)`:

```
first_pred(name,home,age) - (i,i,i) (o,o,o)
```

Here is the declaration for a predicate with either compound flow of an integer list, or plain output flow:

```
p1(integerlist) - ([i,o,i|o]),(o)
```

If a flow pattern is not explicitly specified in a global predicate declaration:

```
my_converter(STRING, INTEGER)
```

Then the default flow pattern (`i, i)` with all input arguments is assumed, like if it is declared like this:

```
my_converter(STRING, INTEGER)- (i,i)
```

Particular determinism mode can be re-declared for every flow pattern. For example:

```
procedure append(ILIST,ILIST,ILIST) -
   (i,i,o)
   determ (i,i,i)
   nondeterm (o,o,i)
```

In this example ***append*** will have **procedure** determinism mode only with `(i,i,o)` flow pattern. Called with the `(i,i,i)` flow pattern it will be **determ**, and called with the `(o,o,i)` flow pattern it will be **nondeterm**

Finally, this declaration specifies compound flow for an object declared as `func(string, integer)` coming from a domain called mydom:

```
pred(mydom) - (func(i,o)) (func(o,i))
```

Note: If any global predicate definition is changed, only the modules, which refer to this predicate need to be recompiled. However, it is rather critical that this recompilation is done; if you change the flow pattern of a predicate, the calls using it will need different code.

It doesn't matter in which module the clauses for global predicates appear, but – as with local predicates – all clauses must appear together.

Projects

When you are using Visual Prolog's Visual Development Environment, then the Application Expert automatically handles creation of new projects and adding/deleting of project modules. The VDE's Make facility automatically manages compilation and linking operations needed to create the target module from the source project modules.

Therefore, we need to explain here only two features that can be important if you use the command line compiler:

1. How Visual Prolog projects use *symbol table*.

   All Visual Prolog modules involved in a project share the same internal symbol table that stores all **symbol** domain terms, which are used in all project modules. The symbol table is generated in the "so called" SYM-file. The SYM-file is an object format file with the .SYM extension. By default, if the project name is *ProjectName* then the VDE accepts that the SYM-file name is *ProjectName.SYM*. This name can be changed with the **SYM File Name** option in the **Compiler Options** dialog.

   When the command line compiler compiles a source file *ModuleName.PRO*, then by default it accepts that the SYM-file name is *ModuleName.SYM*. This default name can be changed by the `-r<ProjectName>` and `–M<SymFileName>` command line compiler options.

2. Visual Prolog programs must have the internal *goal*. Therefore, by default the compiler checks whether a compiled module has the **goal** section, and the compiler generates error if it has not. But in multi-modular projects only one (the *main*) module of the project contains the **goal** section. Hence, to compile other modules the compiler needs to be informed that these modules are parts of the project and thus do not have **goal** sections. This is done by the command line compiler option `-r[ProjectName]`. For example, when compiling a file *ModuleName.PRO* the compiler gets the option

   ```
   -rProjectName
   ```

   then the compiler is notified:

   - The compiled file *ModuleName.PRO* is a module of the project *ProjectName* and so does not have to contain the **goal** section;
   - The SYM-file name is *ProjectName.SYM*.

   In some circumstances we need to use the `–M<SymFileName>` option to override the SYM-file name specified by the `-r<ProjectName>` option. For instance, when linking from the Visual C++ environment, there is a requirement that all object files must have the .OBJ extension.

Errors and Exception Handling

As software quality improves, error handling becomes increasingly important in providing safe and trustworthy programs that users feel they can rely on. In this

section we look at the standard predicates Visual Prolog provides, giving you control over the errors and exceptions that may occur when your application is running. This includes trapping run-time errors and controlling user interruption.

If you look in Visual Prolog's error-message file PROLOG.ERR, you'll see all the error numbers applicable to both compile-time and run-time problems. All numbers above and including 10000 are reserved for user program exit codes, and you may modify and distribute the error message file if required. Additionally, in the include directory you'll find the ERROR.CON include file, containing constant declarations for all run-time error codes (from 1000 till 10000). To guard against future changes, use this file for error codes rather than hard-coding numbers into your application.

Exception Handling and Error Trapping

The cornerstone of error and exception handling is the ***trap*** predicate, which can catch run-time errors as well as exceptions activated by the ***exit*** predicate. You can also use this mechanism to catch signals, such as that generated by **Ctrl-Break** in the textmode platforms, as well as a kind of "block exit" mechanism.

exit/0 and exit/1

A call to ***exit*** has an effect identical to a run-time error.

```
exit                                              /* (no arguments) */
exit(ExitCode)                                             /* (i) */
```

exit without an argument is equivalent to `exit(0)`. If the call to ***exit*** is executed in a direct or indirect subgoal of a ***trap***, the *ExitCode* will be passed to the ***trap***.

The behavior of an untrapped exit depends on the platform. The VPI event handlers do their own trapping, and an exit will be caught here resulting in an error message.

An untrapped call of the ***exit*** predicate on the textmode platforms results in program termination, and the OS return code ('ErrorLevel' in the DOS-related operating systems, '$?' in UNIX ***sh***) will be set to the value used in the call. The maximum value a process can exit with is 254; 255 is reserved for Visual Prolog's ***system*** call, but no checks are performed.

errorexit/0 and errorexit/1

A call to ***errorexit*** performs a run-time error with setting of internal error information.

```
errorexit()                                          /* (no arguments) */
errorexit(ErrorNumber)                                        /* (i) */
```

A call to ***errorexit/1*** has an effect identical to a run-time error, but in contrast to ***exit*** predicates it sets internal error number *ErrorNumber* that can be obtained by a call to ***lasterror/4*** predicate:

```
lasterror(LastErrorNumber, ModuleName, IncludeFileName, Position)
```

This predicate will return *LastErrorNumber* = *ErrorNumber*.

Notice that in order to obtain a correct position all modules in a project must be compiled with the **errorlevel** > 0. (See below.)

Calling ***errorexit/0*** is the same as calling

```
errorexit(1000)
```

If ***errorexit*** is surrounded by a ***trap***, calling ***errorexit*** will jump back to this trap, and *ErrorNumber* will be passed to the error code variable of this trap.

trap/3

trap, which takes three arguments, carries out error trapping and exception handling. The first and the last arguments to ***trap*** are predicate calls, and the second argument is a variable; it takes this format:

```
trap(PredicateCall, ExitCode, PredicateToCallOnError)
```

For example, consider the call:

```
trap(menuact(P1, P2, P3), ExitCode, error(ExitCode, P1)), ...
```

If an error occurs during execution of ***menuact*** – including all further called subgoals – an error code will be returned in the variable *ExitCode*, and the error-handling predicate ***error*** will be called. ***trap*** will then fail on return from ***error***. If ***menuact*** returns successfully, evaluation will continue after the ***trap***, which will no longer be effective.

Before calling the *error* predicate, the system resets the stack, global stack, and trail to the values they had before the goal specified in the trap (*menuact* in the example above) was called. This means that you can use a ***trap*** to catch memory overflows, but you shouldn't rely on big memory consuming operations such as database updates to be in either a complete or unaltered state - a heap-full error may occur anytime.

If **Break** is enabled on the textmode platforms, and a **Break** occurs (because the user pressed **Ctrl-Break** during execution of a predicate with a surrounding ***trap***), the ***trap*** will catch the **Break** and return 0 in the *ExitCode* variable.

Example: catching file-not-open

```
                        /* Program ch10e12.pro */

include "error.con"

DOMAINS
   file = inpfile

PREDICATES
   ioehand(integer,file)
   getline(file,string)

CLAUSES
   ioehand(err_notopen,File):-!,
      write(File," isn't open\n"),
      exit(1).
   ioehand(Err,File):-
      write("Error ",Err," on ",File,'\n'),
      exit(1).

   getline(File,Line):-
      readdevice(Old),
      readdevice(File),
      readln(Line),
      readdevice(Old).

GOAL
      trap(getline(inpfile,First),Err,ioehand(Err,inpfile)),
      write(First).
```

errormsg/4

You can use the ***errormsg*** predicate to access files that are structured the same as Visual Prolog's error-message file.

```
   errormsg(File name, ErrorNo, ErrorMsg, ExtraHelpMsg) /* (i,i,o,o) */
```

A typical use of ***errormsg*** is in error-trapping predicates to obtain an explanation of an error code, as illustrated below.

```
PREDICATES
   error(integer)
   main
   /*....*/

CLAUSES
   error(0) :- !. % discard break.
error(E) :-
      errormsg("prolog.err", E, ErrorMsg, _),
      write("\nSorry; the error\n", E, " : ", ErrorMsg),
      write("\nhas occurred in your program."),
      write("\nYour database will be saved in the file error.sav"),
      save("error.sav").

GOAL
   trap(main, ExitCode, error(Exitcode)).
```

Error reporting

Visual Prolog includes several compiler directives that you can use to control run-time error reporting in your programs. These directives allow you to select the following:

- whether code should be generated to check for integer overflows.
- the level of detail in reporting run-time errors.
- whether code should be generated for stack overflow checking.

You can place these compiler directives at the top of your program, or choose them from the **Compiler Options** dialog.

errorlevel

Visual Prolog has a mechanism to locate the source position where a run-time error occurs. To do this, it generates code before predicate calls to save the source code position where executions are actually performed. Levels of error reporting and storing of source positions are selected by the `errorlevel` compiler directive. The syntax is:

```
errorlevel = d
```

where *d* is one of 0, 1, or 2, representing the following levels:

0 This level generates the smallest and most efficient code. No source positions are saved. When an error occurs, just the error number is reported.

1 This is the default level. When an error occurs, the Visual Prolog system displays its origin (module name and include file, if applicable). The place where the error was detected within the relevant source file is also displayed, expressed in terms of the number of bytes from the beginning of the file.

2 At this level, certain errors not reported at level 1, including stack overflow, heap overflow, trail overflow, etc., are also reported. Before each predicate call code is generated to store the source position.

When a source position are reported, the source program can be loaded into the editor, and you can activate the **Edit** | **Go To Line Number** menu item, where you can enter the position number and the cursor will move to the place where the error occurred.

In a project, the `errorlevel` directive in each module controls that module's detail of error reporting. However, if the errorlevel directive in the main module is higher than that of the other modules, the system might generate misleading error information.

If, for example, an error occurs in a module compiled with errorlevel = 0, which is linked with a main module compiled with errorlevel set to 1 or 2, the system will be unable to show the correct location of the error – instead, it will indicate the position of some previously executed code.

For more information about projects, refer to "Modular Programming" on page 252.

lasterror/4

Hand in hand with **trap** and **errormsg** goes **lasterror**. It returns all relevant information about the most recent error, and looks like this:

```
lasterror(ErrNo,Module,IncFile,Pos)                    /* (i,i,i,i) */
```

where *ErrNo* is the error number, *Module* is the source file name, *IncFile* is the include file name, and *Pos* is the position in the source code where the error occurred. However, the program must be compiled with an ***errorlevel*** greater than 1 in order for the information to be relevant in case of memory overflow. For ordinary errors, an errorlevel of 1 is sufficient.

The primary aim of ***lasterror*** is to ease debugging when subgoals are trapped, but it may equally well form the basis of a cause-of-death indicator in commercially distributed software. Using ***lasterror***, your code can provide the user with a quite sober error message in addition to exact information about what happened.

Handling Errors from the Term Reader

When you call ***consult*** or ***readterm*** and a syntax error occurs in the line read, the predicates will exit with an error. The syntax error could be any one of the following:

- A string is not terminated.
- Several terms are placed on one line.
- A symbol is placed where an integer is expected.
- Upper-case letters are used for the predicate name.
- A symbol is not surrounded by double quotes.
- Etc.

When ***consult*** was originally introduced in Visual Prolog, it was not meant to be used for reading user-edited files: It was designed to read back files that were saved by the ***save*** predicate. In order to make it easier to consult user-created files, we have introduced the two predicates ***readtermerror*** and ***consulterror***. You can call these to obtain information about what went wrong in ***readterm*** or ***consult***, respectively.

If the errors from ***consult*** and ***readterm*** are caught by the ***trap*** predicate, ***consulterror*** and ***readtermerror*** allow you to inspect and possibly edit the cause of the syntax error.

consulterror/3

consulterror returns information about the line containing a syntax error.

```
consulterror(Line, LinePos, Filepos),                    /* (o,o,o) */
```

Line is bound to the line that has the syntax error, *LinePos* is bound to the position in the line where the syntax error was found, and *FilePos* is bound to the position in the file where the line was read.

```
                        /* Program ch10e13.pro */

CONSTANTS
   helpfile = "prolog.hlp"
   errorfile = "prolog.err"

DOMAINS
   dom = f(integer)
   list = integer*
```

```
facts - mydba
   p1(integer, string, char, real, dom, list)

PREDICATES
   handleconsulterr(string, integer)

CLAUSES
   handleconsulterr(File, Err):-
      Err>1400, Err<1410, !,
      retractall(_, mydba),
      consulterror(Line, LinePos, _),
      errormsg(errorfile, Err, Msg, _),
      str_len(Blanks,LinePos),
      write("Syntax error in ",File,'\n',Line,'\n',Blanks,"^\n",Msg,'\n'),
      exit(1).
   handleconsulterr(File,Err):-
      errormsg(errorfile,Err,Msg,_),
      write("Error while trying to consult ",File,":\n",Msg,'\n'),
      exit(2).

GOAL
      File="faulty.dba",
      trap(consult(File, mydba), Err, handleconsulterr(File,Err)),
      write("\nSUCCESS\n").
```

Notice that the file "faulty.dba" should be copied into the \EXE subdirectory of your project if you **Run** the program or into \OBJ subdirectory when you use the **Test Goal**.

readtermerror/2

readtermerror returns information about the ***readterm***-read line containing a syntax error.

```
   readtermerror(Line, LinePos),                                  /* (o,o) */
```

Line is bound to the line that has the syntax error, and *LinePos* is bound to the position in the line where the syntax error was found.

Break Control (Textmode Only)

It is important to understand how the break/signal mechanism is implemented in Visual Prolog. Generally, a break does not immediately abort the current execution. Rather, Visual Prolog has an exception handler installed, which sets a

flag when activated by the signal. Visual Prolog checks this flag in two different cases:

- If the code is compiled with break checking enabled, the status of the break-flag is examined each time a predicate is entered. Break checking may be disabled by using the nobreak directive in the source code, through the **O**ptions/**C**ompiler Directives/**R**un-time check menu item, or from the command line.
- Several of the library routines check the break-flag.

If the break-flag is set, the outcome depends on the break-status, set by the predicate ***break***: If break-status is *OFF*, the signal is ignored for the time being, otherwise the code will exit with an appropriate *exitcode* (discussed below). This exit will of course be caught by a ***trap***, if any is set.

break/1

break enables and disables the sensing of the break-flag during execution. ***break*** takes one of the following forms:

```
break(on)                              /* (i); enables the BREAK key */
break(off)                            /* (i); disables the BREAK key */
break(BreakStatus)          /* (o); returns the current BREAK status */
```

You can read the current break status by calling ***break*** with an output variable. This means that, during critical operations, you can disable break and then return to the original break state afterwards. For example:

```
update :-
   break(OldBreak),
   break(off),
/* .... do the updating, */
   break(OldBreak).
```

For the DOS-related versions, the exitcode resulting from a break will always be 0, as the only signal recognized is the user interrupt. For the UNIX version, SIGINT also results in an exit value of 0, for backwards compatibility with the large base of installed DOS programs written in Visual Prolog. For other signals which the process has been set up to catch, the exitcode is the signal number plus the constant *err_signaloffset*, defined in the include file ERROR.CON.

breakpressed/0

breakpressed succeeds if the break-flag is set, even when the break-state has been turned off by `break(off)` or the program compiled with the nobreak option.

If successful, ***breakpressed*** returns the exitcode generated by the most recently caught signal, and clears the break-flag. For the DOS-related versions of Visual Prolog, this will always be 0; for UNIX, it will be the same value as would otherwise be passed through the exit/trap mechanism, as described above. This too will be 0 if SIGINT is received.

Manual Break and Signal Checking in UNIX

This section, down to page 271, only applies to UNIX and may be skipped by users of the DOS-related versions.

A Visual Prolog program may be configured to catch any of the many different signals a UNIX process can receive (see **signal(S)**). However, as signals may arrive at any time, quite asynchronously from the running process, it's important that they don't interrupt the process while in the middle of something critical, such as memory allocation. The reason for this is that, due to Prolog's modularity, the only means of communication between different predicates is through arguments or through databases. Obviously, an asynchronously executed signal-handler predicate can't communicate to the rest of the program through arguments, leaving only the database. And since databases rely on memory allocation, which invariably is in use by the rest of the program, an asynchronously executed signal-handling predicate could create havoc, if trying to e.g. assert something to indicate that a signal was received, while the interrupted process was in the middle of allocating memory. It really all boils down to Prolog not having global variables, leaving asynchronously executed predicates with no means of communication with the rest of the program.

Therefore, rather than invoking a signal-handling predicate the instant the signal is received, signals are routed through the exit/trap mechanism.

signal/2

Signal-handling in Visual Prolog programs is controlled by the ***signal*** predicate, defined in the include file ERROR.PRE:

```
GLOBAL DOMAINS
   sighand = determ (integer) - (i) language C

GLOBAL PREDICATES
   sighand signal(integer,integer) - (i,i) language C as "_BRK_Signal"
   sighand signal(integer,sighand) - (i,i) language C as "_BRK_Signal"
```

```
CONSTANTS
   sig_default = 0
   sig_ignore = 1
   sig_catch = 2
```

To modify the handling of a specific signal, call ***signal*** with the signal exitcode you want to catch, such as *err_sigalrm*, defined in ERROR.PRE, specifying in the second argument what to do:

- **sig_default** to reset the handling of the signal to the default for the process
- **sig_ignore** to ignore the signal completely
- **sig_catch** to have the signal routed through the exit/trap mechanism
- anything else is taken to be the address of a function to be invoked when the signal occurs

The return value of ***signal*** is the previous handling of the signal in question, which will be one of the values outlined above. The only cases where you may use the fourth alternative (address of function) is when this value was returned by a previous call to ***signal***, or when the function is one you have written yourself in C, exercising the usual precautions when writing signal handlers. In particular, SIGINT is masked out during the execution of the signal handler, so if you intend to do a longjump from a signal handler you're written in C, SIGINT must be enabled first (see ***sigprocmask(S)***). The validity of the function address is not verified at the time ***signal*** is called and results may be highly erratic if it's an invalid address; see ***signal(S)***.

Although the name and argument profile of ***signal*** matches that of ***signal(S)***, it is implemented using ***sigaction(S)*** and SIGINT is ignored during execution of installed signal handlers.

By default, Visual Prolog catches the following signals:

- SIGINT (user interrupt); results in exit of 0 when detected.
- SIGFPE (floating point exception); results in an exit of err_realoverflow immediately after the erroneous calculation.
- SIGBUS and SIGSEGV (memory fault); these signals result from attempting to access memory not belonging to the process, typically due to a faulty pointer. A short message, indicating where in the program the error happened, will be printed if possible (see the **errorlevel** compiler directive), and the process is terminated, leaving a core dump. Unless you have made a mistake in modules you have written yourself in C, this invariably indicates an internal error.

- SIGILL (illegal instruction); the processor encountered an unrecognized or illegal instruction. Other details as for SIGBUS and SIGSEGV.

Any signals caught will be routed through the same function as SIGINT. Note that once catching has been enabled for a signal, it remains in effect until explicitly reset by another call to ***signal***. Receiving and catching a signal will not reset the signal handling.

Needless to say, signal catching should be used very carefully, and the break-state should always be turned off if you intend to receive and test for signals without interrupting the program. In particular, a number of operating system calls will be terminated prematurely if a signal is caught while they're executing. When the break-state is off, the reception of the signal will be noted in the break-flag and the interrupted system call called again, meaning the program should work as expected. However, while every care has been taken to ensure the integrity of this scheme, no guarantees can be given.

Below are two examples, using the alarm clock signal. Both use the ***breakpressed*** predicate, which will be described later.

The first example will print the message "Do something!" every three seconds, until the user enters a character. It doesn't turn the break-state off during the central parts of the program, as the whole purpose is to interrupt a system call.

```
                        /* Program ch10e14.pro */

   /* For UNIX platform only */

include error.con"

GLOBAL PREDICATES
   alarm(integer) - (i) language C                          % See alarm(S)

PREDICATES
   brkclear
   nondeterm repeat
   ehand(integer)
   getchar(char)

CLAUSES
   brkclear:-breakpressed,!.                    % Clear break-flag, if set
   brkclear.

   repeat.
   repeat:-repeat.
```

```
   ehand(2214):-!,
      write("Do something!\n").
   ehand(E):-
      write("\nUnknown exit ",E,'\n'),
      exit(2).

   getchar(C):-
      write("Enter char: "),
      alarm(3),                              % Alarm to go off in 3 seconds
      readchar(C),
                  % This will exit with err_sigalrm when receiving SIGALRM
      alarm(0),                               % Cancel pending alarm signal
      break(off),
      brkclear,                  % Clear break-flag, in case alarm went off
      break(on).       % just before cancellation above.

GOAL
      Old=signal(err_sigalrm,sig_catch),           % Declared in error.con
      repeat,
        trap(getchar(C),Err,ehand(Err)),
      !,
      signal(err_sigalrm,Old),
      write("\nYou entered '",C,"'\n").
```

The next example, which has been deliberately written to be somewhat inefficient, displays program progress during lengthy processing. Break-status is turned off in this program, and the detection of any signals is handled manually, using the ***breakpressed*** predicate.

```
                        /* Program ch10e15.pro */

   /* For UNIX platform only */

include "error.con"

GLOBAL PREDICATES
   alarm(integer) - (i) language C% See alarm(S)

facts
   rcount(unsigned)
   dba(real,real,real)

PREDICATES
   nondeterm repeat
   process_dba
   bcheck
   bcheck1(integer)
```

```
CLAUSES
   repeat.
   repeat:- repeat.

   rcount(0).

   dba(1,1,1).

   process_dba:-
      retract(dba(F1,F2,F3)), !, F = F1 * F2 * F3, assert(dba(F,F,F)),
      retract(rcount(N)), !, NN = N+1, assert(rcount(NN)),
      NN = 25000.      % fail back to repeat in goal

   bcheck:-
      Break = breakpressed(),!,
      bcheck1(Break).
   bcheck.

   bcheck1(err_sigalrm):-!,
      rcount(N),!,
      time(H,M,S,_),
      writef("\r%:%:% % records   ",H,M,S,N),
      alarm(1). % Next alarm in 1 second
   bcheck1(0):-!,
      write("\nInterrupt\n"),
      exit(1).
   bcheck1(Exit):-
      write("\nUnknown exit ",Exit,"; runtime error?\n"),
      exit(2).

GOAL
      break(off),
      Old = signal(err_sigalrm,sig_catch),         % Declared in error.pre
      alarm(1),                                   % First alarm in 1 second
      repeat,
      bcheck, process_dba,
      !,
      alarm(0),                                     % Cancel pending alarm
      signal(err_sigalrm,Old),
      dba(F1,F2,F3), !,
      write('\n',F1,' ',F2,' ',F3,'\n').
```

The ***writef*** predicate is covered in chapter **12**.

Critical Error Handling under DOS Textmode

This section applies only to the DOS textmode platform, and are not relevant for VPI programs.

The DOS-version of Visual Prolog's library contains some default routines for handling error situations, but you can actually substitute the default code with your own clauses. In this section, we describe two routines – ***criticalerror*** and ***fileerror***. DOS will call ***criticalerror*** when a DOS error occurs. The Visual Prolog system calls ***fileerror*** when it gets a file error in the run-time editor. If you define these predicates as global and supply your own clauses for them, the linker will take your code instead of the code from the library. The result is that you gain better control over error situations. Your .EXE program's size might also decrease (because the code for the default routines draw in window routines).

Global declarations for ***criticalerror*** and ***fileerror*** are given in the include file ERROR.PRE shipped with the Visual Prolog system in the include directory.

criticalerror/4

Visual Prolog defines this routine for handling DOS critical errors (DOS interrupt 24H). If you want to use your own version of ***criticalerror***, you should include ERROR.PRE, which gives a global declaration as follows:

```
GLOBAL PREDICATES
   criticalerror(ErrNo, ErrType, DiskNo, Action) - (i, i, i, o) language
c as "_CriticalError_0"
```

Refer to the (see the *Modular Programming* on page 252) for information on how to use global declarations.

The ***criticalerror*** predicate must never fail, and it *works only* from an .EXE file application. The ***criticalerror*** predicate replaces the DOS critical error interrupt handler and has the same restriction as the original interrupt handler. (Refer to the *DOS Technical Reference* for details.) You can only use DOS function calls 01h to 0Ch and 59h ("Get extended error") – that means console I/O and nothing else. If your application uses any other DOS function calls, the operating system is left in an unpredictable state.

Table 10.3: Argument Values for the criticalerror Predicate

Argument	Value	Meaning
ErrNo	= 0 = 1 = 2 = 3 = 4 = 5 = 6 = 7 = 8 = 9 = 10 = 11	Attempt to write on write-protected disk Unknown unit Drive not ready Unknown command CRC error in data Bad drive request structure length Seek error Unknown media type Sector not found = 12 Printer out of paper Write fault Read fault General failure
ErrType	= 0 = 1 = 2	Character device error Disk read error Disk write error
DiskNo	= 0-25	Means device A to Z
Action	= 0 = 1 = 2	Abort current operation Retry current operation Ignore current operation (this could be very dangerous and is not recommended)

fileerror/2

Visual Prolog will activate the predicate ***fileerror*** when a file in the textmode editor action fails.

If you define your own ***fileerror*** predicate, it is not allowed to fail, and it *works only* from an .EXE file application.

The declaration for ***fileerror*** in the ERROR.PRE file is:

```
GLOBAL PREDICATES
   fileerror(integer, string) - (i, i) language c as "_ERR_FileError"
```

Note that this declaration is correct – you *must* specify language c even though the source code will be in Prolog.

Dynamic Cutting

The traditional ***cut*** in Prolog is static. One problem with this is that the effect of the ***cut*** happens when execution passes the `'!'` symbol, and it affects only those clauses in which it was placed (in the source text). There is no way of passing the effect of a cut in an argument to another predicate, where the cut might only be evaluated if some conditions were fulfilled. Another problem with the traditional cut is that it is impossible to cut away further solutions to a subgoal in a clause, without also cutting away the backtracking point to the following clauses in the predicate.

Visual Prolog has a *dynamic cutting mechanism*, which is implemented by the two standard predicates ***getbacktrack*** and ***cutbacktrack***. This mechanism allows you to handle both of these problems. The predicate ***getbacktrack*** returns the current pointer to the top of the stack of backtrack points. You can remove all backtrack points above this place, at some later time, by giving the pointer thus found to the ***cutbacktrack*** predicate.

Examples

Here are some examples that illustrate the use of these two predicates.

1. Suppose you have a database of people and their incomes, and you have registered who their friends are.

   ```
   facts
      person(symbol, income)
      friends(symbol, symbol)
   ```

   If you define a happy person as one who either has some friends or pays little tax, the clauses that return happy persons could be as follows:

   ```
   happy_person(has_friends(P)) :-
      person(P, _),
      friends(P, _).
   happy_person(is_rich(P)) :-
      person(P, Income),
      not(rich(Income)).
   ```

If a person has more than one friend, the first clause will return a multiple number of solutions for the same person. You could, of course, add another predicate `have_friends(P,P)` that has a cut, or you could use the dynamic cut instead.

```
happy_person(has_friends(P)) :-
   person(P, _),
   getbacktrack(BTOP),
   friends(P, _),
   cutbacktrack(BTOP).
```

Although the *friends* predicate might return many solutions if backtracked into, that possibility is cut away with the call to ***cutbacktrack***. A subsequent failure would backtrack into the *person* predicate.

2. The more important use of a dynamic cut is when you pass the backtrack pointer to another predicate and execute the cut conditionally. The pointer is of ***unsigned*** type and can be passed in arguments of ***unsigned*** type.

   As an illustration of this, let's say you want a predicate to return numbers until the user presses a key.

```
PREDICATES
   number(integer)
   return_numbers(integer)
   checkuser(unsigned)

CLAUSES
   number(0).
   number(N) :- number(N1), N = N1+1.

   return_numbers(N) :-
      getbacktrack(BTOP),
      number(N),
      checkuser(BTOP).

   checkuser(BTOP) :-
      keypressed,
      cutbacktrack(BTOP).
   checkuser(_).
```

The compiler does not recognize the ***cutbacktrack*** predicate in the pass that analyzes the clauses for determinism. This means you could get the warning `Non-deterministic clause` when using the ***check_determ*** directive, even if you called ***cutbacktrack***.

You should use dynamic cutting very judiciously. It's all too easy to destroy program structure with dynamic cutting, and careless use will invariably lead to problems that are very hard to track down.

Free Type Conversions

In most cases there is little need to start mixing wildly differing types. However, from time to time, in particular when dealing with system level programming or when interfacing to foreign languages, rather reckless conversions have to be dealt with. To this end the ***cast*** function will convert from anything to anything. No checks are performed on the supplied values, and quite disastrous results will occur if you try to use incorrectly cast variables.

The format of ***cast*** is

```
Result = cast(returnDomain,Expr)
```

where *Expr* is evaluated (if it's a numerical expression), converted to *returnDomain* type, and unified with *Result*.

For instance, a null string pointer (a character pointer with a value of 0; not an empty string, which is a pointer to a byte with a value of 0) can be created using:

```
NullPtr = cast(string,0)
```

Don't try to write the resulting string, you'd most probably get a protection violation, a hung system, or at best garbage characters.

If you don't see any obvious use for ***cast***, don't worry. It plays no part in ordinary Prolog programs.

Programming Style

In this section, we provide some comprehensive guidelines for writing good Visual Prolog programs. After summarizing a few rules of thumb about programming style, we give you some tips about when and how to use the ***fail*** predicate and the ***cut***.

Rules for Efficient Programming

Rule 1. Use more variables rather than more predicates.

This rule is often in direct conflict with program readability. To achieve programs that are efficient (both in their demands upon relatively cheap machines and upon relatively expensive human resources) requires a careful matching of objectives.

Often, the purely declarative style of Prolog leads to code that is significantly less efficient than other (non-declarative) approaches. For instance, if you're writing a predicate to reverse the elements of a list, this code fragment:

```
reverse(X, Y) :- reverse1([], X, Y).              /* More efficient */
reverse1(Y, [], Y).
reverse1(X1, [U|X2], Y) :- reverse1([U|X1], X2, Y).
```

makes less demands upon the stack than the next one, which uses the extra predicate *append*:

```
reverse([], []).                                  /* Less efficient */
reverse([U|X], Y) :- reverse(X, Y1), append(Y1, [U], Y).

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

Rule 2. Try to ensure that execution fails efficiently when no solutions exist.

Suppose you want to write a predicate ***singlepeak*** that checks the integers in a list to see if, in the order given, they ascend to a single maximum and then descend again. With this predicate, the call:

```
singlepeak([1, 2, 5, 7, 11, 8, 6, 4]).
```

would succeed, while the call:

```
singlepeak([1, 2, 3, 9, 6, 8, 5, 4, 3]).
```

would fail.

The following definition for *singlepeak* breaks ***Rule 2***, since the failure of a list to have a single peak is only recognized when *append* has split the list into every possible decomposition:

```
/* Definition 1 - Breaks Rule 2 */

singlepeak(X) :- append(X1, X2, X), up(X1), down(X2).

up[_].
up([U, V|Y]) :- U<V,  up([V|Y]).
```

```
down([]).
down([U]).
down([U, V|Y]) :- U>V,  down([V|Y]).

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

On the other hand, the next definition recognizes failure at the earliest possible moment:

```
/* Definition 2 - Follows Rule 2 */

singlepeak([]).
singlepeak([U, V|Y]) :- U<V, singlepeak([V|Y]).
singlepeak([U, V|Y]) :- U>V, down([V|Y]).

down([]).
down([U]).
down([U, V|Y]) :- U>V, down([V|Y]).
```

The third and final definition shortens *singlepeak* even further by observing Rule 1.

```
/* Definition 3 - Follows Rule 1 */

singlepeak([], _).
singlepeak([H|[]], _).
singlepeak([U, V|W], up) :- U<V, singlepeak([V|W], up).
singlepeak([U, V|W], _)  :- U>V, singlepeak([V|W], down).
```

Using Definition 3, this call to *singlepeak*

```
singlepeak(Y, up)
```

succeeds if *Y* is bound to a single peaked list appended to an ascending list. This call

```
singlepeak(Y, down)
```

succeeds if *Y* is bound to a descending list.

Rule 3. Let Visual Prolog's unification mechanism do as much of the work as possible.

At first thought, you might define a predicate *equal* to test two lists from the same domain for equality as follows:

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

This is unnecessary. Using the definition

```
equal(X, X).
```

or, even simpler, unification by means of =, Visual Prolog's unification mechanism does all the work!

Rule 4. Use backtracking – instead of recursion – for repetition.

Backtracking decreases stack requirements. The idea is to use the ***repeat*** ... ***fail*** combination instead of recursion. This is so important that the next section is dedicated to the technique.

Using the fail Predicate

To evaluate a particular sequence of subgoals repeatedly, it is often necessary to define a predicate like ***run*** with a clause of the form subgoals, evaluating repeatedly

```
run :-
   readln(X),
   process(X, Y),
   write(Y),
   run.
```

This kind of definition incurs unnecessary recursion overheads that cannot be automatically eliminated by the system if ***process(X,Y)*** is non-deterministic.

In this case, the ***repeat*** ... ***fail*** combination avoids the need for the final recursive call. Given

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

you can redefine *run* without recursion as follows:

```
run :-
   repeat,
      readln(X),
      process(X, Y),
      write(Y),
   fail.
```

fail causes Visual Prolog to backtrack to ***process*** and eventually to ***repeat***, which always succeeds. But how do you break out of a ***repeat*** ... ***fail*** combination? Well, in the cases where you want infinite execution (the `run:- ..., ..., run` variety, you will usually only want to quit if some exceptional condition arises. To this end, you can use the ***exit*** predicate in non-interactive programs, or just press break in interactive ones. In other cases, where you have a clear condition of completion, replace the ***fail*** with a test for completion:

```
run:-
   repeat,
      getstuff(X),
      process(X,Y),
      putstuff(Y),
   test_for_completion(Y),
   !.
```

Determinism vs. Non-determinism: Setting the Cut

The compiler directive `check_determ` is useful when you need to decide where to place the ***cut***, since it marks those clauses that give rise to non-deterministic predicates. If you want to make these predicates deterministic, you must insert the ***cut*** to stop the backtracking (which causes the non-determinism).

As a general rule, in such cases, the ***cut*** should always be inserted as far to the left as possible (close to the head of a rule) without destroying the underlying logic of the program.

Keep in mind these two rules used by the compiler to decide that a clause is non-deterministic:

1. There is no ***cut*** in the clause, and there is another clause that can match the input arguments in the clause head.
2. There is a call to another non-deterministic predicate in the clause body, and this non-deterministic call is not followed by a ***cut***.

CHAPTER 11

Classes and Objects

Visual Prolog contains a powerful object-mechanism, that melts together the logic programming and object oriented programming (OOP) paradigms.

You will find some small examples of Visual Prolog programs, which use OOP-technology on the CD in the directory \OOP\EXAMPLES.

Four criteria have to be fulfilled, before a system can be considered to be object orientated: encapsulation, classes, inheritance, and identity.

Encapsulation

The importance of encapsulation and modularity are well known. Encapsulated objects can help building more structured and readable programs because objects are treated like black boxes. Look at complex problems, find a part, which you can declare and describe. Encapsulate it in an object, construct an interface and continue so, until you have declared all the subproblems. When you have encapsulated the objects of the problem, and ensured that they work correctly, you can abstract from them.

OOP is also sometimes known as data-driven programming. You can actually let the objects themselves do the work for you. They contain methods, which are invoked, when they are created, deleted and whenever you call them. Methods can call methods in other objects.

Objects and Classes

The way data is stored in traditional programming languages are usually hard to grasp for humans and not suited for modeling. Objects are much easier to work with, because it is closer to the way humans understand real-world objects and in fact objects themselves are a tool for modeling.

Object is a far more complex data structure than lists. At the basic level an object is a declaration of coherent data. This declaration can also contain predicates, which work on these data. In OOP terminology these predicates are called methods. Each class represents a unique type of objects and the operations (methods) available to create, manipulate, and destroy such objects.

A class is a definition of an object type; it can create objects corresponding to this type. An instance is an actual occurrence of an object. Normally you can define as many instances (objects) of a class as you like.

Example

```
class automobile
   owner string
   brand string
endclass

Actual instance 1
   Owner Beatrice
   brand Morris Mascot
End

Actual instance 2
   Owner John
   brand Rolls Royce
End
```

Inheritance

OOP is a powerful modeling tool. Objects can be defined on the abstraction level that suits best. From this level child-objects can be defined on lower levels, or parent-objects on higher levels. An object can inherit data and methods from objects at higher levels. Objects are thus an easy way to make modular programs.

Identity

Every object is unique. Objects have a changeable state, and since states of objects can be observed by means of their member predicates an object is only identical to itself. I.e. even if the states of two objects are identical, the objects are not identical, because we can change the state of one object without changing the other, and then the objects no longer have identical states.

A very important characteristic of an object is that its identity remains even though its attributes change. We always access an object by means of a reference to an object. An object can be accessed through many different references and since an object is only identical to itself, we can have many references to the same object.

Visual Prolog Classes

Defining a Class in Visual Prolog requires two things: a *class declaration*, and a *class implementation*. The class declaration specifies the interface to the class; that is, what can be seen from the outside. The class implementation contains the Prolog clauses for defining the actual functionality for the class.

That is, the declaration of the interface to a class and the actual definition of the clauses for a class are separated. The class declaration will often be placed in header files (usually, with filename extension .PH or .PRE) that can be included in modules that use the class. The class implementation can be placed in any project module that includes the class declaration.

Class Declarations

A simplified "first-look" syntax for a class declaration is:

```
class class-name [: parentclass-list ]
   domains
      domain_declarations

   [static] predicates
      predicate_declarations

   [static] facts
      fact_declarations

endclass class-name
```

The class declaration section starts with the keyword **class** and terminates with the keyword **endclass**. The optional *parentclass-list* specifies the parent class (or classes) from which the class *class-name* will *derive* (or *inherit*) domains, predicates and facts. If any parent class is specified, the class *class-name* is called a derived class. The *class-name* after the keyword **endclass** is optional.

By default, the access rights of entities declared in the class declaration are public (global to the project); therefore, such domains, predicates and facts can be accessed outside the class (and inherited classes).

Unless predicates and facts sections are preceded with the keyword **static** (see **Static Facts and Predicates** on page 292), the declared facts will belong to objects (class instances), and the predicates will carry an invisible argument identifying the objects to which they belong to.

Class Implementation

A simplified "first-look" syntax for a class implementation is:

```
implement class-name [: parentclass-list ]
   domains
      domain_declarations

   [static] predicates
      predicate_declarations

   [static] facts
      fact_declarations

   clauses
      clause_definitions

endclass class-name
```

Definitions of clauses for a class is done in a section starting with the **implement** keyword and ending with the **endclass** keyword. Multiple domains, predicates, facts, and clauses sections can be placed inside a class implementation. The *class-name* after the **endclass** keyword is optional.

Declarations done in a class implementation work like as they are given in a class declaration, however, in opposite to the class declaration, entities declared in the class implementation will be entirely private to the implementation. The scope of names declared in a class implementation section is this implementation section.

Note also, that it is possible to inherit classes down in the implementation, thus encapsulating details on a class implementation. Base classes specified in a class implementation must not duplicate base classes specified in this class declaration.

Notice that using of global facts is not a safe programming technique. Therefore, we recommend never declare facts in class declarations. We recommend encapsulate facts in class implementations and provide access to them with public class member predicates. According to PDC plans, future versions of Visual Prolog will not support public facts.

Class Instances - Objects

When a class declaration and definition has been made a multiple number of instances (objects) of this class can be created. A new object of a class is made with a call to a *constructor* ***new*** for that class. A call to a constructor predicate ***new*** creates a new object (instance of the class) and returns a reference to the created object, which can later be used to perform operations on this object and

call methods of this object. The syntax for calling a constructor of some class ***my_c*** to create a new class instance is:

```
Object_identifier = my_c :: new,
```

The ordinary syntax for specification of predicates and facts from some formerly created object is:

```
[Object_identifier :] predicate_or_fact_name[(arguments)]
```

Example:

```
                        /* Program ch11e01.pro */

class cCounter
   predicates
      inc()
      dec()
      integer getval()
endclass

implement cCounter
   facts
      single count(integer)

   clauses
      count(0).

      inc:-
         count(X),
         X1=X+1,
         assert(count(X1)).

      dec:-
         count(X),
         X1=X-1,
         assert(count(X1)).

      getval(VAL):-
         count(Val).
endclass

goal
      Obj_counter = cCounter::new,
      Initial = Obj_counter:getval(),
      Obj_counter:inc,
      NewVal = Obj_counter:getval(),
      Obj_counter:delete.
```

Run the program with the Test Goal utility, you will obtain:

```
Obj_counter=145464, Initial=0, NewVal=1
1 Solution
```

In this example the call `Obj_counter = cCounter::new` to the constructor `new` of the class `cCounter` creates a new instance of the class `cCounter` and returns the reference (object identifier) to the created object in the variable `Obj_counter`. Later this reference is used to call methods of the object and access the instance of the fact ***count*** from this object.

Each instance of the class will have its own version of non-static facts. Such fact databases can be manipulated as usual facts by database predicates: ***retract***, ***assert***, ***save***, ***consult***, etc.

Each class has the default constructor ***new*** (without parameters). When a program calls the default constructor to create an object, then Visual Prolog automatically initializes all structures required to the created object. To provide additional functionality to class constructors (initialize facts to special values, etc), the programmer can define explicit constructors to the class. (See *Explicit Constructors and Destructors* on page 297.)

Destroying Objects

Objects have to be deleted explicitly by calling a *destructor* **delete** on an object.

```
Obj = customers::new,
Obj:change_account,
Obj:delete.
```

After calling a destructor ***delete*** on an object *Obj*, the object *Obj* is considered to be deleted, and any attempt to use it is not correct.

Deleting of an object causes automatic retracting of all (non-static) facts from fact databases of this object and deallocation of memory that was used by this object. Hence, it is strictly recommended to delete unnecessary objects in order to free resources.

Notice that once created, objects will survive **fail**. That is, **fail** does not delete created objects and does not deallocate the memory occupied by objects. To delete an object and to free the occupied resources you have to call a class destructor **delete** explicitly.

Each class has the default destructor ***delete*** (without parameters).

Class Domains

The declaration of a class with the name *class_name* automatically generates a global domain *class_name*. This domain can be used to declare adrument domains in predicates that should handle references to objects of this class.

```
CLASS class_name
   ...
ENDCLASS

PREDICATES
   p(class_name)
```

Passing an object identifier in a parameter means just passing a pointer to the object as in normal OOP-programming style.

In most cases, class domains can be used as ordinary global domains and most corresponding rules are applied to them.

- As ordinary global domains, the class domain *class_name* has to be visible only in those project modules (the declaration of class *class_name* should be included in those modules), which explicitly use this *class_name* domain.
- As ordinary global domains, class domains correspondent to *abstract* classes have to be visible in the main module. (see the *Abstract Classes* below).
- But class domains correspondent to non-abstract classes do not have to be visible in the main module if the main module does not use them.

Class domains can be used for type conversions of objects with predicate **val**. Normally, **val** checks correctness of conversions at compile-time; but when **val** converts an object from a *parent* class to a derived (*child*) class, then **val** leaves some freedom to the programmer and checks the correctness only at run-time. If **val** detects that the conversion is incorrect, then it generates the run-time error.

Derived Classes and Inheritance

What you can do if you need an object that is rather similar to an object you already have, but with some extra properties? You can just derive a new class using a class of this similar object as the base class. When you declare a *derived* class *D*, you list parent classes *P1*, *P2* in a comma-delimited parent class list:

```
CLASS D : P1, P2 ...
```

The derived class *D* inherits *public* domains, predicates, and facts of the specified parent classes *P1, P2*. Inherited predicates and facts become public members of

the class *D* and they can be referenced as members of the class *D*. On the other hand, *D* inherits only possibility to use the inherited domains; they cannot be qualified with the name of the class *D*.

Inherited domains, predicates, and facts can be redefined in a derived class. Global predicates also can be redefined inside a class, for example, the build-in global predicates ***beep***, ***concat***, ***retract*** can be overridden inside a class.

When using a child class, it is possible to use predicates and facts from both the parent classes and from the child class. However, the child class might choose to redefine some of predicates or facts from the parent classes. Members of a base class redefined in a derived class can be accessed from the derived class with explicit qualifying of a member name by the base class name:

```
[base_class_name ::] member_name[(arguments)]
```

Example.

```
                        /* Program ch11e02.pro */

class cPerson
   predicates
      procedure add_name( STRING ) - (i)
      procedure add_father( cPerson ) - (i)
      procedure add_mother( cPerson ) - (i)
      procedure write_info()
endclass

class cEmploye : cPerson
   predicates
      procedure add_company(string Name) -(i)
      procedure write_info()
endclass

implement cPerson
   facts
      name( string )
      father( cPerson )
      mother( cPerson )

   clauses
      add_name(Name):-
         assert(name(Name)).

      add_father(Obj_person):-
         assert(father(Obj_person)).
```

```
      add_mother(Obj_person):-
         assert(mother(Obj_person)).

      write_info():-
         name(X),
           write("Name=",X),nl,
         fail.
      write_info():-
         father(F),
           write("Father:\n"),
           F:cPerson::write_info(),
         fail.
      write_info():-
         mother(M),
           write("Mother:\n"),
           M:cPerson::write_info(),
         fail.
      write_info().
endclass

implement cEmploye
   facts
      company(string Name)

   clauses
      add_company(Name):-
         assert(company(Name)).

      write_info():-
           cPerson::write_info(),
         fail.
      write_info():-
         company(X),
           write("Company=",X),nl,
         fail.
      write_info().
endclass
```

```
goal
   F = cPerson::new(),
   F:add_name("Arne"),
   O = cEmploye::new(),
   O:add_name("Leo"),
   O:add_father(F),
   O:add_company("PDC"),
   O:write_info(),
   F:delete(),
   O:delete().
```

The formal syntax for using the *members* of an object is:

```
[ObjectVariable:] [name_of_class::] name_of_member[ ( arguments ) ]
```

The object can be omitted inside the implementation of a class or for calling the class members, which were declared as static. For example, a call in the form:

```
member_name(arguments)
```

will be considered as a call to the corresponding member predicate ***member_name*** of that class (or its parents) in case it exists. Otherwise, if there are no members with the given name, it will be considered as a call to the predicate with the same name, which is declared in some **predicates** or **facts** section outside classes.

Names for members may be redefined in the hierarchy of classes. So, in order to refer to names of members in the previous scope, the name of class defined in call to some member may be used for explicit qualification of class in use:

```
[name_of_class::] name_of_member[ ( arguments ) ]
```

Virtual Predicates

In Visual Prolog all predicates declared in class declarations are what in the C++ terminology is called *virtual* methods. Virtual predicates allow derived classes to provide different versions of a parent class predicates. You can declare a predicate in a parent class and then redefine it in any derived class.

Assume, that a parent class *P* declares and defines a public predicate ***who_am_i***, and class *D* derived from *P* also has a declaration and definitions for ***who_am_i***. If ***who_am_i*** is called on an object of *D*, the call made is `D::who_am_i` even if access to ***who_am_i*** is via a reference to *P*. For example:

```
                        /* Program ch11e03.pro */

class P
   predicates
      test
      who_am_i()
endclass

class D : P
   predicates
      who_am_i()
endclass

implement P
   clauses
      test:-who_am_i().

      who_am_i():-
         write("I am of class P\n").
endclass

implement D
   clauses
      who_am_i():-
         write("I am of class D\n").
endclass

goal
   O = D::new,
   O:test,
   O:delete.
```

The output from the above program would be:

```
   I am of class D
```

Note, that if you define a predicate in a subclass with different domains or number of arguments, Prolog will treat this as a different declaration, and it will not work as a virtual predicate.

Static Facts and Predicates

It is possible to declare class member predicates and facts as being *static*.

Preceding the declaration of a **predicates** or a **facts** section with the keyword **static** specifies that all predicates or facts declared in the section are *static*. For example:

```
static facts
   single counter(integer)

static predicates
   procedure get_counter(integer)
```

Static class members belong to the class, not to the individual objects. This means that when a class declares a static member, then only one such static member exists for a class, rather than one per each object (like it is for non-static (*object*) class members). Static class members do not have any reference to personal objects of the class.

Public static members of a class can be used in any place where the class is visible (in any modules including the class declaration) even without creation of any object of the class.

Outside implementations of classes that inherit (transitively) a static class member, it can be qualified with the class name using the following syntax:

```
class_name :: static_class_member_name[(arguments)]
```

For example, one can access static predicate ***get_counter*** like this:

```
goal
   ...
   xclass::get_counter(Current_Counter),
```

Of course, a static member (as a non-static class members) can be qualified with references to objects of classes obtaining this static member.

Static facts are not generated for each instance of a class. Only one version of each static fact exists for the class and it is accessible for all instances of the class. This is useful, for example, to count the number of instances of a class.

Because, static predicates and static predicate values (see the *Predicate Values* on page 236) do not carry any internal reference to class instances, therefore, "ordinary" predicate domains can be used for declarations of static predicate values. See the example ch11e04.pro

```
                        /* Program ch11e04.pro */

class cCounter
   domains
      p_dom = determ (integer) - (o)  % predicate domain
```

```
   predicates
      procedure new()

   static predicates
      static_p_value : p_dom

endclass

implement cCounter
   static facts - db
      single count(integer)

   clauses
      count(0).

      new():-
         count(N),
         N1 = N+1,
         assert(count(N1)).

      static_p_value(N):-
         count(N),
         write("There are now ",N," instances of class cCounter\n\n").

endclass

goal
   cCounter::static_p_value(_),   % static member predicates can be called
                                                % before an object creation
   O = cCounter::new(),
   O:static_p_value(_),  % qualify static predicate with object identifier
   O2 = cCounter::new(),
   P_value = cCounter::static_p_value, % bound static predicate value
   P_value(_I),                  % call static predicate value by variable
   O:delete,
   O2:delete.
```

The output of the Test Goal for this program will be:

```
   There are now 0 instances of class cCounter
   There are now 1 instances of class cCounter
   There are now 2 instances of class cCounter
   O=6763168, O2=6763190,
   P_value=430BFC,
   _I=2
   1 Solution
```

Reference to the Object Itself (Predicate *this*)

Each non-static predicate has an invisible (to the programmer) extra parameter, which is a pointer (object identifier) to an actual instance of a class (an object) to which this predicate belongs.

In a clause like:

```
implement x
   clauses
      inc:-
         count(X),
         X1=X+1,
         assert(count(X1)).
endclass
```

The object is entirely invisible. If it is necessary to refer to the object itself for instance to access a predicate in an inherited class, it is possible to use the built-in predicate ***this***. The predicate ***this*** allows for an object to get access to any member predicate, which is defined in corresponding class or in its parents. The syntax for making call to ***this*** predicate is:

```
   this( ObjectIdentifier )                                    % (o)
```

The predicate ***this*** can be used only in non-static member predicates. Predicate ***this*** has the single output argument. The variable *ObjectIdentifier* in ***this*** predicate must be free; the anonimous variable (single underscore) is not allowed here. The domain of ***this*** argument is the *class domain* of an implementation section in which ***this*** is called.

For example:

```
IMPLEMENT x
   CLAUSES
      inc:-
         this(ObjectId),
         ObjectId:x::count(X),
         X1=X+1,
         assert(count(X1)).
ENDCLASS
```

This piece of code is functionally identical with the piece of code just above, with the only difference, that you get the object identifier of this object *ObjectId*. The obtained object identifier *ObjectId* can be passed on as a parameter to other predicates.

Class Scopes

A name scope is defined as the area, in which you can access it. Predicate, domain, and fact names may be redefined in the class hierarchy. In order to refer to names in a previous scope, a name can be preceded by a class name for explicit qualification of the class in use. The following notation can be used for explicit naming of a class:

```
   class_name :: member_name(arguments)
```

For example:

```
class parent
   predicates
      p(integer)
endclass

class child : parent
   predicates
      p(string, integer)
endclass

   % IMPLEMENTATION is not shown for clarity

goal
   O = child::new,
   O : parent::p(99)  % Access the definition in parent
```

Classes as Modules

Another usage of the explicit scoping is in using classes with static predicates and static facts as packages, like a module system.

Beginning with version 5.2, Visual Prolog provides possibility to declare domains inside classes. Being able to declare domains in classes opens the possibility to use classes as modules. Public domains declared in a class declaration are global and can be used outside the class.

If a class declares only static entities, then it can be considered a module.

The static entities of a class can be used as ordinary global entities, as long as you remember to qualify them with the class name. One advantage of creating modules this way is that the module will have a separate name space (as the result of qualification with the class name). This means that you can choose names in the module more freely. It also ensures consistent naming of all entities

in the module. Another advantage is that declarations of classes (except for abstract classes) do not have to be included in the main module, even if they contain declarations of public domains.

Example:

```
                        /* Program ch11e05.pro */

class cList
   domains
      ilist = integer*

static predicates
      append(ilist, ilist, ilist) - (i,i,o)
      ilist gen(integer)
endclass

implement cList
   clauses
      append([],L,L).
      append([H|L1],L2,[H|L3]):-
         append(L1,L2,L3).

      gen(0,[]):-!.
      gen(N,[N|L]):-
         N1=N-1,
         L = gen(N1).
endclass

goal
   L1 = cList::gen(3),
   L2 = cList::gen(5),
   cList::append(L1,L2,L3).
```

User-defined Constructors and Destructors

The Visual Prolog system itself allocates and initializes the memory during creation of an object. However there might still be the desire to specify how an object is created. For instance, initialize object facts to special values, create a window on the screen or open a file. In the same way, the programmer may wish to control how an object is deleted, for instance, closing windows or files.

For this purpose, in a class declaration section, it is possible to declare an explicit constructor(s) or destructor(s) for objects of the class. Explicit constructors are made by giving explicit declarations and definitions for predicates ***new*** in a class,

and an explicit destructor is made by giving an explicit definition and declaration for a predicate ***delete***.

```
                        /* Program ch11e06.pro */

CLASS cTest
   PREDICATES
      procedure new(String, String)
      procedure delete()
      add(STRING)
      writeStrings()
ENDCLASS cTest

IMPLEMENT cTest
   FACTS  - privateDB
      nondeterm strDB( STRING )

   CLAUSES
      new(StrHello, StrToLoad):-
         write(StrHello),
         assertz(strDB( StrToLoad )),
         write("Initial string is loaded into the database\n"),
         write("Constructor ended\n").

      delete():-
         writef("Simple destructor startup\n"),
         write("Delete all strings from the database\n"),
         retractall(strDB(_)),
         write("Destructor ended\n").

      add(STR):-
         assertz(strDB(STR)).

      writeStrings:-
         strDB(StringFromDataBase),
         writef("%s\n", StringFromDataBase),
         fail.
      writeStrings.

ENDCLASS cTest

GOAL
   O = cTest::new(
      "Simple constructor startup\n",
      "This is the initial string.\n"),nl,nl,
   O:add("Second String"),
   O:WriteStrings,
   O:delete.
```

Constructors and destructors are not allowed to fail; they must be **procedures**.

Predicates ***new*** and ***delete*** have many features of usual member predicates but they have some specific features.

In clauses of a predicate ***new***, it is possible to call constructors of base classes using the following syntax:

```
base_class_name::new
```

Several constructors and destructors with different arity or argument domains can be declared in a class.

Before doing any references to an object of a class, an object should be created with a call of any constructor ***new*** declared in the class. The compiler checks this.

If a predicate ***new*** is not explicitly declared in a class, then the default constructor ***new*** is used.

If an explicit constructor ***new*** is declared in a class, then the default constructor ***new*** cannot be called explicitly.

If a constructor or destructor exits with a runtime error, the object state is undefined.

An explicitly declared predicate ***new*** can be called in two different ways:

- As a constructor of class objects. The syntax is:

  ```
  Created_Object_Identifier = class_name :: new[(arguments)]
  ```

  For example:

  ```
  O = child::new,      % call new as class "child" constructor
  ```

  When called as the constructor, the predicate ***new*** creates a new object (instance of the class) and returns a reference to the created object.

- As an ordinary member predicate of a class. The syntax is:

  ```
  [Object_Identifier :] [class_name :: ] new[(arguments)]
  ```

  For example:

  ```
  O:parent::new,      % call new() as class "parent" member predicate
  ```

  When the ***new*** is called as a member predicate, it does not create a new instance of the class.

Similarly, an explicitly declared predicate ***delete*** can also be called in two different ways:

- On class objects, as a destructor of class objects. The syntax is:

```
Object_Identifier : [class_name ::] delete[(arguments)]
```

  For example:

```
O:parent::delete,
```

- As an ordinary member predicate of a class. When called as a member predicate ***delete*** does not destroy any object. The syntax is:

```
class_name :: delete[(arguments)]
```

  For example:

```
parent::delete(),
```

Abstract Classes

An abstract class is a class definition without an implementation. It is only inherited by subclasses. Abstract classes are used to provide general concepts (interfaces). . An abstract class is defined by the keyword **abstract** preceding the class declaration. The short syntax is:

```
abstract class <class_name> [: <base_class_list>]
   {[protected] domains <domains_declarations>}
   {[protected] predicates <predicates_declarations>}
endclass [<class_name>]
```

For example, if you want to create a browser that can work on many different kinds of data, you will open the browser by passing to it an object, which methods the browser can call to get the data and move forward or backward. By using an abstract class, the browser knows which predicates it can make calls to.

```
abstract class cBrowseInterface
   predicates
      string get_Current()
      next()
      prev()
endclass
```

```
class cDBinterface : cBrowseInterface
   PREDICATES
      new(Db_Selector,Chain)
      string get_Current()
      next()
      prev()
endclass

class cFileInterface : cBrowseInterface
   predicates
      string get_Current()
      next()
      prev()
endclass

class cBrowser
   predicates
      new(cBrowseInterface)
endclass
```

Another example you can find in OOP\Test\ABSTRACT.PRO.

Remarks:

- In case an abstract class inherits some base classes, these must also be declared abstract.
- It is impossible to declare facts in abstract classes.
- It is impossible to declare static predicates in abstract classes.
- Abstract classes cannot be specified in base class lists for class implementations.
- Declarations of all abstract classes must be included into the main module.

Beginning with Visual Prolog version 5.2, Visual Prolog's class system is much more compatible with COM. Abstract classes provide exactly the same VTABLE as the COM interface.

Protected Predicates, Domains, and Facts

It is possible to declare whether you can access predicates, domains, and facts from outside of a class. By default, all domains, facts and predicates declared in a class declaration are *public*. This means that they are accessible by any predicate in any module, which includes the class declaration.

The default access rights can be changed by preceding the keywords **facts**, **predicates** or **domains** in the section declaration with the keyword **protected**. Protected entities can be used only inside the class in which they are declared and in classes directly inherited from this class.

An example of the usage of protected predicates is callback event handlers, where subclasses might redefine some handling predicates, but it makes no sense to call these from the outside:

```
class window
   protected predicates
      onUpdate(rct)
      onCreate(long)
endclass
```

Derived Class Access Control

An important issue in building the hierarchies of objects correct, so that you can re-use as much code as possible, is inheritance. You can define methods on one level, and these can then be reused on lower levels. If a class inherits from other classes, we say, that this class is a derived class.

Visual Prolog supports so called "multiple inheritance". This means possibility to derive a class from several base classes at once. A resulting derived class thus inherits properties from several derectly inherited classes. Notice that this turns the *inheritance hierarchy* (or *hierarchy of derived classes*) into a *directed (acyclic) inheritance graph*. Notice that in case of simpler "single inheritence" the inheritance hierarchy can be represented as a *directed hierarchy tree*.

When you declare a derived class (for example, *D*), you can list several base classes (for example, *P1*, *P2*, *P3*, and *P4*) in a comma-delimited base class lists both in the declaration and in the implementation of the derived class *D*. For example,

```
   class D : P1, P2
```

and

```
   implement D : P3, P4
```

When base classes *P1* and *P2* are specified in the derived class *D* declaration, then *D* inherits from classes *P1* and *P2* all public domains, predicates, and facts. The inherited predicates and facts become public members of class *D* and they can be referenced as members of class *D*. On the other hand, *D* inherits only

possibility to use inherited public domains; inherited domains cannot be qualified with *D* name.

When base classes *P3* and *P4* are specified in the class *D* implementation, then *D* inherits from *P3* and *P4* all public domains, facts, and predicates, but they become private (local) to *D* and can be used only inside *D* implementation.

Derived class *D* does not have access to private names defined in base classes.

Redefined names can be accessed using scope overrides, if needed.

In case the name of some member can be achieved by several ways with the directed acyclic inheritance graph, the longest way is accepted.

If two or more predicates in a class hierarchy have the same name but different number or types of arguments, we say that this name is overloaded. We then have to consider the scope of the name, here defined as the area in which each predicate is valid. Every usage of name for any *member* of some class must be *unambiguous* (to an approximation of overloading). If a name denotes a predicate, a predicate call must be unambiguous with respect to number and type of arguments. The access to the member of base class is *ambiguous* if the expression of access names more than one member. The test for unambiguous is performed before the *access control*.

In case the synonymously defined name is the name of any overloaded member, then the scope of overloading is performed after the ambiguous control (but before the access control). The ambiguity can be scoped with explicit qualifying name of the *member* by the name of the corresponding *class*. The general syntax is the following:

```
[ObjectVariable:] [name_of_class::] name_of_member[ ( arguments ) ]
```

All predicates from class declarations, except ***new*** and ***delete***, are virtual. Opposite, all predicates declared in implementations become non-virtual.

Object Predicate Values

Visual Prolog supports a notion of *object predicate values*. Object predicate values are a generalization of *predicate values*. (See *Predicate Values* on page 236.) An object predicate value is a non-static predicate of a specific object. This is opposed to "ordinary" predicate values, which are either global, local or static predicates (predicate values).

Object predicate values are declared as instances of object predicate domains (see page 309).

Seen from the outside, an object predicate value looks like an "ordinary" predicate value, because the object to which it belongs is encapsulated by the predicate value itself. That is, an object predicate value consists of both the code and the reference onto the object, to which the code belongs. Therefore, the call of an object predicate value looks exactly the same as the call of an "ordinary" predicate value: It is simply applied to their arguments. But execution will (nevertheless) take place in the context of the specific object to which the object predicate value relates.

So the main reason for using object predicate values over "ordinary" predicate values is that execution will occur in a specific context of an object.

To illustrate this semantics let us consider an example:

First, we declare an object predicate domain:

```
   global domains
      objectIntInt = object procedure integer (integer)
```

The domain *objectIntInt* declares non-static (object) predicates. These predicates (functions) have one input argument from **integer** domain and return **integer** value. They have **procedure** determinism mode. Now let us declare a predicate of this domain. Because a predicate value of such domain must be a non-static member predicate, therefore, it must be declared in a class:

```
class cLast
   predicates
      last : objectIntInt
endclass cLast
```

To illustrate that last is indeed a non-static member predicate, we let it return a value dependent on the state of the object. In fact, we let it return the parameter it was called with at the previous invocation. Thus, we store the parameter from one invocation to the next in a fact:

```
implement cLast
   facts
         % invariant: "lastParameter" holds the parameter value
         % from the previous invocation of "last"
         % initially assume 0 as value from "last" invocation
      single lastParameter(integer Last)

   clauses
      lastParameter(0).
```

```
      last(ThisParameter, LastParameter) :-
         lastParameter(LastParameter),
         assert(lastParameter(ThisParameter)).

endclass cLast
```

So far, the only thing special about this example is the way the predicate ***last*** is declared. And before we will really use ***last*** as an object predicate value, let us use it like a normal non-static member predicate:

```
   predicates
      test1() - procedure ()

   clauses
      test1():-
         O1 = cLast::new(),
         O2 = cLast::new(),
         _1 = O1:last(1),
         _2 = O2:last(2),
         V1 = O1:last(3),
         V2 = O2:last(4),
         writef("V1 = %, V2 = %", V1, V2), nl,
         O1:delete(),
         O2:delete().
```

If we call ***test1*** then it will first create two objects *O1* and *O2*. Then it calls ***last*** on *O1* with parameter `1` and on *O2* with parameter `2`. Both *O1* and *O2* will store their parameter in respective instances of ***lastParameter*** fact. So when ***last*** is called again we will retrieve `1` and `2`, respectively. Subsequently, the call of ***test1*** will produce the output:

```
   V1 = 1, V2 = 2
```

Now let us do the same again, but this time we will use the predicates as values held in variables:

```
   predicates
      test2() - procedure ()
```

```
clauses
   test2() :-
      O1 = cLast::new(),
      O2 = cLast::new(),
      P1 = O1:last, % P1 and P2 are bound to instances
      P2 = O2:last, % of the object predicate domain objectIntInt
      _1 = P1(1),
      _2 = P2(2),
      V1 = P1(3),
      V2 = P2(4),
      writef("V1 = %, V2 = %", V1, V2), nl,
      O1:delete(),
      O2:delete().
```

The first thing to notice is that object predicate values ***P1*** and ***P2*** consist of both an object (*O1* or *O2*) and a predicate (***last***). The call of an object predicate value is, however, completely identical to calls of "ordinary" predicate values, i.e. you do not apply an object predicate value on an object, you simply call it with appropriate arguments.

The effect of running ***test2*** is exactly the same as when running ***test1***: Executing `P1(1)` will store `1` in the ***lastParameter*** fact of *O1*. Likewise, the next call of ***P1*** will retrieve the value stored in the ***lastParameter*** fact of *O1*. And completely similarly ***P2*** will refer to *O2*.

Object predicate values are at least as useful for callbacks as "ordinary" predicate values (please refer to the description of predicate values (see page 236) for a discussion of callbacks). The benefit from using object predicate values (over "ordinary" predicate values) is that the callback comes back to a specific context, namely to the object to which the callback belongs. This makes it possible to deal with several different callbacks of the same kind because each callback will end up in its own context.

Let us walk through an example. Assume that we have a number of "things" that are interested in knowing when a certain value changes. (For the sake of the example this value is simply an integer.) These things want to be notified asynchronously about changes in the value. Therefore, they register a "***dataReady***" listener at a data ready source. In this example we choose to transfer the new value together with the data ready notification, but with more complex data we might let the listener pick up the data itself.

We represent the data ready source by an object of class *cDataReadySource*. If we have several pieces of data that can "become ready", then we can use one instance of *cDataReadySource* per data piece, making it possible to listen to notifications for exactly those of interest. Class *cDataReadySource* supports

registering and unregistering of listeners. It also has a predicate for setting the value in question.

Listeners are represented by object procedure values (i.e. object callbacks).

```
class cDataReadySource
   domains
      dataReadyListener = object procedure
         (cDataReadySource EventSource,integer NewValue)

   predicates
      addDataReadyListener(dataReadyListener Listener) - procedure (i)
      removeDataReadyListener(dataReadyListener Listener) - procedure (i)

   predicates
      setValue(integer NewValue) - procedure (i)

endclass cDataReadySource
```

The implementation is quite straightforward. We store the currently registered listeners in a fact, and when the data is changed, we notify all registered listeners about this.

```
implement cDataReadySource
   facts
         % Invariant: listener_db contains the currently registered
         % listeners (multiple registrations are ignored)
      listener_db(dataReadyListener Listener)

   clauses
      addDataReadyListener(Listener):-
         listener_db(Listener), % already registered
         !.
      addDataReadyListener(Listener):-
         assert(listener_db(Listener)).

      removeDataReadyListener(Listener) :-
         retractAll(listener_db(Listener)).

   predicates
      dataIsReady(integer NewValue) - procedure (i)

   clauses
      dataIsReady(NewValue):-
         this(This),
         listener_db(Listener),
         Listener(This, NewValue),
         fail.
      dataIsReady(_).
```

```
   clauses
      setValue(NewValue) :-
         dataIsReady(NewValue).

endclass cDataReadySource
```

Let us also try to use the class above in a context. Assume that we have a system, which counts how many users are active, this count is used in a number of places. One of these places is a status window, which displays the count. For the sake of the example, we imagine that there is a global predicate ***getUserCountSource***, which will return a *cDataReadySource* object corresponding to the count.

```
   global predicates
      cDataReadySource getUserCountSource() - procedure ()
```

We implement our status window as a class *cStatusWindow*. The declaration of *cStatusWindow* is not very interesting in this context; all we are concerned with is the implementation. In the implementation we put our ***dataReadyListener*** and in the constructor of the class we register this listener with the user count data source. We, of course, also unregister the listener in the destructor.

```
implement cStatusWindow
   predicates
      updateWindow(integer NewValue) - procedure (i)

   clauses
      updateWindow(NewValue) :- ... % window updating code

   predicates
      onUserCountChanged : cDataReadySource::dataReadyListener

   clauses
      onUserCountChanged(_Source, NewValue) :-
         updateWindow(NewValue).

   clauses
      new() :-
         UserCountSource = getUserCountSource(),
         UserCountSource:addDataReadyListener(onUserCountChanged).
             % THIS is subsumed

      delete() :-
         UserCountSource = getUserCountSource(),
         UserCountSource:removeDataReadyListener(onUserCountChanged).

endclass cStatusWindow
```

No matter how many status windows we create they all will be updated when the user count changes.

Object Predicate Domain Declarations

An *object predicate domain* declares a type of non-static (object) member predicates.

The object predicate domains can be used for declarations of non-static predicates that can be used as *object predicate values* (see page 303). These object predicate values can be passed as arguments to other predicates.

Object predicate domains can be declared both outside and inside classes, but in contrast to "ordinary" predicate domains, object predicate domains can be used for declarations of object predicate values only inside classes. This is because, when an object predicate value is passed as an argument, it should carry the reference to the actual class instance (object) to which the predicate value belongs.

The declaration of an object predicate domain is of the form:

```
[global] domains
   PredDom = object
      DetermMode [ReturnDom] (ArgList) [- [FlowPattern]] [Language]
```

Here the keyword **object** states declaration of the object predicate domain. This is the only difference from syntax for declaration of "ordinary" predicate domains described with all details on page 238. Therefore, here we only shortly remind used terms and point to differences. ***PredDom*** declares the name of the predicate domain. ***DetermMode*** specifies the determinism mode. ***ReturnDom*** defines the domain of the return value for functions. ***ArgList*** defines domains for arguments. ***FlowPattern*** specifies argument flows. It must be `'i'` for input arguments and `'o'` for output arguments. Only one flow pattern can be specified. If it is absent, then the default flow pattern with all input arguments is accepted implicitly. *Language* is of the form:

```
language { pascal | stdcall | asm | c | syscall | prolog}
```

The default calling convention is **pascal**. Notice the difference with declarations of predicates, where the default is **prolog**.

Restriction. If an object predicate value is declared as an instance of an object predicate domain with calling conventions **prolog**, **syscall**, **c** or **asm**, then this object predicate value cannot be called on arguments if it is passed in a variable.

For example, the declaration of object predicate domain for deterministic predicates taking an **integer** as argument and returning an **integer**, would be:

```
   domains
      list_process = object determ integer (integer) - (i)
```

The example ch11e07.pro demonstrates using of object predicate domain *obj_dom* for the declaration of object predicate value ***opv_a***.

```
                        /* Program ch11e07.pro */

domains
   obj_dom = object procedure (string) - (i) % object predicate domain

class cl_a
   predicates
      opv_a : obj_dom  % declaration of object predicate value opv_a

endclass cl_a

implement cl_a
   clauses
      opv_a( S ) :-
         write( S ), nl.

endclass cl_a

class cl_b : cl_a
   predicates
      p_ns_b(obj_dom,string)

   static predicates
      p_s_b(obj_dom,string)   % static predicate p_s_b

endclass cl_b

implement cl_b
   clauses
      p_ns_b(OP_value,S) :-
         OP_value(S), nl.

      p_s_b(OP_value,S) :-
         OP_value(S), nl.
endclass cl_b

   predicates
      p_local(obj_dom, string) - procedure (i,i)
```

```
   clauses
      p_local(OP_value, S):-
      OP_value(S).

goal
   O_a = cl_a::new(),
   O_a:opv_a("I am opv_a!"),
   write("\t Object Predicate (OP) <opv_a> is called directly."), nl,nl,
   OP_value_a = O_a:opv_a,
   OP_value_a("OP value <opv_a> is called as variable"), nl,
   p_local(O_a:opv_a,
      "OP <opv_a> is explicitly specified in predicate argument"),nl,
   p_local(OP_value_a,
      "OP value <opv_a> is passed to predicate in variable !"),nl,
   cl_b::p_s_b(O_a:opv_a,
      "OP <opv_a> is specified as argument to static predicate!"),nl,
   O_b = cl_b::new(),
   O_b:p_ns_b(OP_value_a,
      "OP value <opv_a> is passed in variable to non-static predicate!" ),
   nl,
   O_a:delete(),
   O_b:delete().
```

Formal Syntax for Classes

```
<class> ::=
   <class_declaration> <class_implementation>

<abstract_class> ::=
   <abstract_class_declaration>

<class_declaration > ::=
   <class_declaration_begin>
   <class_declaration_body>
   <class_end>

<abstract_class_declaration> ::=
   <abstract_class_declaration_begin>
   <abstract_class_declaration_body>
   <class_end>

<class_declaration_begin > ::=
   CLASS <class_header>

<abstract_class_declaration_begin>  ::=
   ABSTRACT CLASS <class_header>
```

```
<class_header> ::=
   <class_name> [: <base_class_list>]

<class_name > ::=
   identifier

<base_class_list > ::=
   <base_class_name_1> [,<base_class_name_2>]*

<class_declaration_body> ::=
   <class_declaration_section>*

<abstract_class_declaration_body> ::=
   <abstract_class_declaration_section>*

<class_declaration_section> ::=
   <class_declaration_domains_section>
   | <class_declaration_facts_section>
   | <class_declaration_predicates_section>

<abstract_class_declaration_section> ::=
   <class_declaration_domains_section>
   | <abs_class_declaration_predicates_section>

<class_declaration_domains_section> ::=
   [PROTECTED] DOMAINS
      <domains_declarations>

<class_declaration_facts_section > ::=
   [STATIC] [PROTECTED ] FACTS [ - <facts_section_name>]
      <facts_declarations>
<facts_section_name> ::=
   identifier

<class_declaration_predicates_section > ::=
   [STATIC] [PROTECTED] PREDICATES
      <predicates_declarations>

<abs_class_declaration_predicates_section> ::=
   [PROTECTED] PREDICATES
      <predicates_declarations>

<class_end> ::=
   ENDCLASS [ <class_name> ]

<class_implementation> ::=
   <class_implementation_begin>
   <class_implementation_body>
   <class_end>
```

```
<class_implementation_begin> ::=
   IMPLEMENT <class_header>

<class_implementation_body> ::=
   <class_implementation_section>*

<class_implementation_section> ::=
   <class_implementation_domains_section>
   | <class_implementation_facts_section>
   | <class_implementation_predicates_section>
   | <CLAUSES_section>

<class_implementation_facts_section> ::=
   [STATIC ] FACTS [ - <facts_section_name>]
      <facts_declarations>

<class_implementation_predicates_section>  ::=
   [STATIC] PREDICATES
      <predicates_declarations>
```

Here:

- The square brackets indicate optional items.
- The curly braces indicate arbitrary number of items (zero or more items).
- The symbol '|' means that one of the items separated by the symbol '|' must be chosen.
- The asterisk symbol '*' indicates arbitrary number of the immediately preceding item (zero or more items).

PART 3

Tutorial Chapters 12 – 17: Using Visual Prolog

CHAPTER 12

Writing, Reading, and Files

In this chapter, we first cover the basic set of built-in predicates for writing and reading. Next we describe how the file system works in Visual Prolog and show how you can redirect both input and output to files. We also discuss the ***file*** domain and some predefined files.

Writing and Reading

In these tutorials, most of the Input/Output has been interactive via screen and keyboard. In this section, we provide formal coverage of the standard predicates you use for I/O, including predicates for file operations.

Writing

Visual Prolog includes three standard predicates for writing. These predicates are ***write, nl*** and ***writef***.

write/* and nl

The predicate ***write*** can be called with an arbitrary number of arguments:

```
write(Param1, Param2, Param3, ..., ParamN)
                                              /* (i, i, i, ..., i) */
```

These arguments can either be constants from standard domains or they can be variables. If they're variables, they must be input parameters.

The standard predicate ***nl*** (for new line) is often used in conjunction with ***write***; it generates a newline on the display screen. For example, the following subgoals:

```
pupil(PUPIL, CL),
write(PUPIL," is in the ",CL," class"),
nl,
write("----------------------------------").
```

could result in this display:

```
Helen Smith is in the fourth class
---------------------------------
```

while this goal:

```
....,
write("List1= ", L1, ", List2= ", L2 ).
```

could give:

```
List1= [cow,pig,rooster], List2= [1,2,3]
```

Also, if ***My_sentence*** is bound to

```
sentence(subject(john),sentence_verb(sleeps))
```

in the following program

```
DOMAINS
   sentence = sentence(subject, sentence_verb)
   subject = subject(symbol) ; ......
   sentence_verb = sentence_verb(verb) ; ......
   verb = symbol

CLAUSES
   ....
   write( " SENTENCE= ", My_sentence ).
```

you would obtain this display:

```
SENTENCE= sentence(subject(john),sentence_verb(sleeps))
```

Note that with respect to strings, the backslash (\) is an escape character. To print the backslash character verbatim, you must type two backslashes. For example, to designate the DOS directory path name A:\PROLOG\MYPROJS\MYFILE.PRO in a Visual Prolog program, you would type `a:\\prolog\\myprojs\\myfile.pro`.

If a backslash is followed by one of a few specially recognized characters, it will be converted to a print control character. These are

```
'n'     newline and carriage return
't'     tab
'r'     carriage return
```

Alternatively, the backslash may be followed by up to three decimal digits, specifying a particular ASCII code. However, avoid putting \0 into a string

unless you know what you're doing. Visual Prolog uses the C convention with 0-terminated strings.

Be very careful with the '\r' option. It sets the current write position back to the start of the current line, but if you accidentally do that in between writing different things, it may happen so quickly that the first thing you write becomes overwritten before you even notice it's there. Also, if you write something which is too long for a single line, causing the output to wrap, the '\r' will set the cursor back to the beginning of the last line, not the beginning of the line where the writing started.

Often ***write*** does not, by itself, give you as much control as you'd like over the printing of compound objects such as lists, but it's easy to write programs that give better control. The following four small examples illustrate the possibilities.

Examples Demonstrating the write Predicate

These examples show how you can use ***write*** to customize your own predicates for writing such things as lists and compound data structures.

1. Program ch11e01.pro prints out lists without the opening bracket ([) and closing bracket (]).

```
                        /* Program ch12e01.pro */

DOMAINS
   integerlist = integer*
   namelist    = symbol*

PREDICATES
   writelist(integerlist)
   writelist(namelist).

CLAUSES
   writelist([]).
   writelist([H|T]):-
      write(H, " "),
      writelist(T).
```

   Notice how this program uses recursion to process a list. Load the program and try this goal:

```
   writelist([1, 2, 3, 4]).
```

2. The next example, Program ch11e02.pro, writes out the elements in a list with no more than five elements per line.

```
                        /* Program ch12e02.pro */

DOMAINS
   integerlist = integer*

PREDICATES
   writelist(integerlist)
   write5(integerlist,integer)

CLAUSES
   writelist(NL):-
      nl,
      write5(NL,0),nl.
   write5(TL,5):-!,
      nl,
      write5(TL, 0).
   write5([H|T],N):-!,
      write(H," "),
      N1=N+1,
      write5(T,N1).
   write5([],_).
```

If you give the program this goal:

```
writelist([2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]).
```

Visual Prolog responds with:

```
2 4 6 8 10
12 14 16 18 20
22
```

3. Frequently, you may want a predicate that displays compound data structures in a more readable form. Program ch11e03.pro displays a compound object like:

```
plus(mult(x, number(99)), mult(number(3), x))
```

in the form:

```
x*99+3*x
```

(This is known as *infix notation*.)

```
                        /* Program ch12e03.pro */

DOMAINS
   expr = number(integer); x; log(expr);
   plus(expr, expr); mult(expr, expr)
```

```
PREDICATES
   writeExp(expr)

CLAUSES
   writeExp(x):-write('x').
   writeExp(number(No)):-write(No).
   writeExp(log(Expr)):-
      write("log("), writeExp(Expr), write(')').
   writeExp(plus(U1, U2)):-
      writeExp(U1),write('+'),writeExp(U2).
   writeExp(mult(U1,U2)):-
      writeExp(U1),write('*'),writeExp(U2).
```

4. Program ch11e04.pro is similar to Program ch11e03.pro.

```
                        /* Program ch12e04.pro */

DOMAINS
   sentence   = sentence(nounphrase, verbphrase)
   nounphrase = nounp(article, noun); name(name)
   verbphrase = verb(verb); verbphrase(verb, nounphrase)
   article, noun, name, verb = symbol

PREDICATES
   write_sentence(sentence)
   write_nounphrase(nounphrase)
   write_verbphrase(verbphrase)

CLAUSES
   write_sentence(sentence(S,V)):-
      write_nounphrase(S),write_verbphrase(V).
   write_nounphrase(nounp(A,N)):-
      write(A,' ',N,' ').
   write_nounphrase(name(N)):-write(N,' ').
   write_verbphrase(verb(V)):-write(V,' ').
   write_verbphrase(verbphrase(V,N)):-
      write(V,' '),write_nounphrase(N).
```

   Try Program ch11e04.pro with this goal:

```
      write_sentence(sentence(name(bill), verb(jumps))).
```

Exercise

Write a Visual Prolog program that, when given a list of addresses contained in the program as clauses of the form:

```
address("Sylvia Dickson", "14 Railway Boulevard","Any Town", 27240).
```

displays the addresses in a form suitable for mailing labels, such as:

```
Sylvia Dickson
14 Railway Boulevard
Any Town
27240
```

writef/*

The ***writef*** predicate allows you to produce formatted output; it uses this format:

```
writef(FormatString, Arg1, Arg2, Arg3, ...,ArgN)
                                              /* (i, i, i, i, ..., i) */
```

Arg1 to *ArgN* must be constants or bound variables belonging to standard domains; it is not possible to format compound domains. The format string contains *ordinary characters* and *format specifiers;* ordinary characters are printed without modification, and format specifiers take the following form:

```
%-m.pf
```

The characters in the format specifiers following the % sign are optional and have these meanings:

hyphen (-)	Indicates that the field is to be left-justified (right-justified is the default).
m field	Decimal number specifying a minimum field length.
.*p* field	Decimal number specifying either the precision of a floating-point number or the maximum number of characters to be printed from a string.
f field	Specifies formats other than the default format for the given object. For example, the *f* field can specify that ***integers*** should be printed as unsigned or hexadecimal numbers.

Visual Prolog recognizes the following format specifiers in the f field:

`f`	reals in fixed-decimal notation (such as 123.4 or 0.004321)
`e`	reals in exponential notation (such as 1.234e2 or 4.321e-3)
`g`	reals in the shorter format of f or e (this is the default for reals)

d	integral domains as a signed decimal number
D	integral domains as a signed long decimal number
u	integral domains as an unsigned decimal integer
U	integral domains as an unsigned long decimal integer
o	integral domains as an octal number
O	integral domains as an octal long number
x	integral domains as a hexadecimal number
X	integral domains as a long hexadecimal number
c	integral domains as a char
s	as a string (***symbols*** and ***strings***)
R	as a database reference number (***ref*** domain only)
B	as a binary (***binary*** domain only)
P	as a predicate value

The ***ref*** domain will be described in chapter **14**, and the ***binary*** and predicate values in chapter **10**.

For the integral domain specifiers, an uppercase format letter denotes that the associated object is a long type. If no format letter is given, Visual Prolog will automatically select a suitable format.

Examples of Formatted Output

1. The following example program illustrates the effect of different format specifiers on output formatted with ***writef***.

```
                        /* Program ch12e05.pro */

% Note that the format strings in this example specify 16bit integers

GOAL
      A = one,
      B = 330.12,
      C = 4.3333375,
      D = "one two three",
      writef("A = '%-7' \nB = '%8.1e'\n",A,B),
      writef("A = '%' \nB = '%8.4e'\n",A,B),nl,
      writef("C = '%-7.7g' \nD = '%7.7'\n",C,D),
      writef("C = '%-7.0f' \nD = '%0'\n",C,D),
      writef("char: %c, decimal: %d, octal: %o, hex: %x",'a','a','a','a').
```

When run, this program will produce the following output:

```
A = 'one    '
B = ' 3.3E+02'
A = 'one'
B = '3.3012E+02'
C = '4.3333375'
D = 'one two'
C = '4      '
D = 'one two three'
char: a, decimal: 97, octal: 141, hex: 61
```

2. Here's another example, showing how you can use ***writef*** to format your output.

```
                        /* Program ch12e06.pro */

facts
   person(string,integer,real)

CLAUSES
   person("Pete Ashton",20,11111.111).
   person("Marc Spiers",32,33333.333).
   person("Kim Clark",28,66666.666).

GOAL
      % Name   is left-justified, at least 15 characters wide
      % Age    is right-justified, 2 characters wide
      % Income is right-justified, 9 characters wide, with 2
      %        decimal places, printed in fixed-decimal notation
```

```
person(N, A, I),
writef("Name= %-15, Age= %2, Income= $%9.2f \n",N,A,I),
fail
;
true.
```

This produces the following:

```
Name= Pete Ashton    , Age= 20, Income= $  11111.11
Name= Marc Spiers    , Age= 32, Income= $  33333.33
Name= Kim Clark      , Age= 28, Income= $  66666.67
```

Reading

Visual Prolog includes several standard predicates for reading. The four basic ones are ***readln*** for reading whole lines of characters, ***readchar*** for reading single characters/keystrokes, ***readint*** for reading integers, and readreal for reading floating point numbers. Additionally, ***readterm*** will read any term, including compound objects. These predicates can all be redirected to read from files.

Another, more specialized, predicate that belong in the reading category is ***file_str*** for reading a whole text file into a string.

readln/1

readln reads a line of text; it uses this format:

```
readln(Line)                                                    /* (o) */
```

The domain for the variable *Line* will be a string. Before you call ***readln***, the variable *Line* must be free. readln reads up to 254 characters (plus a carriage return) from the keyboard, up to 64K from other devices. If **Esc** is pressed during input from the keyboard, ***readln*** will fail.

readint/1, readreal/1, and readchar/1

readint reads an integer value, using this format:

```
readint(X)                                                      /* (o) */
```

The domain for the variable X must be of ***integer*** type, and X must be free prior to the call. ***readint*** will read an integer value from the current input device (probably the keyboard) until the **Enter** key is pressed. If the line read does not correspond to the usual syntax for integers, ***readint*** fails and Visual Prolog

invokes its backtracking mechanism. If **Esc** is pressed during input from the keyboard, ***readint*** will fail.

readreal does as its name conveys: it reads a ***real*** number (as opposed to ***readint***, which reads an integer). ***readreal*** uses the following format:

```
readreal(X)                                                    /* (o) */
```

The domain for the variable X must be ***real***, and X must be free prior to the call. ***readreal*** will read a real value from the current input device until the **Enter** key is pressed. If the input does not correspond to the usual syntax for a real number, ***readreal*** fails. If **Esc** is pressed during input from the keyboard, ***readreal*** will fail.

readchar reads a single character from the current input device, using this format:

```
readchar(CharParam)                                            /* (o) */
```

CharParam must be a free variable before you call ***readchar*** and must belong to a domain of ***char*** type. If the current input stream is the keyboard, ***readchar*** will wait for a single character to be typed before it returns. If Esc is pressed during input from the keyboard, ***readchar*** will fail.

readterm/2

readterm reads a compound term and converts it to an object; it takes this format:

```
readterm(DomainName, Term)                                  /* (i, i) */
```

You call ***readterm*** with two arguments: a domain name and a term. ***readterm*** reads a line and converts it to an object of the given domain. If the line does not look like ***write*** would have formatted the object, ***readterm*** gives an error. The standard predicate ***readtermerror*** may be used in connection with a trap to produce customized error handling for ***readterm***. See chapter **10**.

readterm is useful for handling terms in text files. For example, you can implement you own version of ***consult***.

file_str/2

file_str reads characters from a file and transfers them to a variable, or creates a file and writes the string into the file. It uses this format:

```
file_str(Filename, Text)                              /* (i, o), (i, i) */
```

If, before your program calls ***file_str***, the variable *Text* is free, ***file_str*** reads the entire contents of the file *Filename* into *Text*. In the DOS-related versions of Visual Prolog, an eof character (**Ctrl+Z**) will terminate reading when encountered and will not be included in the string.

For example, the call

```
file_str("T.DAT", My_text)
```

binds *My_text* to the contents of the file T.DAT. The file size can't exceed the maximum length of a string, which is 64 Kbytes on the 16-bit platforms. If the file exceeds the maximum size, ***file_str*** will return an error message.

With *My_text* bound to the text in T.DAT, `file_str("T.BAK", My_text)` will create a file called T.BAK that contains the text from `T.DAT`. If T.BAK already exists it will be overwritten.

Examples

These examples demonstrate how you can use the standard reading predicates to handle compound data structures and lists as input.

1. Program ch11e07.pro illustrates assembling of compound objects from individually read pieces of information.

```
                        /* Program ch12e07.pro */

DOMAINS
   person = p(name, age, telno, job)
   age = integer
   telno, name, job = string

PREDICATES
   readperson(person)
   run

CLAUSES
   readperson(p(Name,Age,Telno,Job)):-
      write("Which name ? "), readln(Name),
      write("Job ? "), readln(Job),
      write("Age ? "), readint(Age),
      write("Telephone no ? "), readln(Telno).

   run :-
      readperson(P),nl,write(P),nl,nl,
      write("Is this compound object OK (y/n)"),
      readchar(Ch),Ch='y', !.
```

```
   run :-
      nl,write("Alright, try again"),nl,nl,run.

GOAL
   run.
```

2. This next example reads one integer per line until you type a non-integer (such as the X key), then ***readint*** will fail and Visual Prolog displays the list.

```
                        /* Program ch12e08.pro */

DOMAINS
   list = integer*

PREDICATES
   readlist(list)

CLAUSES
   readlist([H|T]):-
      write("> "),
      readint(H),!,
      readlist(T).
   readlist([]).

GOAL
      write("*************** Integer List *****************"),nl,
      write(" Type in a column of integers, like this:"),nl,
      write("  integer (press ENTER)\n  integer (press ENTER)\n"),
      write("  etc.\n\n Type X (and press ENTER) to end the list.\n\n"),
      readlist(TheList),nl,
      write("The list is: ",TheList).
```

Load Program ch12e08.pro and run it. At the prompt, enter a column of integers (such as 1234 **Enter** 567 **Enter** 89 **Enter** 0 **Enter**), then press X **Enter** to end the list.

Exercise

Write and test clauses for a predicate ***readbnumb***, which, in the call:

```
   readbnumb(IntVar)
```

converts a user-input, 16-bit binary number like "1111 0110 0011 0001" to a corresponding integer value to which *IntVar* is bound. Check your work by writing a program that contains readbnumb.

Binary Block Transfer

Three standard predicates allow reading and writing of binary blocks, or byte sequences of a given length. They all use the binary standard domain. This emulates an array, with a word (***dword*** on the 32-bit versions of Visual Prolog) in front, holding the size. For a complete description of the ***binary*** domain, see chapter **10**.

readblock/2

readblock has the following format:

```
readblock(Length,BTerm)                                    /* (i, o) */
```

where *Length* is the number of bytes to read and *BTerm* is the returned binary term. As described in chapter 11, there are no restrictions on the contents of a binary term, and it will contain whatever was read from the file including zeros and DOS eof-markers (**Ctrl+Z**).

The current input device must be assigned to a file (see ***readdevice*****).**

If *Length = 0* is specified, the ***readblock*** attempts to read maximum possible number of bytes from an input device. (Remember that *BinBlock* < 64 K on 16-bit platforms).

If *Length* is larger than the actual remaining number of bytes in the file - then the ***readblock*** generates the run-time error 1111: "Wrong number of bytes read from file".

writeblock/2

writeblock complements readblock:

```
writeblock(Length,BTerm)                                   /* (i, i) */
```

As with ***readblock***, there are no restrictions on the contents of *BTerm*. The *Length* specifies how many bytes to write; a length of 0 will write the whole term.

For compatibility with previous versions of Visual Prolog, where binary blocks were disguised as strings, ***writeblock*** may be called with a string argument instead of a binary term. In this case, it is imperative that *Length* contains the correct number of bytes to write.

file_bin/2

file_bin will read a whole file into a binary term, and vice versa. It takes two arguments, the filename and the binary term:

```
file_bin(FileName,BinTerm)                          /* (i, o) (i, i) */
```

Visual Prolog's File System

In this section, we give you a look at Visual Prolog's file system and the standard predicates relevant to files. We also introduce I/O redirection, an efficient method for routing input and output to various devices. With a few exceptions, the file system works identically in the different versions of Visual Prolog.

Visual Prolog uses a *current_read_device*, from which it reads input, and a *current_write_device*, to which it sends output. Normally, the keyboard is the current read device, and the screen is the current write device. However, you can specify other devices. For instance, input could be read from an externally stored file (on disk perhaps). Not only can you specify other devices, you can even reassign the current input and output devices while a program is running.

Regardless of what read and write devices you use, reading and writing are handled identically within a Visual Prolog program. To access a file, you must first open it. A file can be opened in one of four ways:

- for reading
- for writing
- for appending
- for modification

A file opened for any activity other than reading must be closed when that activity is finished, or the changes to the file might be lost. You can open several files simultaneously, and input and output can be quickly redirected between open files. In contrast, it takes much longer to open and close a file than to redirect data between files.

When Visual Prolog opens a file, it connects a symbolic name to the actual file name. Visual Prolog uses this symbolic name when directing input and output. Symbolic file names must start with a lower-case letter and must be declared in the ***file*** domain declaration like this:

```
file = file1; source; auxiliary; somethingelse
```

Only one ***file*** domain is allowed in any program. Visual Prolog recognizes five predefined ***file*** alternatives:

keyboard	reading from the keyboard (default)
screen	writing to the screen
stdin	reading from standard input
stdout	writing to standard output
stderr	writing to standard error

These predefined alternatives must not appear in the ***file*** domain declaration; they don't need to be opened and they should not be closed. Note, that when using the VPI strategy, only the screen alternative can be used.

Opening and Closing Files

The following sections describe the standard predicates for opening and closing files.

Note: In the DOS-related versions of Visual Prolog, remember that the backslash character, used to separate subdirectories, is an escape character. You must always use two backslash characters when you give a path in the program, for example, the string

```
"c:\\prolog\\include\\iodecl.con"
```

represents the path name

```
c:\prolog\include\iodecl.con
```

openread/2

openread opens the file *OSFileName* for reading, using this format:

```
openread(SymbolicFileName, OSFileName)                    /* (i, i) */
```

Visual Prolog refers to the opened file by the symbolic name *SymbolicFileName* declared in the ***file*** domain. If the file can't be opened, Visual Prolog returns an error message.

openwrite/2

openwrite opens the file *OSFileName* for writing; it takes this format:

```
openwrite(SymbolicFileName, OSFileName)                    /* (i,i) */
```

If the file already exists, it is erased. Otherwise, Visual Prolog creates a new file and makes an entry in the appropriate directory. If the file can't be created, the predicate exits with an error message.

openappend/2

openappend opens the file *OSFileName* for writing at the end, using this format:

```
openappend(SymbolicFileName, OSFileName)                    /* (i, i) */
```

If the file can't be opened for write access, Visual Prolog issues an error message.

openmodify/2

openmodify opens the file *OSFileName* for both reading and writing; if the file already exists, it won't be overwritten. ***openmodify*** takes this format:

```
openmodify(SymbolicFileName, OSFileName)                    /* (i, i) */
```

If the system can't open *OSFileName*, it issues an error message. ***openmodify*** can be used in conjunction with the ***filepos*** standard predicate to update a random-access file.

filemode/2

When a file has been opened, ***filemode*** sets the specified file to text mode or binary mode, using this format:

```
filemode(SymbolicFileName, FileMode)                        /* (i, i) */
```

If *FileMode* = 0, the file specified by *SymbolicFileName* is set to binary mode; if *FileMode* = 1, it's set to text mode.

In text mode, new lines are expanded to carriage- return/line-feed pairs during writes, and carriage-return/line-feed pairs are converted to newlines during reads.

Carriage return	= ASCII 13
Line feed	= ASCII 10

In binary mode, no expansions or conversions occur. To read a binary file, you can only use ***readchar*** or the binary file-access predicates discussed in chapter **10**.

filemode is only relevant in the DOS-related versions of Visual Prolog. In the UNIX version it has no effect.

closefile/1

closefile closes the indicated file; it takes this format:

```
closefile(SymbolicFileName)                                    /* (i) */
```

This predicate always succeeds, even if the file has not been opened.

readdevice/1

readdevice either reassigns the *current_read_device* or gets its name; the predicate takes this format:

```
readdevice(SymbolicFileName)                                   /* (i), (o) */
```

readdevice reassigns the current read device if *SymbolicFileName* is bound and has been opened for reading. If *SymbolicFileName* is free, ***readdevice*** binds it to the name of the current active read device.

writedevice/1

writedevice either reassigns or gets the name of the *current_write_device*; it takes this format:

```
writedevice(SymbolicFileName)                                  /* (i), (o) */
```

writedevice reassigns the current write device if the indicated file has been opened for either writing or appending. If *SymbolicFileName* is free, ***writedevice*** binds it to the name of the current active write device.

Examples

1. The following sequence opens the file MYDATA.FIL for writing, then directs all output produced by clauses between the two occurrences of ***writedevice*** to that file. The file is associated with the symbolic file name *destination* appearing in the declaration of the ***file*** domain.

```
DOMAINS
   file = destination

GOAL
   openwrite(destination, "MYDATA.FIL"),
   writedevice(OldOut),                /* gets current output device */
   writedevice(destination),         /* redirects output to the file */
      :
      :
   writedevice(OldOut),                      /* resets output device */
```

2. Program ch11e09.pro uses some standard read and write predicates to construct a program that stores characters typed at the keyboard in the file TRYFILE.ONE. Characters typed are not echoed to the display; it would be a good exercise for you to change the program so that characters are echoed. The file is closed when you press the # key.

```
                    /* Program ch12e09.pro */

DOMAINS
   file = myfile

PREDICATES
   readloop

CLAUSES
   readloop:-
      readchar(X),
      X<>'#',!,
      write(X),
      readloop.
   readloop.

GOAL
      write("This program reads your input and writes it to"),nl,
      write("tryfile.one. For stop press #"),nl,
      openwrite(myfile,"tryfile.one"),
      writedevice(myfile),
      readloop,
      closefile(myfile),
      writedevice(screen),
      write("Your input has been transferred to the file tryfile.one"),nl.
```

Redirecting Standard I/O

The ***file*** domain has three additional options: stdin, stdout, and stderr. The advantage of these file streams is that you can redirect I/O at the command line.

stdin Standard input is a read-only file - the keyboard, by default. `readdevice(stdin)` directs the input device to stdin.

stdout Standard output is a write-only file that defaults to the screen. `writedevice(stdout)` directs the output device to stdout.

stderr Standard error is a write-only file that defaults to the screen. `writedevice(stderr)` directs the output device to stderr.

Working with Files

In this section, we describe several other predicates used for working with files; these are: ***filepos, eof, flush, existfile, searchfile, deletefile, renamefile, disk,*** and ***copyfile***.

filepos/3

filepos can control the position where reading or writing takes place; it takes the form

```
filepos(SymbolicFileName, FilePosition, Mode)    % (i, i, i), (i, o, i)
```

With *FilePosition* bound, this predicate can change the read and write position for the file identified by *SymbolicFileName*. It can return the current file position if called with *FilePosition* free. *FilePosition* is a ***long*** value.

Mode is an ***integer*** and specifies how the value of *FilePosition* is to be interpreted, as shown in Table 12.1.

Table 12.1: Mode and FilePosition

Mode	FilePosition
0	Relative to the beginning of the file.
1	Relative to current position.
2	Relative to the end of the file. (The end of the file is position 0.)

When returning *FilePosition*, ***filepos*** will return the position relative to the beginning of the file irrespective of the value of Mode. **Note**: In the DOS-related versions of Visual Prolog, ***filepos*** does not consider files in text mode to be different from files in binary mode. No translation of DOS newline conventions takes place, and a newline in a file following DOS newline conventions consists of two characters.

Example

1. The following sequence writes the value of *Text* into the file SOMEFILE.PRO (referred to by Prolog as *myfile*), starting at position 100 (relative to the beginning of the file).

```
      Text = "A text to be written in the file",
      openmodify(myfile, "somefile.pro"),
      writedevice(myfile),
      filepos(myfile, 100, 0),
      write(Text),
      closefile(myfile).
```

2. Using ***filepos***, you can inspect the contents of a file on a byte-by-byte basis, as outlined in Program ch11e10.pro. This program requests a file name, then displays the contents of positions in the file as their position numbers are entered at the keyboard.

```
                        /* Program ch12e10.pro */

DOMAINS
   file = input

PREDICATES
   inspect_positions(file)

CLAUSES
   inspect_positions(UserInput):-
      readdevice(UserInput),
      nl,write("Position No? "),
      readln(X),
      term_str(ulong,Posn,X),
      readdevice(input),
      filepos(input,Posn,0),
      readchar(Y),nl,
      write("Char is: ",Y),
      inspect_positions(UserInput).

GOAL
      write("Which file do you want to work with ?"),nl,
      readln(FileName),
      openread(input, FileName),
      readdevice(UserInput),
      inspect_positions(UserInput).
```

eof/1

eof checks whether the file position is at the end of the file, in which case ***eof*** succeeds; otherwise, it fails. ***eof*** has the form

```
   eof(SymbolicFileName)                                    /* (i) */
```

eof gives a run-time error if the file has been opened with write-only access. Note that it doesn't consider a DOS eof character (**Ctrl+Z**) to have any particular meaning.

Example

eof can be used to define a predicate ***repfile*** that's handy when operating with files. ***repfile*** generates backtrack points as long as the end of the file has not been reached.

```
PREDICATES
   repfile(FILE)

CLAUSES
   repfile(_).
   repfile(F):- not(eof(F)), repfile(F).
```

The following program converts one file to another where all the characters are upper-case.

```
                        /* Program ch12e11.pro */

DOMAINS
   file = input; output

PREDICATES
   convert_file
   nondeterm repfile(FILE)

CLAUSES
   convert_file :-
      repfile(input),
      readln(Ln),
      upper_lower(LnInUpper,Ln),    /* converts the string to uppercase */
      write(LnInUpper),nl,
      fail.
   convert_file.

   repfile(_).
   repfile(F):-
      not(eof(F)),
      repfile(F).
```

```
GOAL
      write("Which file do you want convert ?"),
      readln(InputFileName),nl,
      write("What is the name of the output file ?"),
      readln(OutputFileName),nl,
      openread(input, InputFileName),
      readdevice(input),
      openwrite(output, OutputFileName),
      writedevice(output),
      convert_file,
      closefile(input),
      closefile(output).
```

flush/1

flush forces the contents of the internal buffer to be written to the named file. It takes this format:

```
flush(SymbolicFileName)                                    /* (i) */
```

flush also requests the operating system to flush its buffers. For versions of DOS previous to 3.30, this entails closing and re-opening the file. For newer versions of DOS, as well as the other platforms, the appropriate operating system function is called.

existfile/1

existfile succeeds if *OSFileName* exists. It takes this format:

```
existfile(OSFileName)                                      /* (i) */
```

where *OSFileName* may contain a directory path and the name itself may contain wildcards, e.g. `c:\psys\*.cfg`. ***existfile*** fails if the name does not appear in the directory. However, note that although ***existfile*** finds all files, including those with the 'system' and 'hidden' attribute set, it doesn't find directories. This may be accomplished using the directory search predicates described later on.

You can use the following sequence to verify that a file exists before attempting to open it.

```
open(File, Name) :-
   existfile(Name), !,
   openread(File, Name).
open(_, Name) :-
   write("Error: the file ", Name," is not found").
```

existfile/2

In UNIX, ***existfile*** is also available in a two- arity version:

```
existfile(OSFileName,AccessMode)                          /* (i, i) */
```

with *AccessMode* specifying the type of access desired. This should be one of the following constants:

- **f_ok** to test for existence
- **x_ok** to test for execute permission
- **w_ok** to test for write permission
- **r_ok** to test for read permission

These constants are declared in the include file IODECL.CON.

existfile with only one argument tests for file-existence only.

searchfile/3

searchfile is used to locate a file along a path list, and is a kind of automated ***existfile***. It takes three arguments, as follows:

```
searchfile(PathList,Name,FoundName)                       /* (i,i,o) */
```

The *PathList* is a string containing one or more paths, separated by semicolons (or colons, for UNIX), and *Name* is the name of the file to locate. If found, *FoundName* will be bound to the fully qualified name, otherwise ***searchfile*** will fail. For instance, for DOS

```
SearchFile(".;..;C:\\","autoexec.bat",FoundName),
```

will - provided autoexec.bat is located in the root of drive C - set *FoundName* to `C:\AUTOEXEC.BAT`.

The file name may contain wildcards. In that case, *FoundName* is bound to the fully qualified wildcard name, which may subsequently be used as input to the directory matching predicates described later on. For instance, if the name is specified as `*.bat` instead of `autoexec.bat` in the above example, *FoundName* will be bound to `C:\*.BAT`.

deletefile/1

deletefile removes the file specified by its argument:

```
deletefile(OSFileName)                                     /* (i) */
```

deletefile gives an error if it can't remove the file. The *OSFileName* cannot contain wildcards.

renamefile/1

renamefile renames the file *OldOSFileName* to *NewOSFileName*. It takes this format:

```
renamefile(OldOSFileName, NewOSFileName)                    /* (i, i) */
```

renamefile succeeds if a file called *NewOSFileName* doesn't already exist and both names are valid file names; otherwise, it gives an error.

disk/1

disk is used to change the current disk and/or directory; it takes this format:

```
disk(Path)                                                  /* (i) (o) */
```

Called with a free variable, ***disk*** will return the current directory. In the DOS-related versions, to change to another disk without changing the existing current directory on that disk, use `D:`. Where *D* is the drive letter.

copyfile/2

copyfile is used to copy a file. It takes two file names as follows:

```
copyfile(SourceName,DestinationName)                         /* (i,i)*/
```

The names may be partly or fully qualified file names, including disks and directories. However, no wildcards are allowed. The copied file will have the same attributes (and permissions) as those of the source.

File Attributes

Although the standard file open predicates described previously cover all general cases, there may be a need to open or create files with specialized attributes and non-obvious sharing modes. To this end Visual Prolog incorporates a general-purpose open predicate, but before discussing that we need to look at file attributes and sharing modes.

The attributes and access modes used by Visual Prolog use the same values as your operating system, with the exception of the default ('normal') attribute in the NonUNIX-related versions of Visual Prolog. However, for easy porting to other environments, you should avoid coding inherently non-portable constructs such

as file attributes (and even the fact that files have attributes) all over an application. Rather, wrap things up nicely and write your own intermediate level of predicates, getting and setting information in transparent ways.

The attributes and sharing modes are found in the include file IODECL.CON.

Opening and creating files

When opening or creating a file, the OS needs to know the file's attributes (e.g. 'hidden'), the type of use or access (e.g. 'read'), and how the file may be shared with other programs while open (e.g. 'deny write'). Don't confuse these - they are three different pieces of information, only partly related:

Attributes

The **file attributes** are the permanent attributes relating to the physical file on disk, whether currently in use by a program or not. In DOS there's only a few attributes, such as 'read only' and 'hidden'. These attributes inform the operating system about how it may handle the file. Network and multi-user operating systems, such as UNIX, typically have a much wider range of attributes. These may include access allowed by other users (e.g. 'execute-only', no read or write, giving copy-protection) and direct instructions to the OS ('this is an executable program').

The attributes have no effect when opening an existing file, as files are unique based on names only. They only apply when creating a new file.

The standard predicates described in the previous section all reference 'normal' files. However, when a file has been modified the archive bit will automatically be set by the operating system when the file is closed.

Access Modes

The **access modes** indicate how the file will be used. The OS will combine this information with the files physical attributes, to determine if the access requested is acceptable. For instance, opening a file having the read-only physical attribute set, with either *fm_access_wo* or *fm_access_rw* will not be accepted.

Sharing Modes

The **sharing modes** indicate how this process views sharing with other processes. The OS will combine the sharing and access modes with the sharing and access modes specified by other processes, if the file is already in use, to

determine if the open call should succeed. If successful, the modes will restrict future open attempts.

Note that conceptually the sharing and access modes work both ways to form a combined set of restrictions on the file: they specify both what the process wants from a file and what it will allow from other processes. For instance, if a file has been opened with 'deny write' and 'read only', an open attempt with 'deny none' and 'write only' will fail because the first process has specified 'deny write' - in this case it is the existing restriction on the file that rejects the open attempt. On the other hand, an open attempt with 'deny read' and 'read only' will fail because the file is already open with read access - in this case it is the current requirement that rejects the open attempt.

Note that the *fm_sh_denyrw* denies all modes from other processes; it doesn't mean 'deny read- write, but allow read-only or write-only'.

All the standard predicates described in the previous section specify the sharing mode as 'deny write'.

Special File Modes for DOS >= 4.0 and UNIX

DOS versions greater than or equal to 4.0, have a special *fm_returnerr* mode:

The *fm_returnerr* specify that "media" errors occurring after the file has been opened should return an error to the program, rather than be reported through a pop-up window. Media errors are those indicating a malfunction of the device, e.g. if writing to a floppy and the drive door is opened - this generates the well-known 'Drive not ready' error.

UNIX and DOS >= 4.0 also have a write- through mode:

The *fm_writethru* specifies that no write buffering should take place. In this case, every single byte written to the file cause both the physical disk image and the directory entry to be updated, giving a secure file. However, disk writes performed with write-through are excessively slow compared to ordinary writes.

openfile/5

With the general-purpose ***openfile*** predicate, files may be created or opened in nonstandard ways. ***openfile*** looks like this:

```
openfile(SymbolicName,OSName,OpenMode,Attributes,Creation)
                                                    /* (i,i,i,i,i) */
```

The *SymbolicName* and *OSName* are the same as for the previously described standard predicates. The rest of the arguments are as follows (please refer to the section on File Attributes a few pages back):

- **OpenMode** is the access and sharing modes for the file. It is formed by adding together one of the fm_access_XX values, one of the fm_sh_XXXXXX and optionally fm_returnerr and fm_writethru. If no access mode is specified, it will be set to 'read only'. If no sharing mode is specified, it will be set to 'deny write'.
- **Attributes** are the attributes for the physical disk file. Valid attributes on DOS are fa_rdonly, fa_hidden, fa_system, fa_arch and fa_normal. If nothing (0) is specified, the attributes will be set to fa_normal. The system and the hidden attributes both have the same effect, namely to hide the file when a 'dir' command is executed. Note that DOS automatically sets the archive attribute when an updated file is closed. For UNIX, the attributes correspond to the file's permissions.
- **Creation** specifies how the presence or absence of an existing file with the same name is to be handled. It is formed by adding at most one from the cr_ex_XX group and at most one from the cr_noex_XX group. Pay attention to Creation defaults - if nothing (0) is specified. Note that this is the equivalent of specifying cr_ex_fail and cr_noex_fail, i.e. fail if it exists and fail if it doesn't exist. But remember that the actual default Creation action will be set according to the access mode as follows:

```
fm_access_ro   ->   cr_ex_open + cr_noex_fail
fm_access_wo   ->   cr_ex_replace + cr_noex_create
fm_access_rw   ->   cr_ex_open + cr_noex_create
```

  A sensible *Creation* default for read-write access is a bit tricky: If read-write is specified because the file is opened for 'modification', an existing file of the same name should be opened, not replaced. This is therefore the default. However, if read-write is specified because one wants bidirectional access to a new file, an existing file of the same name should be deleted. This is possible with a call to ***openfile*** as follows:

```
:
FMode = fm_access_rw + fm_sh_denywr + fm_returnerr,
FCrea = cr_ex_replace + cr_noex_create,
openfile(dbfile,"salient.dba",FMode,fa_normal,FCrea),
:
```

File and Path Names

A set of standard predicates ease file name handling and enable searching for files on a disk.

filenamepath/3

filenamepath is used to compose and decompose a fully qualified name around its path and file name. It takes three arguments, as follows:

```
filenamepath(QualName,Path,Name)                    /* (i,o,o)  (o,i,i)*/
```

filenamepath converts between *QualName* on one side, and *Path* and *Name* on the other. The programs ch11e12.pro and ch11e13.pro contain examples for DOS and UNIX respectively; both examples do essentially the same thing:

```
                        /* Program ch12e12.pro */

GOAL
      QualName="c:\\vip\\bin\\prolog.err",
      fileNamePath(QualName,Path,Name),
      write("\nQualName=",QualName),
      write("\nPath=",Path),
      write("\nName=",Name),
      fileNamePath(NewName,Path,Name),
      write("\nConverted back: ",NewName),nl.

                        /* Program ch12e13.pro */

GOAL
      QualName="/usr/bin/prolog.err",
      fileNamePath(QualName,Path,Name),
      write("\nQualName=",QualName),
      write("\nPath=",Path),
      write("\nName=",Name),
      fileNamePath(NewName,Path,Name),
      write("\nConverted back: ",NewName),nl.
```

This will set *Path* to C:\VIP\BIN and name to PROLOG.ERR; finally, *NewName* will be set to C:\VIP\BIN\PROLOG.ERR. Note that under DOS, all Visual Prolog file name handling converts the name to upper case. This is because there has in the past been confusion with respect to upper and lower case versions of some foreign characters.

Please, using the `(o,i,i)` flow pattern of this predicate, take into account some special cases described in the ***filenamepath*** topic in VDE help.

filenameext/3

filenameext is used to compose and decompose a (fully qualified) file name around its extension, defined by a dot. It takes three arguments, as follows:

```
   filenameext(Name,BaseName,Ext)                    /* (i,o,o)  (o,i,i)*/
```

Here is the DOS example:

```
                        /* Program ch12e14.pro */

GOAL
      Name="c:\\vip\\bin\\win\\16\\vip.exe",
      FileNameExt(Name,BaseName,Ext),
      write("\nName=",Name),
      write("\nBaseName=",BaseName),
      write("\nExt=",Ext),
      FileNameExt(NewName,BaseName,Ext),
      write("\nConverted back: ",NewName),
      % Override the old extension
      FileNameExt(NewName1,"VIP.EXE",".HLP"),
      write("\nNewName1=",NewName1),nl.
```

This will set *BaseName* to `C:\VIP\BIN\WIN\16\VIP` and *Ext* to `.EXE`; then *NewName* is set to `C:\VIP\BIN\WIN\16\VIP.EXE` and finally *NewName1* demonstrates a direct extension override - it isn't necessary to explicitly remove the old extension first. Note that the dot is considered a part of the extension and that - as with ***filenamepath*** - in the DOS version, everything is converted to upper case.

Directory Searching

Visual Prolog includes directory search predicates, enabling file name matching with wildcards. In addition, the predicates return all relevant information about the directory entries found.

Directory searching is very file system dependent and you should therefore guard yourself against future changes by isolating these predicates when they're used. Don't spread them all over your application, and don't rely on their arguments and functionality remaining unchanged.

Basically, to find matching files the directory has to be opened; this is done with the ***diropen*** predicate, specifying the file name mask and the attributes to look for. Then, by calling the ***dirmatch*** predicate, the matching files are found one by one. Finally, the directory is closed by a call to the ***dirclose*** predicate.

Generally, the predicates behave identically irrespective of platform: a file name - optionally containing wildcards - is used to specify the names to match, and a set of search attributes refine the match (for a list of attributes, see the section on

File Attributes earlier in this chapter). However, unlike the DOS directory search mechanism, the search attributes don't increase the search beyond 'normal' files. Visual Prolog considers all attributes as strictly informational, and they may all be used for file selection. When using the directory search predicates, you should therefore specify the attributes you are interested in: if you for instance want everything with the archive bit set, specify `fa_arch`; if you want everything with the system bit set, specify `fa_system`; if you want 'normal' files, specify `fa_normal`, etc. You should be aware, though, that the attributes specified are inclusive of each other: if several attributes are combined, the directory search will find everything matching at least one of the attributes, but the entry found won't necessarily match all the attributes. In other words, using set terminology, it is the union of the matching files, not the intersection, which is returned. Exactly what is found may be determined by bitwise testing of the returned attribute.

UNIX users should be aware that only one kind of directory entry (such as a normal file, a pipe, a directory, etc.) may be searched for at a time. No permissions of the entries are considered, and none should be specified.

diropen/3

diropen is used to gain access to a directory. It takes the following format:

```
diropen(Wild,Attrib,Block)                              /* (i,i,o) */
```

where *Wild* is a file name, optionally containing wildcards, *Attrib* are the required search attributes, and *Block* is an information block used by subsequent calls to ***dirmatch***. To the compiler this block looks like a string, but it contains more information than meets the eye. Therefore, it cannot be asserted in a fact database and then retracted for use at a later stage - as long as the directory is open, it must be held in a local variable (or an argument, which is the same thing). ***diropen*** will fail if there are no files matching the specification; however, if the file name includes a directory that does not exist, ***diropen*** will exit with an error.

Several ***diropens*** may be active simultaneously; in other words, they may be nested and used recursively.

dirmatch/10

dirmatch will, after ***diropen*** has returned an information block, return the name and other information for each matching file, one at each call. It looks as follows:

```
dirmatch(Block,Name,Attr,Hour,Min,TwoSec,Year,Month,Day,Size)
                                              /* (i,o,o,o,o,o,o,o,o,o) */
```

The *Block* is the information block returned by ***diropen***, *Name* is the matching name, and *Attr* are the attributes for the entry found. The rest of the arguments should be self-explanatory - they're all unsigned integers, apart from *Size*, which is an unsigned long. Note that DOS uses only 5 bits to encode the seconds part of the time stamp, giving at most 32 different values - hence the *TwoSec* name.

Upon each call, ***dirmatch*** returns the next matching directory entry. When there are no more matches, ***dirmatch*** fails; if this happens, ***dirmatch*** will automatically close the directory.

You should be aware that if searching for subdirectories with a name specification of e.g. `"*.*"`, ***dirmatch*** will always return the entries `"."` and `".."` if these are returned by the operating system. Therefore, ***dirmatch*** is likely to find directories in all directories except perhaps the root.

dirclose/1

dirclose will close a previously opened directory. It takes the form:

```
dirclose(Block)                                          /* (i) */
```

where *Block* is the information block returned by ***diropen***. Note that if ***dirmatch*** is repeatedly called until it fails (because there are no more matching files), ***dirclose*** should not be called, as ***dirmatch*** will have closed the directory.

Example

The following demonstrates the use of the directory matching predicates, to make an ***existdir*** predicate to complement the ***existfile*** standard predicate described previously.

```
                        /* Program ch12e16.pro */

include "iodecl.con"

PREDICATES
   existdir(string)
   exd1(string)
   exd2(string,string)

CLAUSES
   existdir(Wild):-
      diropen(Wild,fa_subdir,Block),
      exd1(Block),
      dirclose(Block).
```

```
exd1(Block):-
   dirmatch(Block,Name,_,_,_,_,_,_,_,_),
   exd2(Block,Name).

exd2(_,Name):-
   not(frontchar(Name,'.',_)),!.
exd2(Block,_):-
   exd1(Block).
```

Given for instance the goal `existdir("C:\\*.*")` in DOS, it will - unless you have a rather extraordinary disk organization - say 'yes'. However, it will only find subdirectories in existing paths - if you ask for e.g. `existdir(C:\\JNK\\*.*")` without having a directory called 'JNK' in the root of drive C, it will exit with an error. You should also be aware that in DOS the root itself cannot be matched: there is no directory called `'\'`, and `existdir("c:\\")` will fail. This is an operating system defined restriction of DOS, and is not relevant in UNIX where `'/'` does exist.

Note, by the way, how the current and parent directory entries (`"."` and `".."`) are filtered out in the example.

dirfiles/11

Having presented the hard way of finding files, here is the easy way. ***dirfiles*** is a non-deterministic standard predicate which, upon backtracking, returns all matching files one by one. It looks as follows:

```
dirfiles(Wild,Attrib,Fnam,RetAttr,Hour,Min,Sec,
Year,Month,Day,Size)                    /* (i,i,o,o,o,o,o,o,o,o,o) */
```

The use of ***dirfiles*** obviates the need to open and close the directory as this is handled automatically, but there is a condition attached: in order to use it correctly, it must be backtracked into until it fails. It is the final failure of the predicate, which automatically closes the directory. You should be aware that neither the compiler nor the code supporting a running program has any way of detecting if this won't happen - it is entirely the programmer responsibility. Having said that, no serious harm will come from leaving a couple of directories open, but eventually the system will run out of handles.

As with ***diropen***, calls to ***dirfiles*** may be nested and used recursively.

DOS Example

Below is a sample program, which will traverse all directories on, drive C, searching for entries having the 'system' or 'hidden' attribute set. The OS will

typically have a couple of hidden files in the root directory. However, if there are hidden files elsewhere on the disk, be suspicious! They're probably harmless copy- protection or configuration files for commercial software you have installed, but why hide any files?

```
                        /* Program ch12e17.pro */

CONSTANTS
   fa_hidden = $02    /* Hidden file          */
   fa_system = $04    /* System file          */
   fa_subdir = $10    /* Subdirectory         */

   fa_hidsys = $06    /* hidden + system */

PREDICATES
   findhidden(string,string)
   wrattr(integer)

CLAUSES
   wrattr(A):-
      bitand(A,fa_hidden,AA),
      AA<>0,write('H'),fail.
   wrattr(A):-bitand(A,fa_system,AA),
      AA<>0,write('S'),fail.
   wrattr(A):-
      bitand(A,fa_subdir,AA),
      AA<>0,write('D'),fail.
   wrattr(_).

   findhidden(CurrPath,Wild):-
      write(CurrPath,":\n"),
      filenamepath(FileSpec,CurrPath,Wild),
      dirfiles(FileSpec,fa_hidsys,FileName,RetAttr,_,_,_,_,_,_,_),
      wrattr(RetAttr),
      write('\t',FileName,'\n'),
      fail.
   findhidden(CurrPath,Wild):-
      filenamepath(DirSpec,CurrPath,"*.*"),
      dirfiles(DirSpec,fa_subdir,Name,_,_,_,_,_,_,_,_),
      not(frontchar(Name,'.',_)),
      filenamepath(DirName,CurrPath,Name),
      findhidden(DirName,Wild),
      fail.
   findhidden(_,_).

GOAL
   findhidden("C:\\","*.*").
```

This example also demonstrates decoding the returned attribute (in the ***wrattr*** predicate), by means of bitwise testing.

Manipulating File Attributes

A standard predicate enables getting and setting the (informational) attributes of files. Although documentation for DOS and MS Windows frequently talks about a "directory attribute", a file cannot be changed to a directory just by clearing this attribute.

fileattrib/2

Depending on dataflow ***fileattrib*** will get or set the attributes for a file. In UNIX this corresponds to the file mode, meaning permissions, sticky bits, etc; see ***chmod(S)***.

```
fileattrib(Name,Attrib)                                /* (i,o) (i,i) */
```

The *Name* must specify an existing file, otherwise ***fileattrib*** exits with an error. Note that the definition of getting or setting attributes is entirely operating system defined; in particular, you cannot set the file attributes for a directory. The attributes for the file appear in *Attrib* as an unsigned short. This may then be decomposed and/or changed using bitwise manipulation. For instance, the following will clear the system attribute for the DOS file "JNK":

```
constants
   fa_system = $04    /* System file          */
   fa_notsys = $FFFB  /* ~system file.        */

GOAL
   fileattrib("jnk",FA),
   bitand(FA,fa_notsys,Plain),
   fileattrib("jnk",Plain).
```

The constant *fa_notsys* is the bitwise negation of *fa_system*. If you don't know how to find the negation, use the ***bitxor*** (see chapter 16) standard predicate:

```
constants
   fa_system = $04      /* System file          */
```

```
GOAL
   bitxor(fa_system,$FFFF,NotSys),
   fileattrib("jnk",FA),
   bitand(FA,NotSys,Plain),
   fileattrib("jnk",Plain).
```

Handling terms in text files

The ***readterm*** predicate makes it possible to access facts in a file. ***readterm*** can read any object written by the ***write*** predicate and takes the form

```
readterm(<name>,TermParam).
```

where <*name*> is the name of a domain. The following code excerpt shows how readterm might be used.

```
domains
   name,addr = string
   one_data_record = p(name, addr)
   file = file_of_data_records

predicates
   person(name, addr)
   moredata(file)

clauses
   person(Name,Addr) :-
      openread(file_of_data_records, "DD.DAT"),
      readdevice(file_of_data_records),
      moredata(file_of_data_records),
      readterm(one_data_record, p(Name, Addr)).

   moredata(_).
   moredata(File) :-
      not(eof(File)),
      moredata(File).
```

If the file DD.DAT contains facts belonging to the *one_data_record* domain, such as

```
p("Peter","28th Street")
p("Curt","Wall Street")
```

the following are examples of goals, which retrieves information from that file. Run the **Test Goal** adding the following **goal** section into the program code:

```
Goal
   person("Peter",Address).
```

The **Test Goal** will reply:

```
Address="28th Street"
1 Solution
```

Now run the **Test Goal** adding the following **goal**:

```
Goal
   person("Peter","Not an address").
```

The **Test Goal** will reply:

```
no
```

Manipulating Facts Like Terms

Facts that describe fact database predicates can also be manipulated as though they were terms. This is made possible by automatical declaration by the Visual Prolog compiler of domains corresponding to names of facts sections (and the special ***dbasedom*** domain to the unnamed facts section).

Visual Prolog generates one domain alternative for each predicate in a facts section. It describes each database predicate by a functor and by the domains of the arguments in that predicate. For example, given this facts section declaration:

```
facts
   person(name, telno)
   city(cno, cname)
```

Visual Prolog generates the corresponding ***dbasedom*** domain:

```
DOMAINS
   dbasedom = person(name, telno) ; city(cno, cname)
```

A named facts section similarly generates a domain corresponding to the facts section name, as in the following example:

```
facts - test
   person(name, telno)
   city(cno, cname)
```

generates the domain

```
DOMAINS
   test = person(name, telno) ; city(cno, cname)
```

These domains can be used like any other predefined domains.

Example

The following example shows how you could construct a predicate ***my_consult***, similar to the standard predicate ***consult***.

```
/* User-defined Predicate my_consult using readterm */

domains
   file = dbase

facts - dba1
   /*  ... Declare database predicates to be read from file */

predicates
   my_consult(string)
   repfile(file)

clauses
   my_consult(FileName) :-
      openread(dbase, FileName),
      readdevice(dbase),
      repfile(dbase),
      readterm(dba1, Term),
      assertz(Term),
      fail.
   my_consult(_) :- eof(dbase).

   repfile(_).
   repfile(F):-not(eof(F)),repfile(F).
```

If, for example, the facts section contains the declaration

```
p(string, string)
```

and a file called DD.DAT exists (with contents as described above), you could try the following goals, which retrieves information from that file. Add the following **goal** section into the program code and run the **Test Goal** :

```
goal
   my_consult("dd.dat").
```

The **Test Goal** will reply:

```
yes
```

Now run the **Test Goal** adding the following **goal**:

```
Goal
   my_consult("dd.dat"),
   p(X,Y).
```

The **Test Goal** will reply:

```
X=Peter, Y=28th Street
X=Curt, Y=Wall Street
2 Solutions
```

Summary

These are the important points covered in this chapter:

1. Visual Prolog provides three standard predicates for basic writing:
   a. ***write*** (for plain output)
   b. ***writef*** (for output formatted according to format specifiers)
   c. ***nl*** (for generating new lines)
2. Visual Prolog basic standard predicates for reading are
   a. ***readln*** (for reading whole lines of characters)
   b. ***readint, readreal***, and ***readchar*** (for reading integers, reals, and characters, respectively)
   c. ***readterm*** (for reading compound objects)
   d. ***file_str*** (for reading a whole text file into a string)
3. Additionally, binary blocks may be transferred using
   a. ***readblock*** (reads a binary block from a file)
   b. ***writeblock*** (writes a binary block to a file)
   c. ***file_bin*** transfers between whole files and binary blocks
4. Visual Prolog uses a *current_read_device* (normally the keyboard) for reading input, and a *current_write_device* (normally the screen) for sending output. You can specify other devices, and can reassign the current input and output devices at run time. (This reassigning is known as *redirecting I/O*.)
5. Visual Prolog's basic file-access predicates are:

a. ***openread*** (open a file for reading)

b. ***openwrite*** (open a file for writing)

c. ***openappend*** (open a file for appending)

d. ***openmodify*** (open a file for modification)

e. ***openfile*** (general purpose file opener)

f. ***filemode*** (set a file to text mode or binary mode)

g. ***closefile*** (close a file)

h. ***readdevice*** (reassign the *current_read_device* or get its name)

i. ***writedevice*** (reassign the *current_write_device* or get its name)

6. To access files you use the FILE domain, which has five predefined alternatives:

   a. *keyboard* (for reading from the keyboard)

   b. *screen* (for writing to the screen)

   c. *stdin* (for reading from standard input)

   d. *stdout* (for writing to standard output)

   e. *stderr* (for writing to standard error)

7. To work within and manipulate files, you use the following standard predicates:

   a. ***filepos*** (controls the position where reading or writing takes place)

   b. ***eof*** (checks whether the file position during a read operation is at the end of the file)

   c. ***flush*** (forces the contents of the internal buffer to be written to a file)

   d. ***existfile*** (verifies that a file exists)

   e. **searchfile** (locates a file among several directories)

   f. ***deletefile*** (deletes a file)

   g. ***renamefile*** (renames a file)

   h. ***copyfile*** (copies a file)

   i. ***fileattrib*** (gets or sets file attributes)

   j. ***disk*** (changes the current disk and directory/subdirectory)

8. To search directories, the following standard predicates are available:

a. ***diropen*** (opens a directory for searching)

b. ***dirmatch*** (finds matching files in a directory)

c. ***dirclose*** (closes a directory)

d. ***dirfiles*** (non-deterministically matches files in a directory)

9. To manipulate file names, the following are available:

   a. ***filenamepath*** (joins or splits qualified file names)

   b. ***filenameext*** (joins or splits file names and extensions)

10. The standard predicate ***readterm*** allows your program to access facts in a file at run time. ***readterm*** can read any object written by ***write***, plus facts describing fact database predicates.

CHAPTER 13

String-Handling in Visual Prolog

Visual Prolog provides several standard predicates for powerful and efficient string manipulations. In this chapter, we've divided these into two groups: the family of basic string-handling predicates, and a set of predicates used for converting strings to other types and vice versa. Strings may also be compared to each other, but this is covered in chapter **9**.

String Processing

A few formalities apply to strings and string processing, in that the backslash acts as an escape character, allowing you to put non-keyboardable characters into strings. Please see the description on page 106.

Basic String-Handling Predicates

The predicates described in this section are the backbone of string-handling in Visual Prolog; as such, they serve several purposes:

- dividing a string into component strings or tokens
- building a string from specified strings or tokens
- verifying that a string is composed of specified strings or tokens
- returning a string, token, or list of these from a given string
- verifying or returning the length of a string
- creating a blank string of specified length
- verifying that a string is a valid Visual Prolog name
- formatting a variable number of arguments into a string variable

frontchar/3

frontchar operates as if it were defined by the equation

```
String1 = the concatenation of Char and String2
```

It takes this format:

```
frontchar(String1,Char,String2)
                          /* (i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i) */
```

frontchar takes three arguments; the first is a string, the second is a ***char*** (the first character of the first string), and the third is the rest of the first string.

frontchar can be used to split a string up into a series of characters, or to create a string from a series of characters, and to test the characters within a string. If the argument *String1* is bound to a zero-length string, the predicate fails.

Example

In Program `ch13e01.pro`, ***frontchar*** is used to define a predicate that changes a string to a list of characters. Try the goal

```
string_chlist("ABC", Z)
```

This goal will return *Z* bound to ['A','B','C'].

```
                        /* Program ch13e01.pro */

DOMAINS
   charlist = char*

PREDICATES
   string_chlist(string, charlist)

CLAUSES
   string_chlist("", []):-!.
   string_chlist(S, [H|T]):-
      frontchar(S,H,S1),
      string_chlist(S1,T).
```

fronttoken/3

fronttoken performs three related functions, depending on the type of flow pattern you use when calling it.

```
fronttoken(String1, Token, Rest)
                          /* (i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i) */
```

In the `(i,o,o)` flow variant, ***fronttoken*** finds the first token of *String1*, binds it to *Token*, and binds the remainder of *String1* to *Rest*. The `(i,i,o)`, `(i,o,i)`, and `(i,i,i)` flow variants are tests; if the bound arguments are actually bound to the

corresponding parts of *String1* (the first token, everything after the first token, or both, respectively), ***fronttoken*** succeeds; otherwise, it fails.

The last flow variant `(o,i,i)` constructs a string by concatenating *Token* and *Rest*, then binds *String1* to the result.

A sequence of characters is grouped as one token when it constitutes one of the following:

- a name according to normal Visual Prolog syntax
- a number (a preceding sign is returned as a separate token)
- a non-space character

fronttoken is perfectly suited for decomposing a string into lexical tokens.

Example

Program `ch13e02.pro` illustrates how you can use ***fronttoken*** to divide a sentence into a list of names. If 2 is given the goal:

```
string_namelist("bill fred tom dick harry", X).
```

X will be bound to:

```
[bill, fred, tom, dick, harry]
```

```
                        /* Program ch13e02.pro */

DOMAINS
   namelist = name*
   name  = symbol

PREDICATES
   string_namelist(string, namelist)

CLAUSES
   string_namelist(S,[H|T]):-
      fronttoken(S,H,S1),!,
      string_namelist(S1,T).
   string_namelist(_,[]).
```

frontstr/4

frontstr splits *String1* into two parts. It takes this format:

```
frontstr(NumberOfChars, String1, StartStr, EndStr)
                                                    /* (i,i,o,o) */
```

StartStr contains the first *NumberOfChars* characters in *String1*, and *EndStr* contains the rest. When ***frontstr*** is called, the first two parameters must be bound, and the last two must be free.

concat/3

concat states that *String3* is the string obtained by concatenating *String1* and *String2*. It takes this format:

```
concat(String1, String2, String3)
                                /* (i,i,o), (i,o,i), (o,i,i), (i,i,i) */
```

At least two of the parameters must be bound before you invoke ***concat***, which means that ***concat*** always gives only one solution (in other words, it's deterministic). For example, the call

```
concat("croco", "dile", In_a_while)
```

binds *In_a_while* to *crocodile*. In the same vein, if *See_ya_later* is bound, the call

```
concat("alli", "gator", See_ya_later)
```

succeeds only if *See_ya_later* is bound to *alligator*.

str_len/2

str_len can perform three tasks: It either returns or verifies the length of a string, or it returns a string of blank spaces of a given length. It takes this format:

```
str_len(StringArg, Length)                    /* (i,o), (i,i), (o,i) */
```

str_len binds *Length* to the length of *StringArg* or tests whether *StringArg* has the given *Length*. The *Length* is an unsigned integer. In the third flow version, ***str_len*** returns a string of spaces with a given length; this can be used to allocate buffers, etc. allocating buffers with ***str_len***, but ***makebinary*** is preferable especially for binary data.

isname/1

isname verifies that its argument is a valid name in accordance with Visual Prolog's syntax; it takes this format:

```
isname(String)                                              /* (i) */
```

A name is a letter of the alphabet or an underscore character, followed by any number of letters, digits, and underscore characters. Preceding and succeeding spaces are ignored.

format/*

format performs the same formatting as ***writef*** (see page 321), but ***format*** delivers the result in a string variable.

```
format(OutputString,FormatString,Arg1,Arg2,Arg3,...,ArgN)
                                                /* (o,i,i,i,...,i) */
```

subchar/3

subchar returns the character at a given position in a string; it takes the form:

```
subchar(String,Position,Char)                          /* (i,i,o) */
```

The first character has position 1. For example,

```
subchar("ABC",2,Char)
```

will bind *Char* to B. If the position specifies a character beyond the end of the string, ***subchar*** exits with an error.

substring/4

substring returns a part of another string; it takes the form:

```
substring(Str_in,Pos,Len,Str_out)                    /* (i,i,i,o) */
```

Str_out will be bound to a copy of the string starting with the *Pos*'th character, *Len* characters long, in *Str_in*. For example

```
substring("GOLORP",2,3,SubStr)]
```

binds *SubStr* to OLO. If *Pos* and *Len* specify a string partly or wholly outside of *Str_in*, ***substring*** exits with an error. However, it is not an error to ask for 0 bytes at the extreme end of the string:

```
substring("ABC",4,0,SubStr)]
```

will bind SubStr to an empty string (""), while

```
substring("ABC",4,1,SubStr)/* WRONG */]
```

is an error. By the way, so is

```
   substring("ABC",5,-1,SubStr)/* WRONG */]
```

searchchar/3

searchchar returns the position of the first occurrence of a specified character in a string; it takes the form:

```
   searchchar(String,Char,Position)                          /* (i,i,o) */]
```

For example,

```
   searchchar("ABEKAT",'A',Pos)]
```

will bind Pos to 1. If the character isn't found, ***searchchar*** will fail. Note that ***searchchar*** is not re-satisfiable (i.e. if there are more occurrences of the specified character in the string, backtracking won't find them), but you can easily make your own:

```
                        /* Program ch13e03.pro */

PREDICATES
   nondeterm nd_searchchar(string,char,integer)
   nondeterm nd_searchchar1(string,char,integer,integer)
   nondeterm nd_sc(string,char,integer,integer,integer)
   run

CLAUSES
   nd_searchchar(Str,Ch,Pos):-
      nd_searchchar1(Str,Ch,Pos,0).

   nd_searchchar1(Str,Ch,Pos,Old):-
      searchchar(Str,Ch,Pos1),
      nd_sc(Str,Ch,Pos,Pos1,Old).

   nd_sc(_,_,Pos,Pos1,Old):- Pos = Pos1+Old.
   nd_sc(Str,Ch,Pos,Pos1,Old):-
      frontstr(Pos1,Str,_,Rest),
      Old1 = Old + Pos1,
      nd_searchchar1(Rest,Ch,Pos,Old1).

GOAL
      nd_searchchar("abbalblablabbala",'a',P),
      write(P,'\n'),
      fail.
```

This implements a non-deterministic predicate ***nd_searchchar***, which is plug-compatible with ***searchchar***; if you don't mind typing the extra argument (*Old*) to *nd_searchchar1* yourself, you can of course discard a level of calls.

searchstring/3

searchstring returns the position of the first occurrence of a string in another string; it takes the form:

```
searchstring(SourceStr,SearchStr,Pos)                    /* (i,i,o) */]
```

For example,

```
searchstring("ABEKAT","BE",Pos)]
```

will bind Pos to 2. If the search string isn't found in, or is longer than, the source string, ***searchstring*** will fail. As with ***searchchar***, ***searchstring*** isn't re-satisfiable, but you can easily make your own. As a matter of fact, all that's necessary is to take 3 and do a global substitution with 'string' replacing 'char', and change the 'a' in the goal to a suitable search string, e.g. `"ab"`:

```
GOAL
   nd_searchstring("abbalblablabbala","ab",P),
   write(P,'\n'),
   fail.
```

Type Conversion

In this section, we summarize the standard predicates available for type conversion. The predicates are ***char_int***, ***str_char***, ***str_int***, ***str_real***, ***upper_lower***, and finally ***term_str***, which converts between terms of any kind and strings.

char_int/2

char_int converts a character into an integer or an integer into a character; it takes this format:

```
char_int(Char, Integer)                        /* (i,o), (o,i), (i,i) */
```

With both its arguments bound, ***char_int*** tests that the arguments correspond. With one argument bound and the other free, ***char_int*** performs the conversion and binds the output argument to the converted form of the input one.

Note: This predicate is really not needed in newer versions of Visual Prolog because there is automatic conversion between characters and integers. We've left ***char_int*** in to be compatible with older versions.

str_char/2

str_char converts a string containing one and only one character into a character, or converts a single character into a string of one character; it takes this format:

```
str_char(String, Char)                          /* (i,o), (o,i), (i,i) */]
```

In the `(i,i)` flow variant, ***str_char*** succeeds if *String* is bound to the single-character ***string*** equivalent of *Char*. If the length of the string is not 1, ***str_char*** fails.

str_int/2

str_int converts a string containing an integer into an integer, or converts an integer into its textual representation; it takes this format:

```
str_int(String, Integer)                        /* (i,o), (o,i), (i,i) */]
```

In the `(i,i)` flow variant, ***str_int*** succeeds if *Integer* is bound to the ***integer*** equivalent of the integer represented by *String*.

str_real/2

str_real converts a string containing a real number into a real number, or converts a real number into a string; it takes this format:

```
str_real(String, Real)                          /* (i,o), (o,i), (i,i) */]
```

In the `(i,i)` flow variant, ***str_real*** succeeds if *Real* is bound to the ***real*** equivalent of the real number represented by *String*.

upper_lower/2

upper_lower converts an upper-case (or mixed) string or character to all lower-case, or a lower-case (or mixed) string or character to all upper-case; it takes this format:

```
upper_lower(Upper, Lower)                       /* (i,o), (o,i), (i,i) */]
```

With both its arguments bound, ***upper_lower*** succeeds if *Upper* and *Lower* are bound to strings that – except for the case of the letters – are identical; for instance, the goal:

```
Str1=samPLEstrING,
Str2=sAMpleSTRing,
upper_lower(Str1, Str2)}
succeeds. Otherwise, it fails.
```

term_str/3

term_str is a general-purpose conversion predicate and will convert between terms of a specified domain and their string representations. It looks like this:

```
term_str(Domain,Term,String)                    /* (i,i,o),(i,_,i) */]
```

where *Domain* specifies which domain the term belongs to. ***term_str*** could replace the various ***str_**** predicates above, for instance, ***str_real*** could be implemented as `str_real(S,R):- term_str(real,R,S)`. However, ***term_str*** is a somewhat heavier mechanism.

The *Domain* need not be one of the standard domains; it can be any user-defined domain:

```
                        /* Program ch13e04.pro */

DOMAINS
   intlist = integer*

GOAL
      write("Input list (example [66,73,76,83]): "),
      readln(L),nl,
      str_len(L,Len),
      write("The stringlength of ",L),
      write(" is ",Len,'\n').
```

Examples

1. This example defines the predicate ***scanner***, which transforms a string into a list of tokens. Tokens are classified by associating a functor with each token. This example uses the predicates ***isname***, ***str_int***, and ***str_len*** to determine the nature of the tokens returned by ***fronttoken***.

```
                        /* Program ch13e05.pro */

   DOMAINS
      tok  = numb(integer); name(string); char(char)
      toklist = tok*
```

```
PREDICATES
   nondeterm scanner(string, toklist)
   nondeterm maketok(string, tok)

CLAUSES
   scanner("",[]).
   scanner(Str,[Tok|Rest]):-
      fronttoken(Str, Sym, Str1),
      maketok(Sym, Tok),
      scanner(Str1, Rest).

   maketok(S,name(S)):-isname(S).
   maketok(S,numb(N)):-str_int(S,N).
   maketok(S,char(C)):-str_char(S, C).

GOAL
   write("Enter some text:"),nl,
   readln(Text),nl,
   scanner(Text,T_List),
   write(T_List).
```

2. Conversions between the domain types ***symbol*** and ***string***, and between ***char***, ***integer***, and ***real***, are handled automatically when using standard predicates and during evaluation of arithmetic expressions. Reals will be rounded during automatic conversions. Visual Prolog performs this automatic conversion as necessary when a predicate is called, as in the following example:

```
PREDICATES
   p(integer)

CLAUSES
   p(X):- write("The integer value is ",X,'\n').
```

   With this example, the following goals have the same effect:

```
X=97.234, p(X).
X=97, p(X).
X='a', p(X).
```

3. The following very simple English parser is a practical example of string parsing. This example directly parses strings; if the parser were to be extended, the string should be tokenized using a scanner similar to the one used in Program `ch13e04.pro`. Whether you're parsing tokens or strings, the algorithm in this program is a good example of how to start.

If you are interested in English-language parsing, we recommend that you take a look at the Sentence Analyzer and Geobase programs in the VPI\PROGRAMS subdirectory.

```
                    /* Program ch13e06.pro */

DOMAINS
   sentence    = s(noun_phrase,verb_phrase)
   noun_phrase = noun(noun) ; noun_phrase(detrm,noun)
   noun        = string
   verb_phrase = verb(verb) ; verb_phrase(verb,noun_phrase)
   verb        = string
   detrm       = string

PREDICATES
   nondeterm s_sentence(string,sentence)
   nondeterm s_noun_phrase(string,string,noun_phrase)
   nondeterm s_verb_phrase(string,verb_phrase)
   d(string)
   n(string)
   v(string)

CLAUSES
   s_sentence(Str,s(N_Phrase,V_Phrase)):-
      s_noun_phrase(Str, Rest, N_Phrase),
      s_verb_phrase(Rest, V_Phrase).

   s_noun_phrase(Str,Rest,noun_phrase(Detr,Noun)):-
      fronttoken(Str,Detr,Rest1),
      d(Detr),
      fronttoken(Rest1,Noun,Rest),
      n(Noun).

   s_noun_phrase(Str,Rest,noun(Noun)):-
      fronttoken(STR,Noun,Rest),
      (Noun).

   s_verb_phrase(Str, verb_phrase(Verb,N_Phrase)):-
      fronttoken(Str,Verb,Rest1),
      v(Verb),
      s_noun_phrase(Rest1,"",N_Phrase).

   s_verb_phrase(Str,verb(Verb)):-
      fronttoken(STR,Verb,""),
      v(Verb).
```

```
/* determiner */
d("the").
d("a").

/* nouns */
n("bill").
n("dog").
n("cat").

/* verbs */
v("is").
```

Load and run this program, and enter the following goal:

```
Goal s_sentence("bill is a cat", Result).
```

The program will return:

```
Result = s(noun("bill"),verb_phrase("is", noun_phrase("a","cat")))
1 Solution
```

Summary

These are the important points covered in this chapter:

1. Visual Prolog's string-handling predicates are divided into two groups: basic string manipulation and string type conversions.
2. The predicates for basic string manipulation are summarized here:
   a. ***frontchar***, ***fronttoken***, and ***concat*** for dividing a string into components, building a string from specified components, and testing if a string is composed of specified components; these components can be characters, tokens, or strings
   b. ***subchar*** and ***substring*** for returning a single character from, or a part of, another string
   c. ***searchchar*** and ***searchstring*** for finding the first occurrence of a character, or a string, in a string
   d. ***str_len*** for verifying or returning the length of a string, or creating a blank string of specified length
   e. ***frontstr*** for splitting a string into two separate strings
   f. ***isname*** for verifying that a string is a valid Visual Prolog name

g. ***format*** for formatting a variable number of arguments into a string variable

Several of the basic string manipulation predicates have different flow variants. The variants with only input parameters perform tests that succeed when the string in question is made up of the specified components (or is of the specified length).

3. The predicates for type conversion are listed here:

   a. ***char_int*** for converting from a character to an integer, or vice versa

   b. ***str_char*** for converting a single character into a string of one character, or vice versa

   c. ***str_int*** for converting from an integer to its textual representation, or vice versa

   d. ***str_real*** for converting from a real number to a string, or vice versa

   e. ***upper_lower*** for converting a string to all upper-case or all lower-case characters, or testing case-insensitive string equality

   f. ***term_str*** for conversion between arbitrary domains and strings

   The type conversion predicates each have three flow variants; the `(i,o)` and `(o,i)` variants perform the appropriate conversions, and the `(i,i)` variants are tests that succeed only if the two arguments are bound to the converted representations of one another

CHAPTER 14

The External Database System

In this chapter, we cover Visual Prolog's external database system. An *external database* is composed of an external collection of chained terms; these chains give you direct access to data that is not a part of your Prolog program. The external database can be stored in any one of three locations: in a file, in memory, or in EMS-type expanded memory under DOS. The external database supports B+ trees, which provide fast data retrieval and the ability to sort quickly, and it supports multi-user access by a mechanism for serializing the file accesses inside transactions.

External Databases in Visual Prolog

Visual Prolog's internal fact database, which uses ***asserta***, ***assertz***, ***retract***, and ***retractall***, is very simple to use and suitable for many applications. However, the RAM requirements of a database can easily exceed the capacity of your computer; the external database system has been designed partly with this problem in mind. For example, you might want to implement one or more of the following:

- a stock control system with an large number of records
- an expert system with many relations but only a few records with complicated structures
- a filing system in which you store large text files in the database
- your own database product – which maybe has nothing to do with a relational database system – in which data is linked together in other, nonrelational ways
- a system including several of these possibilities

Visual Prolog's external database system supports these different types of applications, while meeting the requirement that some database systems must not lose data during update operations – even in the event of power failure.

Visual Prolog's external database predicates provide the following facilities:

- efficient handling of very large amounts of data on disk
- the ability to place the database in a file, in memory, or in EMS-type expanded memory cards under DOS
- multi-user access
- greater data-handling flexibility than provided by the sequential nature of Visual Prolog's automatic backtracking mechanism
- the ability to save and load external databases in binary form

An Overview: What's in an External Database?

A Visual Prolog external database consists of two components: the data items – actually Prolog *terms* – stored in chains, and corresponding B+ trees, which you can use to access the data items very quickly.

The external database stores data items in chains (rather than individually) so that related items stay together. For example, one chain might contain part numbers to a stock list, while another might contain customer names. Simple database operations, such as adding new items or replacing and deleting old items, do not require B+ trees. These come into play when you want to sort data items or search the database for a given item; they are covered in detail later in this chapter.

Naming Convention

The names of all the standard predicates concerned with database management follow a certain convention.

- The first part of the name (**db_**, **chain_**, **term_**, and so on) is a reminder of what you must specify as input.
- The second part of the name (**flush**, **btrees**, **delete**, and so on) is a reminder of what action occurs or what is returned or affected.

For example, ***db_delete*** deletes a whole database, ***chain_delete*** deletes a whole chain, and ***term_delete*** deletes a single term.

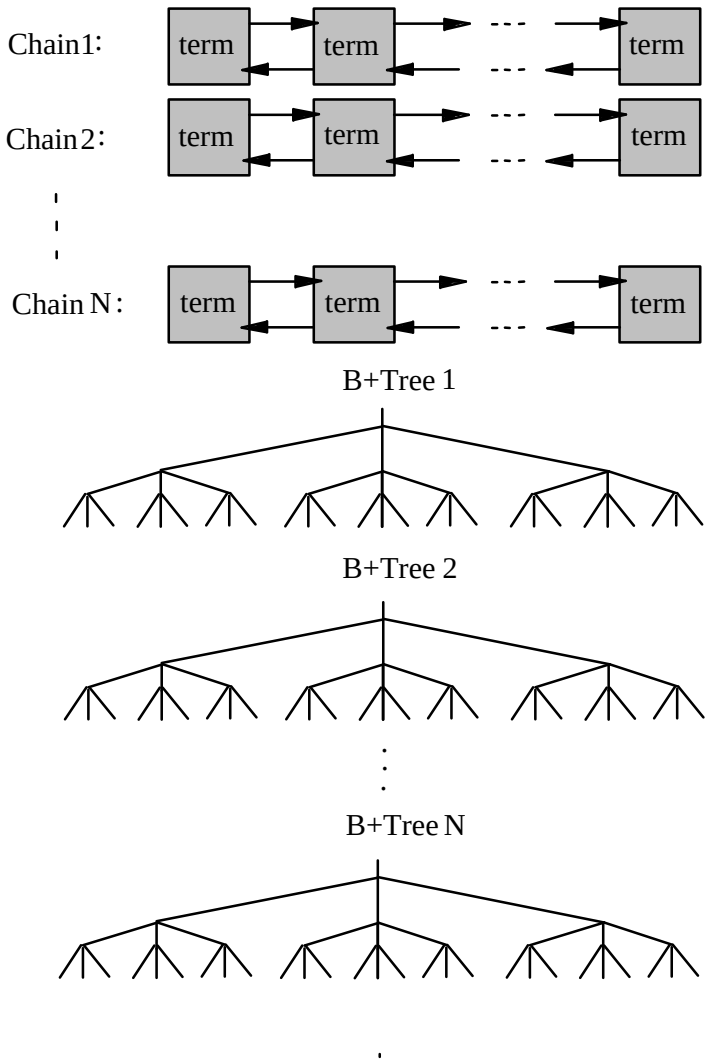


Figure 14.1: Structure of a Visual Prolog External Database

External Database Selectors

It is possible to have several external databases simultaneously in memory, on disk, and in an EMS-type memory expansion card under DOS. With this flexibility, you can place external databases where they give the best speed and space compromise.

In order to distinguish between several open databases, you use a *selector* in every call to an external database standard predicate. You must declare these selectors in a domain called ***db_selector***. This works like the ***file*** domain in the file system. For example, the following domains, declarations, external databases domain declaration declares ***customers*** and ***parts*** to be external database selectors:

```
DOMAINS
   db_selector = customers; parts
```

Chains

An external database is a collection of Prolog *terms*. Some examples of terms are ***integers***, ***reals***, ***strings***, ***symbol*** values, and compound objects; for instance, `32, -194, 3.1417, "Wally", wages`, and `book(dickens, "Wally goes to the zoo")`.

Inside an external database, the terms are stored in *chains*. A chain can contain any number of terms, and an external database can contain any number of chains. Each chain is selected by a name, which is simply a string.

The following figure illustrates the structure of a chain called *MY_CHAIN*.

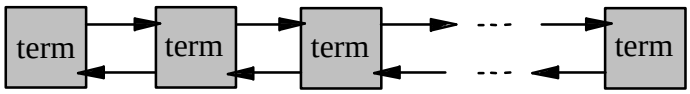


Figure 14.2: Structure of a Chain

Database relations and database tables are modeled by chains of terms. For example, suppose you have a customer, supplier, and parts database, and you want to put all the data into a single database with three relations: one for customers, one for suppliers, and one for parts. You do this by putting the customers in one chain called *customers*, the suppliers in another chain called *suppliers*, and the parts in a chain called *parts*.

To insert a term in an external database, you must insert the term into a named chain. On the other hand, you can retrieve terms without explicitly naming the containing chain. In both cases, you must specify the domain to which the term belongs. In practice, it is best if all terms in the chain belong to the same domain, but there is actually no restriction on how terms are mixed in a database. It's up to you to ensure that a term you retrieve belongs to the same domain as it did when you inserted it.

The following is a simple example of setting up two chained databases, *dba1* and *dba2*. In this example, all the customer data is in *dba1* and all the parts data in *dba2*. For now, just look over this example. We need to introduce a lot more information before we can explain what's happening here.

```
                        /* Program ch14e01.pro */

DOMAINS
   db_selector = dba1 ; dba2
   customers = customer(customer_name, address)
   parts = part(part_name, ID, customer_name)
   customer_name, part_name = symbol
   ID = integer
   address = string

PREDICATES
   access

CLAUSES
   access:-
      chain_terms(dba1,chain1,customers,customer(Name, ADDR),_),
      chain_terms(dba2,chain2,parts,part(Part, Id, Name),_),
      write("send ",Part," part num ",Id," to ",Addr), nl,
      fail.
   access.

GOAL
      % create the databases dba1 and dba2
      db_create(dba1, "dd1", in_memory),
      db_create(dba2, "dd1.bin", in_file),

      % insert customer facts into chain1 in dba1
      chain_insertz(dba1, chain1, customers,
      customer("Joe Fraser","123 West Side"), _),
      chain_insertz(dba1, chain1, customers,
      customer("John Smith","31 East Side"), _),
      chain_insertz(dba1, chain1, customers,
      customer("Diver Dan","1 Water Way"), _),
      chain_insertz(dba1, chain1, customers,
      customer("Dave Devine","123 Heaven Street"), _),
```

```
% insert parts facts into chain2 in dba2
chain_insertz(dba2, chain2, parts, part("wrench", 231,
              "John Smith"), _),
chain_insertz(dba2, chain2, parts, part("knife", 331,
              "Diver Dan"), _),
access,
db_close(dba1), db_close(dba2),
db_delete("dd1", in_memory),
db_delete("dd1.bin", in_file).
```

This program first creates the databases ***dba1*** (in memory) and ***dba2*** (in a disk file). It then inserts facts into two chains: *chain1* and *chain2*. After inserting the facts, it looks in these chains for a customer and the part ordered by that customer; finding these, it returns the address to which the shipper should ship the part. Finally, it closes and deletes the two databases.

External Database Domains

The external database uses six standard domains, summarized here:

Domain	What It's Used For
db_selector	Domain for declaring database selectors
bt_selector	Domain for declaring B+ tree selectors
place	Location of the database: in RAM, in a file, or in an extended memory system (EMS card under DOS)
accessmode	Decides how the file will be used.
denymode	Determines how other users can open the file.
ref	Reference to the location of a term in a chain

Database Reference Numbers

Every time you insert a new term into an external database, Visual Prolog assigns it a *database reference number*. You can use the term's database reference number to retrieve, remove, or replace that term, or to get the next or previous term in the chain. You can also insert a database reference number in a B+ tree (as described later in this chapter), and then use the B+ tree to sort some terms or to carry out a fast search for a term.

Database reference numbers are independent of the database location and any possible packing operations. Once a reference number has been associated with a

term, you can use that number to access that term – no matter which database management operations are subsequently carried out – until the term is deleted.

The ref Domain

Database reference numbers are special because you can insert them in facts sections and write them out with ***write*** or ***writef***, but you can't type them in from the keyboard. You must declare the arguments to predicates handling database reference numbers as belonging to the standard domain ***ref***.

When you delete a term with ***term_delete***, the system will reuse that term's reference number when it inserts the next term into the external database. This happens automatically; however, if reference numbers have been stored in the facts section or in a B+ tree for some reason, it is your responsibility to ensure that a given reference is associated with the correct term.

To assist you in this, there is an error-checking option, enabled with the ***db_reuserefs*** standard predicate:

db_reuserefs/2

db_reuserefs has the following form:

```
db_reuserefs(DBase,ReUse)                                   /* (i,i)*/
```

where *DBase* is a ***db_selector*** and *ReUse* is an unsigned integer. This should be set to 0 to enable checking for use of released terms, or 1 do disable this. The overhead of having the check enabled is very small (4 bytes per term, virtually no CPU overhead), but those 4 bytes will never be released. If you constantly create and release terms, your database will therefore grow at a steady rate. ***db_reuserefs***'s primary purpose is to assist you in tracking down bugs during development of programs.

Manipulating Whole External Databases

When you create a new external database, or open an existing one, you can place it in a file, in memory, or in EMS-type expanded memory under DOS, depending on the value of the *Place* argument in your call to ***db_create*** or ***db_open***. After you've finished working with the external database, you close it with a call to ***db_close***.

When you place an external database in main or expanded memory, closing the database with ***db_close*** does not delete the database from memory. You must do this explicitly with a call to ***db_delete***, to free the memory the database occupies.

If you close such an external database but don't delete it, you can later reopen it with the ***db_open*** predicate.

Since the external database system relies on the DOS buffer system, it will be very slow if no buffers have been allocated. To allocate 40 buffers (which isn't an excessive number), include the following line in your CONFIG.SYS file (a part of the DOS environment):

```
buffers = 40
```

In this section, we discuss the predicates ***db_create***, ***db_open***, ***db_copy***, ***db_loadems***, ***db_saveems***, ***db_close***, ***db_delete***, ***db_openinvalid***, ***db_flush***, ***db_garbagecollect***, ***db_btrees***, ***db_chains***, and ***db_statistics***.

db_create/3

db_create creates a new database.

```
db_create(Dbase, Name, Place)                                   /* (i,i,i) */
```

If the database is placed in a disk file, the name of the file will be *Name*; if it's placed in memory or EMS under DOS, you'll need *Name* if you close the database and want to open it later. *Dbase* and *Name* correspond to the internal and external names for files.

Where you place an external database is determined by the *Place* argument. *Place* can take one of the following values:

in_file	The external database is placed in a disk file, and there will be only a minimum of main memory overhead.
in_memory	The external database is placed in the main memory – usually this will be done to achieve maximum performance.
in_ems	The database is placed in EMS-type expanded memory, if a suitable card is installed in the computer. *in_ems* is only relevant for DOS. On other platforms it has the same effect as *in_memory*

These values, *in_file*, *in_memory*, and *in_ems*, are elements of the pre-declared domain ***place***, which corresponds to the following declaration:

```
DOMAINS
   place = in_file; in_memory; in_ems
```

For example, here are two different calls to ***db_create***:

```
db_create(db_sel1,"MYFILE.DBA",in_file)
                                        /* Creates disk file MYFILE.DBA */
db_create(db_sel2,"SymName2",in_memory)
                                    /* Creates memory database SymName2 */
```

db_open/3

db_open opens a previously created database, identified by *Name* and *Place*.

```
db_open(Dbase, Name, Place)                                  /* (i,i,i) */
```

If *Place* is *in_memory* or *in_ems*, *Name* is the database's symbolic file name; if *Place* is *in_file*, *Name* is the actual DOS-style file name.

db_copy/3

Irrespective of where you initially place an external database, you can later move it to another location with the ***db_copy*** predicate.

```
db_copy(Dbase, Name, Place)                                  /* (i,i,i) */
```

For example, in this call to ***db_copy***

```
db_copy(my_base, "new_EMSbase", in_ems)
```

Visual Prolog copies the database identified by the database selector *my_base* into the new database file *new_EMSbase*, which is placed in EMS under DOS.

When you copy a database, the original still exists; you will have two copies until you explicitly delete the original.

Once you've moved a database, all processing can continue as if nothing happened, since all reference numbers to the external database terms will still be valid. In this way, if you're maintaining an external database in main memory, and free storage is running short, you can copy the database to a file and continue execution with the database in the file. An index set up to the external database in internal memory is still valid, even after you've copied the database to a file.

db_copy has several uses; you can use it to do the following:

- Load a database from disk to memory and later save it again in binary form, instead of using **save** and **consult** with text files.
- Copy a medium-sized database from disk to memory for faster access.
- Pack a database containing too much free space; when the database is copied to another file, all free space will be eliminated.

db_loadems/2 and db_saveems/2

db_copy performs a full-scale record-by-record copy of the database in question. This has the advantage that the resulting database will be compacted and without unused space, but for large databases the process can be time consuming.

For DOS only, ***db_loadems*** and ***db_saveems*** will transfer complete images of databases between disk and EMS:

```
db_loadems(FileName,EmsName                                  /* (i,i) */
db_saveems(EmsName,FileName)                                 /* (i,i) */
```

The only restriction on their use is that there can be no more than one database in EMS.

db_openinvalid/3

db_openinvalid allows you to open a database that's been flagged as invalid.

```
db_openinvalid(Dbase, Name, Place)                        /* (i,i,i) */
```

If the power to the computer fails while a database is being updated, all the data in the database may be lost because part of some buffer has not been written to disk. A flag in the database indicates if it's in an invalid state after an update.

A database is recorded as being invalid after a call to any of the predicates that change the content in the database. These include ***chain_inserta***, ***chain_insertz***, ***chain_insertafter***, ***term_replace***, ***term_delete***, ***chain_delete***, ***bt_create***, ***key_insert***, and ***key_delete***. The database is recorded as being valid once again when it is closed with ***db_close***, or when ***db_flush*** is called to flush out the buffers.

By using ***db_openinvalid***, it is sometimes possible to continue execution when a database is marked as invalid. This might make it possible to recover some data if all your backups have disappeared. However, all attempts to use an invalid database after opening it with ***db_openinvalid*** might yield unexpected results.

db_flush/1

db_flush flushes the buffers and writes their contents to the appropriate destination in your database.

```
db_flush(Dbase)                                              /* (i) */
```

When a database is updated it will be marked as invalid, and it remains flagged as invalid until it is either flushed with ***db_flush***, or closed.

The level of security you employ for a given database will, of course, depend on how important its data is. The most basic level of data security is to keep backups on disk. At the intermediate level, you could call ***db_flush*** after each important database update. However, flushing the buffers is a relatively slow operation; if it's done too often, your database system will grind to a halt. Finally, if the contents of an external database are especially valuable, you could record all changes in a special log file or maintain two identical databases – perhaps on different disks.

db_close/1

A call to ***db_close*** closes an open database.

```
db_close(Dbase)                                          /* (i) */
```

If the database *Dbase* is placed in a disk file, the file will be closed. The database won't be deleted, even if it is placed in memory or in an EMS-type memory expansion card, and you can reopen it later through a call to ***db_open***. You can use ***db_delete*** to remove a closed database from memory.

db_delete/1

When the database is situated in memory or in an EMS-type memory expansion card, ***db_delete*** releases all the occupied space.

```
db_delete(Name, Place)                                   /* (i,i) */
```

When the database is situated in a file, ***db_delete*** erases the file. ***db_delete*** gives an error if the database Name does not exist in the given Place.

db_garbagecollect/1

db_garbagecollect scans through the free lists in the database garbage collect and tries to merge some of the free space together into larger pieces.

```
db_garbagecollect(Dbase)                                 /* (i) */
```

This scanning and merging is done automatically when the database is placed in memory or in an EMS card.

Under normal circumstances, there should be no need to call this predicate. However, if there seems to be too much free space in the database that is not being reused when new terms are inserted, ***db_garbagecollect*** can regain some extra space.

db_btrees/2

During backtracking, ***db_btrees*** successively binds *BtreeName* to the name of each B+ tree in the *Dbase* database.

```
nondeterm db_btrees(Dbase, BtreeName)                    /* (i,o) */
```

The names are returned in sorted order. B+ trees are described later in this chapter.

db_chains/2

During backtracking, ***db_chains*** successively binds *ChainName* to the name of each chain in the *Dbase* database.

```
nondeterm db_chains(Dbase, ChainName)                    /* (i,o) */
```

The names are returned in sorted order.

db_statistics/5

db_statistics returns statistical information for the database *Dbase*.

```
db_statistics(Dbase, NoOfTerms, MemSize, DbaSize, FreeSize)
                                                  /* (i,o,o,o,o) */
```

The arguments to ***db_statistics*** represent the following:

NoOfTerms is bound to the total number of terms in the database.

MemSize	is bound to the size – in bytes – of the internal tables stored in memory for the database.
DbaSize	is bound to the total number of bytes that the terms and descriptors in the *Dbase* database occupy. If *Dbase* is stored in a disk file, and *DbaSize* gets a value much smaller than the size of that file, the file can be compressed by using ***db_copy***.

FreeSize becomes bound to a value representing the free memory space; the value depends on where the database *Dbase* is currently placed.

- When Dbase is placed in memory, FreeSize is bound to the number of unused bytes between the top of the global stack and the top of the heap. (**Note:** There might be some additional free bytes that are not included in this count.)
- If Dbase is placed in EMS-type expanded memory, FreeSize is bound to the number of unoccupied bytes in that expansion memory.
- When Dbase is placed in a file, FreeSize is bound to the number of unused bytes on the disk containing the file.

Manipulating Chains

To insert terms into an external database chain, you use the predicates ***chain_inserta***, ***chain_insertz***, or ***chain_insertafter***. You can successively bind the terms in a chain, and their reference numbers, to the arguments of ***chain_terms***, while ***chain_delete*** allows you to delete a whole chain of terms from the external database.

Four standard predicates return database reference numbers. These are ***chain_first***, ***chain_last***, ***chain_next***, and ***chain_prev***.

chain_inserta/5 and chain_insertz/5

The predicates ***chain_inserta*** and ***chain_insertz*** correspond to ***asserta*** and ***assertz***, respectively. These take the following form:

```
chain_inserta(Dbase, Chain, Domain, Term, Ref)        /* (i,i,i,i,o) */
chain_insertz(Dbase, Chain, Domain, Term, Ref)        /* (i,i,i,i,o) */
```

chain_inserta inserts the term *Term* at the beginning of the chain *Chain*, while ***chain_insertz*** inserts *Term* at the chain's end. *Dbase* is the ***db_selector*** of the database, *Domain* is the domain of *Term*, and *Ref* is the database reference number corresponding to *Term*. For example, if *my_dba* is declared to be in the domain ***db_selector***, like this:

```
DOMAINS
   db_selector = my_dba; ....
```

then in this call to ***chain_inserta***

```
chain_inserta(my_dba, customer, person, p(john,
                                          "1 The Avenue", 32), NewRef)
```

customer is the name of the chain, and all customers are stored in one chain. It would be perfectly all right to store the suppliers as terms from the domain *person* but in a different chain, perhaps called supplier. *person* is the name of the domain to which `p(john, "1 The Avenue", 32)` belongs, as shown in this domain declaration:

```
DOMAINS
   person = p(name, address, age)
```

If *Chain* doesn't already exist, these predicates will automatically create it.

chain_insertafter/6

chain_insertafter inserts a term after a specified term, returning the inserted term's new reference number. It takes this format:

```
chain_insertafter(Dbase, ChainName, Domain, Ref, Term, NewRef)
                                                  /* (i,i,i,i,i,o) */
```

chain_insertafter inserts the term *Term* after the chain element specified by *Ref*, while *NewRef* is bound to the database reference number corresponding to *Term* after it's been inserted.

chain_terms/5

During backtracking, ***chain_terms*** successively binds *Term* and *Ref* to each term and its associated database reference number in the specified *Chain*. ***chain_terms*** takes the form:

```
chain_terms(Dbase, Chain, Domain, Term, Ref)          /* (i,i,i,o,o) */
```

chain_delete/2

chain_delete deletes a specified chain from a given external database; this predicate takes the form:

```
chain_delete(Dbase, Chain)                              /* (i,i) */
```

chain_first/3 and chain_last/3

chain_first and ***chain_last*** return the database reference number for the first and last terms in a given chain, respectively.

```
chain_first(Dbase, Chain, FirstRef)                    /* (i,i,o) */
chain_last(Dbase, Chain, LastRef)                      /* (i,i,o) */
```

chain_next/3 and chain_prev/3

chain_next returns the reference number of the term following the given one, while ***chain_prev*** returns the reference number of the term preceding the given one.

```
chain_next(Dbase, Ref, NextRef)                        /* (i,i,o) */
chain_prev(Dbase, Ref, PrevRef)                        /* (i,i,o) */
```

Manipulating Terms

Three standard predicates for external database management are all concerned with terms; these are ***term_replace***, ***term_delete***, and ***ref_term***. Whenever you call any of the term-handling external database standard predicates, you must give the domain of the term as one of the arguments. Because of this, it's usually a good idea to declare all terms in a given database as alternatives in *one* domain, as in this declaration:

```
DOMAINS
   terms_for_my_stock_control_database =
      customer(Customer, Name, ZipCode, Address);
      supplier(SupplierNo, Name, Address);
      parts(PartNo, Description, Price, SupplierNo)
```

Note that there are no restrictions on mixing types (domains) in an external database. One chain can contain text strings, another integers, a third some kind of compound structures, and so on. However, external database data items are not stored with type descriptors; for example, integers don't necessarily occupy just two bytes. It's your responsibility to retrieve a term into the same domain as that from which it was inserted. A run-time error will usually result if you attempt to mix domains.

term_replace/4

term_replace replaces an old term (referenced by *Ref*, a database reference number) with a new term, *Term*.

```
term_replace(Dbase, Domain, Ref, Term)                    /* (i,i,i,i) */
```

term_delete/3

term_delete erases the term stored under *Ref*, a given database reference number.

```
term_delete(Dbase, Chain, Ref)                              /* (i,i,i) */
```

The storage occupied by the term will be released, and there must be no further references to *Ref*.

ref_term/4

ref_term binds *Term* to the term stored under a given reference number, *Ref*.

```
ref_term(Dbase, Domain, Ref, Term)                        /* (i,i,i,o) */
```

A Complete Program Example

The following example program `ch14e02.pro` uses nearly all the external database predicates introduced so far. Working first in memory, this program goes through the following sequence of operations:

1. Writes 100 terms in a database.
2. Reads them back.
3. Replaces every second term.
4. Doubles the number of terms.
5. Erases every second term.
6. Examines every term with ***ref_term***.
7. Calculates the size of the database.

This program then copies the database to a disk file and carries out the same sequence of activities twice with the database held on disk. Finally, it calculates – in hundredths of a second – the total time taken to carry out these activities. Note, however, that for illustration the program generates large amounts of output, which slows it down considerably. The true speed is only revealed if you remove the output statements. The program `ch14e03.pro` is for UNIX, as time-calculation is done differently in UNIX, and terminal output is significantly slower in UNIX than in DOS.

Run the program to see what happens, and then try to alter the number of terms and study your system's performance. The DOS program appears below.

```
                        /* Program ch14e02.pro */

DOMAINS
   my_dom = f(string)
   db_selector = my_dba

PREDICATES
   write_dba(integer)
   read_dba
   rd(Ref)
   count_dba(integer)
   count(Ref, integer, integer)
   replace_dba
   replace(Ref)
   double_dba
   double(Ref)
   half_dba
   half(Ref)
   mixture

CLAUSES
   write_dba(0):-!.
   write_dba(N):-
      chain_inserta(my_dba,my_chain,my_dom,f("Prolog system"),_),
      chain_insertz(my_dba, my_chain, my_dom, f("Prolog Compiler"), _),
      N1=N-1,
      write_dba(N1).

   read_dba:-
      db_chains(my_dba, Chain),
      chain_terms(my_dba, Chain, my_dom, Term, Ref),nl,
      write("Ref=", Ref),
      write(", Term=", Term),
      fail.
   read_dba:-
      db_chains(my_dba, Chain),
      chain_first(my_dba, Chain, Ref),
      rd(Ref),
      fail.
      read_dba.
```

```
rd(Ref):-
   ref_term(my_dba, my_dom, Ref, Term), nl,
   write(Term),
   fail.
rd(Ref):-
   chain_next(my_dba,Ref,Next),!,rd(Next).
rd(_).

replace_dba:-
   chain_first(my_dba, my_chain, Ref),
   replace(Ref).

replace(Ref):-
   term_replace(my_dba, my_dom, Ref, f("Prolog Toolbox")),
   chain_next(my_dba, Ref, NN),
   chain_next(my_dba, NN, Next),!,
   replace(Next).
   replace(_).

half_dba:-
   chain_last(my_dba, my_chain, Ref),
   half(Ref).

half(Ref):-
   chain_prev(my_dba, Ref, PP),
   chain_prev(my_dba, PP, Prev), !,
   term_delete(my_dba, my_chain, Ref),
   half(Prev).
half(_).

double_dba:-
   chain_first(my_dba, my_chain, Ref),
   double(Ref).

double(Ref):-
   chain_next(my_dba, Ref, Next),!,
   chain_insertafter(my_dba, my_chain, my_dom, Ref,f("Programmers
                    Guide"), _),
   double(Next).
   double(_).

count_dba(N):-
   chain_first(my_dba, my_chain, Ref),
   count(Ref, 1, N).
```

```
   count(Ref, N, N2):-
      chain_next(my_dba, Ref, Next),!,
      N1=N+1,
      count(Next, N1, N2).
   count(_, N, N).

   mixture :-nl,
      write("Replace every second term:"),
      replace_dba,nl,
      write("Double the number of terms:"),
      double_dba,nl,
      write("Erase every second term:"),
      half_dba,nl,
      write("Use ref_term for all terms:"),
      read_dba,
      count_dba(N),nl,
      write("There are now ", N, " terms in the database"),
      db_statistics(my_dba, NoOfTerms, MemSize, DbaSize, FreSize),nl,
      writef("NoOfTerms=%, MemSize=%, DbaSize=%, FreeSize=%", NoOfTerms,
             MemSize,DbaSize,FreSize).

GOAL
      nl,nl,nl,
      write("\tTEST OF DATABASE SYSTEM\n\t**********************\n\n"),
      time(H1, M1, S1, D1),
      db_create(my_dba, "dd.dat", in_memory),nl,nl,
      write("Write some terms in the database:"),
      write_dba(50),
      read_dba,
      mixture,nl,nl,

      write("Copy to file"),
      db_copy(my_dba, "dd.dat", in_file),
      db_close(my_dba),  db_delete("dd.dat", in_memory),
      db_open(my_dba, "dd.dat", in_file),
      mixture,
      db_close(my_dba),nl,nl,nl,

      write("Open the database on file"),
      db_open(my_dba, "dd.dat", in_file),
      mixture,
      db_close(my_dba),

      time(H2, M2, S2, D2),
      Time = (D2-D1)+100.0*((S2-S1)+60.0*((M2-M1)+ 60.0*(H2-H1))),nl,nl,
      write("Time = ", Time, "/100 Sec" ), nl.
```

B+ Trees

A B+ tree is a data structure you can use to implement a very efficient method for sorting, large amounts of data efficient method for sorting large amounts of data; B+ trees enable a correspondingly efficient searching algorithm. You can think of a B+ tree as providing an *index* to a database, which is why B+ trees are sometimes referred to as indices.

In Visual Prolog, a B+ tree resides in an external database. Each entry in a B+ tree is a pair of values: a *key string* key string and an associated *database reference number*. When building your database, you first insert a record in the database and establish a key for that record. The Visual Prolog B+tree predicates may then be used to insert this key and the database reference number corresponding to this record into a B+ tree.

When searching a database for a record, all you have to do is to obtain a key for that record, and the B+ tree will give you the corresponding reference number. Using this reference number, you can retrieve the record from the database. As a B+ tree evolves, its entries are kept in key order. This means that you can easily obtain a sorted listing of the records.

A B+ tree is analogous to a binary tree, with the exception that in a B+ tree, more than one key string is stored at each node. B+ trees are also balanced; this means that the search paths to each key in the leaves of the tree have the same length. Because of this feature, a search for a given key among more than a million keys can be guaranteed, even in the worst case, to require accessing the disk only a few times – depending on how many keys are stored at each node.

Although B+ trees are placed in an external database, they don't need to point to terms in the same database. It is possible to have a database containing a number of chains, and another database with a B+ tree pointing to terms in those chains.

Pages, Order, and Key Length

In a B+ tree, keys are grouped together in *pages*; each page has the same size, and all pages can contain the same number of keys, which means that all the stored keys for that B+ tree must be the same size. The size of the keys is determined by the *KeyLen* argument, which you must supply when creating a B+ tree. If you attempt to insert strings longer than *KeyLen* into a B+ tree, Visual Prolog will truncate them. In general, you should choose the smallest possible value for *KeyLen* in order to save space and maximize speed.

When you create a B+ tree, you must also give an argument called its *Order*. This argument determines how many keys should be stored in each tree node; usually, you must determine the best choice by trial and error. A good first try for *Order* is 4, which stores between 4 and 8 keys at each node. You must choose the value of *Order* by experimentation because the B+ tree's search speed depends on the values *KeyLen* and *Order*, the number of keys in the B+ tree, and your computer's hardware configuration.

Duplicate Keys

When setting up a B+ tree, you must allow for all repeat occurrences of your key. For example, if you're setting up a B+ tree for a database of customers in which the key is the customer's last name, you need to allow for all those customers called *Smith*. For this reason, it is possible to have duplicate keys in a B+ tree.

When you delete a term in the database, you must delete the corresponding entry in a B+ tree with duplicate keys by giving both the key and the database reference number.

Multiple Scans

In order multiple, scans of B+ trees to have more than one internal pointer to the same B+ tree, you can open the tree more than once. Note, however, that if you update one copy of a B+ tree, for which you have other copies currently open, the pointers for the other copies will be repositioned to the top of the tree.

The B+ Tree Standard Predicates

Visual Prolog provides several predicates for handling B+ trees; these predicates work in a manner that parallels the corresponding ***db_...*** predicates.

bt_create/5 and bt_create/6

You create new B+ trees by calling the ***bt_create*** predicate.

```
bt_create(Dbase, BtreeName, Btree_Sel, KeyLen, Order)
                                                    /* (i,i,o,i,i) */
bt_create(Dbase, BtreeName, Btree_Sel, KeyLen, Order, Duplicates)
                                                  /* (i,i,o,i,i,i) */
```

The *BtreeName* argument specifies the name for the new tree. You later use this name as an argument for ***bt_open***. The arguments *KeyLen* and *Order* for the B+ Tree are given when the tree is created and can't be changed afterwards. If you

are calling ***bt_create*/*5*** or ***bt_create*/*6*** with the *Duplicates* argument set to 1, duplicates will be allowed in the B+Tree. If you call ***bt_create*/*6*** with the *Duplicates* argument set to 0 you will not be allowed to insert duplicates in the B+Tree.

bt_open/3

bt_open opens an already created B+ tree in a database, which is identified by the name given in ***bt_create***.

```
bt_open(Dbase, BtreeName, Btree_Sel)                    /* (i,i,o) */
```

When you open or create a B+ tree, the call returns a selector *(Btree_Sel)* for that B+ tree. A B+ tree selector belongs to the predefined domain ***bt_selector*** and refers to the B+ tree whenever the system carries out search or positioning operations. The relationship between a B+ tree's name and its selector is exactly the same as the relationship between an actual file name and the corresponding symbolic file name.

You can open a given B+ tree more than once in order to handle several simultaneous scans. Each time a B+ tree is opened, a descriptor is allocated, and each descriptor maintains its own internal B+ tree pointer.

bt_close/2 and bt_delete/2

You can close an open B+ tree with a call to ***bt_close*** or delete an entire B+ tree with ***bt_delete***.

```
bt_close(Dbase, Btree_Sel)                              /* (i,i) */
bt_delete(Dbase, BtreeName)                             /* (i,i) */
```

Calling ***bt_close*** releases the internal buffers allocated for the open B+ tree with *BtreeName*.

bt_copyselector

bt_copyselector gives you a new pointer for an already open B+ tree selector (a new scan).

```
bt_copyselector(Dbase,OldBtree_sel,NewBtree_sel)        /* (i,i,o) */
```

The new selector will point to the same place in the B+ tree as the old selector. After the creation the two B+ tree selectors can freely be repositioned without affecting each other.

bt_statistics/8

bt_statistics returns statistical information for the B+ tree identified by *Btree_Sel*.

```
bt_statistics(Dbase,Btree_Sel,NumKeys,NumPages,          /* (i,i,o,o, */
              Depth,KeyLen,Order,PgSize)                  /* o,o,o,o) */
```

The arguments to ***bt_statistics*** represent the following:

Dbase	is the **db_selector** identifying the database.
Btree_Sel	is the **bt_selector** identifying the B+ tree.
NumKeys	is bound to the total number of keys in the B+ tree *Btree_Sel*.
NumPages	is bound to the total number of pages in the B+ tree.
Depth	is bound to the depth of the B+ tree.
KeyLen	is bound to the key length.
Order	is bound to the order of the B+ tree.
PgSize	is bound to the page size (in bytes).

key_insert/4 and key_delete/4

You use the standard predicates ***key_insert*** and ***key_delete*** to update the B+ tree.

```
key_insert(Dbase, Btree_Sel, Key, Ref                    /* (i,i,i,i) */
key_delete(Dbase, Btree_Sel, Key, Ref)                   /* (i,i,i,i) */
```

By giving both *Key* and *Ref* to ***key_delete***, you can delete a specific entry in a B+ tree with duplicate keys.

key_first/3, key_last/3, and key_search/4

Each B+ tree maintains an internal pointer to its nodes. ***key_first*** and ***key_last*** allow you to position the pointer at the first or last key in a B+ tree, respectively. ***key_search*** positions the pointer on a given key.

```
key_first(Dbase, Btree_Sel, Ref)                          /* (i,i,o) */
key_last(Dbase, Btree_Sel, Ref)                           /* (i,i,o) */
key_search(Dbase, Btree_Sel, Key, Ref)         /* (i,i,i,o)(i,i,i,i) */
```

If the key is found, ***key_search*** will succeed; if it's not found, ***key_search*** will fail, but the internal B+ tree pointer will be positioned at the key immediately after where Key would have been located. You can then use ***key_current*** to return the key and database reference number for this key. If you want to position

on an exact position in a B+ tree with duplicates you can also provide the *Ref* as an input argument.

key_next/3 and key_prev/3

You can use the predicates ***key_next*** and ***key_prev*** to move the B+ tree's pointer forward or backward in the sorted tree.

```
key_next(Dbase, Btree_Sel, NextRef)                    /* (i,i,o) */
key_prev(Dbase, Btree_Sel, PrevRef)                    /* (i,i,o) */
```

If the B+ tree is at one of the ends, trying to move the pointer further will cause a fail, but the B+ tree pointer will act as if it were placed one position outside the tree.

key_current/4

key_current returns the key and database reference number for the current pointer in the B+ tree.

```
key_current(Dbase, Btree_Sel, Key, Ref)              /* (i,i,o,o) */
```

key_current fails after a call to the predicates ***bt_open***, ***bt_create***, ***key_insert***, or ***key_delete***, or when the pointer is positioned before the first key (using ***key_prev***) or after the last (with ***key_next***).

Example: Accessing a Database via B+ Trees

The following example program handles several text files in a single database file at once. You can select and edit the texts as though they were in different files. A corresponding B+ tree is set up for fast access to the texts and to produce a sorted list of the file names.

```
                        /* Program ch14e04.pro */

DOMAINS
   db_selector = dba

PREDICATES
   % List all keys in an index
   list_keys(db_selector,bt_selector)
```

```
CLAUSES
   list_keys(dba,Bt_selector):-
      key_current(dba,Bt_selector,Key,_),
      write(Key,' '),
      fail.
   list_keys(dba,Bt_selector):-
      key_next(dba,Bt_selector,_),!,
      list_keys(dba,Bt_selector).
   ist_keys(_,_).

PREDICATES
   open_dbase(bt_selector)
   main(db_selector,bt_selector)
   ed(db_selector,bt_selector,string)
   ed1(db_selector,bt_selector,string)

CLAUSES
   % Loop until escape is pressed
   main(dba,Bt_select):-
      write("File Name: "),
      readln(Name),
      ed(dba,Bt_select,Name),!,
      main(dba,Bt_select).
   main(_,_).

   % The ed predicates ensure that the edition will never fail.
   ed(dba,Bt_select,Name):-
      ed1(dba,Bt_select,Name),!.
   ed(_,_,_).

   %* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   % There are three choices:
   %% a) The name is an empty string - list all the names
   % b) The name already exists - modify the contents of the file
   % c) The name is a new name - create a new file
   %* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
   ed1(dba,Bt_select,""):-!,
      key_first(dba,Bt_select,_),
      list_keys(dba,Bt_select),
      nl.
   ed1(dba,Bt_select,Name):-
      key_search(dba,Bt_select,Name,Ref),!,
      ref_term(dba,string,Ref,Str),
      edit(Str,Str1,"Edit old",NAME,"",0,"PROLOG.HLP",RET),
      clearwindow,
      Str><Str1, RET=0,
      term_replace(dba, string, Ref, Str1).
   ed1(dba,Bt_select,Name):-
      edit("",STR1,"Create New",NAME,"",0,"PROLOG.HLP",RET),
      clearwindow,
      ""><Str1, RET=0,
      chain_insertz(dba,file_chain,string,Str1,Ref),
      key_insert(dba,Bt_select,Name,Ref).

   open_dbase(INDEX):-
      existfile("dd1.dat"),!,
      db_open(dba,"dd1.dat",in_file),
      bt_open(dba,"ndx",INDEX).

   open_dbase(INDEX):-
      db_create(dba,"dd1.dat",in_file),
      bt_create(dba,"ndx",INDEX,20,4).

GOAL
      open_dbase(INDEX),
      main(dba,INDEX),
      bt_close(dba,INDEX),
      db_close(dba).
```

External Database Programming

In this section, we provide seven examples that illustrate some general principles and methods for working with Visual Prolog's external database system. This is a summary of what the following sections cover:

"*Scanning through a Database*" shows you the way to perform a sequential scan through a chain or a B+ tree in an external database.

"*Displaying the Contents of a Database*" defines a predicate you can use to display the current state of an external database.

"*Making a Database That Won't Break Down*" illustrates how to protect your database from unexpected system power failure and other potential catastrophes.

"*Updating the Database*" provides an example that makes it easy to change, add to, and protect your database.

"*Using Internal B+ Tree Pointers*" supplies you with some predicates for positioning a pointer within an open B+ tree.

"*Changing the Structure of a Database*" offers an alternative to the old copy-while-changing method of changing the structure of a database.

Scanning through a Database

When you are using the database system, it is important to keep Visual Prolog's storage mechanisms storage mechanisms in mind. Every time Visual Prolog retrieves a term from an external database with the ***ref_term*** predicate, it places that term on the global stack. The system won't release the space occupied by that term until the program fails and backtracks to a point before the call to ***ref_term***. This means that, to do a sequential scan through a chain in an external database, you should always use a structure like the following:

```
/* Structure for sequentially scanning through a chain */

scan(db_selector, Chain, ....) :-
   chain_first(db_selector, Chain, Ref),
   scanloop(db_selector, Ref).

scanloop(db_selector, Ref) :-
   ref_term(db_selector, mydom, Ref, Term),
   /* ... do your processing ... */
   fail.

scanloop(db_selector, _) :-
   chain_next(db_selector, Ref, NextRef),
   scanloop(db_selector, NextRef).
```

Similarly, for a sequential scan through an index, you should use a structure like this:

```
/* Structure for sequentially scanning through an index */

scan(db_selector, Bt_selector) :-
   key_first(db_selector, Bt_selector, FirstRef),
   scanloop(db_selector, Bt_selector, FirstRef).
```

```
scanloop(db_selector, Bt_selector, Ref) :-
   ref_term(db_selector, mydom, Ref, Term),
   /* ... do your processing ... */
   fail.

scanloop(db_selector, Bt_selector, _) :-
   key_next(db_selector, Bt_selector, NextRef),
   scanloop(db_selector, Bt_selector, NextRef).
```

You can also carry out a sequential scan through a chain in the database by using ***chain_terms***, like this:

```
/* Another way to sequentially scan through a chain */

scan(db_selector, Chain) :-
   chain_terms(db_selector, Chain, mydom, Term, Ref),
/* ... do your processing ... */
   fail.
scan(_, _).
```

To scan through a B+ tree, you could have also defined and used the predicate ***bt_keys***. During backtracking, this predicate returns (for a given B+ tree and database) each key in the tree and its associated database reference number.

```
                /* This fragment goes with ch14e05.pro */

PREDICATES
   bt_keys(db_selector, bt_selector, string, ref)
   bt_keysloop(db_selector, bt_selector, string, ref)

CLAUSES
   bt_keys(Db_selector, Bt_selector, Key, Ref):-
      key_first(Db_selector, Bt_selector, _),
      bt_keysloop(Db_selector, Bt_selector, Key, Ref).

   bt_keysloop(Db_selector, Bt_selector, Key, Ref):-
      key_current(Db_selector, Bt_selector, Key, Ref).

   bt_keysloop(Db_selector, Bt_selector, Key, Ref):-
      key_next(Db_selector, Bt_selector, _),
      bt_keysloop(Db_selector, Bt_selector, Key, Ref).
```

Displaying the Contents of a Database

You can use the predicate ***listdba***, defined in the following program fragment, to display the current state of an external database. ***listdba*** has one argument: the selector of a database assumed to be open. All terms in the database must belong

to the same domain. In the example, the domain is called ***mydom***; when you use this predicate, you must replace ***mydom*** with the actual name of the appropriate domain in your program.

```
                        /* Program ch14e05.pro */

CONSTANTS
   filename = "\\vip\\vpi\\programs\\register\\exe\\register.bin"

DOMAINS
   db_selector = mydba
   mydom = city(zipcode, cityname);
           person(firstname, lastname, street, zipcode, code)
   zipcode, cityname, firstname, lastname, street, code  =  string

PREDICATES
   listdba(db_selector)
   nondeterm bt_keys(db_selector,bt_selector,string,ref)
   nondeterm bt_keysloop(db_selector,bt_selector,string,ref)

CLAUSES
   listdba(Db_selector):-nl,
      write("****************************************"),nl,
      write("           DATABASE LISTING"),nl,
      write("****************************************"),
      db_statistics(Db_selector,NoOfTerms,MemSize,DbaSize,FreeSize),nl,nl,
      write("Total number of records in the database: ",NoOfTerms),nl,
      write("Number of bytes used in main memory: ",MemSize),nl,
      write("Number of bytes used by the database: ",DbaSize),nl,
      write("Number of bytes free on disk: ",FreeSize),nl,
      fail.

   listdba(Db_selector):-
      db_chains(Db_selector,Chain),nl,nl,nl,nl,
      write("******* Chain LISTING *************"),nl,nl,
      write("Name=",Chain),nl,nl,
      write("CONTENT OF: ",Chain),nl,
      write("-----------------------------\n"),
      chain_terms(Db_selector, Chain, mydom,Term, Ref),
      write("\n", Ref, ": ",Term),
      fail.
```

```
   listdba(Db_selector):-
   db_btrees(Db_selector,Btree),           /* Returns each B+ tree name */
      bt_open(Db_selector,Btree,Bt_selector),
      bt_statistics(Db_selector,Bt_selector,NoOfKeys,
                    NoOfPages,Dept,KeyLen,Order,PageSize),nl,nl,nl,
      write("******** INDEX LISTING **************"),nl,nl,
      write("Name=     ", Btree),nl,
      write("NoOfKeys= ", NoOfKeys),nl,
      write("NoOfPages=", NoOfPages),nl,
      write("Dept=     ", Dept),nl,
      write("Order=    ", Order),nl,
      write("KeyLen=   ", KeyLen),nl,
      write("PageSize= ", PageSize), nl,
      write("CONTENT OF: ", Btree),nl,
      write("----------------------------\n"),
      bt_keys(Db_selector,Bt_selector,Key,Ref),
      write("\n",Key, " - ",Ref),
      fail.
   listdba(_).

   bt_keys(Db_selector,Bt_selector,Key, Ref):-
      key_first(Db_selector,Bt_selector,_),
      bt_keysloop(Db_selector,Bt_selector,Key,Ref).

   bt_keysloop(Db_selector,Bt_selector,Key,Ref):-
      key_current(Db_selector,Bt_selector,Key,Ref).

   bt_keysloop(Db_selector,Bt_selector,Key,Ref):-
      key_next(Db_selector,Bt_selector,_),
   bt_keysloop(Db_selector,Bt_selector,Key,Ref).

GOAL
      db_open(mydba,filename,in_file),
      listdba(mydba).
```

Implementing a Database That Won't Break Down

If you enter a lot of new information into a database, it is important to ensure that this information won't be lost if the system goes down. In this section, we illustrate one way of doing this – by logging all changes in another file.

Making a change involves first updating the database, and then flushing it. If this operation succeeds, the system then records the change in the log file and flushes the log file itself. This means that only one file is unsafe at any given time. If the database file becomes invalid (because the system went down before the file was flushed, for example), you should be able to reconstruct it by merging the log file

with a backup of the database file. If the log file becomes invalid, you should create a new log file and make a backup of the database file.

If you record the date and time in the log file, together with the old values from a modification involving replacement or deletion, you should be able to reconstruct the database to its state at a given time.

```
            /* This program fragment goes with ch14e05.pro */

DOMAINS
   logdom = insert(relation,dbdom,ref);
         replace(relation,dbdom,ref,dbdom);
         erase(relation,ref,dbdom)

PREDICATES
   logdbchange(logdom)

CLAUSES
   logdbchange(Logterm):-
      chain_insertz(logdba,logchain,logdom,Logterm,_),
      db_flush(logdba).
```

Updating the Database

As a general principle, you shouldn't spread database updating throughout the program but should keep it in some user-defined predicates. This makes it easier to change the database and/or to add new B+ trees. When you confine updating this way, it's also easier to make a robust database system because your program involves only a small piece of code in which the database is unsafe.

The following example handles updating two different relations, whose objects are all strings:

```
   person(firstname, lastname, street, zipcode, code)

   city(zipcode, cityname)
```

It handles the updating with the following indexes (keys) on the *person* and *city* relations:

```
   Person's Name............Last Name plus First Name
   Person's Address.........Street Name
   City Number..............Zip Code
```

In this example, we assume that the B+ trees are already open, and that their *bt_selectors* have been asserted in the database predicate ***indices***.

Before this program initiates the updating, it eliminates the possibility of a BREAK with the ***break*** predicate. After updating is finished, the program flushes the database with ***db_flush***. Although ***db_flush*** makes the updating a slow process (thanks to DOS), the file will be safe after this call.

To make the system as secure as possible, the program logs changes in a special file through a call to ***logdbchange***.

```
                        /* Program ch14e06.pro */

      /* Logging database operations */

DOMAINS
   logdom = insert(relation,dbdom,ref);
   replace(relation,dbdom,ref,dbdom);
   erase(relation,ref,dbdom)

PREDICATES
   logdbchange(logdom)

CLAUSES
   logdbchange(Logterm):-
      chain_insertz(logdba,logchain,logdom,Logterm,_),
      db_flush(logdba).

DOMAINS
   dbdom = city(zipcode, cityname);
   person(firstname, lastname, street, zipcode, code)
   zipcode, cityname, firstname, lastname = string
   street, code = string
   indexName = person_name; person_adr; city_no
   relation  = city; person
   db_selector = dba; logdba

facts
   % This takes and index  name (a key) that is a person's name or address
   %or a city number; it also takes a B+ tree selector
   indices(IndexName, bt_selector)

PREDICATES
   %and a first name (10 characters)
   % This predicate creates an index name from a last name (20 characters)
   xname(FirstName,LastName,string)

CLAUSES
   xname(F,L,S):-
      str_len(L,LEN),LEN>20,!,
      frontstr(20,L,L1,_),
      format(S,"%-20%",L1,F).
```

```
   xname(F,L,S):-
   format(S,"%-20%",L,F).

PREDICATES
   ba_insert(relation, dbdom)
   dba_replace(relation, dbdom, Ref)
   dba_erase(relation, Ref)

CLAUSES
   dba_insert(person,Term):-!,
      break(OldBreak),
      break(off),
      indices(person_name,I1),
      indices(person_adr,I2),!,
      Term = person(Fname,Lname,Adr,_,_),
      xname(Fname,Lname,Xname),
      chain_insertz(dba,person,dbdom,Term,Ref),
      key_insert(dba,I1,Xname,Ref),
      key_insert(dba,I2,Adr,Ref),
      db_flush(dba),
      logdbchange(insert(person,Term,Ref)),
      break(OldBreak).

   dba_insert(city,Term):-
      break(OldBreak),
      break(off),
      indices(city_no,I),!,
      Term = city(ZipCode,_),
      chain_insertz(dba,city,dbdom,Term,Ref),
      key_insert(dba,I,ZipCode,Ref),
      db_flush(dba),
      logdbchange(insert(city,Term,Ref)),
      break(OldBreak).
```

```
dba_replace(person,NewTerm,Ref):-!,
   break(OldBreak),
   break(off),
   indices(person_name,I1),
   indices(person_adr,I2),!,
   ref_term(dba,dbdom,Ref,OldTerm),
   OldTerm=person(OldFname,OldLname,OldAdr,_,_),
   xname(OldFname,OldLname,OldXname),
   key_delete(dba,I1,OldXname,Ref),
   key_delete(dba,I2,Oldadr,Ref),
   NewTerm=person(NewFname,NewLname,NewAdr,_,_),
   xname(NewFname,NewLname,NewXname),
   term_replace(dba,dbdom,Ref,NewTerm),
   key_insert(dba,I1,NewXname,Ref),
   key_insert(dba,I2,NewAdr,Ref),
   db_flush(dba),
   logdbchange(replace(person,NewTerm,Ref,OldTerm)),
   break(OldBreak).

dba_replace(city,NewTerm,Ref):-!,
   break(OldBreak),
   break(off),
   indices(city_no,I),!,
   ref_term(dba,dbdom,Ref,OldTerm),
   OldTerm=city(OldZipCode,_),
   key_delete(dba,I,OldZipCode,Ref),
   NewTerm=city(ZipCode,_),
   term_replace(dba,dbdom,Ref,NewTerm),
   key_insert(dba,I,ZipCode,Ref),
   db_flush(dba),
   logdbchange(replace(city,NewTerm,Ref,OldTerm)),
   break(OldBreak).
```

```
dba_erase(person,Ref):-!,
   break(OldBreak),
   break(off),
   indices(person_name,I1),
   indices(person_adr,I2),!,
   ref_term(dba, dbdom, Ref, OldTerm),
   OldTerm=person(OldFname,OldLname,OldAdr,_,_),
   xname(OldFname,OldLname,OldXname),
   key_delete(dba,I1,OldXname,Ref),
   key_delete(dba,I2,OldAdr,Ref),
   term_delete(dba,person,Ref),
   db_flush(dba),
   logdbchange(erase(person, Ref, OldTerm)),
   break(OldBreak).

dba_erase(city,Ref):-
   break(OldBreak),
   break(off),
   indices(city_no,I),!,
   ref_term(dba,dbdom,Ref,OldTerm),
   OldTerm=city(OldZipCode,_),
   key_delete(dba,I,OldZipCode,Ref),
   term_delete(dba,city,Ref),
   db_flush(dba),
   logdbchange(erase(city,Ref,OldTerm)),
   break(OldBreak).
```

Using Internal B+ Tree Pointers

Each open B+ tree has an associated pointer to its nodes. When you open or update the B+ tree, this pointer is positioned before the start of the tree. When you call ***key_next*** with the pointer at the last key in the tree, the pointer will be positioned after the end of the tree. Whenever the pointer moves outside the tree, ***key_current*** fails. If this arrangement is not appropriate for a particular application, you can model other predicates.

You can use ***mykey_next***, ***mykey_prev*** , and ***mykey_search***, defined in this example, to ensure that the B+ tree pointer is always positioned inside the B+ tree (provided there are any keys in the tree).

```
PREDICATES
   mykey_next(db_selector, bt_selector, ref)
   mykey_prev(db_selector, bt_selector, ref)
   mykey_search(db_selector, bt_selector, string, ref)
```

```
CLAUSES
   mykey_prev(Dba, Bt_selector, Ref) :-
      key_prev(Dba, Bt_selector, Ref), !.
   mykey_prev(Dba, Bt_selector, Ref) :-
      key_next(Dba, Bt_selector, Ref), fail.

   mykey_next(Dba, Bt_selector, Ref) :-
      key_next(Dba, Bt_selector, Ref), !.
   mykey_next(Dba, Bt_selector, Ref) :-
      key_prev(Dba, Bt_selector, Ref), fail.

   mykey_search(Dba, Bt_selector, Key, Ref) :-
      key_search(Dba, Bt_selector, Key, Ref), !.
   mykey_search(Dba, Bt_selector, _, Ref) :-
      key_current(Dba, Bt_selector, _, Ref), !.
   mykey_search(Dba, Bt_selector, _, Ref) :-
      key_last(Dba, Bt_selector, Ref).
```

You can use the ***samekey_next*** and ***samekey_prev*** predicates, defined in the next example, to move the index pointer to the next identical key in a B+ tree that has duplicate keys.

```
PREDICATES
   samekey_next(db_selector, bt_selector, ref)
   try_next(db_selector, bt_selector, ref, string)
   samekey_prev(db_selector, bt_selector, ref)
   try_prev(db_selector, bt_selector, ref, string)

CLAUSES
   samekey_next(Dba, Bt_selector, Ref) :-
      key_current(Dba, Bt_selector, OldKey, _),
      try_next(Dba, Bt_selector, Ref, OldKey).
   try_next(Dba, Bt_selector, Ref, OldKey) :-
      key_next(Dba, Bt_selector, Ref),
      key_current(Dba, Bt_selector, NewKey, _),
      NewKey = OldKey, !.

   try_next(Dba, Bt_selector, _, _) :-
      key_prev(Dba, Bt_selector, _),
      fail.

   samekey_prev(Dba, Bt_selector, Ref) :-
      key_current(Dba, Bt_selector, OldKey, _),
      try_prev(Dba, Bt_selector, Ref, OldKey).
```

```
try_prev(Dba, Bt_selector, Ref, OldKey) :-
   key_prev(Dba, Bt_selector, Ref),
   key_current(Dba, Bt_selector, NewKey, _),
   NewKey = OldKey, !.

try_prev(Dba, Bt_selector, _, _) :-
   key_next(Dba, Bt_selector, _),
   fail.
```

Changing the Structure of a Database

One way to change the structure of a database is to write a small program that copies the old database to a new one while making external databases, changing structure of the changes. Another way, which we'll describe here, is to first dump the database into a text file, make any necessary modifications to the database with a text editor, and then read the modified database back into a new file.

You can use the predicate ***dumpDba***, defined in the next program fragment, to dump the contents of an external database into a text file if the database satisfies the following conditions:

- Every chain in the database models a relation.
- All terms in the database belong to the same domain.

This method does not dump the B+ trees into the text file; we assume, given the first condition, that B+ trees can be generated from the relations. In this example, all terms belong to the generic domain ***mydom***; when you implement this method, replace ***mydom*** with the actual name and a proper declaration.

This code writes the contents of the database to a text file opened by ***outfile***. Each line of the text file contains a term and the name of the containing chain. The term and the chain names are combined into the domain ***chainterm***.

```
                        /* Program ch14e07.pro */

CONSTANTS
   filename = "\\vip\\vpi\\programs\\register\\exe\\register.bin"

DOMAINS
   Db_selector = myDba
   chainterm  = chain(string, mydom)
   file = outfile
   mydom = city(zipcode, cityname);
   person(firstname, lastname, street, zipcode, code)
   zipcode, cityname, firstname, lastname = string
   street, code = string
```

```
PREDICATES
   wr(chainterm)
   dumpDba(string,string)

CLAUSES
   wr(X):-
      write(X),nl.

   dumpDba(Db_selector,OutFile):-
      db_open(myDba,Db_selector,in_file),
      openwrite(outfile,OutFile),
      writedevice(outfile),
      db_chains(myDba,Chain),
      chain_terms(myDba,Chain,mydom,Term,_),
      wr(chain(Chain,Term)),
      fail.

   dumpDba(_,_):-
      closefile(outfile),
      db_close(myDba).

GOAL
   dumpDba(filename,"register.txt").
```

Now, using your customized version of this code, you can generate the text file by calling ***dumpDba***, and you can reload the database by using ***readterm*** with the ***chainterm*** domain. The predicate ***dba_insert***, which we defined in "Updating the Database" (page 399), takes care of the updating.

```
   DOMAINS
      chainterm = chain(string, dbdom)

   PREDICATES
      nondeterm repfile(file)
      copyDba
      loadDba(string)

   CLAUSES
      repfile(_).
   repfile(File) :- not(eof(File)), repfile(File).
```

```
loadDba(OutFile) :-
   openread(Prn_file, OutFile),
   readdevice(Prn_file),
   repfile(Prn_file),
   readterm(Chainterm, chain(Chain, Term)),
   write(Term), nl,
   Dba_insert(Chain, Term),
   fail.

loadDba(_) :-
   closefile(Prn_file).

copyDba :-
   createDba,
   db_open(Dba, "register.bin", in_file),
   open_indices,
   loadDba("register.txt"),
   db_close(Dba).
```

Filesharing and the External Database

Visual Prolog supports file-sharing the external database. This means that a file can be opened by several users or processes simultaneously, which will be useful if you are using the external database in a LAN-application or with one of the multitasking platforms. UNIX developers should take note that Visual Prolog uses advisory file-locking.

Visual Prolog provides the following file-sharing facilities:

- opening an existing database with two different access modes and three different sharing modes for optimal speed.
- grouping database accesses in transactions to ensure consistency
- predicates that make it possible to check whether other users have updated the database.

Filesharing Domains

The two special domains, which are used for file-sharing have the alternatives:

Domain	Functors
accessmode	= read; readwrite
denymode	= denynone; denywrite; denyall

Opening the Database in Share Mode

In order to access the external database in share mode, you must open an already existing database file with the four-arity version of ***db_open***, specifying *AccessMode* and *DenyMode*.

If *AccessMode* is *read* the file will be opened as **readonly**, and any attempts to update the file will result in a run-time error, if it is *readwrite* the file is opened for both reading and writing. *AccessMode* is also used with the predicate ***db_begintransaction***.

If *DenyMode* is *denynone* all other users will be able to both update and read the file, if it is *denywrite*, other users will not be able to open the file in *AccessMode* = *readwrite*, but you will be able to update the file providing it was opened in *AccessMode* = *readwrite*. If ***db_open*** is called with *DenyMode* = *denyall* no other users will be able to access the file at all.

The first user that opens the file determines *DenyMode* for all subsequent attempts to open the file, and a run-time error will occur if reopened in an incompatible mode. The following table summarizes the results of opening and subsequently attempting to reopen the same file for all combinations of *DenyMode* and *AccessMode*:

2ND, 3RD, ... REOPEN

			denyAll		denyWrite		denyNone	
			R	RW	R	RW	R	RW
1st OPEN	deny-All	R	N	N	N	N	N	N
		RW	N	N	N	N	N	N
	deny-Write	R	N	N	**Y**	N	**Y**	N
		RW	N	N	N	N	**Y**	N
	deny-None	R	N	N	**Y**	**Y**	**Y**	**Y**
		RW	N	N	N	N	**Y**	**Y**

```
R : AccessMode = read
RW: AccessMode = readwrite
Y : Open by 2ND, 3RD ... allowed
N : Open by 2ND, 3RD ... not allowed
```

Transactions and Filesharing

If a database file is opened in share mode, all database predicates that access the database file in any way, must be grouped inside "transactions" this is done by surrounding the calls to the predicates with ***db_begintransaction*** and ***db_endtransaction***.

Dependent on the combination of the chosen *AccessMode* and *DenyMode* the shared file may be locked for the duration of the transaction. Again dependent on the severity of the lock, other users may not be able to either read or update the file, while your transaction takes place. This is of course necessary to avoid conflicts between reading and writing, but if file-sharing is to have any meaning, no excessive locking ought to take place. This can be avoided by keeping the transactions small (as short as possible) and only include those predicates that access the database inside the transaction.

The concept of transactions in relation to file-sharing is very important. Two often conflicting requirements, namely, database consistency and a minimum of file locking, must be fulfilled at the same time.

db_begintransaction ensures that database consistency is maintained and that an appropriate locking of the file is effectuated. Several readers can access the file at the same time, but only one process at the time is allowed to update the database. The predicate ***db_setretry*** can be called to set for how long ***db_begintransaction*** will wait to gain access to the file before returning with a run-time error. Calling ***db_begintransaction*** with *AccessMode* set to *readwrite* with a file opened with *AccessMode* set to *read* will also result in a run-time error. If ***db_begintransaction*** is called, ***db_endtransaction*** must be called before a new call to ***db_begintransaction*** for the same database, otherwise a run-time error will occur.

The following table summarizes the actions taken by ***db_begintransaction*** with different combinations of *AccessMode* and *DenyMode*:

		AccessMode	
		read	readWrite
Deny-	denyNone	WLock\Reload	RWLock\Reload

Mode	denyWrite	None	RWLock
	denyAll	None	None

Actions :

WLock : No write. Read allowed.
RWLock : No read or write allowed.
Reload : Reloading of file descriptors.

Since reloading and locking takes time, *AccessMode* and *DenyMode* should be selected with care. If no users are going to update the database, set *AccessMode* to *read* and *DenyMode* to *denywrite* for a minimal overhead.

Filesharing Predicates

In this section we discuss the file sharing predicates ***db_open***, ***db_begintransaction***, ***db_endtransaction***, ***db_updated***, ***bt_updated***, and ***db_setretry***.

db_open/4

This four-arity version of ***db_open*** opens an existing database on file in share mode.

```
db_open(Dbase, Name, AccessMode, DenyMode)              /* (i,i,i,i) */
```

After creating an external database (*in_file*) with ***db_create*** it can be opened in share mode, where *Dbase* is a *db_selector*, *Name* is the DOS-style file name, *AccessMode* is *read* or *readwrite*, and *DenyMode* is *denynone*, *denywrite*, or *denyall*.

db_begintransaction/2

```
db_begintransaction(Dbase, AccessMode)                    /* (i,i) */
```

This predicate marks the beginning of a transaction, and must be called prior to any form of access to a database opened in share mode, even if opened with *denyall*. In addition to the *db_selector* for the database, ***db_begintransaction*** must be called with *AccessMode* bound to either *read* or *readwrite*.

db_endtransaction/1

```
db_endtransaction(Dbase)                                  /* (i) */
```

db_endtransaction marks the end of a transaction and carries out the appropriate unlocking of the database. A call of ***db_endtransaction*** without a prior call to ***db_begintransaction*** for the db_selector *Dbase* will result in a run-time error.

db_updated/1

```
db_updated(Dbase)                                    /* (i) */
```

If other users have updated the database, a call of ***db_begintransaction*** will ensure that database consistency is maintained. Changes can be detected with the predicate ***db_updated***, which succeeds if called inside a transaction where changes made by other users since your last call of ***db_begintransaction***. If no changes have been made, ***db_updated*** will fail. If called outside a transaction a run-time error will occur.

bt_updated/2

```
bt_updated(Dbase,Btree_Sel)                          /* (i,i) */
```

Similar to ***db_updated/1***, but only succeeds if the named B+ tree has been updated.

db_setretry/3

```
db_setretry(Dbase,SleepPeriod,RetryCount)            /* (i,i,i) */
```

If access to a file is denied, because another process has locked the file, you can have your process wait for a period of time and then try again. The predicate ***db_setretry*** changes the default settings of *SleepPeriod*, which is the interval in centiseconds between retries, and *RetryCount*, which is the maximum number of times access will be attempted. The default settings are 100 for *RetryCount* and 10 for *SleepPeriod*.

Programming with Filesharing

Great care must be taken when using the file sharing predicates. Although they, when used properly, ensure low-level consistency in a shared database, it is the application programmers responsibility to provide the demanded high level consistency for a given application. The term "transaction" is used here for a group of file accesses, but it should be kept in mind that no back out facilities are provided, and that program interruption caused by either software or hardware failure, may cause inconsistencies in the database file.

When several processes share a database, special attention must also be paid to the domains involved. It's crucial that they are identical and use identical alignment.

To avoid unnecessary locking of the database file the transactions should be kept fairly small, in order to ensure that the file will be locked for as short a time as possible. At the same time it is important that predicates used to locate and access an item in the database are grouped inside the same transaction:

```
.....
db_begintransaction(dba,readwrite),
  key_current(dba,firstindex,Key,Ref),
  ref_term(dba,string,Ref,Term),
db_endtransaction(dba),
write(Term),
.....
```

In this example the predicates ***key_current*** and ***ref_term*** should **not** be placed inside different transactions, as the term stored under **Ref** may be deleted by another user between transactions.

If a B+ tree is updated by another user and the file buffers are reloaded, the B+ tree will be repositioned before the first element of the tree. By calling the predicate ***bt_updated*** you can detect when to reposition your B+ tree. It is still possible to list the entire index and at the same time keep the transactions small, by temporarily storing the current key in the internal database, as shown in the following program fragment. It works under the assumption that no duplicate keys exist.

```
DOMAINS
   db_selector = dba

facts
   determ currentkey(string)

PREDICATES
   list_keys(bt_selector)
   list_index(bt_selector)
   check_update(bt_selector,string)

CLAUSES
   check_update(Index,Key):-
      not(bt_updated(dba,Index)),!,
      key_next(dba,Index,_).
   check_update(Index,Key):-
      key__search(dba,Index,Key,_),!. % Will fail if current was deleted
   check_update(_,_).                 %by another user
```

```
list_keys(Index):-
   currentkey(Key),
   write(Key),nl,
   db_begintransaction(dba,read),
      check_update(Index,Key),
      key_current(dba,Index,NextKey,_),
   db_endtransaction(dba),!,
   retract(currentkey(_)),
   assert(currentkey(NextKey)),
   list_keys(Index).
list_keys(_):-
   db_endtransaction(dba).

list_index(Index):-
   db_begintransaction(dba,read),
      key_first(dba,Index,_),
      key_current(dba,Index,Key,_),
   db_endtransaction(dba),
   retractall(currentkey(_)),
   assert(currentkey(Key)),
   list_keys(Index).
list_index(_).
```

key_search is used to reposition the B+ tree at the key that was listed previously. The ***my_search*** predicate insures that the B+ tree will be correctly positioned even if ***currentkey*** was deleted by another user.

The example above also illustrates another important point. A ***db_endtransaction*** must be used after each, and before the next, call of ***db_begintransaction***. In the predicate ***list_keys*** above, the listing stops when ***key_next*** fails, indicating that all the keys have been listed. As ***db_begintransaction*** had to be called prior to accessing the database, ***db_endtransaction*** has to be called as well after accessing is completed. The second **list_keys**-clause ensures that ***db_endtransaction*** will be called when ***key_next*** fails.

Implementing High-level Locking

The examples shown so far have illustrated some of the problems involved in file sharing, and how they can be avoided.

You are allowed to do all the same operations on a shared database file as if you were the only user with access to the file. Grouping the accesses to the file inside ***db_begintransaction*** and ***db_endtransaction*** ensures that the Visual Prolog system has consistency in its descriptor tables. But on a higher level you must

yourself ensure that the various logical constraints you have on your application are conserved over a network with multiple users.

We call this high level locking or application level locking. By using the primitives ***db_begintransaction*** and ***db_endtransaction*** you have many ways of implementing a high level locking facility.

A common example of where high level locking is needed is in a database system where a user wants to change a record. When he has decided that he wants to change a record he should perform some kind of action so the application will place a lock on that record until the user has finished the changes to the record so the new record can be written back to disk, and the record unlocked.

Some suggestions for implementing an application-level lock of this type are:

- Have a special field in that record to tell whether it is locked.
- Have a special B+Tree or a chain where you store all references to all the records that are locked by users.
- Associated with a REF store a list of references to all records that are locked.

You might need to implement a kind of supervisor mechanism so a special user can unlock locked records.

This was just an example, you might want to implement locking on a higher level like tables or groups of tables, - or knowledge groups etc.

Note: If you want to delete a B+ tree in a database file opened in share mode, it is up to you to ensure by high level locking that no other users have opened this B+ Tree. In the Visual Prolog system there is no check for a B+Tree selector being no longer valid because the B+Tree has been deleted by another user.

A Complete Filesharing Example

In the following large example it will be shown how file sharing can be done more easily by implementing your own locking system. If you manage your own locks, needless file locking can be avoided, and other users won't have to wait for access to the file because it is locked.

The example is the file-sharing version of the previous ch14e04.pro example. The program lets several users create, edit, view and delete texts from a single shared file. When creating and editing a text, it will be locked until editing is complete. Other users cannot delete or edit a text while it is locked, but they will be able to view the text. Run the program and experiment with different settings for ***db_open*** and ***db_setretry***.

```
                        /* Program ch14e08.pro */

facts - indexes
   determ lockindex(bt_selector)
   determ index(bt_selector)
   determ mark(real)

DOMAINS
   my_dom = f(string)
   db_selector = dba

PREDICATES
   nondeterm repeat
   wr_err(integer)

% List texts and their status
   list
   list_texts(bt_selector,bt_selector)
   show_textname(string,bt_selector)

CLAUSES
   show_textname(Key,LockIndex):-
      key_search(dba,LockIndex,Key,_),!,
      write("\n*",Key).
   show_textname(Key,_):-
      write("\n ",Key).

   list_texts(Index,LockIndex) :-
      key_current(dba,Index,Key,_),
      show_textname(Key,LockIndex),
      key_next(dba,Index,_),!,
      list_texts(Index,LockIndex).
   list_texts(_,_).

   list:-nl,
      write("***************** TEXTS (*=Locked) ******************"),nl,
      index(Index),
      lockindex(LockIndex),
      key_first(dba,Index,_),!,
      list_texts(Index, LockIndex),nl,
      write("*****************************************************"),nl.
   list.

   repeat.
   repeat:-repeat.
```

```
   wr_err(E):-
      errormsg("PROLOG.ERR",E,Errormsg,_),
      write(Errormsg),
      readchar(_).

PREDICATES
%Logical locking of files
   lock(string,bt_selector,bt_selector)

CLAUSES
   lock(Name,Index,LockIndex):-
      not(key_search(dba,LockIndex,Name,_)),!,
      key_search(dba,Index,Name,Ref),
      key_insert(dba, LockIndex, Name, Ref).
   lock(Name,_,_):-
      db_endtransaction(dba),
      write(Name," is being updated by another user.\n Access denied"),
      fail.

PREDICATES
   ed(db_selector, bt_selector, bt_selector, string)
   ed1(db_selector, bt_selector, bt_selector, string)

CLAUSES
% The ed predicates ensure that the edition will never fail.
   ed(dba,Index,LockIndex,Name):-
      ed1(dba,Index,LockIndex,Name),!.
   ed(_,_,_,_).
```

```
   /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   * There are two choices:                                         *
   *                                                               *
   * 1) The name already exists - modify the contents of the      *
   *  file                                                         *
   * 2) The name is a new name - create a new file                 *
   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
   ed1(dba, Index,LockIndex, Name) :-
      db_begintransaction(dba,readwrite),
      key_search(dba, Index, Name, Ref),!,
      ref_term(dba, string, Ref, Str),
      lock(Name,Index,LockIndex),
      list,
      db_endtransaction(dba),nl,
      write("*****************************************************"),nl,
      write("*  EDIT               ",Name,"                      *"),nl,
      write("*****************************************************"),nl,
      write(Str),nl,
      write("< Press 'r' to replace this string ; else any key >"),nl,
      readchar(X),X='r',nl,
      write("Enter string and press <ENTER>"),nl,
      readln(Str1),nl,
      db_begintransaction(dba,readwrite),
      term_replace(dba, string, Ref, Str1),
      key_delete(dba, LockIndex, Name, Ref), %unlock
      list,
      db_endtransaction(dba).

%New file
   ed1(dba, Index,LockIndex, Name):-
      chain_insertz(dba, file_chain, string, "", Ref),
      key_insert(dba, Index, Name, Ref),
      list,
      db_endtransaction(dba),
      ed1(dba,Index,LockIndex, Name).

PREDICATES
   main(db_selector, bt_selector, bt_selector)
   interpret(char, bt_selector, bt_selector)
   check_update_view
   update_view
   get_command(char)
```

```
CLAUSES
% Loop until 'Q' is pressed
   main(dba,Index,LockIndex) :-
      check_update_view,
      get_command(Command),
      trap(interpret(Command,Index,LockIndex),E,wr_err(E)),!,
      main(dba,Index,LockIndex).
   main(_,_,_).

   check_update_view:-
      mark(T),timeout(T),!,
      db_begintransaction(dba,read),
      update_view,
      db_endtransaction(dba),
      marktime(100,Mark),
      retractall(mark(_)),
      assert(mark(Mark)).
   check_update_view.

   update_view:-nl,
      write("******* COMMANDS E:Edit V:View D:DeleteQ:Quit *******"),nl,
      write("COMMAND>"),
      db_updated(dba),!,
      list.
   update_view.

   get_command(Command):-
      readchar(C),!,
      upper_lower(Command,C),
      write(Command),nl.
   get_command(' ').
```

```
%interpret commandlineinput
   interpret(' ',_,_):-!.
   interpret('Q',_,_):-!,fail.
   interpret('E',Index,LockIndex):-!,
      write("\nFile Name: "),
      readln(Name),nl,
      ed(dba,Index,LockIndex,Name).
   interpret('V',Index,_):-
      write("\nFile Name: "),
      readln(Name),nl,
      db_begintransaction(dba,read),
      key_search(dba,Index,Name,Ref),!,
      ref_term(dba,string,Ref,Str),
      db_endtransaction(dba),
      write("*****************************************************"),nl,
      write("*      VIEW            ",Name,"                       "),nl,
      write("*****************************************************"),nl,
      write(Str),nl.

   interpret('V',_,_):-!,
      db_endtransaction(dba).
   interpret('D',Index,_):-
      write("\nDelete file: "),
      readln(Name),nl,
      db_begintransaction(dba,readwrite),
      key_search(dba,Index,Name,Ref),!,
      %    not(key_search(dba,LockIndex,Name,_)),!,
      key_delete(dba,Index,Name,Ref),
      term_delete(dba,file_chain,Ref),
      list,
      db_endtransaction(dba).
   interpret('D',_,_):-!,
   db_endtransaction(dba).
   nterpret(_,_,_):-beep.

PREDICATES
   open_dbase(bt_selector,bt_selector)

CLAUSES
   open_dbase(INDEX,LOCKINDEX):-
      existfile("share.dba"),!,
      db_open(dba, "share.dba",readwrite,denynone),
      db_begintransaction(dba,readwrite),
      bt_open(dba, "locks", LOCKINDEX),
      bt_open(dba, "ndx", INDEX),
      db_endtransaction(dba).
```

```
   open_dbase(INDEX,LOCKINDEX):-
      db_create(dba,"share.dba" , in_file),
      bt_create(dba, "locks",TEMPLOCKINDEX,20, 4),
      bt_create(dba, "ndx",TEMPINDEX , 20, 4),
      bt_close(dba, TEMPINDEX),
      bt_close(dba, TEMPLOCKINDEX),
      db_close(dba),
      open_dbase(INDEX,LOCKINDEX).

GOAL
      open_dbase(INDEX,LOCKINDEX),
      assert(index(INDEX)),
      assert(lockindex(LOCKINDEX)),
      marktime(10,Mark),
      assert(mark(Mark)),
      db_setretry(dba,5,20),
      db_begintransaction(dba,read),
      list,nl,
      db_endtransaction(dba),
      main(dba, INDEX,LOCKINDEX),
      db_begintransaction(dba,read),
      bt_close(dba, INDEX),
      bt_close(dba, LOCKINDEX),
      db_endtransaction(dba),
      db_close(dba).
```

Implementation Aspects of Visual Prolog Filesharing

Filesharing in Visual Prolog is efficient and fast, because only the necessary parts of the database file descriptors are loaded after an update by another user. As was shown earlier in this chapter it is only under certain circumstances that any reloading of file buffers and locking of files has to be done at all, and the complex internal management of the database file ensures that after an update a minimum of disk activity is needed.

The database has a serial number, which is a six-byte integer, that is incremented and written to disk each time an update occurs. The ***db_begintransaction*** predicate compares the local copy of the serial number with the one on the disk, and if they differ, the descriptors are reloaded. Locking is done in an array with room for 256 readers. When a reader wishes to access the file, an unlocked space is located in this lock array, and locked for the duration of the transaction. This allows several readers to access the file simultaneously. If ***db_begintransaction*** is called with *AccessMode* = *readwrite*, it will wait until all present readers have

unlocked their space, and then lock the entire array, allowing no other users to access the file.

Miscellaneous

Finally, we have provided a couple of small predicates that are handy in special circumstances. The predicate ***availableems*** will in DOS return the amount of available EMS. This can be used before a call to ***db_open*** or ***db_create*** in order to see if there is enough space for placing the database ***in_ems***.

```
availableems(Size)                                  /* (real)-(o) */
```

Another predicate ***str_ref*** can be used to convert a database reference number to a string so it can be inserted in a B+ Tree.

```
str_ref(Str,Ref)                    /* (string,ref)-(i,o)(o,i)(i,i) */
```

Summary

Visual Prolog's external database system adds power, speed, and efficiency to your database applications. These are the major points covered in this chapter:

1. External database terms are stored in *chains*, which you can access directly with *database reference numbers*; these reference numbers belong to the predefined ***ref*** domain.
2. Individual databases are identified by a database selector, which belongs to the standard domain ***db_selector***.
3. You can store your external database in three locations, depending on which alternative you use for the predefined ***place*** domain:
   a. *in_file* places the database in a disk file
   b. *in_memory* places it in memory
   c. and *in_ems* places it in EMS-type expanded memory (same effect as *in_memory* on non-DOS platforms
4. If you want to sort the terms in your database, you'll use B+ trees. Like databases, individual B+ trees are identified by a *B+ tree selector* of the standard domain ***bt_selector***.

5. Each entry in a B+ tree node consists of a *key* string (also known as an *index*), which identifies a record, and the database reference number, associated with that record.

6. B+ tree keys are grouped in pages, and the number of keys stored at a node is specified by the tree's *order*.

7. File sharing is done by grouping the predicates that access the database into transactions.

CHAPTER 15

System-Level Programming

Visual Prolog provides several predicates that allow you to access your PC's operating system and - to the extent that operating system allows - the hardware directly. We summarize those predicates in this chapter, first taking a look at the ones giving access to the OS, then those that perform bit-level logical and shifting operations. After that, we discuss a set of predicates that provide low-level support for manipulating the DOS BIOS, memory, and other hardware elements. We end this chapter with a couple of examples that demonstrate how to use some of these predicates within a Visual Prolog application.

Access to the Operating System

With a handful of predicates, you can access the operating system while running the Visual Prolog integrated environment, as well as build the ability to access the run-time computer's operating system right into your Visual Prolog applications. You can execute any external program with a call to ***system***, call the date and time facilities with ***date*** and ***time***, investigate the environment table with ***envsymbol***, and read the command-line arguments with ***comline***. Furthermore, you can establish the start-up directory and exe-filename of the program by calling ***syspath***, and the ***marktime***, the ***timeout*** and the ***sleep*** predicates provide time-tunneling capacity. Then there's the inevitable ***sound*** and ***beep*** predicates, and finally ***osversion*** returning the operating system version, ***diskspace*** returning the amount of free disk space, and three versions of ***storage*** used to determine memory used.

This section describes each of these predicates in detail and provides some practical examples that demonstrate how to use them.

system/1

Visual Prolog programs provide access to the OS through the ***system*** predicate, which takes the following form:

```
system("command")                                    /* (i) */
```

If the argument is an empty string (""), a new command interpreter will be run in interactive mode.

Examples

1. To copy the file B:\ORIGINAL.FIL to a file A:\NEWCOPY.FIL from within the Visual Prolog system, your program can call the predicate

   ```
   system("").
   ```

   then in activated the OS session the user can copy the file using the usual OS command,

   ```
   copy B:\ORIGINAL.FIL A:\NEWCOPY.FIL
   ```

   You could then return to Visual Prolog by typing

   ```
   exit
   ```

   after which you are back in your program again.

2. To rename the file (without going out to the OS), you could give the command

   ```
   system("ren newcopy.fil newcopy.txt"),
   ```

system/3

This extended version of the ***system*** predicate provides two extra features: one for returning the OS error level, and one for resetting the run-time system's video mode. The latter has no effect in Windows. In UNIX, this argument is used to indicate that the process has no interaction with the terminal, and hence that there's no need to clear and reset it. This is a somewhat unfortunate dual use of the same argument, but it fulfills the typical needs of users.

system/3 takes this format:

```
system(CommandString, ResetVideo, ErrorLevel)            /* (i,i,o) */
```

The error level is returned in *ErrorLevel*. This is the program return code known by the OS at the time control returns to the program issuing the **system** call. In DOS this is only available for .COM and .EXE files.

In textmode DOS, *ResetVideo* controls whether your program should reset the video hardware to the state it was in before ***system/3*** was called. *ResetVideo* = 1 resets the video mode; *ResetVideo* = 0 does not. When *ResetVideo* = 0, your program will run in the new video mode you set, even if that's a mode not

specifically supported by Visual Prolog. (For information about setting the run-time system's video mode, refer to the reference manual for the video hardware.)

In other words, if your external program MYSETMD sets the video hardware to a mode not specifically supported by Visual Prolog, and you place the following calls to ***system*** in your Visual Prolog program (running from the development environment), you can actually make your program run in that unsupported mode:

```
system("mysetmd", 0, ErrorLevel),
```

Note: The external program must be compatible with the hardware at least at the BIOS level (updating the BIOS variables *rows* and *columns* on-screen).

envsymbol/2

The ***envsymbol*** predicate searches for environment symbols in the application's environment table; the SET (OS) commands set these symbols. ***envsymbol*** takes this format:

```
envsymbol(EnvSymb, Value)                                   /* (i,o) */
```

For example, the command

```
SET SYSDIR=C:\FOOL
```

sets the symbol SYSDIR to the string C:\FOOL, and the goal

```
/*...*/
envsymbol("SYSDIR", SysDir),
/*...*/
```

searches the environment for the symbol SYSDIR, binding *SetValue* to C:\FOOL.

envsymbol will fail if the symbol does not exist.

time/4 and date

Visual Prolog has three more handy OS-related standard predicates: two forms of ***date*** and ***time***. The ***date/3*** and ***time/3*** predicates can be used in two ways, depending on whether their parameters are free or bound on entry.

With input flow, ***time*** and ***date*** will set the internal system clock to the time specified (in UNIX you need root privileges to do this). If all variables are free, the system will bind them to the internal clock's current values.

```
time(Hours, Minutes, Seconds, Hundredths)
                                          /* (i,i,i,i), (o,o,o,o) */
```

Note that the UNIX version of ***time*** doesn't return anything useful in the *Hundredths* argument.

date/3 also relies on the internal system clock and operates similarly to ***time***; it takes the following form:

```
date(Year, Month, Day)                          /* (i,i,i), (o,o,o) */
```

date/4 only has an output flow version. The fourth argument is the weekday number, but what numbering scheme is used is operating system dependent. However, it's fairly common that 0 is Sunday, 1 is Monday, etc.

```
date(Year, Month, Day, WeekDay)                       /* (o,o,o,o) */
```

Example

Program `ch15e02.pro` uses ***time*** to display the time elapsed during a listing of the default directory.

```
                        /* Program ch15e02.pro */

GOAL
      time(H1,M1,S1,_),nl,
      write("Start time is: ",H1,":",M1,":",S1),nl,
      /* This is the activity that is being timed */
      system("dir /s/b c:\\*.*"),
      time(H2,M2,S2,_),
      Time = S2-S1 + 60*(M2-M1 + 60*(H2-H1)),
      write("Elapsed time: ",Time," seconds"),nl,
      time(H3,M3,S3,_),
      write("The time now is: ",H3,":",M3,":",S3).
```

comline/1

comline reads the command-line parameters used when invoking a program; this is its format:

```
comline(CommandLine)                                    /* (o) */
```

where ***CommandLine*** is a string.

syspath/2

syspath returns the start-up directory and name of the program calling it. ***syspath*** looks as follows:

```
syspath(HomeDir,ExeName)                                    /* (o,o) */
```

The main use for ***syspath*** is to provide programs the possibility of loading files from their home directory, as well as constructing helpful command-line error messages: `<progname>: Usage: [-foul] <blah> <blah> <blah>.`

On UNIX, the start-up directory is not directly available to a program. In order to use ***syspath*** on UNIX, an initialization predicate, ***initsyspath***, must be called. In particular, this must be called before the program changes its working directory, if this becomes necessary. If ***initsyspath*** isn't called, ***syspath*** will exit with an error code of 1138.

Timing Services

Visual Prolog provides two different timing services: execution suspension, and elapsed-time testing. Some special notes apply to UNIX (see the description of ***difftime*** below).

sleep/1

sleep suspends program execution for a specified length of time. It looks like this

```
sleep(CentiSecs)                                              /* (i) */
```

where *CentiSecs* is the time (in centiseconds, i.e. 1/100ths) to sleep. The exact length of time the program will wait may vary, depending on CPU / OS activity, and you shouldn't expect greater accuracy than 20-50 milliseconds.

In UNIX, ***sleep*** uses the ***nap(S)*** system call for delays and fractions of delays less than 1 second. This call uses the kernel's callout table, and it may be necessary to increase the size of this (kernel parameter NCALL) to prevent overflows if more than 10-20 processes simultaneously use ***sleep*** with fractional delays or nap(S).

marktime/2

marktime returns a time-stamp, which may later be tested for expiration using the ***timeout*** predicate. ***marktime*** has the following format:

```
marktime(CentiSecs,Ticket)                                  /* (i,o) */
```

where *CentiSecs* is the required length of time *Ticket* should last. The *Ticket* is an implementation-defined structure holding the timing information, currently masquerading as a real number.

timeout/1

timeout tests a time-ticket returned by ***marktime*** for expiration. If it has expired, ***timeout*** succeeds, otherwise it fails. ***timeout*** looks like this:

```
timeout(Ticket)                                          /* (i) */
```

As with ***sleep***, don't expect too fine a granularity.

difftime

On UNIX, the standard predicate ***time*** doesn't provide a resolution in 100ths, so any timing calculations will be rather rough. However, the UNIX version of Visual Prolog has a standard predicate ***difftime***:

```
difftime(real,real,real)                                 /* (i,i,o) */
```

which returns the difference between the 1st and the 2nd timemark, in hundredths of seconds as a floating-point number. The first timemark should be the younger, and the second the older, i.e. the usage is

```
marktime(0,M1), lengthy_process, marktime(0,M2),
difftime(M2,M1,Diff).
```

In order for ***marktime*** and ***difftime*** to work, they must know how many clock-ticks the machine has per second. For UNIX executables, they establish this by calling the *sysconf* library function (see sysconf(S)), which is a very safe mechanism. However, for XENIX executables they have to call the library function *gethz* (see gethz(S)), which in it's current implementation simply examines a shell variable called HZ. Thus it is critical that this variable has the correct value, which, unless it's a whole new world when you read this, is 60. If *gethz* fails (e.g. because HZ doesn't exist), ***marktime*** will exit with error 1136. The same applies to ***difftime*** if either ***marktime*** has never been called, or if ***marktime*** exited due to failure in *gethz*.

The granularity of ***sleep*** and the ***marktime*** and ***timeout*** predicates is system-defined, currently being 1/60th of a second. Note that timemarks do not survive reboots. Under UNIX they're the number of machine clock-ticks since "an arbitrary point in the past" which in practice means system start-up. With 60 ticks/second, this also means that the tick count wraps around zero after approx. 2.26 years.

Example

Program `ch15e04.pro` below demonstrates ***marktime*** and ***timeout***.

```
                        /* Program ch15e04.pro */

PREDICATES
   ttimeout(real)

CLAUSES
   ttimeout(TM):-timeout(TM),!.
   ttimeout(TM):-
      write("No timeout, sleep 0.5 secs"),nl,
      sleep(50),
      ttimeout(TM).

GOAL
      marktime(400,TM),                                    % 4 secs
      ttimeout(TM),
      write("\nBINGO!\n").
```

sound/2

sound generates a sound in the PC's speaker:

```
   sound(Duration,Frequency)                              /* (i,i) */
```

where *Duration* is the time in 1/100ths of a second.

On UNIX, ***sound*** works only on the ANSI console; whether you're running on this is established by examining the *TERM* shell variable. On other terminals, ***sound*** is equivalent to ***beep***.

beep/0

```
   beep                                                /* (no arguments) */
```

In the DOS-related versions of Visual Prolog, ***beep*** is equivalent to sound(50,1000).

On UNIX, ***beep*** writes the bell character to the file used for terminal output. If the program is in terminal mode, all buffering will be bypassed.

osversion/1

osversion returns the current operating system version and looks like this:

```
   osversion(VerString)                                   /* (o) */
```

The format for *VerString* is operating system defined. For DOS, it consists of the major and minor version numbers, separated by a dot (full stop), e.g. "3.30". In UNIX, the string contains the information returned by uname(S).

diskspace/2

diskspace returns as an unsigned long the available disk space, using the following format:

```
diskspace(Where,Space)                                  /* (i,o) */
```

The space is reported in bytes.

In the DOS-related versions of Visual Prolog, Where should be a character specifying the drive letter. In the UNIX version, it should be the name of a file residing on the file system you want to query (see statfs(S)). You may use simply "/" for the root file system, or an empty or null string in which case information is retrieved for the file system holding the current working directory. The space reported will be the smaller of the actual space and the ulimit for the process (see ulimit(S)).

storage/3 and storage/11

The standard predicate ***storage*** returns information about the three run-time memory areas used by the system (stack, heap, and trail, respectively) as unsigned longs:

```
storage(StackSize,HeapSize,TrailSize)                   /* (o,o,o) */
```

The values are all in bytes.

In all versions of Visual Prolog, *TrailSize* contains the amount of memory used by the trail.

In the DOS-related versions, *StackSize* indicates how much stack space is left. In UNIX, *StackSize* is the exact opposite, namely how much stack that's been used so far.

Finally, the *HeapSize* shows how much memory is available to the process.

In UNIX this is the difference between the current break value and the maximum possible break value (see ulimit(S) and brk(S)), which again is set by the kernel parameter MAXUMEM. It does not include memory held in free-lists in the heap.

In DOS, the *HeapSize* is the unallocated physical memory between the top of the GStack and the bottom of the heap. It does not include memory held in free lists

in the heap. The ***storage*** predicate returns the size that you can be sure of having when you're loading a file or going out to the operating system.

The ***storage/11*** is extended version of ***storage/3*** predicate. The ***storage/11*** returnes more detailed information about the run-time memory areas used by the application. Description of this predicate you can find in Visual Prolog's on-line help.

storage/0

The 0-arity version of ***storage*** is primarily intended for debugging purposes. It prints in the current window an overview of the amount of memory in use by the different parts of Visual Prolog's memory management, as well as the number of backtrack points.

Bit-Level Operations

Visual Prolog provides six predicates for bit-level operations: ***bitor***, ***bitand***, ***bitnot***, ***bitxor***, ***bitleft***, and ***bitright***. These predicates have one flow variant each, operate on unsigned integers, and must be used in prefix notation.

bitnot/2

bitnot performs a bit-wise logical NOT.

```
bitnot(X, Z)                                        /* (i,o) */
```

With *X* bound to some integral value, *Z* will be bound to the bit-wise negation of *X*.

Operator	X	Z
bitnot	1	0
	0	1

bitand/3

bitand performs a bit-wise AND.

```
bitand(X, Y, Z)                                     /* (i,i,o) */
```

With *X* and *Y* bound to some integral values, *Z* will be bound to the result of bit-wise ANDing the corresponding bits of *X* and *Y*.

Operator	X	Y	Z
bitand	1	1	1
	1	0	0
	0	1	0
	0	0	0

bitor/3

bitor performs a bit-wise OR.

```
bitor(X, Y, Z)                                          /* (i,i,o) */
```

With *X* and *Y* bound to some integral values, *Z* will be bound to the result of bit-wise ORing the corresponding bits of *X* and *Y*.

Operator	X	Y	Z
bitor	1	1	1
	1	0	1
	0	1	1
	0	0	0

bitxor/3

bitxor performs a bit-wise XOR.

```
bitxor(X, Y, Z)                                         /* (i,i,o) */
```

With *X* and *Y* bound to some integral values, *Z* will be bound to the result of bit-wise XORing the corresponding bits of *X* and *Y*.

Operator	X	Y	Z
bitxor	1	1	0
	1	0	1
	0	1	1
	0	0	0

bitleft/3

bitleft performs a bit-wise left shift.

```
bitleft(X, N, Y)                                        /* (i,i,o) */
```

With *X* and *N* are bound to some integral values, *Y* is bound to the result of shifting the bit-wise representation of *X N* places to the left. The new bits will be zero-filled.

bitright/3

bitright performs a bit-wise right shift.

```
bitright(X, N, Y)                                       /* (i,i,o) */
```

With *X* and *N* are bound to some integral values, *Y* is bound to the result of shifting the bit-wise representation of *X N* places to the right. The new bits will be zero-filled.

Exercise

Write a Visual Prolog program to test the theory that

```
myxor(A, B, Result) :-
   bitnot(B, NotB), bitand(A, NotB, AandNotB),
   bitnot(A, NotA), bitand(NotA, B, NotAandB),
   bitor(AandNotB, NotAandB, Result).
   behaves like
   bitxor(A, B, Result)
```

Access to the Hardware: Low-Level Support

The DOS ROM-BIOS (Read Only Memory-Basic Input/Output System) provides an interface between programs and the operating system to perform various functions, including disk, file, printer, and screen I/O. For specific information on the ROM-BIOS, refer to the *DOS Technical Reference Manual*.

Visual Prolog provides six built-in predicates that give low-level access to the operating system, I/O ports, and hardware. These predicates are ***bios*** (2 versions), ***ptr_dword***, ***memword***, ***membyte***, and ***port_byte***.

This section describes each of these predicates in detail and provides some practical examples that demonstrate how to use them.

bios/3 and bios/4

bios gives access to the PC's low-level BIOS (Basic I/O System) routines. For information about these routines, refer to your *DOS Reference Manual*. Note that the **bios** predicates only relate to DOS. Under UNIX, it's possible to access routines in shared libraries using the *nlist* library call (see nlist(S)). However, the process is rather involved and won't be described here. See NLIST.PRO in the PROGRAMS directory for an example.

Information passes to and from the BIOS functions through the predefined compound object ***reg(...)***. The ***bios*** predicate takes the following forms:

```
bios(InterruptNo, RegistersIn, RegistersOut)              /* (i,i,o) */
bios(InterruptNo, RegistersIn, RegistersOut, OutFlags)
                                                        /* (i,i,o,o) */
```

where *RegistersIn* and *RegistersOut* are data structures defined as follows:

```
/* RegistersIn */
   reg(AXi, BXi, CXi, DXi, SIi, DIi, DSi, ESi)
     /* (i,i,i,i,i,i,i,i) */

/* RegistersOut */
   reg(AXo, BXo, CXo, DXo, SIo, DIo, DSo, ESo)
                                              /* (o,o,o,o,o,o,o,o) */
```

The ***bios*** predicates use the arguments

- AXi, BXi, CXi, DXi, SIi, DIi, DSi, and ESi to represent the PC's hardware register values passed to the BIOS.
- AXo, ... , ESo for those register values returned by the BIOS.

The flow pattern for ***bios/3*** is `(i,i,o)`; for ***bios/4*** it is `(i,i,o,o)`. When you make a call to the BIOS, each argument of *RegistersIn* must be instantiated (bound to a value), and each argument of *RegistersOut* must be free (not bound to a value).

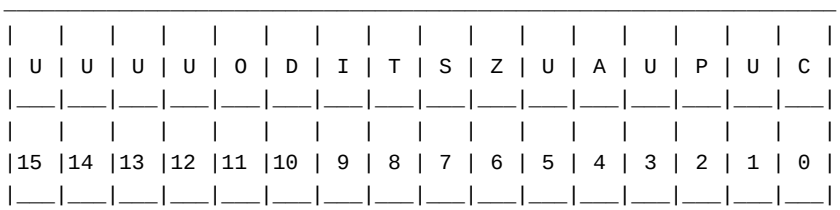
The domain for the *RegistersIn* and *RegistersOut* compound objects (*reg(AX, BX, ...)*) is the ***reg*** domain, a predefined data structure created by Visual Prolog specifically for the ***bios*** predicate. Internally, Visual Prolog defines this data structure as

```
DOMAINS
   reg = reg(integer, integer, integer, ..., integer)
```

The optional *OutFlag* argument in the ***bios/4*** predicate is packed coding for the 8086 flag register (see Figure 15.1). *OutFlag* allows you to read the contents of

the status flags after return from the interrupt. The flags are packed in an integer value as shown here:

Figure 15.1: Packing the 8086 Flag Register in an Integer



Code	Flag	Flag's Purpose
U	Undefined; indeterminate value	Not used
O	Overflow flag	Indicates arithmetic overflow
D	Direction flag	Controls left/right direction in repeated operations
I	Interrupt enable flag	Enables interrupts (when set)
T	Trap flag	Generates a trap at end of each instruction (for trace)
S	Sign flag	Indicates negative result or comparison if set
Z	Zero flag	Indicates zero result or equal comparison
A	Auxiliary flag	Need adjustment in BCD (Binary Coded Decimal) operations
P	Parity flag	Indicates even number of bits set
C	Carry flag	Indicates arithmetic carry out bit

ptr_dword

ptr_dword returns the internal address of *StringVar*, or creates the string ("the char pointer") *StringVar* based on the supplied address.

```
ptr_dword(StringVar, Seg, Off)                  /* (o,i,i), (i,o,o) */
```

When *StringVar* is bound, ***ptr_dword*** returns the internal segment and offset for the string. When *Seg* and *Off* are bound, ***ptr_dword*** binds *StringVar* to the string stored at that location. On 32-bit platforms the segment is ignored. ***ptr_dword*** has to a considerable extent been superseded by the ***cast*** function.

A string in Visual Prolog is a series of ASCII values terminated by a zero value. You can use the low-level routines in this chapter on abnormal strings (those that contain several zero bytes). However, you can't write abnormal strings out or assert them in the fact database.

membyte, memword, memdword

Visual Prolog provides three predicates for looking at (peeking) and modifying (poking) specific elements in memory. ***membyte***, ***memword*** and ***memdword*** access ***byte***, ***word*** and ***dword*** sized locations respectively. All predicates have two formats:

```
membyte(Segment, Offset, Byte)                  /* (i,i,i), (i,i,o) */
memword(Segment, Offset, Word)                  /* (i,i,i), (i,i,o) */
memdword(Segment, Offset, DWord)                /* (i,i,i), (i,i,o) */
```

and

```
membyte(StringVar, Byte)                           /* (i,i), (i,o) */
memword(StringVar, Word)                           /* (i,i), (i,o) */
memdword(StringVar, DWord)                         /* (i,i), (i,o) */
```

The *Segment* is an ***ushort***, the *Offset* is an ***unsigned***, and *Byte*, *Word* and *DWord* are ***byte***, ***word*** and ***dword*** respectively. Many of the ***bios*** calls require pointers to be passed as *Segment:Offset* pairs. ***membyte*** and ***memword*** also require pointers in this format. In real mode DOS, Memory locations are calculated as `((Segment * 16) + Offset)`.

The ***mem**** predicates have to a large extent been superseded by the ***get*entry*** and ***set*entry*** predicates for the ***binary*** domain.

port_byte/2

The ***port_byte*** predicate allows you to read or write a byte to a specific I/O port. The DOS format for ***port_byte*** is

```
port_byte(PortAddress, Byte)                    /* (i,i), (i,o) */
```

where *PortAddress* and *Byte* are defined as **unsigned**s. If you don't know what to use ***port_byte*** for, don't worry and don't think about using it. It's intended for access to (custom) hardware using ports for I/O.

Summary

These are the major points covered in this chapter:

1. Visual Prolog includes several predicates that
   a. give access to the OS
   b. perform bit-level logical and shifting operations
   c. provide low-level support for manipulating the BIOS, memory, and other hardware elements
2. These are the predicates giving access to the OS:
   a. ***system*** (execute external program)
   b. ***time*** (get or set the system clock)
   c. ***date*** (get or set the internal calendar)
   d. ***envsymbol*** (look up an environment variable)
   e. ***comline*** (read the command-line arguments)
   f. ***syspath*** (return start-up directory and name of .EXE file)
   g. ***osversion*** (returns operating system version number)
   h. ***diskspace*** (returns disk space available)
3. These are the predicates that perform bit-level operations:
   a. ***bitor*** (bit-wise OR)
   b. ***bitand*** (bit-wise AND)
   c. ***bitnot*** (bit-wise NOT)
   d. ***bitxor*** (bit-wise XOR)
   e. ***bitleft*** (bit-wise LEFT SHIFT)
   f. ***bitright*** (bit-wise RIGHT SHIFT)

4. These are the predicates that provide low-level support for various hardware elements:
    a. ***bios*** (accesses the PC's low-level BIOS routines)
    b. ***ptr_dword*** (returns the internal address of its argument or places the argument at a specified memory location)
    c. ***membyte*** (peeks or pokes a one-byte value)
    d. ***memword*** (peeks or pokes a two-byte value)
    e. ***memdword*** (peeks or pokes a four-byte value)
    f. ***port_byte*** (reads or writes a byte to a specific I/O port)

CHAPTER 16

Example Prolog Programs

In this final tutorial chapter, we present some small example programs intended to stimulate your own ideas and to further illustrate the topics covered in the earlier tutorial chapters. Nearly all of the examples offer plenty of room for expansion; your own ideas can grow into full-blown programs using one of these programs as a starting point.

Building a Small Expert System

In this first example, we show you how to construct a small expert system that figures out which of seven animals (if any) the system's user has in mind. The expert system will figure out the animal by asking questions then making deductions from the replies given. This example demonstrates backtracking – using facts – and how to use ***not*** effectively.

A typical user dialogue with this expert system might be:

```
has it hair?
yes
does it eat meat?
yes
has it a fawn color?
yes
has it dark spots?
yes

Your animal may be a cheetah!
```

Visual Prolog's ability to check facts and rules will provide your program with the reasoning capabilities germane to an expert system. The first step is to provide the knowledge with which the system can reason; this is known as the *inference engine* and is shown in ch16e01.pro.

```
                        /* Program ch16e01.pro */

global facts
   xpositive(symbol,symbol)
   xnegative(symbol,symbol)

predicates
   nondeterm animal_is(symbol)
   nondeterm it_is(symbol)
   ask(symbol,symbol,symbol)
   remember(symbol,symbol,symbol)
   positive(symbol,symbol)
   negative(symbol,symbol)
   clear_facts
   run

clauses
   animal_is(cheetah):-
      it_is(mammal),
      it_is(carnivore),
      positive(has,tawny_color),
      positive(has,dark_spots).

   animal_is(tiger):-
      it_is(mammal),
      it_is(carnivore),
      positive(has, tawny_color),
      positive(has, black_stripes).

   animal_is(giraffe):-
      it_is(ungulate),
      positive(has,long_neck),
      positive(has,long_legs),
      positive(has, dark_spots).

   animal_is(zebra):-
      it_is(ungulate),
      positive(has,black_stripes).

   animal_is(ostrich):-
      it_is(bird),
      negative(does,fly),
      positive(has,long_neck),
      positive(has,long_legs),
      positive(has, black_and_white_color).
```

```
animal_is(penguin):-
   it_is(bird),
   negative(does,fly),
   positive(does,swim),
   positive(has,black_and_white_color).

animal_is(albatross):-
   it_is(bird),positive(does,fly_well).

it_is(mammal):-
   positive(has,hair).
it_is(mammal):-
   positive(does,give_milk).

it_is(bird):-
   positive(has,feathers).
it_is(bird):-
   positive(does,fly),
   positive(does,lay_eggs).

it_is(carnivore):-
   positive(does,eat_meat).

it_is(carnivore):-
   positive(has,pointed_teeth),
   positive(has, claws),
   positive(has,forward_eyes).

it_is(ungulate):-
   it_is(mammal),
   positive(has,hooves).

it_is(ungulate):-
   it_is(mammal),
   positive(does,chew_cud).

positive(X,Y):-
   xpositive(X,Y),!.
positive(X,Y):-
   not(xnegative(X,Y)),
   ask(X,Y,yes).

negative(X,Y):-
   xnegative(X,Y),!.
negative(X,Y):-
   not(xpositive(X,Y)),
   ask(X,Y,no).
```

```
   ask(X,Y,yes):-
      !,
      write(X," it ",Y,'\n'),
      readln(Reply),nl,
      frontchar(Reply,'y',_),
      remember(X,Y,yes).
   ask(X,Y,no):-
      !,
      write(X," it ",Y,'\n'),
      readln(Reply),nl,
      frontchar(Reply,'n',_),
      remember(X,Y,no).

   remember(X,Y,yes):-
      assertz(xpositive(X,Y)).
   remember(X,Y,no):-
      assertz(xnegative(X,Y)).

   clear_facts:-
      write("\n\nPlease press the space bar to exit\n"),
      retractall(_,dbasedom),readchar(_).

   run:-
      animal_is(X),!,
      write("\nYour animal may be a (an) ",X),
      nl,nl,clear_facts.
   run :-
      write("\nUnable to determine what"),
      write("your animal is.\n\n"),
      clear_facts.

goal
   run.
```

Each animal is described by a number of attributes that it has (or has not). Those questions that the user is to reply to are the `positive(X,Y)` and `negative(X,Y)` ones. The system, therefore, might ask something like this:

```
   Does it have hair?
```

Having received a reply to such a question, you want to be able to add the answer to the database, so the system will be able to use the previously gathered information when reasoning.

For simplicity, this example program will only consider positive and negative replies, so it uses a fact database containing two predicates:

```
facts
   xpositive(symbol, symbol)
   xnegative(symbol, symbol)
```

The fact that the animal doesn't have hair is represented by

```
xnegative(has,hair).
```

The rules of ***positive*** and ***negative*** then check to see if the answer is already known, before asking the user.

```
positive(X,Y) :-
   xpositive(X,Y), !.
positive(X,Y) :-
   not(xnegative(X,Y)),
   ask(X,Y,yes).

negative(X,Y) :-
   xnegative(X,Y), !.
negative(X,Y) :-
   not(xpositive(X,Y)),
   ask(X,Y,no).
```

Notice that the second rule for both ***positive*** and ***negative*** ensures that a contradiction won't arise before asking the user.

The ***ask*** predicate asks the questions and organizes the remembered replies. If a reply begins with the letter *y*, the system assumes the answer is `Yes`; if it begins with *n*, the answer is `No`.

```
            /* Asking Questions and Remembering Answers */

ask(X, Y, yes) :- !, write(X, " it ", Y, '\n'),
   readln(Reply),
   frontchar(Reply, 'y', _),
   remember(X, Y, yes).

ask(X, Y, no)  :-  !, write(X, " it ", Y, '\n'),
   readln(Reply),
   frontchar(Reply, 'n', _),
   remember(X, Y, no).

remember(X, Y, yes) :- assertz(xpositive(X, Y)).
remember(X, Y, no)  :- assertz(xnegative(X, Y)).
```

```
                        /* Clearing Out Old Facts */

   clear_facts :- write("\n\nPlease press the space bar to exit\n"),
      retractall(_,dbasedom), readchar(_).
```

For practice, type in the preceding inference engine and knowledge clauses. Add appropriate declarations to make a complete program, and then try out the result. The completed animal expert system is provided as ch16e01.pro.

An example expert systems shell (GENI.PRO) is also provided with Visual Prolog in the PROGRAMS directory; this shell is based on the same techniques introduced in this example, with the added feature that it allows you to dynamically change the rules.

Prototyping: A Simple Routing Problem

Suppose you want to construct a computer system to help decide the best route between two U.S. cities. You could first use Visual Prolog to build a miniature version of the system (see 2), since it will then become easier to investigate and explore different ways of solving the problems involved. You will use the final system to investigate questions such as:

Is there a direct road from one particular town to another?

Which towns are situated less than ten miles from a particular town?

The following program is a classic example of using backtracking and recursion to solve route planning.

```
                        /* Program ch16e02.pro */

DOMAINS
   town     = symbol
   distance = integer

PREDICATES
   nondeterm road(town,town,distance)
   nondeterm route(town,town,distance)

CLAUSES
   road(tampa,houston,200).
   road(gordon,tampa,300).
   road(houston,gordon,100).
   road(houston,kansas_city,120).
   road(gordon,kansas_city,130).
```

```
route(Town1,Town2,Distance):-
   road(Town1,Town2,Distance).
route(Town1,Town2,Distance):-
   road(Town1,X,Dist1),
   route(X,Town2,Dist2),
   Distance=Dist1+Dist2,
   !.
```

Figure 16.1 shows a simplified map for the prototype.

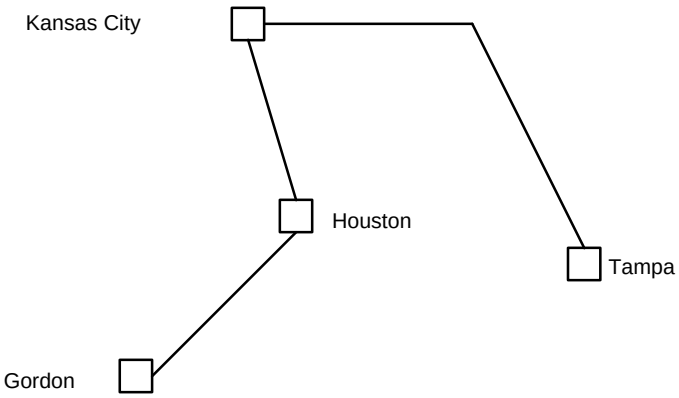


Figure 16.1: Prototype Map

Each clause for the ***road*** predicate is a fact that describes a road of a certain length (in miles) that goes from one town to another.

The ***route*** clauses indicate that it is possible to make a route from one town to another over several stretches of road. Following the route, the driver travels a distance given by the third parameter, ***distance***.

The ***route*** predicate is defined recursively; a route can simply consist of one single stretch of road, as in the first clause. In this case, the total distance is merely the length of the road.

You can also construct a route from *Town1* to *Town2* by driving first from *Town1* to *X*, then following some other route from *X* to *Town2*. The total distance is the sum of the distance from *Town1* to *X* and the distance from *X* to *Town2*, as shown in the second clause for ***route***.

Specify the **goal** section like this and run the program ch16e02.pro with the **Test Goal**:

```
goal
   route(tampa, kansas_city, X).
```

Can the program handle all possible combinations of starting point and destination? If not, can you modify the program to avoid any omissions?

The next example will give you ideas about how to get this routing program to make a list of towns visited enroute. Making such a list prevents Visual Prolog from choosing a route that involves visiting the same town twice, thereby avoiding going around in circles, and ensures that the program doesn't go into an infinite loop. When you've solved problems of this type, you can enlarge the program by adding more cities and roads.

Adventures in a Dangerous Cave

You're an adventurer, and you've heard that there is a vast gold treasure hidden inside a cave. Many people before you have tried to find it, but to no avail. The cave is a labyrinth of galleries connecting different rooms in which there are dangerous beings, like monsters and robbers. In your favor is the fact that the treasure is all in one room. Which route should you follow to get to the treasure and escape unhurt with it? Consider the following map of the cave:

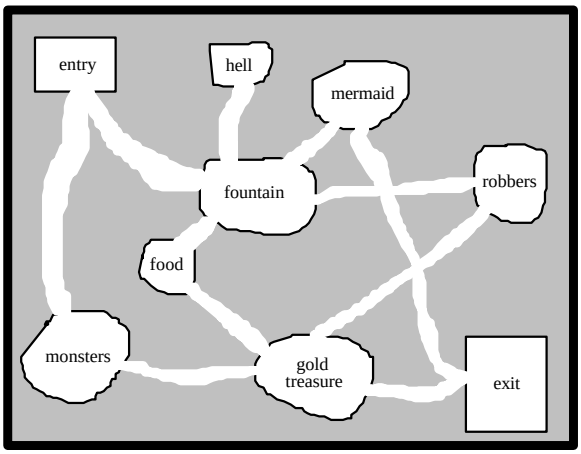


Figure 16.2: The Labyrinth of Galleries

You can construct a Visual Prolog representation of the map to help you find a safe route. Each gallery is described by a fact. The predicates ***go*** and ***route*** give rules. Give the program the goal

```
go(entry, exit).
```

The answer will consist of a list of the rooms you should visit to capture the treasure and return safely with it.

An important design feature of this program is that the rooms already visited are collected in a catalog. This happens thanks to the ***route*** predicate, which is defined recursively. If you're standing in the exit room, the third parameter in the ***route*** predicate will be a list of the rooms you've already visited. If the *gold_treasure* room is a member of this list, you'll have achieved your aim. Otherwise, the list of rooms visited is enlarged with *Nextroom*, provided *Nextroom* is not one of the dangerous rooms and has not been visited before.

```
                        /* Program ch16e03.pro */

DOMAINS
   room     =  symbol
   roomlist = room*

PREDICATES
   nondeterm gallery(room,room)
      % There is a gallery between two rooms
      % Necessary because it does not matter
      % which direction you go along a gallery
   nondeterm neighborroom(room,room)
   avoid(roomlist)
   nondeterm go(room,room)
   nondeterm route(room,room,roomlist)
      % This is the route to be followed.
      % roomlist consists of a list of rooms already visited.
   nondeterm member(room,roomlist)

CLAUSES
   gallery(entry,monsters).          gallery(entry,fountain).
   gallery(fountain,hell).           gallery(fountain,food).
   gallery(exit,gold_treasure).      gallery(fountain,mermaid).
   gallery(robbers,gold_treasure).   gallery(fountain,robbers).
   gallery(food,gold_treasure).      gallery(mermaid,exit).
   gallery(monsters,gold_treasure).  gallery(gold_treasure,exit).

   neighborroom(X,Y):-gallery(X,Y).
   neighborroom(X,Y):-gallery(Y,X).
```

```
avoid([monsters,robbers]).

go(Here,There):-route(Here,There,[Here]).
go(_,_).

route(Room,Room,VisitedRooms):-
   member(gold_treasure,VisitedRooms),
   write(VisitedRooms),nl.
route(Room,Way_out,VisitedRooms):-
   neighborroom(Room,Nextroom),
   avoid(DangerousRooms),
   not(member(NextRoom,DangerousRooms)),
   not(member(NextRoom,VisitedRooms)),
   route(NextRoom,Way_out,[NextRoom|VisitedRooms]).

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

After verifying (using the **Test Goal**) that the program does find a solution to the goal

```
go(entry, exit).
```

you might want to try adding some more galleries, for example,

```
gallery(mermaid, gold_treasure).
```

Or you can add some additional nasty things to avoid.

Even though – once you've made these additions – there is more than one possible solution to the problem, your program will only come up with one solution. To obtain all the solutions, you must make Visual Prolog backtrack as soon as it has found one solution. You can do this by adding the ***fail*** predicate to the first rule for ***route***:

```
route(Room, Room, VisitedRooms) :-
   member(gold_treasure, VisitedRooms),
   write(VisitedRooms), nl, fail.
```

To get a neater output, you could use a list-writing predicate, ***write_a_list***, to write the list of names without the containing square brackets (`[` and `]`) or the separating commas. However, the rooms you've visited are collected in the *VisitedRooms* list in reverse order (*exit* first and *entry* last). Therefore, you need to reverse the list or make the list-writing predicate write the list in reverse.

Hardware Simulation

Every logical circuit can be described with a Visual Prolog predicate, where the predicate indicates the relationship between the signals on the input and output terminals of the circuit. The fundamental circuits are described by giving a table of corresponding truth values (see the ***and_***, ***or_***, and ***not_*** predicates in Program `ch16e04.pro`).

Fundamental circuits can be described by indicating the relationships between the internal connections, as well as the terminals. To see how this works, construct an exclusive OR circuit from AND, OR, and NOT circuits, and then check its operation with a Visual Prolog program. The circuit is shown in Figure 16.3.

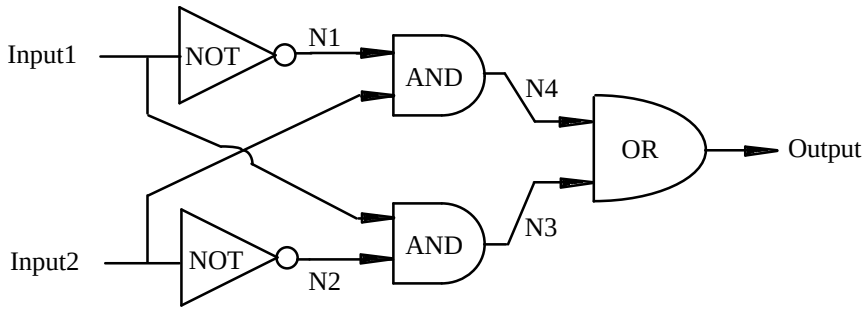


Figure 16.3: Fundamental XOR Circuit

In Program `ch16e04.pro`, this network is described by the ***xor*** predicate.

```
/* Program ch16e04.pro */

DOMAINS
   d = integer

PREDICATES
   nondeterm not_(D,D)
   and_(D,D,D)
   or_(D,D,D)
   nondeterm xor(D,D,D)
```

```
CLAUSES
   not_(1,0).      not_(0,1).
   and_(0,0,0).    and_(0,1,0).
   and_(1,0,0).    and_(1,1,1).
   or_(0,0,0).     or_(0,1,1).
   or_(1,0,1).     or_(1,1,1).

   xor(Input1,Input2,Output):-
      not_(Input1,N1),
      not_(Input2,N2),
      and_(Input1,N2,N3),
      and_(Input2,N1,N4),
      or_(N3,N4,Output).

goal
   xor(Input1, Input2, Output).
```

Run this program with the Test Goal and it yields the following result:

```
Input1=1, Input2=1, Output=0
Input1=1, Input2=0, Output=1
Input1=0, Input2=1, Output=1
Input1=0, Input2=0, Output=0
4 Solutions
```

Interpreting this result as a truth table, you can see that the circuit does indeed perform as expected.

Towers of Hanoi

The solution to the Towers of Hanoi puzzle is a classic example of recursion. The ancient puzzle of the Towers Of Hanoi consists of a number of wooden disks mounted on three poles, which are in turn attached to a baseboard. The disks each have different diameters and a hole in the middle large enough for the poles to pass through. In the beginning, all the disks are on the left pole as shown in Figure 16.4.

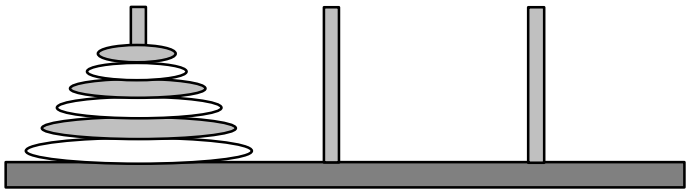


Figure 16.4: The Towers of Hanoi

The object of the puzzle is to move all the disks over to the right pole, one at a time, so that they end up in the original order on that pole. You can use the middle pole as a temporary resting place for disks, but at no time is a larger disk to be on top of a smaller one. It's easy to solve the Towers of Hanoi with two or three disks, but the process becomes more difficult with four or more disks.

A simple strategy for solving the puzzle is as follows:

- You can move a single disk directly.
- You can move N disks in three general steps:
- Move N-1 disks to the middle pole.
- Move the last (Nth) disk directly over to the right pole.
- Move the N-1 disks from the middle pole to the right pole.

The Visual Prolog program to solve the Towers Of Hanoi puzzle uses three predicates:

- **hanoi**, with one parameter that indicates the total number of disks you are working with.
- **move**, which describes the moving of N disks from one pole to another – using the remaining pole as a temporary resting place for disks.
- **inform**, which displays what has happened to a particular disk.

```
                    /* Program ch16e05.pro */

DOMAINS
   loc =right;middle;left

PREDICATES
   hanoi(integer)
   move(integer,loc,loc,loc)
   inform(loc,loc)
```

```
CLAUSES
   hanoi(N):-
      move(N,left,middle,right).

   move(1,A,_,C):-
      inform(A,C),!.

   move(N,A,B,C):-
      N1=N-1, move(N1,A,C,B),
      inform(A,C),move(N1,B,A,C).

   inform(Loc1, Loc2):-nl,
      write("Move a disk from ", Loc1, " to ", Loc2).
```

To solve the Towers of Hanoi with three disks, give the

```
goal
   hanoi(3).
```

Run the **Test Goal**. The output is:

```
Move a disk from left to right
Move a disk from left to middle
Move a disk from right to middle
Move a disk from left to right
Move a disk from middle to left
Move a disk from middle to right
Move a disk from left to right

yes
```

Dividing Words into Syllables

Using a very simple algorithm that involves looking at the sequence of vowels and consonants a word contains, a computer program can decide how to divide words into syllables. For instance, consider the two sequences:

1. vowel consonant vowel

   In this case, the word is divided after the first vowel. For example, this rule can be applied to the following words:

   ```
   ruler    >   ru-ler
   prolog >   pro-log
   ```

2. vowel consonant consonant vowel

In this case, the word is divided between the two consonants. For example,

```
number  > num-ber
panter    > pan-ter
console  > con-sole
```

These two rules work well for most words but fail with words like *handbook* and *hungry*, which conform to neither pattern. To divide such words, your program would have to use a library containing all words.

Write a Visual Prolog program to divide a word into syllables. The program will first ask for a word to be typed in, and then attempt to split it into syllables using the two rules just given. As we've mentioned, this will not always produce correct results.

First, the program should split the word up into a list of characters. You therefore need the following domain declarations:

```
DOMAINS
   letter = symbol
   word= letter*
```

You must have a predicate that determines whether the letter is a vowel or a consonant. However, the two rules given can also work with the vocals (the usual vowels – *a*, *e*, *i*, *o*, and *u* – plus the letter *y*). The letter *y* sounds like (and is considered to be) a vowel in many words, for example, *hyphen*, *pity*, *myrrh*, *syzygy*, and *martyr*. To account for the vocals, you have the clauses

```
vocal(a).      vocal(e).      vocal(i).
vocal(o).      vocal(u).      vocal(y).
```

for the predicate ***vocal***. A consonant is defined as a letter that is not a vocal:

```
consonant(L) :- not(vocal(L)).
```

You also need two more predicates. First, you need the ***append*** predicate.

```
append(word, word, word)
```

Second, you need a predicate to convert a string to a list of the characters in that string:

```
string_word(string, word)
```

This predicate will use the standard predicate ***frontstr*** (described in chapter 13), as well as the standard predicates ***free*** and ***bound*** (where ***free(X)*** succeeds if *X* is a free variable at the time of calling, and ***bound(Y)*** succeeds if *Y* is bound), to control which clause to activate, dependent on the flow-pattern.

Now you're ready to attack the main problem: defining the predicate ***divide*** that separates a word into syllables. ***divide*** has four parameters and is defined recursively. The first and second parameters contain, respectively, the *Start* and the *Remainder* of a given word during the recursion. The last two arguments return, respectively, the first and the last part of the word after the word has been divided into syllables.

As an example, the first rule for ***divide*** is:

```
divide(Start, [T1, T2, T3|Rest], D, [T2, T3|Rest]) :-
   vocal(T1), consonant(T2), vocal(T3),
   append(Start, [T1], D).
```

where *Start* is a list of the first group of characters in the word to be divided. The next three characters in the word are represented by *T1*, *T2*, and *T3*, while *Rest* represents the remaining characters in the word. In list *D*, the characters *T2* and *T3*, and the list *Rest* represent the complete sequence of letters in the word. The word is divided into syllables at the end of those letters contained in *D*.

This rule can be satisfied by the call:

```
divide([p, r], [o, l, o, g], P1, P2)
```

To see how, insert the appropriate letters into the clause:

```
divide([p, r], [o, l, o|[g]], [p, r, o], [l, o | [g]]) :-
   vocal(o), consonant(l), vocal(o),
   append([p, r], [o], [p, r, o]).
```

The ***append*** predicate concatenates the first vocal to the start of the word. *P1* becomes bound to *[p, r, o]*, and *P2* is bound to *[l, o, g]*.

The second rule for ***divide*** is shown in the complete program, 6.

```
                        /* Program ch16e06.pro */

DOMAINS
   letter = char
   word_ = letter*

PREDICATES
   nondeterm divide(word_,word_,word_,word_)
   vocal(letter)
   consonant(letter)
   nondeterm string_word(string,word_)
   append(word_,word_,word_)
   nondeterm repeat
```

```
CLAUSES
   divide(Start,[T1,T2,T3|Rest],D1,[T2,T3|Rest]):-
      vocal(T1),consonant(T2),vocal(T3),
      append(Start,[T1],D1).
   divide(Start,[T1,T2,T3,T4|Rest],D1,[T3,T4|Rest]):-
      vocal(T1),consonant(T2),consonant(T3),vocal(T4),
      append(Start,[T1,T2],D1).
   divide(Start,[T1|Rest],D1,D2):-
      append(Start,[T1],S),
      divide(S,Rest,D1,D2).

   vocal('a'). vocal('e'). vocal('i').
   vocal('o'). vocal('u'). vocal('y').

   consonant(B):-
      not(vocal(B)),B <= 'z','a' <= B.

   string_word("",[]):-!.
   string_word(Str,[H|T]):-
      bound(Str),frontchar(Str,H,S),string_word(S,T).
   string_word(Str,[H|T]):-
      free(Str),bound(H),string_word(S,T),frontchar(Str,H,S).

   append([],L,L):-!.
   append([X|L1],L2,[X|L3]):-
      append(L1,L2,L3).

   repeat.
   repeat:-repeat.

GOAL
      repeat,
        write("Write a multi-syllable word: "),
        readln(S),nl,
        string_word(S,Word),
        divide([],Word,Part1,Part2),
        string_word(Syllable1,Part1),
        string_word(Syllable2,Part2),
        write("Division: ",Syllable1,"-",Syllable2),nl,
      fail.
```

The N Queens Problem

In the N Queens problem, the object is to place *N* queens on a chessboard in such a way that no two queens can take each other. Accordingly, no two queens can be placed on the same row, column, or diagonal.

To solve the problem, you'll number the rows and columns of the chessboard from 1 to *N*. To number the diagonals, you divide them into two types, so that a diagonal is uniquely specified by a type and a number calculated from its row and column numbers:

```
Diagonal = N + Column - Row (Type 1)
Diagonal = Row + Column - 1 (Type 2)
```

When you view the chessboard with row 1 at the top and column 1 on the left side, Type 1 diagonals resemble the backslash (\) character in shape, and Type 2 diagonals resemble the shape of slash (/). Figure 16.5 shows the numbering of Type 2 diagonals on a 4x4 board.

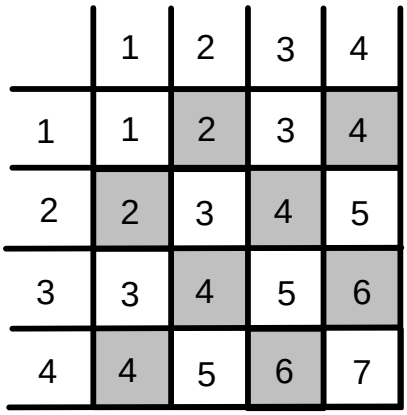


Figure 16.5: The N Queens Chessboard

To solve the N Queens Problem with a Visual Prolog program, you must record which rows, columns, and diagonals are unoccupied, and also make a note of where the queens are placed.

A queen's position is described with a row number and a column number as in the domain declaration:

```
queen = q(integer, integer)
```

This declaration represents the position of one queen. To describe more positions, you can use a list:

```
queens = queen*
```

Likewise, you need several numerical lists indicating the rows, columns, and diagonals not occupied by a queen. These lists are described by:

```
freelist = integer*
```

You will treat the chessboard as a single object with the following domain declaration:

```
board = board(queens, freelist, freelist, freelist, freelist)
```

The four *freelist*s represent the free rows, columns, and diagonals of Type 1 and Type 2, respectively.

To see how this is going to work, let ***board*** represent a 4 * 4 chessboard in two situations: (1) without queens, and (2) with one queen at the top left corner.

1. ***board*** without queens

```
board([], [1,2,3,4], [1,2,3,4], [1,2,3,4,5,6,7], [1,2,3,4,5,6,7])
```

2. ***board*** with one queen

```
board([q(1,1)], [2,3,4], [2,3,4], [1,2,3,5,6,7], [2,3,4,5,6,7])
```

You can now solve the problem by describing the relationship between an empty board and a board with *N* queens. You define the predicate

```
placeN(integer, board, board)
```

with the two clauses following. Queens are placed one at a time until every row and column is occupied. You can see this in the first clause, where the two lists of *freerows* and *freecols* are empty:

```
placeN(_, board(D, [], [], X, Y), board(D, [], [], X, Y)) :- !.

placeN(N, Board1, Result) :-
   place_a_queen(N, Board1, Board2),
   placeN(N, Board2, Result).
```

In the second clause, the predicate ***place_a_queen*** gives the connection between *Board1* and *Board2*. (*Board2* has one more queen than *Board1*). Use this predicate declaration:

```
place_a_queen(integer, board, board)
```

The core of the N Queens Problem lies in the description of how to add extra queens until they have all been successfully placed, starting with an empty board. To solve this problem, add the new queen to the list of those already placed:

```
[q(R, C)|Queens]
```

Among the remaining free rows, *Rows*, you need to find a row *R* where you can place the next queen. At the same time, you must remove *R* from the list of free rows, resulting in a new list of free rows, *NewR*. This is formulated as:

```
findandremove(R, Rows, NewR)
```

Correspondingly, you must find and remove a vacant column *C*. From *R* and *C*, you can calculate the numbers of the occupied diagonals. Then you can determine if *D1* and *D2* are among the vacant diagonals.

This is the ***place_a_queen*** clause:

```
place_a_queen(N, board(Queens, Rows, Columns, Diag1, Diag2),
   board([q(R, C)|Queens], NewR, NewS, NewD1, NewD2)) :-
      findandremove(R, Rows, NewR),
      findandremove(C, Columns, NewC),
      D1=N+S-R, findandremove(D1, Diag1, NewD1),
      D2=R+S-1, findandremove(D2, Diag2, NewD2).
```

Program `ch16e07.pro` is the complete program. It contains a number of smaller additions to define ***nqueens***, so you only need to give a goal section like:

```
nqueens(5).
```

to obtain a possible solution (in this case, for placing five queens on a 5 * 5 board).

```
                        /* Program ch16e07.pro */

domains
   queen    = q(integer, integer)
   queens   = queen*
   freelist = integer*
   board    = board(queens, freelist, freelist, freelist, freelist)
```

```
predicates
   nondeterm placeN(integer, board, board)
   nondeterm place_a_queen(integer, board, board)
   nondeterm nqueens(integer)
   nondeterm makelist(integer, freelist)
   nondeterm findandremove(integer, freelist, freelist)
   nextrow(integer, freelist, freelist)

clauses
   nqueens(N):-
      makelist(N,L),Diagonal=N*2-1,makelist(Diagonal,LL),
      placeN(N,board([],L,L,LL,LL),Final), write(Final).

   placeN(_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-!.
   placeN(N,Board1,Result):-
      place_a_queen(N,Board1,Board2),
      placeN(N,Board2,Result).

   place_a_queen(N,board(Queens,Rows,Columns,Diag1,Diag2),
   board([q(R,C)|Queens],NewR,NewC,NewD1,NewD2)):-
      nextrow(R,Rows,NewR),
      findandremove(C,Columns,NewC),
      D1=N+C-R,findandremove(D1,Diag1,NewD1),
      D2=R+C-1,findandremove(D2,Diag2,NewD2).

   findandremove(X,[X|Rest],Rest).
   findandremove(X,[Y|Rest],[Y|Tail]):-
      findandremove(X,Rest,Tail).

   makelist(1,[1]).
   makelist(N,[N|Rest]) :-
      N1=N-1,makelist(N1,Rest).
      nextrow(Row,[Row|Rest],Rest).

goal
   nqueens(5),
   nl,readchar(_).
```

PART 4

Programmer's Guide

CHAPTER 17

Elements of the Language

In this chapter, we summarize the elements of the Visual Prolog compiler and language. We discuss some fundamental elements of the language: names, program sections, compiler directives, and memory management.

We give an introduction to handling modules in Visual Prolog, and how a program can be split up into several modules, which you can then compile separately and link together.

We've written this chapter for programmers who've already worked some with Visual Prolog. To get the most benefit out of this chapter, you should be familiar with the material in the first chapters of the *Visual Prolog Language.*

Names

In Prolog, *names* are used to denote symbolic constants, domains, predicates, and variables. A name consists of a letter, or an underscore character, followed by any combination of zero or more letters, digits, and underscores. Two important restrictions are imposed on names:

- Names of symbolic constants must start with a lower-case letter.
- Names of variables must start with an upper-case letter or an underscore.

Except for these restrictions, you can use upper-case and lower-case letters in your programs as you please. For instance, you could make a name more readable by using mixed upper-case and lower-case, as in the variable

```
MyLongestVariableNameSoFar
```

or by using underscores, as in

```
pair_who_might_make_a_happy_couple(henry_viii, ann_boleyn)
```

The Visual Prolog compiler does not make a distinction between upper and lower case letters, except for the first letter. This means that the two variables:

```
SourceCode
```

and

```
SOURCECODE
```

are the same.

Keywords

The following are reserved words; you must not employ them as user-defined names:

```
abstract        domains         if              object
align           elsedef         ifdef           or
as              endclass        ifndef          procedure
and             enddef          implement       protected
class           erroneous       include         predicates
clauses         facts           language        reference
constants       failure         multi           single
database        global          nocopy          static
determ          goal            nondeterm       struct
                                                this
```

Specially-Handled Predicates

The following predicates are handled specially by the compiler.

```
assert              chain_terms     free            retractall
asserta             consult         msgrecv         save
assertz             db_btrees       msgsend         term_bin
bound               db_chains       not             term_replace
chain_inserta       fail            readterm        term_str
chain_insertafter   findall         ref_term        trap
chain_insertz       format          retract         write
                                                    writef
```

Program Sections

A Visual Prolog program consists of several modules and each module consists of several program sections. Each program section is identified by a keyword, as shown in this table.

Table 17.1: Contents of Program Sections

Section	Contents
compiler options	Options are given at the top of a module.
constants section	Zero or more symbolic constants.
domains section	Zero or more domain declarations.
facts section	Zero or more declarations of fact database predicates.
predicates section	Zero or more predicate declarations.
goal section	The program goal.
clauses section	Zero or more clauses.
class section	Zero or more declarations of public (and protected) predicates, facts and domains
implement section	Zero or more declarations of private predicates, facts and domains. Zero or more clauses implementing public and private predicates (and initializing facts).
abstract class section	Zero or more declarations of public predicates and domains. Should not have an implementation.

To generate an executable stand-alone application, your program must contain a **goal**. Usually, a program requires at least a **predicates** and a **clauses** section. For most programs, a **domains** section is needed to declare lists, compound structures and your own names for the basic domains.

For modular programming, you can prefix the keywords **domains, predicates** and **facts** with the keyword **global,** indicating that the subsequent declarations have global scope of visibility and the declared names can be used in all modules that include declarations of these global sections. (Modular programming is discussed on page 252).

A program can contain several **domains, predicates, facts** or **clauses** sections and several declarations and implementation of classes, provided you observe the following restrictions:

- Compiler options must precede all other sections.

- Constants, domains (including implicitly defined domains defined by names of classes and facts section) and predicates should be defined before you use them. However, within the **domains** section you can refer to domains that are declared at a later point.
- All predicates with the same name, but with different domains or arity must be declared in one **predicates** section.
- You cannot give the same name to predicates declared in **predicates** and **facts** sections.
- All clauses that describe predicates with the same name and arity (but with different domains) must occur in sequence (one after the other).
- **Facts** sections can be named, but a given name can only appear once. Because the default name is **dbasedom,** there can only be one unnamed database section and it must be global. Initializing clauses for **global** facts can be done only after the **goal** section in the main module.
- One and only one goal must be specified in a program. However, the goal can appear anywhere.
- If a compiled file (project module) does not contain a goal section, then the compiler has to be informed that this file is a project module. This can be done with the `–r<-ProjectName>` command line compiler option (the VDE makes this automatically) or with the `project "ProjectName"` compiler directive.
- Beginning with version 5.2, Visual Prolog provides enhanced handling of global declarations:
- Global and local declarations can be given in any order. (Elder versions required that all global declarations must come before any local declarations.)
- The main project module (with the **goal**) must contain declarations of all global domains, global facts sections, and **abstract** classes that are declared in all project submodules.
- Any other project module may contain declarations of only those global domains, global facts sections, and classes, which are used in this module.
- If any global declaration (class declaration) is changed, all modules including this declaration must be recompiled.

The Domains Section

A **domains** section contains domain declarations. Seven generic formats are used:

```
my_name = d                                    % standard domain synonyms

my_list = elementDom*                                    % list domain

my_CompDom = f_1(d_11,d_12,...,d_1n);               % multi-alternative
                                                      % compound domain
             f_m(d_m1,d_m2,...,d_mn)

my_SingleAltCompDom = struct f_single(d1,d2,...,dn)% single-alternative
                                                      % compound domain

predefdom = name1;name2;...;nameN      % specially handled domains like
                                              % db_selector and file

my_PredicateDom = determSpec retDom (args) - flow langSpec
                                                       % predicate domain

my_ObjPredicateDom = object determSpec retDom (args) - flow langSpec
                                                % object predicate domain
```

Shortening Domain Declarations

The left side of a domain declaration (except for specially handled predefined domains **file** and **db_selector**) can consist of a list of names, like this:

```
mydom1, mydom2, ... , mydomN = ...
```

This feature allows you to declare several domains at the same time. For example:

```
firstname, lastname, address = string
```

Synonyms to Standard Domains

```
my_name = d
```

This declaration specifies a domain ***my_name,*** which consists of elements from a standard domain ***d;*** the domain ***d*** must be **char, real, ref, string, symbol, binary** or one of the integral standard domains: **byte**, **sbyte**, **short**, **ushort**, **word**, **integer**, **unsigned**, **dword**, **long**, **ulong**. For some integral domains also can be used the following syntax:

```
my_name = [signed | unsigned] {byte | word | dword}
```

These declarations are used for objects that are syntactically alike but semantically different. For instance, *Rows* and *Columns* could both be represented as integers, and consequently be mistaken for one another. You can avoid this by declaring two different domains of **unsigned** type, like this:

```
Rows, Columns = unsigned
```

Declaring different domains in this way allows Visual Prolog to perform domain checks to ensure, for example, that *Rows* and *Columns* are never inadvertently mixed. However, both domains can interchangeably be mixed with **unsigned**, and you can use the equal sign to convert between *Rows* and *Columns.*

General Syntax for Synonyms to Standard Domains is:

```
my_dom [, my_dom1] = [reference] <basicdom>
```

Here we use square brackets to indicate optional items. The keyword **reference** is used for declaration of *reference* domains (see the *Declaring Reference Domains* on page 470).

List Domains

```
mylist = elementDom*
```

This is a convenient notation for declaring a list domain. *mylist* is a domain consisting of lists of elements, from the domain *elementDom.* The domain *elementDom* can be either a user-defined domain, or one of the standard types of domain. You read the asterisk `'*'` as "list". For example, this domain declaration:

```
numberlist = integer*
```

declares a domain for lists of integers, such as `[1, -5, 2, -6]`.

General Syntax for List Domains is:

```
mylist [, mylist1]= [reference] [align {byte|word|dword}] elementDom*
```

Here we use square brackets to indicate optional items. The keyword **reference** is used for declaration of *reference* domains (see the *Declaring Reference Domains* on page 470). The optional keyword **align** indicates the type of *memory alignment* (see below in this chapter) to be used.

Multi-alternative Compound Domains

```
myCompDom=f_1(d_11, .., d_1N); f_2(d_21, d_22, .., d_2N); ...
```

To declare a domain that consists of compound elements, you state functors and the domains for all the subcomponents.

For example, you could declare a domain of ***owners*** made up of elements like this:

```
owns(john, book(wuthering_heights, bronte))
```

with this declaration:

```
owners = owns(symbol, book)
book = book(symbol,symbol)
```

where ***owns*** is the functor of the compound object, and ***symbol*** and ***book*** are domains of the subcomponents.

The right side of this type of domain declaration can define several alternatives, separated by a semicolon `';'`. Each alternative must contain a unique functor and a description of the domains for the actual subcomponents of the functor. For example, the following domain declaration could be used to say, "For some predicates a ***key*** is either *up*, *down*, *left*, *right* or a *char* with a character value."

```
key = up; down; left; right; char(char)
```

There is a possibility to include a comment after the domain, for instance

```
person= p(string name, integer age).
```

General Syntax for Multi-alternative Compound Domains is:

```
my_dom [,my_dom_M]* = [reference] [align {byte|word|dword}]
                       alternative_1 [;alternative_N]*
```

Here:

my_dom_M

are names of declared domains. They can be any legal Visual Prolog names.

alternative_N

are declarations of domain alternatives in the form:

```
alternative_functor([subComponent_1 [, subComponent_2]* ])
```

Here `alternative_functor` is the program unique functor naming the alternative. It can be any legal Visual Prolog name. An alternative subcomponents `subComponent_N` must be of the form `sub_Domain_N`

`[sub_Name_N]`, where `sub_Domain_N` can be either standard or any user-defined domain. At last, `sub_Name_N` is an optional name of the subcomponent.

The optional keyword **reference** indicates declarations of reference domains.

The optional keyword **align** indicates the type of *memory alignment* to be used.

Here we use:

- Square brackets indicate optional items, and curly braces mean that one of the items (separated by the symbol '|') must be chosen.
- An asterisk symbol `'*'` indicates arbitrary number of the immediately preceding item (zero or more items)
- The semicolon `';'` is read as "or".

Single-alternative Compound Domains

By prefixing a compound domain declaration with the keyword **struct**, you declare a special *single-alternative compound domain*

```
my_FunctorlessDom = struct [align {byte|word|dword}]
      singleFunctor(dom1 [name1],dom2 [name2],...,domN [nameN])
```

The main difference is that the single-alternative compound domain can declare only one domain "alternative" with a single functor. Therefore, internal representation of single-alternative terms need not store the functor number, which is the reason of other name *functorless domains* for single-alternative compound domains. Such functorless internal representation is directly compatible with C **structs** and the primary aim of using functorless terms is for interfacing with other languages.

Notice that functorless domains (for technical reasons) cannot be declared as **reference** domains. In all other aspects terms of functorless domains can be used just like other terms in your programs.

Domains FILE and DB_SELECTOR

```
file = symbolicFileName1; symbolicFilename2;...;symbolicFilenameN
```

A ***file*** domain must be defined when you need to refer to files (other than the predefined ones) by symbolic names. A program can have only one domain of this type, which must be called ***file.*** Symbolic file names are then given as alternatives for the ***file*** domain. For example, this declaration:

```
file = sales ; salaries
```

introduces the two symbolic file names *sales* and *salaries.*

The following alternatives are predefined in the ***file*** domain:

keyboard *stdin*
screen *stdout*
stderr

Notice that VDE generates the default definitions for **file** and **db_selector** domains in the <ProjectName>.INC file. It is:

```
global domains
   db_selector = browselist_db                  % For treebrowser tool
   file = fileselector1; fileselector2                  % To be edited
```

Therefore, if you need to use files of external databases in your program, then you should append your alternatives to these default declarations.

Specially Handled Predefined Domains

There are several predefined domains; some are handled specially, like the ***file*** domain and the ***db_selector*** domain. Here's a summary of these special predefined domains:

Table 17.2: Specially Handled Predefined Domains

dbasedom	implicitly generated domain for terms in the unnamed global fact database
bt_selector	returned binary tree selector
db_selector	user-defined external database selectors
place	*in_memory; in_ems; in_file*
accessmode	read; readwrite
denymode	denynone; denywrite; denyall
ref	domain for database reference numbers
file	symbolic file names
reg	reg(AX,BX,CX,DX,SI,DI,DS,ES) used with bios/4

Declaring Reference Domains

A *reference domain* is one that can carry unbound variables as input arguments. To declare a reference domain, precede the right side of the domain declaration with the keyword **reference.**

```
DOMAINS
   reflist = reference refint*
   refint  = reference integer
   term    = reference int(refint); symb(refsymb)
   refsymb = reference symbol
```

When you declare a compound domain as a reference domain, all its subdomains are automatically (implicitly) declared as reference domains.

If some domain, for example **integer**, is implicitly redeclared as reference, then variables of this domain are treated as references on the original domain values. For example, integer variables will contain not integer values but references to integers. This can be dangerous for multi-module projects. If a global predicate defined in other module (for example in C library) does not know that **integer** arguments should be not ordinary integer values (but pointers to integer values), then calls to this predicate can generate errors. Therefore, you should always explicitly declare all domains intended to be reference domains in the **domains** section; and you should never use basic domains as reference, instead you should declare a domain being a reference domain to the desired base domain, for instance, as *refint* in the above example.

Declaring Predicate Domains

A *predicate domain* declares a type of predicates. In a subsequent predicate declaration you may then declare one or more predicates as belonging to such a predicate domain, and these may then be specified as arguments to other predicates. Those other predicates will hence be able to do a variable call. Therefore, we say that predicates declared as instances of predicate domains can be used as *predicate values*. (See the *Predicate Values* on page 236.)

In declarations and implementations of classes, these "ordinary" predicate domains can be used for declarations of **static** predicate values. (See the *Static Facts and Predicates* on page 292.) For declarations of **non-static** (object) class member predicates you should use *object predicate domains* (See the *Object Predicate Values* on page 303)

The declaration for *object* predicate domains is of the form:

```
[global] domains
   PredDom = object
      DetermMode [ReturnDom] (ArgList) [- [FlowPattern]] [Language]
```

The syntax for "ordinary" predicate domains is almost the same

```
   PredDom = DetermMode [ReturnDom] (ArgList) [- [FlowPattern]] [Language]
```

The only difference is the additional keyword **object** stating the declaration of *object* predicate domains.

This syntax is described in all details in the section *Predicate Domains* on page 238. ***PredDom*** states the name of the declared predicate domain. ***DetermMode*** specifies the determinism mode for predicates. Must be specified one of the keywords: **procedure**, **determ**, **nondeterm**, **failure**, **erroneous**, and **multi**. ***ReturnDom*** states the domain of the return value (when declaring functions). ***ArgList*** defines domains for arguments; brackets surrounding the arguments should be given even when the ***ArgList*** is empty.

The *Language* specification tells the compiler which calling convention to use, and is only required when declaring domains for routines written in other languages (see the *Interfacing with Other Languages* chapter). The calling convention defaults to **pascal** if omitted, but this should not be relied upon if a particular convention is desired.

The *flowpattern* specifies how each argument is to be used. It should be the letter `'i'` for an argument with input flow and `'o'` for one with output flow. You can have no more than one flow pattern declaration for a predicate domain, and it must be given unless the argument list is empty or all arguments have input flow.

Hence, the predicate domain declaration for a deterministic predicate values taking an integer as input argument and returning an integer, would be:

```
   domains
      list_process = determ integer (integer) - (i)
```

This predicate domain is now known as **list_process**. In classes **list_process** domain can be used only for declarations of statis predicate values, if you need using object predicate values, then you should declare correspondent object predicate domain like this:

```
      objectIntInt = object procedure integer (integer)
```

The object predicate domain *objectIntInt* can be used inside classes for declarations of object predicate values. These predicates (functions) have one input argument from **integer** domain and returns an **integer** value. They have

procedure determinism mode. As non-static class members, they have a hidden argument pointing onto objects (class instances) to which these predicate values belong.

The Predicates Section

In Visual Prolog, the sections introduced by the keyword **predicates** contain predicate declarations. You declare a predicate by its name and the domains of its arguments, like this:

```
predicates
   predname(domain1 [Name1], domain2 [Name2],...,domainN [NameN])
```

In this example, ***predname*** represents the new predicate name and *domain1*, ..., *domainN* stand for user-defined or pre-defined domains. Optionally after each argument domain, you can specify a mnemonic argument name *NameN*. You can use these argument names to improve readability; the compiler just deletes them on the preprocessing phase.

Multiple declarations for one predicate are also allowed. As an example, you could declare that the predicate ***member*** works both on numbers *and* names by giving the following declarations:

```
PREDICATES
   member(name, namelist)
   member(number, numberlist)
```

In this example, the arguments *name, namelist*, *number*, and *numberlist* are user-defined domains.

You can declare a predicate with several different arities.

```
hanoi              % chooses 10 slices as default
hanoi(integer)     % moves N slices
```

If you give more than one declaration for the same name (with different domains or arities), these declarations must come right after each other. However, in classes hierarchy you can declare overloading predicates (different domains or arities) and overriding predicates (with the same domains and arity).

Determinism Modes

You can declare predicates as being deterministic by preceding the predicate declaration with the keywords **procedure**, **determ**, **failure**, or **erroneous**. From the other hand, you can declare a predicate as being non-deterministic by

preceding a declaration by the keywords **nondeterm** or **multy**. If you declare a predicate to be deterministic, the compiler will issue a warning/error if it finds any non-deterministic clauses for the predicate. On the other hand, when you declare a predicate as non-deterministic, the compiler will not complain on non-deterministic clauses in this predicate. The compiler *default determinism mode* for predicates is **determ**. You can change this default to **nondeterm** or **procedure** in the VDE's **Compiler Options** dialog with an appropriate selection in the **Default Predicate Type** radio button group. You can also directly specify the default determinism mode to the compiler with the compiler option **–Z{pro|dtm|ndt}**. (Here: **pro** corresponds to the **procedure**; **dtm** to **determ**, and **ndt** to **nondeterm**.)

The compiler implicitly accepts this default determinism mode to all predicates declared without an explicit determinism mode specification.

```
nondeterm repeat              /*repeat is non-deterministic by design*/
determ menuact(Integer,String)            /*menuact is deterministic*/
```

By default, the compiler checks user-defined deterministic predicates (**determ**, **procedure**, **failure** and **erroneous**) and gives warnings/errors for clauses that can result in a non-deterministic predicate. This checking is extremely important for effective and reliable code generation; therefore, it cannot be switched OFF from VDE. (It can be switched OFF only by the compiler option **–udtm**).

By default, the compiler checks user-defined predicates declared with the keywords **procedure**, **multi**, and **erroneous** and gives warning/errors if it cannot guarantee that a predicate *cannot fail*. To switch OFF this checking from the VDE, you can uncheck the **Check Type of Predicates** in the **Compiler Options** dialog. (It can be switched OFF also by the command-line compiler option **–upro**).

The short description of determinism modes:

nondeterm

The keyword **nondeterm** defines non-deterministic predicates that *can backtrack* and generate multiple solutions. Predicates declared with the keyword nondeterm *can fail*.

procedure

The keyword **procedure** defines predicates called *procedures* that always have one and only one solution. (But run-time errors are possible.) Procedures *cannot fail* and do not produce *backtrack points*. Most built-in Visual Prolog predicates are internally declared as procedures.

determ

The keyword **determ** defines deterministic predicates that can *succeed* or *fail*, but cannot produce *backtrack points*. That is, predicates declared with the keyword determ have no more then one solution.

multi

The keyword **multi** defines non-deterministic predicates that can backtrack and generate multiple solutions. Predicates declared with the keyword **multi** cannot fail.

erroneous

A predicate declared with the keyword erroneous should never fail and should not produce solution. Typical used for error-handling purposes. Visual Prolog supplies built-in **erroneous** predicates ***exit*** and ***errorexit***.

failure

A predicate declared with the keyword **failure** should not produce a solution but it *can fail*. Visual Prolog supplies the built-in **failure** predicate ***fail***. Failure predicates are usually used to enforce backtracking to the nearest backtrack point. The predicate ***failure1*** demonstrates the difference between **failure** and **erroneous** predicates:

```
failure1(0) :-                          %This predicate can fail
   errorexit().
failure1(_) :-
   fail().
```

Flow Patterns

In a given predicate call, the known arguments are called *input* arguments (i), and the unknown arguments are called *output* arguments (o). The set of the input and output arguments in a given predicate call is called the *flow pattern*.

Starting from the **goal** section, the Visual Prolog compiler analizes all calls of each local user-defined predicate and internally defines all possible flow patterns with which a predicate can be used. For each flow pattern that a predicate is called with, the flow analyzer goes through that predicate clauses with the variables from the head set to either input or output (depending on the flow pattern being analyzed) and defines flow patterns for calls of used predicates. When the flow analyzer recognizes that a standard predicate or a user-defined predicate is called with a nonexistent flow pattern, it issues an error message.

For local predicates (since a local predicate is used only in one compiling module) the compiler can determine all flow patterns a predicate is called with; therefore, it is not required to specify flow patterns for local predicates. But if you know that a local predicate is only valid with special flow patterns, it is a good idear to explicitly specify these flow patterns in the predicate declaration. Then the flow analyzer will catch any wrong usage of this predicate.

You can also specify separate determinism mode before each flow pattern. For example, you can give the following detalization to the declaration of the predicate **plus/3** in the example ch10e02.pro:

```
PREDICATES
   nondeterm plus(integer, integer, integer) –
      procedure (i,i,o)
      procedure (o,i,i)
      procedure (i,o,i)
      determ (i,i,i)
      (o,o,i) (i,o,o) (o,i,o) (o,o,o)
```

If a special determinism mode is explicitly declared before a flow pattern (for example, `procedure (i,i,o)`) then it overrides the "main" predicate determinism mode declared before the predicate name, otherwise, the flow pattern has the determinism mode declared before the predicate name (for example, `(o,i,o)` has **nondeterm** mode).

The General Form of Local Predicate Declarations is:

```
predicates
   [DeterminismMode] PredicateName [(ArgList)]
      [- [[FlowDetermMode] FlowPattern] [[,][FlowDetermMode] FlowPattern]*]
```

All elements used in this declaration are explained with all details in the *Global Predicates* on page 254).

Functions

By prefixing a predicate declaration with a domain name, you declare a function. The return value is taken from the last argument in the final clause executed, and this argument must not be present in the predicate declaration. A function returning the cube of its argument would hence be declared as:

```
   predicates
      procedure integer cube(integer)
```

And the clause for this function would be:

```
clauses
   cube(In,Out):- Out = In*In*In.
```

A function can return any domain.

Predicate Values

If you have declared a *predicate domain* in the domain section, you may declare one or more predicates as belonging to that domain. The syntax for this is.

```
PREDICATES
   pred1: p_domain
   pred2: p_domain
   ...
```

where ***pred1***, ***pred2*** etc. are the predicate names and *p_domain* is the predicate domain declared in the domain section. As "plain" predicates, these predicates ***pred1***, ***pred2*** can be called with appropriate arguments. But also these predicates can be considered as *predicate values* since they have the following value properties:

- They can be passed as parameters and returned from predicates and functions.
- They can be stored in facts.
- They can be held in variables.
- They can be compared for identity.

Object Predicate Values

If you have declared an *object predicate domain* (for example, *my_objDom*) in a domains section, you may (in a class declaration or in a class implementation) declare one or more non-static class member predicates as belonging to this object predicate domain. The syntax is.

```
class c
   predicates
      my_obj_pred1: my_obj_Dom
      my_obj_pred2: my_obj_Dom
```

The Facts Section

A **facts** section (**database** section in PDC Prolog) declares predicates just as the **predicates** section does. However, the clauses for such predicates can only consist of plain facts, they cannot have an associated body. These facts can be

inserted at run time by ***assert, asserta, assertz,*** or ***consult*** predicates into *internal fact databases*, and you can remove them again with ***retract*** or ***retractall.*** The predicates ***save/1*** and ***save/2*** save the contents of internal fact databases to a file. The predicates ***consult/1*** and ***consult/2*** read facts from a file and assert them into internal fact databases.

You can have several facts sections in your program; some of them can be global and some local. You can name facts sections, and each name must be unique within the module. If you do not give a name for a facts section, the compiler will give it the default name ***dbasedom***. Only one such unnamed facts section is possible and it must be declared **global** in multi-module programs (this is the default for projects created by the Visual Development Environment). When a facts section is declared, the compiler internally declares the domain with the same name as the facts section name; this allows predicates to handle facts as terms.

Notice that inside the implementation and the declaration of a class you can declare several unnamed facts sections, for such fact databases the compiler generates special internal names that cannot be referenced from the program code.

The General Form of Facts Section Declarations is:

```
[global] facts [ - <databasename>]
   [nocopy] [{ determ | single | nondeterm }] dbPredicate ['(' [Domain
   [ArgumentName]]* ')']
      ...
```

Each facts section can declare any number of database predicates.

By their basic nature, fact database predicates are nondeterministic; therefore, the default determinism mode of facts is **nondeterm**. You can precede declaration of a database predicate with the keywords:

- **determ** if you know that there can be only one fact for that database predicate.
- **single** if you know that there always will be one and only one fact for that database predicate.

This enables the compiler to produce better code, and you will not get non-deterministic warnings for calling such predicates. This is useful for flags, counters, etc.

nocopy

Normally, when a database predicate is called to bind a variable to a string or a compound object, then the referenced data are copied from the heap to the Visual Prolog global stack (GStack). The **nocopy** declares that the data will not be copied and variables will reference directly to the fact's data stored in the heap. This can increase efficiency, but should be used carefully. If a copy was not made, the variable would point to garbage after the fact retraction was made.

global

Determines a global facts section. (See **Modular Programming** on page 252.) Notice that safe programming techniques require that you do not use global facts. Instead, you can use global predicates handling local facts.

An example is:

```
facts - tables
   part(name,cost)
   salesperson(name,sex)

predicates
   write_table_element(tables)

clauses
   write_table_element(part(Name,Cost)):-
     writef("\nPart's Name= % Cost = %",Name,Cost).
   write_table_element(salesperson(Name,Sex)):-
     writef("\nSalesperson's Name= % Sex = %",Name,Sex).
```

The Clauses Section

A clause is either a fact or a rule corresponding to one of the declared predicates. In general, a clause consists of either 1) a fact or 2) a clause head followed first by a colon and hyphen (**:-**), then by a list of predicate calls separated by commas (**,**) or semicolons (**;**). Both facts and rules must be terminated by a period (**.**).

The fact:

```
same_league(ucla, usc).
```

consists of a predicate name ***same_league***, and a bracketed list of arguments `(ucla, usc)`.

Notice that for compatibility with other Prologs you can use the keywords: **if** instead of (**:-**), **and** instead of (**,**), **or** instead of (**;**).

Simple Constants

Simple constants belong to one of the following standard domains:

char	A character (an 8-bit ASCII character enclosed between a pair of single quotation marks) belongs to the ***char*** domain. An ASCII character is indicated by the *escape character* (\) followed by the ASCII code for that character. *\n, \t, \r* produce a newline , a tab and a carriage return character, respectively. A backslash (\) followed by any other character produces that character ('\\' produces \ and '\"' produces ').
integral numbers	positive and negative numbers can be represented in the Visual Prologs integral number domains shown in the following table.
real	A real number belongs to the ***real*** domain and is a number in the range 1.7*10^-307 to 1.7*10^+308. Real numbers are written with a sign, a mantissa, a decimal point, a fractional part, an *e*, a sign, and an exponent, all without included spaces. For example, the real value -12345.6789 * 10^3 can be written as `-1.23456789e+7`. The sign, fractional, and exponent parts are optional (though if you omit the fractional part, you must leave out the decimal point, too). Visual Prolog automatically converts integers to real numbers when necessary.
string	A string (any sequence of characters between a pair of double quotation marks) belongs to the string domain. Strings can contain characters produced by an escape sequence (as mentioned under char); strings can be up to 64 K in length.

symbol	A symbolic constant (a name starting with a lower-case letter) belongs to the symbol domain type. Strings are accepted as symbols too, but symbols are kept in an internal table for quicker matching. The symbol table takes up some storage space, as well as the time required to make an entry in the table. However, if the same symbols are frequently compared, it's well worth the investment.
binary	A binary constant belongs to the binary domain. It is written as a comma-separated list of integral values, each less than or equal to 255, enclosed in square brackets prefixed with a dollar sign: $[1,0xff,'a'].
predicate value	A predicate value is the name of a predicate previously declared as belonging to a predicate domain. It is written simply as the name of the predicate, with no argument list or brackets.
object predicate value	An object predicate value is the name of a predicate previously declared as belonging to an object predicate domain. It is written simply as the name of the predicate, with no argument list or brackets.

Table 17.3: Integral Standard Domains

Domain	Description and implementation		
short	A small, signed, quantity.		
	All platforms	16 bits,2s comp	32768 .. 32767
ushort	A small, unsigned, quantity.		
	All platforms	16 bits	0 .. 65535
long	A large signed quantity		
	All platforms	32 bits,2s comp	-2147483648 .. 2147483647
ulong	A large, unsigned quantity		
	All platforms	32 bits	0 .. 4294967295
integer	A signed quantity, having the natural size for the machine/platform architecture in question.		
	16bit platforms	16 bits,2s comp	-32768 .. 32767
	32bit platforms	32 bits,2s comp	-2147483648 .. 2147483647
unsigned	An unsigned quantity, having the natural size for the machine/platform architecture in question.		
	16bit platforms	16 bits	0 .. 65535
	32bit platforms	32 bits	0 .. 4294967295
byte			
	All platforms	[3] 8 bits	0 .. 255
word			
	All platforms	16 bits	0 .. 65535
dword			
	All platforms	32 bits	0 .. 4294967295

An integral value may be preceded by `0x` or `0o`, indicating hexadecimal and octal syntax respectively.

Terms

A term is, strictly speaking, any Prolog entity. Either an object from one of the domains of standard type, a list, a variable, or a compound term, i.e., a functor followed by a list of terms (optional arguments) enclosed in parentheses and separated by commas. Facts can be handled as terms when using the domains internally generated for names of facts sections. In practice we tend to mean those (variable) entities holding data or non-compiled information, or compound terms (consisting of a functor and optional arguments).

Variables

Variables are names starting with an upper-case letter or underscore or, to represent the anonymous variable, a single underscore character underscore character). The anonymous variable is used when the value of that variable is not of interest. A variable is said to be *free* when it is not yet associated with a term, and *bound* or *instantiated* when it is unified with a term.

The Visual Prolog has an option so it can give a warning when it detects that a variable has been used only once in a clause. This warning will not be given if the variable starts with an underscore. Note that when a variable name starts with an underscore like ***_Win***, it is still a normal variable, which unlike the anonymous variable can be used to pass values from one call to another.

Compound Terms

A compound term is a single object that consists of a collection of other terms (called *subcomponents)* and a describing name (the *functor).* The subcomponents are enclosed in parentheses and separated by commas. The functor is written just before the left parenthesis. For example, the following compound term consists of the functor ***author*** and three subcomponents:

```
author(emily, bronte, 1818)
```

A compound term belongs to a user-defined domain. The domain declaration corresponding to the ***author*** compound term might look like this:

```
DOMAINS
   author_dom = author(firstname, lastname, year_of_birth)
   firstname, lastname = symbol
   year_of_birth = integer
```

Functorless Compound Terms

By prefixing a compound domain declaration with the directive **struct,** you declare a single alternative compount domain.

```
domains
   author_dom = struct author(firstname, lastname, year_of_birth)
```

There can be no alternatives in a single alternative compound domain. Hence, the internal representation of a term of a single alternative compound domain has no functor. Therefore, such domains can be named *functorless domains*. Functorless terms can be used just like other terms in your source code, but their primary aim is to be directly compatible with C **structs**.

Lists – a Special Kind of Compound Terms

Lists are a common data structure in Prolog and are actually a form of compound terms. Syntactically, it is written as a sequence of comma-separated arguments, enclosed in square brackets. A list of integers would appear as follows:

```
[1, 2, 3, 9, -3, 2]
```

Such a list belongs to a user-defined domain, such as:

```
DOMAINS
   ilist = integer*
```

If the elements in a list are of mixed types (for example, a list containing both characters and integers), you must state this in a corresponding domain declaration. For example, the following declarations

```
DOMAINS
   element = c(char) ; i(integer)
   list = element*
```

would allow lists like this one:

```
[i(12), i(34), i(-567), c('x'), c('y'), c('z'), i(987)]
```

Memory Alignment

By prefixing a compound or list declaration with an alignment specification, you can override the default alignment. The syntax is:

```
DOMAINS
   dom = align { byte | word | dword } domdecl
```

where *domdecl* is a normal domain declaration:

```
DOMAINS
   element = align byte c(char) ; i(integer)
   list = align dword element*
```

This would make the internal representation for *elements* byte-aligned and *list* dword-aligned.

If you want to override the default alignment for a functorless domain, the **struct** directive must precede the **align** directive.

```
DOMAINS
   bbdom = struct align byte blm(char,integer)
```

The primary aim of overriding alignment is to make compound objects compatible with external code using a different alignment than the default for your platform. If several programs share an external database or communicate over pipes, the domains involved must use the same alignment.

The Goal Section

The **goal** section is declared with the keyword **goal**. A **goal** is a special variant of a clause, sometimes called a goal clause.

Essentially, the **goal** section is the same as the body of a clause. The goal section simply contains a list of subgoals separated by commas `','` (logical **AND**) or semicolons `';'` (logical **OR**). However in contrast to a rule, the goal section does not contain `':-'`. When the program starts, Visual Prolog automatically calles the goal and the program runs, trying to satisfy the body of the goal rule. If the subgoals in the **goal** section all succeed, then the program terminates *successfully*. Otherwise, we say that the program *fails*. For example,

```
goal
   write("Hello world"), readchar(_).
```

The Constants Section

You can define and use constants in your Visual Prolog programs. A constant declaration section is indicated by the keyword **constants,** followed by the declarations themselves, using the following syntax:

```
<Id> =  <definition>
```

Each <*definition*> is terminated by a newline character, so there can be only one constant declaration per line. Constants declared in this way can then be referred to later in the program.

Consider the following program fragment:

```
CONSTANTS
   blue  = 1
   green = 2
   red   = 4
   grayfill = [0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 ]
   language = english
   project_module = true
```

Before compiling your program, Visual Prolog will replace each constant with the actual string to which it corresponds. For instance:

```
...
menu_colors(red,green,blue),
my_fill_pattern(grayfill),
text_convert(prolog, language),
status(project_module),
...
```

will be handled by the compiler in exactly the same way as:

```
...
menu_colors(4, 2, 1),
my_fill_pattern([0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55]),
text_convert(prolog, english),
status(true),
...
```

There are a few restrictions on the use of symbolic constants.

- The definition of a constant can't refer to itself. For example:

  ```
  list = [1, 2|list].     /* Is not allowed  */
  ```

  will generate the error message `Recursion in constant definition.` The system does not distinguish between upper-case and lower-case in a constant declaration. Consequently, when a constant identifier is used in the clauses section of a program, the first letter must be lower-case to avoid ambiguity with variables. So, for example, the following is a valid construction:

  ```
  CONSTANTS
     Two = 2
  ```

```
GOAL
   A=two, write(A).
```

- There can be several **constants** sections in a program, but each constant must be declared before it is used.
- Constant identifiers are global for the rest of the module and can only be declared once. Multiple declarations of the same identifier will result in an error message. You can use constants to redefine names of domains and predicates, except the specially-handled predicates. Refer to "Specially-Handled Predicates" earlier in this chapter.

Predefined Constants

Depending on the target platform selected for compilation, one or more constants will be predefined:

Table 17.4: Predefined Constants

Constant	Target selections causing it to be defined
os_dos	DOS, Phar Lap and 16-bit Windows
os_os2	32-bit or 16-bit OS/2
os_nt	32-bit MS Windows SCO UNIX and Linux
os_unix	32-bit or 16-bit MS Windows
ws_win	Presentation Manager
ws_pm	Phar Lap286
dosx286	16-bit platforms
platform_16bit	32-bit platforms
platform_32bit	COFF object format: 32-bit Visual C++.
use_coff_objformat	ELF object format is used on Linux
use_elf_objformat	OMF object format
use_omf_objformat	

Selecting DOS as your target will cause *os_dos* to be defined, and selecting 16-bit MS Windows will cause both *os_dos* and *ws_win* to be defined.

These predefined constants enable you to control platform-dependent conditional compilation with **ifdef**/**ifndef** compiler directives.

Conditional Compilation

You use conditional compilation when you need to generate different versions of the same program; for example, one version that uses graphics and another that only uses text mode. The syntax for conditional compilation directives is:

```
[ifdef | ifndef] <constantID>
   ...
elsedef
   ...
enddef
```

<constantID> represents a constant identifier declared in a constants section. The value of the constant is irrelevant; only its presence matters. The ifdef directive succeeds if the constant is defined, while the ifndef directive succeeds if the constant *is not* defined. The elsedef part is optional. The following program shows a typical use of the conditional compilation directives.

```
CONSTANTS
   restricted = 1

ifdef restricted              /* if restricted is defined, use this */

savebase(_):-
   write("\nBase cannot be saved in demo version"),
   readchar(_).

elsedef                                    /* otherwise, use this */

savebase(Name):-
   write("\nSaving ",Name),
   save(Name).

enddef
```

Including Files in Your Program

You use `include` to include the contents of another file in your program during compilation. The syntax is:

```
include "OSFileName"
```

The *OSFileName* can include a path name, but you must remember that the backslash character used to give subdirectories in the DOS-related versions of Visual Prolog is an escape character in Visual Prolog. Because of this, you must always give two backslash characters when you use the backslash in a path inside the source text.

```
include "\\vip\\include\\error.con"
```

Under **Options | Project | Directories** you can give one or more paths separated by semicolons (colons under UNIX) to indicate where the Prolog system should look for the include files (Here, of course, only a single backslash is required). If you don't give an absolute path in your *OSFileName*, the compiler will in turn try to concatenate each of the paths given in the include directory to your filename in order to locate the file.

You can only use **include** files on section boundaries in a program, so **include** can appear only where the keywords **constants**, **domains, predicates, goal, facts, clauses**, **class** or **implement** are permitted. An include file itself can contain further **include** directives. However, include files must not be used recursively in such a way that the same file is included more than once during a module compilation.

Include files can contain any sections, provided the restrictions on program structure are observed (see page 463).

Modules and Global Programming Constructions

Compilation Units

The programmer writes all code in source files. The source files are presented to the compiler in "chunks" called *compilation units*. Each compilation unit is compiled separately into an intermediate file called an object file. These intermediate files are linked together (maybe with other files) by the PDC Link Editor (or other linker) to create the target file. The Visual Prolog's target files can be executables or dynamic link libraries (DLLs).

Until the moment, in this chapter we mainly have spoken about programs having one compilation unit. We say that such programs consist of one *module*. Of course, one-module programs can consist of several source files that are included (with **include** derectives) during the compilation into the "root" source file of the module.

When your program consists of a single module, then you need only local declarations (that were discussed in this chapter). Because all user-defined names are defined in this module, the compiler sees all of them while compiling the module. The scope of local names is the module where they are defined. However, real programs rare consist of one module. There are several reasons: restrictions on numbers of domains and predicates that can be declared in a module, necessity to use tools supplied as separate modules, etc. Notice that any VPI (Visual Programming Interface) program contains as less two modules.

Names with Global Scope

Since the scope of local names is a module in which they are defined, hence, to interact between two modules they need some "large-scale" entities that can be accessed from both modules. In Visual Prolog such large-scale names are:

- *Global* domains and predicates declared in **global domains** or **global predicates** sections (and also global facts).
- *Public* domains and predicates (and facts) declared in *declarations of classes*.

Declarations of global domains and predicates is discribed in the *Modular Programming* on page 252). From the inter-module communication perspective, public predicates and domains declared in classes have the same behavior as global ones. (Of course, declared in classes entities need to be qualified with names of classes or with objects to which they belong.) The common rules are the following:

1. A module can use all names declared in the scope of the module. That is:
    - a module can use all global predicates and domains (and facts), which declarations are included in the module;
    - a module can use all public predicates, domains (and facts) that are declared in the classes, which declarations are included in the module. The module can also use all domains corresponding to names of these classes.
2. The main project module (with the **goal**) must contain declarations of all global domains, global facts sections, and **abstract** classes that are declared in all project submodules.
3. Any other project module may contain declarations of only those global domains, global facts sections, **abstract** and "ordinary" classes, which are used in this module.
4. If any global declaration or a class declaration is changed, all modules including this declaration must be recompiled.

Include Structure of Multi-modular Programs

Let us consider include "styles" of multi-modular programs, which fit the above requirements.

Include All Global Declarations in each Module

First, let us start with the most simple to use "style", which fit well for beginners, small-scale programming, prototyping, etc. This is the VDE's default structuring mechanism. According to it, every module includes a <ProjectName>.INC file, which then in turn includes <ModuleName>.DOM and a <ModuleName>.PRE files for each module. In the DOM file you should declare the global domains (and facts) introduced in the module. In the PRE file you should declare the global predicates and place class declarations. Since every module includes the INC file, which includes all DOM and PRE files, you have access to every global domain and every global predicate in each module. To follow this mechanism, creating a new module you simply need in VDE's **File Inclusion for Module** dialog:

1. Check ON the "**Create <ModuleName>.DOM**" for each source module, which may introduce new global domains and
   - check ON the "**Include <ModuleName>.DOM**", to specify that the **include** statement for this file must be generated in the **<ProjectName>.INC** file.
2. Check ON the "**Create <ModuleName>.PRE**" for each source module, which may introduce new global predicates or declare a new classes and
   - check ON the "**Include <ModuleName>.PRE**", to specify that the **include** statement for this file must be generated in the **<ProjectName>.INC** file.

This is nice for "easy programming" (programming in the small).

Notice that regarding to proper handling of global domains this was the best possible mechanism in Visual Prolog versions prior to 5.2. Because older versions required that all global domains were declared in all modules (and even in the same order). Class names are considered global domains (in some respects), and (until Visual Prolog v.5.2) class declarations also had to be included in every module. Notice that every thing we say here about global domains also count for domains corresponding to names of global facts sections.

Where-used Strategy for Including Global Declarations

When you make large programs, there are serious drawbacks of the above style:

- If you change any global declaration of a class declaration, you have to recompile all modules of the project.
- You soon meet the restrictions on numbers of predicates and domains that can be used in a module.
- When creating larger programs (especially with many programmers), having access to everything from everywhere tends to lead to spaghetti programs.

That is, programs where every part of a program references to all other parts of the program.

Therefore, it is quite common to structure larger programs differently. When creating larger programs you would often move the includes of all the PRE and DOM files out of the INC file, and include them only into those modules (PRO files) that actually use the corresponding PRE and DOM files. (Recall that until VIP v. 5.2, you could not do this with DOM files.) Notice that now it is also recommended to place declarations of classes into **<ModuleName>.PH** files (separately from declarations of global predicates).

The benefit from moving includes of PRE, DOM, and PH files from the INC file into the using them PRO files are:

- When you change declaration of a global predicate, a global domain or a class, only those modules that include the changed declaration need to be recompiled. This can considerably decrease recompilation time.
- Since the overall numbers of predicates and domains that can be declared in a module are limited; therefore, more global predicates and domains can be used in a project and more local predicates and domains can be declared in each module.
- If a module does not include a certain PRE, DOM or PH file, then you cannot call the predicates or use domains declared in this file. Subsequently it is easier to maintain a certain usage structure between modules, with reduction in spaghetti code as a consequence. Especially it is easier to avoid that new programmers on a project, who do not yet know the program structure, accidentally circumvent a good program structure. New programmers on a project can also learn about the overall structure of a program by investigating the include structure.

Now when the main module is the only one that has to include declarations of all global domains and abstract classes, you might consider making the main module very small (so that recompilation will be fast). An easy way to make the main module small is to replace the goal in its original place with a clause of a new global predicate (lets name it **run**).

Then create a new module containing *all* includes required to the main module and a goal:

```
goal
   run().
```

The declaration of the global predicate ***run*** must, of course, be visible both in the old main module and in the new one.

Qualification Rules for Public Class Members

This manual provides rather compact description of syntax for classes and objects in Visual Prolog (see the *Classes and Objects* on page 282). Therefore, we will not repeat it here. The only important to modular programming feature that should be summarized here are scoping and qualification rules for names declared in classes.

1. Any class entities (domains, predicates, facts) declared in a class implementation is local to that implementation, and inside this implementation these class entities are used just like any other domains, predicates or facts.
2. Any public class entities declared in a class declaration can be used directly in all classes that inherit (transitively) from that class (including the class itself). Domains declared in a class declaration can be used both in the declaration and in the implementation of this class.
3. In classes that do not inherit (transitively) from the class declaring the domain (and outside classes), the domain and all functors have to be qualified with the class name of the declaring class. The qualification have to go to the class that really declares the domain, it cannot go to a class that merely inherit from that class. In other words, the derived classes only "inherit" the ability to use a domain.
4. Static class members belong to the class, not to the individual objects. Outside implementations of classes that inherit (transitively) a public static class member, it can be qualified with the class name using the following syntax:

   ```
   class_name :: static_class_member[(arguments)]
   ```

   Public static members of a class can be used in any place where the class declaration is visible (in any modules including the class declaration) even without creation of any object of the class. Of course, qualification with reference to an object is also possible.
5. In opposite, non-static class members belong to the instances (objects). Therefore, to use a public non-static class member outside implementations of classes that inherit (transitively) this member, an object obtaining this member should be created, and in the call of this non-static member an object identifier should be specified using the following syntax:

   ```
   object : non_static_class_member[(arguments)]
   ```

6. In Visual Prolog all public predicates (except ***new*** and ***delete***) are *virtual*. Virtual predicates allow derived classes providing different versions of a parent class predicates. You can declare a predicate in a parent class and then redefine it in any derived class. Calling a virtual predicate you may need to qualify explicitly the class name. The general syntax is:

```
[ObjectVariable:] [name_of_class::] virtual_pred_name[(arguments)]
```

Notice that all domains declared in classes are static/class entities. The domain belongs to the class not to the individual objects (but like other static/class entities they can of course be used freely by objects).

Being able to declare domains in classes opens the possibility to use classes as modules. If a class only declare static entities, then it can be considered a module. The static/class entities of a class can however be used as ordinary global entities, as long as you remember to qualify them with the class name. One advantage of creating modules this way is that the module will have a separate name space (as the result of qualification with the class name). This means that you can choose names in the module more freely. Another advantage is that classes do not have to be included in the main module, even if they contain public domains

Compiler Options for Multi-modular Projects

To manage multi-modular programs Visual Prolog's compiler uses the concept of *projects*. It is used to handle the following two tasks:

1. Define the SYM-file name.

   Visual Prolog stores **symbol** domain terms in a so-called *symbol table*, which is stored in the special (object format) SYM-file. By default, its filename extension is SYM. This symbol table is shared by all modules. Hence, the compiler must collect in the same SYM-file all symbol terms used in all modules. Therefore, while compiling each project module the compiler has to know the name of the common SYM-file.

2. To notify the compiler that compiled modules are parts of a project and so do not obliged to contain the **goal** section.

   Visual Prolog programs must have the internal **goal**. Therefore, by default the compiler requires that each compiled module has the **goal** section, but in projects only the *main* module contains the **goal**.

These can be done by the command line compiler option `-r[ProjectName]`. For example, let the compiler, while compiling a file *ModuleName.PRO*, get the option

```
-rProjectName
```

It notifies the compiler that:

- the compiled file *ModuleName.PRO* is a module of the project *ProjectName* and so does not have to contain the **goal** section;
- the SYM-file name is *ProjectName*.*SYM*.

These also can be done with the **project** compiler directive (see below in this chapter), but the compiler option `-r` overrides it. The `-r` command line compiler option is much more convenient to the Visual Development Environment; therefore, the **project** compiler directive is almost never used now.

If you use the VDE, the Application expert automatically specifies the required command line compiler options. Therefore, if you use the VDE, then you need not any knowledge about command line compiler options. Otherwise, you can find descriptions of command line compiler options in the on-line help. (Search for the topic *Command Line Compiler Options*.)

Compiler Directives

A number of compiler features are controlled through *compiler directives.* You can introduce one or more of the following directives at the beginning of the program text:

```
check_determ      errorlevel        nowarnings
code              heap              printermenu
config            gstacksize        project
diagnostics       nobreak
```

Many of the compiler directives can be set both in the Visual Prolog development environment (from the menus), through command-line options and in the source code. If a compiler directive exists in the source code, its setting will override values set elsewhere.

Note, that most of the compiler directives are now obsolete for VPI Programs.

check_determ

When you specify **check_determ**, the Visual Prolog compiler will give a warning for all non-deterministic clauses, unless the predicates have explicitly been declared as **nondeterm**. This compiler directive can override the command-line compiler option **-udtm-** or with the VDE's option **Options | Project | Compiler Options | Check Type of Predicates**.

You can use **check_determ** to guide the setting of cuts. Visual Prolog itself performs extensive tests to decide whether a predicate is deterministic or non-deterministic, so you don't need to fill your programs with cuts merely to save stack space (as is necessary in many other Prolog implementations). Visual Prolog offers effective determinism monitoring based on declarations of determinism modes of predicates and facts. Six different determinism modes can be declared for predicates with the keywords **multi**, **nondeterm**, **determ**, **procedure**, **failure** or **erroneous**. To declare the predicate determinism mode one can precede the predicate declaration with one of these keywords. Notice that a separate determinism mode can be declared for each flow pattern of a predicate by preceding a flow pattern with the required determinism mode keyword. For example:

```
predicates
   procedure difModes(string,integer) – determ (i,i) (i,o) (o,i)
```

This example declares predicate `difModes` as **determ** for `(i,i)` flow and as **procedure** for all other possible flow patterns. You can also use keywords **determ** or **single** declaring database predicates intended to have no more than one fact or one and only one fact respectively. Calling and retracting of facts declared deterministic are deterministic; you can use declarations of **determ** or **single** facts to avoid warnings about non-deterministic clauses.

The default determinism mode for predicates is **determ**. You can change it by the **z[Value]** command line compiler option to **procedure** (`Value=pro`) or **nondeterm** (`Value=ndt`). Visual Development Environment generates these options, when you check the correspondent check box for the **Default Predicate Mode** in the **Compiler Options** dialog.

Notice that if a predicate is declared with the mode **determ**, **procedure**, **failure** or **erroneous**, then the compiler always checks the predicate for non-deterministic clauses.

There are two kinds of non-deterministic clauses:

1. If a clause does not contain a cut, and there are one or more clauses that can match with the same input arguments for that flow pattern.
2. If a clause calls a non-deterministic predicate, and that predicate call is not followed by a cut.

code

The code directive specifies the size of the internal code array. The default is 4000 paragraphs (16-byte units) for the 16-bit versions of Visual Prolog,

otherwise 10000 paragraphs. For larger programs you might need to specify a larger size.

```
code = Number_of_paragraphs
```

where *Number_of_paragraphs* represents the number of memory paragraphs (16 bytes each) required in the code array. For example, the directive:

```
code = 1024
```

sets the size of the code array to 16 Kbytes.

The code directive has no influence on the size of an executable file, it simply controls how much memory the compiler should allocate for the compilation.

When the code size exceeds the value 4095, the compiler will switch over to generating FAR calls inside that module. For this reason, you should only use a code size above 4095 if it is really needed. For 32-bit code generation, the size of the code array is practically irrelevant. All code is NEAR, and the operating system will only allocate physical memory when the allocated virtual memory is actually used.

config

This option is only relevant for old DOS textmode windowing applications!

To let a stand-alone DOS application read a configuration file (which defines default window attributes, keyboard setup, etc.) place the directive:

```
config "<ConfigFileName>.cfg"
```

in your program.

diagnostics

Corresponds to the VDE's option **Options | Project | Compiler Options | Diagnostics Output**. When you specify **diagnostics**, the compiler will display an analysis of your program containing the following information:

- the names of the predicates used
- whether a predicate is local, global or defined externally
- whether a predicate is deterministic or non-deterministic
- the size of the code for each predicate
- the domain types of the parameters

- the flow patterns for each predicate

The diagnostics will also produce a listing of which domains are treated as reference domains, and for which domains the compiler generates internal unification predicates. These predicates are generated when you unify two terms in certain non-simple ways. As an example if you are writing L1=L2 where both L1 and L2 are bound to a list, the compiler needs to test all the elements of the list for equality.

Here's an example of a diagnostics display:

```
DIAGNOSTICS FOR MODULE: /usr/pdev/test/flow1.pro

Predicate Name   Type   Determ Size  Domains – flowpattern
---------------- ------ ------ ----- ---------------------------
_PROLOG_Goal     local  yes      216  --
_p1_0            global yes      112 integerlist – [i,o,i|o]
_p1_1            global yes      128 integerlist – o
---------------- ------ ------ ----- ---------------------------
Total size                      460

Size of symbol table=    324 bytes
Size of PROCONST segment=1705 bytes
```

Under **Options | Global | Environment**, it is possible to log the diagnostics output into a file.

errorlevel

Corresponds to the VDE's option **Options | Project | Compiler Options | Errorlevel**. The compiler directive `errorlevel` enables you to control how detailed the error reporting should be. The syntax is:

```
errorlevel = d
```

where *d* is one of 0, 1, or 2, representing the following levels:

d	Level of Error Reporting
0	Generates the most efficient code. No cursor information will be placed in the code and only the error number will be displayed in an error occurs.

1	This is the default. When an error occurs, its origin (module name and include file, if applicable) will be displayed. The place where the error was detected within the relevant source file will also be displayed, expressed in terms of the number of bytes from the beginning of the file.
2	At this level, certain errors not reported at level 1, including stack overflow heap overflow, trail overflow, etc., are also reported.

In a project, it is the error-level option in each module that controls that module's detail of saving the cursor information. If, however, the error-level option in the main module is higher than that of the submodules, Visual Prolog might generate misleading error information. For example, if an error occurs in a module compiled with error level 0, which is included in a main module compiled with error level 1 or 2, the system will be unable to show the correct location of the error. Instead, it will indicate the position of some previously executed code.

heap

Note: this is relevant only if you are going to implement a DOS TSR program.

Corresponds to the VDE's option **Options | Project | Compiler Options | Heap Size**. The `heap` directive specifies how much memory your .EXE file should allocate when it is started from DOS. If you don't use the heap directive, or if you set it to the value 0, the program will allocate all available memory. This is normally the right way to do it, but if you want to implement a RAM-resident Visual Prolog program, your program should only allocate the necessary memory. The format is:

```
heap = Number_of_paragraphs
```

nobreak

Note: this is only relevant for DOS textmode programs.

Corresponds to the VDE's option **Options | Project | Compiler Options | Break Check**. In the absence of the `nobreak` compiler directive, the Visual Prolog system will generate code to check the keyboard before each predicate call, to ensure that the **Ctrl+Break** key combination has not been pressed. This slows down program execution slightly and takes up a little extra program space.

The `nobreak` directive prevents this automatic generation of code. When nobreak is in operation, the only way to escape an endless loop is to reboot the computer (DOS, PharLap) or kill the process in some other way. `nobreak` should only be used after a program has been thoroughly tested.

nowarnings

Corresponds to the VDE's option **Options | Project | Compiler Options | Unused Variables**. The `nowarnings` directive suppresses the warnings given when a variable occurs only once in a clause.

If a variable occurs only once, either it is a mistake or it should be replaced by the anonymous variable - or a variable starting with an underscore.

printermenu

Note: this is only relevant for DOS textmode programs.

Corresponds to the VDE's option **Options | Project | Compiler Options | Print menu in DOS .EXE**. When this compiler directive appears in the program, Visual Prolog will place code in the executable file for handling the Alt-P key. This means that the user will be able to send screen output to the printer or capture it to a log file.

project

The `project` compiler directive is used in modular programming. All Visual Prolog modules involved in a project need to share an internal symbol table. If the project is called MYPROJ, the symbol table will be placed in a file called MYPROJ.SYM. The project directive must appear on the first line of a module to specify which project that module belongs to. For example, the following line of code states that a module belongs to the MYPROJ project:

```
project "myproj"
```

The project name is not allowed to have a path or an extension.

If the name of the .SYM file is given in the VDE with the option **Options|Project|Compiler Options|.SYM File Name** or with the command line compiler option (**-r**<*ProjectName*> or **–M**<*SymFileName*>), the **project** compiler directive will be ignored.

See page 252 for more details about modular programming.

Visual Prolog Memory Management

Visual Prolog uses the following memory areas:

Stack the stack is used for transferring arguments and return addresses for predicate calls. The stack also holds the information for backtrack points.

Heap the heap holds all objects that are more or less permanent, such as database facts, window buffers, file buffers etc.

GStack the global stack, normally called GStack, is the place where lists, compound structures and strings are placed. The GStack is only released during backtracking.

Trail The trail is only used when the program uses reference variables. It holds information about which reference variables must be unbound during backtracking. The trail is allocated in the heap.

Memory Restrictions

If you get a Memory overflow error, either correct your program or increase the size of the correspondent memory area.

Stack Size

- On 32-bit Windows platforms, the linker should specify the total stack allocation in the application virtual memory (Stack Size). This determines the maximal possible Stack size for the process. This reservation specifies the continuous range of addresses, in the 2GB virtual address space of the processes, which is reserved for the Stack. Without this type of protection, operations like loading DLLs could occupy Stack addresses and jeopardize availability for later use of them for stack needs. Notice that the reserved Stack Size cannot be increased dynamically by running processes; therefore, if a process attempts to allocate for stack more memory than the reserved Stack Size, then the memory error is generated. The default Stack size is 1 MB. This value can be changed with VDE's option **Options | Project | Compiler Options | Miscellaneous | Stack Size**. This VDE option is equivalent to "`-s[StackSize]`" PDC linker option. The **STACKSIZE** definition file directive (if specified) overwrites this option.

  Notice that such addresses reserving in a process virtual memory, does not allocate physical memory pages and no space is reserved in the paging (swap) file. It is simply saving a free address range until needed by Stack, protecting

the addresses from other allocation requests. Therefore, since no resources are allocated during the operation, this is a quick operation, completely independent of the size of the virtual address range (whether a 500MB or a 4K) being reserved.

Also notice that, this reserving of a range of virtual addresses for Stack does not guarantee that at a later time there will be physical memory available to allocate to those addresses. While a program execution the Stack will dynamically expand to the physical memory available from OS.

- In the 16-bit protected mode versions (16-bit Windows, PharLap DOS extended) 64 Kbytes will always be allocated for the stack.
- For plain DOS programs the default stack size is 600 paragraphs (16-byte units). You can increase the default stack size using Visual Prolog VDE option Options | Project | Compiler Options | Stack Size or with the "-S[Value]" compiler option.
- In the UNIX version, all stack allocation and sizing is handled automatically by the operating system. The virtual stack can grow almost without limit.

GStack Size

- For 32-bit Windows targets, Visual Prolog compiler should specify the total amount of addresses reserved for the Global Stack (GStack) in a process virtual memory. This determines the maximal possible GStack size for the process. The default GStack size is 100 MB. This value can be changed with VDE's option **Options | Project | Compiler Options | Miscellaneous | GStack Size**. This VDE option is equivalent to "`-k[GStackSize]`" command line compiler option. The **gstacksize** compiler directive (if specified in the text of the main program module) overwrites this option. The minimum valid value for this option is 128KB.
- Notice that, as in the case of Stack size, such addresses reserving in a process virtual memory does not allocate physical memory pages and no space is reserved in the paging (swap) file. Therefore, this is a quick operation completely independent of the specified GStack range of the virtual address range, but also, this reserving does not guarantee that at a later time there will be physical memory available to allocate to those addresses.
- In 16-bit applications, GStack can be dynamically expanded to the memory available. Notice that at once no more than 64 KB can be allocated.

Heap Size

- The **heap** compiler directive can be used for DOS real mode terminate and stay resident programs to specify the Heap Size. In all other cases, the Heap will dynamically expand to the memory available. Under 32-bit platforms, all addresses from the process virtual address space that are not reserved for the Stack and Gstack are available to the Heap.

  Under 16-bit platforms, at once no more than 64 KB can be allocated.

Releasing Spare Memory Resources

During program execution, the memory is used for several different purposes; depending upon the purpose, spare memory resources can be released in separate ways.

- To minimize stack use, avoid unnecessary non-determinism; use the `check_determ` directive to guide the setting of cuts. Also, take advantage of tail-recursion elimination by writing your predicates so they are tail-recursive.
- The global stack is used for building strings and structures. In order to save global stack space, write your program so that the outer loop is a **repeat...fail** loop. The ***fail*** predicate effectively releases unused GStack. More flexible releasing of the GStack can be done by ***mem_MarkGStack*** and ***mem_ReleaseGStack*** predicates.
- The trail will seldom be a problem in Visual Prolog. In all versions of Visual Prolog, the trail is dynamically allocated (in the heap), and will be increased in size when necessary. However, in the 16-bit versions the trail is limited to 64K and the first thing to do if the system complains about trail overflow is to avoid using reference domains. If you want to use reference domains, you should decrease the number of backtrack points by using some cuts (use check_determ). The **repeat...fail** combination will also release the trail. As a last resort, rearrange your predicate calls so that you create less reference variables.
- The heap is used when facts are inserted in fact databases and to store window buffers, file buffers, etc. These areas are automatically released when facts are retracted, windows are closed, and so on.

CHAPTER 18

Interfacing with Other Languages

Although Visual Prolog is an excellent tool for many purposes, there are still reasons to use other languages. For example, it's easier to perform numeric integration in C, and interrupt-handling and low-level stuff is perhaps better done in Assembly language. Moreover, if you've developed a large program in another language that already solves some aspect of the problem, this work should not be wasted. For these reasons, Visual Prolog allows you to interface your programs with other languages, as long as those languages produce standard object files and follows the conventions outlined in this chapter.

In this chapter you will find a number of examples of interfacing C and Visual Prolog. Their source files are in the DOC\EXAMPLES directory or the FOREIGN directory of your distribution. In order to run them, you need to have the appropriate development system and/or C compiler and libraries installed.

The process to compile and link the examples varies considerably between the different operating systems and the different C compilers. In the *foreign* subdirectory of your distribution you will find thorough instructions and examples for the different platforms and compilers. Read these instructions carefully.

When using the Visual Prolog Development Environment, you don't, strictly speaking, need to know how to compile C programs, how to run the linker, or which libraries to specify and how to do it. This is handled automatically. However, you should have a fairly thorough understanding about C, and be able to write, compile and link multi-module C programs yourself.

Using DLL's

A dynamic-link library (DLL) is a binary file that acts as a shared library of predicates that can be used simultaneously by multiple applications.

Visual Prolog can generate DLL's and link in DLL's staticly or load DLL's dynamicly. For more information about Visual Prolog and dll's please see the examples VPI\EXAMPLES\DLL and VPI\TOOLEXAMP\BUILD.

Calling Other Languages from Visual Prolog

In this section, we cover what you need to know to call C, Pascal and assembler routines from Visual Prolog.

Before calling routines and functions written in other languages, you need to declare them as external predicates in Visual Prolog. You also need to understand the correct calling conventions and parameter-pushing sequences, and you need to know how to name the different flow variants of your external predicates.

Declaring External Predicates

To inform the Visual Prolog system that a given global predicate is implemented in another language, you need to append a language specification to the **global predicates** declaration, as briefly mentioned in chapter 17:

```
GLOBAL PREDICATES
   add(integer,integer,integer) - (i,i,o),(i,i,i) language c
   scanner(string,token) - (i,o) language pascal
   triple(integer,real)  - (i,o) language asm
```

In Visual Prolog, you explicitly list the interfaced language; this simplifies the problems inherent in calling conventions, such as activation record format, naming convention and returning conventions.

Calling Conventions and Parameter Passing

The 80x86 processor family gives programmers a choice between NEAR and FAR subroutine calls, when running 16-bit programs. Visual Prolog requires all global routines to be FAR. The same applies to pointers to data objects. Many 16-bit compilers for the 80x86 family require you to choose between 16-bit and 32-bit pointers, where the 16-bit pointers refer to a default segment. In order to access all of memory, Visual Prolog always uses 32-bit pointers.

For 32-bit programs, "NEAR" means 32 bits and the above considerations are irrelevant.

Input parameters

For input parameters, the value is pushed directly, and the size of the parameter depends on its type.

Output parameters

An output parameter is pushed as a 32-bit pointer to where a value must be assigned.

Return Values

Visual Prolog follows the most widely adopted register convention for function values on the 80x86 CPU family. This should not be of any concern in most cases, but is included here for completeness.

Table 18.1: Registers for Return Values

Operand Size	Program Type
16 bit	32 bit
byte (8 bits)	AL
word (16 bits)	AX
dword (32 bits)	DX:AX [3] EAX

Pointers are 32 bits in size and are handled as dwords. The Program Type is determined by the operating system,

Floating point values are exceedingly troublesome to handle. They may be returned in registers, on the (emulated) coprocessor stack, and the pascal calling convention will frequently return them through pointers. Currently pascal functions cannot return floating point values. See the notes in the FOREIGN subdirectory of your distribution for any special considerations for your platform.

In any case, floating point values can always be returned in arguments. However, take special note that Visual Prolog's ***real*** corresponds to a C **double** (8 bytes).

You should also be aware that currently external C functions cannot return C structs (but they may of course return pointers to structs).

Multiple declarations

In Visual Prolog, a predicate can have several type variants, arities, and flow variants, and a separate procedure is needed for each type and flow variant. When you implement predicates, having several versions, in C, each C function must have a name corresponding to the name generated by Visual Prolog. The naming convention used by Visual Prolog is straightforward; the predicate name is used as the root, and the suffix _X is appended to signify the variant number,

where *X* is an integer starting at 0. If there is only one variant, no suffix is appended.

Consider the following program:

```
GLOBAL PREDICATES
   add(integer,integer,integer) –
      (i,i,o),(i,o,i),(o,i,i),(i,i,i) language c
   square(integer,integer) - (i,o)

GOAL
   add(2,3,X), write("2 + 3 = ",X), nl,
   add(2,Y,5), write("5 - 2 = ",Y), nl,
   add(Z,3,5), write("5 - 3 = ",Z), nl,
   add(2,3,5), write("2 + 3 is 5"), nl,
   square(5,Sq), write("5 squared is ",Sq).
```

A module linked with this program should contain the following C functions:

add_0 for the first flow pattern `(i,i,o)`

add_1 for the `(i,o,i)` flow pattern

add_2 for `(o,i,i)`

add_3 for `(i,i,i)`

square

As an example, the following C module implements *square* as well as all flow patterns for ***add****:*

```
add_0(int x, int y, int *z)                /* (i,i,o) flow pattern */
{ *z = x + y; }

add_1(int x, int *y, int z)                /* (i,o,i) flow pattern */
{ *y = z - x; }

add_2(int *x, int y, int z)                /* (o,i,i) flow pattern */
{ *x = z - y; }

add_3(int x, int y, int z)                 /* (i,i,i) flow pattern */
{ if ( (x + y) != z ) RUN_Fail(); }

square(int i,int *i_sq)
{ *i_sq = i*i; }
```

Parameter pushing order

When interfacing to a routine written in C, the parameters are pushed onto the stack in reverse order and, after return, the stack pointer is automatically adjusted by Visual Prolog.

When calling languages other than C, the parameters are pushed in the normal order, and the called function is responsible for removing the parameters from the stack.

Leading underscored

On the 16-bit platforms, C compilers will prefix the name of public C functions with an underscore. Therefore, global predicates declared as `language c` will also have their names prefixed with an underscore if the target platform is one of these.

32-bit Windows naming convention

Win32 (Windows 95/98/NT/2000) API functions use the special C calling convention **__stdcall**. The **__stdcall** uses the following name-decoration rules:

- the function name is prefixed with an underscore '_' and
- the function name is suffixed with the **@NN**, where **NN** is the number of bytes in the argument list.

This means that a function ***fName***, which has two integer arguments, should have the object name ***_fName@8***.

When the Visual Prolog compiler generates object files for Windows32 Target Platform, specifying the **language stdcall** in a global predicate (or global predicate domain) declaration, you can specify that the predicate must have the same calling convention as **__stdcall** in C.

Therefore, it is recommended to use **language stdcall** declaring global predicates for calling 32-bit Windows API functions.

Notice that when **language stdcall** is specified, then the compiler provides the advanced handling for the explicit specification of predicate names in object code `as "object_name"`.

- The compiler checks and (if needed) adds/corrects the leading underscore and
- The compiler adds/corrects (if needed) the suffixing '**@**' with number of bytes pushed in the argument list.

For instance, with the following declaration of the global predicate to call the Win32 API function ***sndPlaySoundA***:

```
sndPlaySound(STRING, UNSIGNED)
   - (i,i) language stdcall as "sndPlaySoundA"
```

the compiler generates the object name ***_sndPlaySoundA@8***.

Notice that when a predicate has several variants (for example, several flow patterns), then the standard naming convention for multiple declarations is used. The predicate name is used as the root, and the suffix `_N` is appended to specify the variant number. For example, with the declaration:

```
pName(integer) - (i), (o) language stdcall
```

the compiler generates the following object names ***_pName_0@4*** for the first `(i)` flow pattern and ***_pName_1@4*** for the second `(o)` flow pattern.

Converting the name to Uppercase (Pascal)

PASCAl uses the convention, that the name is converted to uppercase. So if language **PASCAl** is used, the name will during .OBJ module generation be converted to uppercase.

Adjustment of stackpointer

There are two possibilities of adjusting the SP register. This can be done either by the called function or the calling function. Traditionally PASCAL does this in the called function, while C does it in the calling function.

Table 18.2: Calling conventions

	Convert name to upper case	**Add leading under Score**	**Push args Reversed**	**Adjust SP after return**	**NT naming convention**
pascal	X				
c		X	X	X	
stdcall		X	X		X
syscall			X	X	

The AS "external_name" Declaration

As an alternative to the automatic naming convention, you can use the **as** keyword in the global declaration, like this:

```
GLOBAL PREDICATES
   scanner(string,token) - (i,o) language c as "_myscan"
```

The result of this is that Visual Prolog will refer to the name *_myscan* in the object file instead of *_scanner*. You would still refer to the name *scanner* in your Visual Prolog source.

You can only use the **as** option if there is a single flow variant for the predicate.

Domain Implementation

Most types normally used in C form a subset of Visual Prolog domains, and hence have direct equivalents. Below we discuss both simple and complex domain equivalents in C.

Simple Domains

The implementation of Visual Prolog's simple domains are outlined in the following table:

Table 18.3: Visual Prolog Simple Domains

Domain	**Implementation**	
	16-bit OS	32-bit OS
char, byte	1 byte (see note)	1 byte (see note)
(u)short, word	2 bytes	2 bytes
(u)long, dword	4 bytes	4 bytes
unsigned, integer	2 bytes	4 bytes
real	8 bytes (IEEE format)	8 bytes (IEEE format)
ref	4 bytes	4 bytes

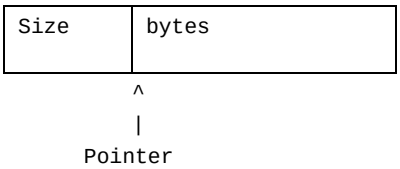
Note: The ***char*** and ***byte*** domains occupy a machine word when pushed on the stack (2 bytes for 16-bit programs, 4 bytes for 32-bit programs).

Complex Domains

All non-simple domains are implemented as pointers to things.

The ***string*** and ***symbol*** domains are pointers to null-terminated character arrays, with ***symbols*** being hashed and stored in the symbol table in the heap.

The ***binary*** domain is a pointer to a block of memory, prefixed by a dword (32bit platforms) or word (16bit platforms) indicating the net size of the block.



Special consideration must be given to allocation of memory for complex domains. This will be detailed in a later section.

Ordinary Compound Objects and Structures

User-defined compound objects are pointers to records (structs and unions in C). The general format of these is a byte representing the functor (domain alternative), followed by the individual components of the term. These will vary, depending on which alternative we're dealing with. In any case, components belonging to simple domains are stored directly in the term record itself, while complex components are themselves stored as pointers. For example, in this code fragment:

```
DOMAINS
   mydom = i(integer); c(char); s(string)
```

the functor number will be 1 for the first alternative, *i(integer)*, 2 for the second, *c(char)*, and 3 for the third.

A suitable C **typedef** for *mydom* would be:

```
typedef struct {
   unsigned char func;
   union {
      int i;
      char c;
      char *s;
   } u;
} MYDOM;
```

Here *func* will have the value 1, 2 or 3, depending on which domain alternative we're dealing with. This then indicates which of the union's components it's appropriate to access.

Functorless Terms (structs)

By prefixing a compound domain declaration with the compiler directive ***struct***, the terms belonging to that domain will not carry functors with them:

```
DOMAINS
   rec = struct record(d1,d2,...)
```

Apart from the ***struct*** directive, terms belonging to a functorless domain are used and written exactly like other terms in your program, except that there can be no alternatives in a functorless domain.

Functorless terms allow you to duplicate C structs when interfacing to C routines and libraries using predefined structs. Apart from that, they'll save you a bit of memory if you don't need alternatives in the domain.

Lists

Lists are implemented exactly like ordinary compound domains, with a field at the end of the record pointing to the next. This is known as linked lists in C terminology. From a Prolog perspective lists are merely a notational convenience. Given for instance a declaration for a list of strings:

```
DOMAINS
   strlist = string*
```

the C structures relevant for this are identical to those for:

```
DOMAINS
   strlist = elem(string,strlist); endoflist()
```

The records for the *elem* alternative will contain:

1. a functor
2. a pointer to a string
3. a pointer to the next element in the list

and the record for the *endoflist* alternative will only contain a functor.

This collection of fields is reflected in the C data structure for ***strlist:***

```
struct node {
   unsigned char functor;                              /* The type */
   char *value;                                  /* A string pointer */
   struct node *next;                    /* A pointer to struct node */
} strlist;
```

The *functor* field indicates the type of list record. The value is 1 if it's a list element, and 2 if it's the end of the list.

Memory Considerations

While all memory considerations are handled automatically when you write pure Prolog code, you need to take special care when interfacing to foreign languages. In this section we'll describe several of these aspects.

Memory Alignment

C compilers for 32-bit platforms will usually align data on dword boundaries, while those for 16-bit platforms will usually align on byte boundaries. The reason

for aligning on word or dword boundaries is speed. On a 32-bit platform, dword alignment will give up to 10-12 percent faster execution than byte alignment.

For simple variables alignment isn't important, but in order for Prolog terms and C structures to be compatible, the data contained in the records must be identically aligned. To this end, Visual Prolog gives you the option of selecting a different alignment than what's the default for your platform. The default is dword if you're using a 32-bit version of Visual Prolog, otherwise byte. Notice that some C compilers for 32-bit platforms (for example, Visual C++) by default align data at 8 Bytes; therefore, you should correct this to 4 Bytes to be compatible with DWord alignment of Visual Prolog.

The alignment scheme may be selected with the help of the **Options | Project | Compiler Options** menu item, or through the -A command line option. Additionally, the ***align*** compiler directive may be used to override alignment on selected domains, like this:

```
DOMAINS
   dom = align { byte | word | dword } func(d1,d2,...) [; func1(...);
... ]
```

The ***align*** directive must appear before any alternatives in the domain, and all alternatives will have the alignment specified. It's not possible to specify different alignment for individual alternatives.

For functorless terms, the ***align*** directive should appear after the ***struct*** directive.

Note that when several processes share a database or communicate over pipes, it's crucial that the domains involved use identical alignment.

Example

Byte alignment is easy: each element is simply put right after the previous one. Given the declaration `dom = struct my_struct(char,short,char,long)` (recall that the ***struct*** directive declares the term to be functorless), the term `my_struct('P',29285,'B',1702063209)` is stored in memory like this:

```
Byte number:

      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |


       _____________________________
      |   |       |   |               |
      |'P'| 29285 |'B'|  1702063209   |
      |___|_______|___|_______________|
```

Word and dword alignment is a bit trickier. Here, items are stored in memory so that accessing them won't cross a word or dword boundary. That means that the individual elements of terms may be followed by a number of unused bytes, depending on the size of the following element. With dword alignment, the term above would be stored like this:

```
Byte number:

      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|


       _____________________________________________
      |   |   |       |   |   |   |   |               |
      |'P'|PAD| 29285 |'B'|PAD|PAD|PAD|  1702063209   |
      |___|___|_______|___|___|___|___|_______________|
```

The PADs indicate unused bytes, allowing the values following them to be stored on suitable boundaries.

Notice that it's sufficient for the value 29285 to be aligned on a word boundary, because it's a ***short*** (16 bits); accessing it on a word boundary won't cross any undesirable boundaries.

Memory Allocation

When you create and return compound objects to Visual Prolog, memory for the objects must normally be allocated on the Global Stack. This memory will automatically be released if you fail back to a point previous to its allocation. GStack memory is allocated using:

```
void *MEM_AllocGStack(unsigned size);
```

You would typically use C's sizeof function to determine how much memory to allocate. Given for instance the *mydom* domain discussed previously, the Prolog declarations for a C routine returning a term belonging to that domain in an argument would be:

```
                        /* Program mydom_p.pro */

project "mydom"

global domains
   mydom = i(integer); c(char); s(string)

global predicates
   determ make_mydom(mydom) - (o) language C
```

```
goal
   make_mydom(MD), write(MD), nl.
```

And the C code for *mydom* and *make_mydom* could be:

```
                          /* Program mydom_c.c */

   typedef struct {
     unsigned char func;
     union {
       int i;
       char c;
       char *s;
     } u;
   } MYDOM;

   void *MEM_AllocGStack(unsigned);
   char *MEM_SaveStringGStack(char *);

   void make_mydom(register MYDOM **md)
   {
      *md = MEM_AllocGStack(sizeof(MYDOM));
      (*md)->func = 3;
      (*md)->u.s = MEM_SaveStringGStack("wombat");
   }
```

Notice that, as terms are handled through pointers in Prolog, the argument to *make_mydom* is a pointer to a term pointer. This example also makes use of another GStack-related function, ***MEM_SaveStringGStack,*** which allocates GStack space for the string (based on its length), then copies the string into the allocated space, returning a pointer to it. There are some other handy functions in Visual Prolog's library:

```
   char *MEM_SaveStringHeap(char *String);
   /* Copies String to heap */

   unsigned STR_StrLen(char *String);
   /* Returns length (excluding terminating null byte) of String */

   void MEM_MovMem(void *Source,void *Dest,unsigned Len);
   /* Moves Len bytes from Source to Dest; these may overlap */
```

Pre-allocation of Memory

Many C library functions require you to specify a pointer to a structure, which the C routine then fills in. In this case the compound flow pattern for global predicates should be used to specify what's happening:

```
GLOBAL DOMAINS
off_t, time_t = long
   dev_t = short
   stat = struct stat(dev_t,ushort,ushort,short,ushort,ushort,
                      dev_t,off_t,time_t,time_t,time_t)

GLOBAL PREDICATES
   determ integer stat(string,stat) -
                              (i,stat(o,o,o,o,o,o,o,o,o,o)) language C
```

When you call *stat*

```
..., 0 = stat("/unix",Stat), !, write(Stat).
```

Visual Prolog will allocate memory for the *stat* structure before the call.

The sizeof function

Visual Prolog has a sizeof function that duplicates C's sizeof function, returning the size of the specified domain or variable.

For a compound ***domain*** with alternatives, sizeof will return the size of the largest alternative. Given a second argument of a functor from one of the domain alternatives, sizeof will return the size of that particular alternative.

Given a variable, ***sizeof*** will return the size of the corresponding domain (alternative), except that for string variables or constants, ***sizeof*** will return the number of bytes in the string including the terminating zero byte.

The program ALIGN.PRO illustrates alignment selection and the use of the ***sizeof*** function:

```
   /* Program align.pro */

DOMAINS
   dom = struct f(char,integer)
   dom1 = align word f(integer,integer,long); g(string)
   refint = reference integer

predicates
   refint(refint)

clauses
   refint(_).

goal                              % Find the size of a functorless domain
   A = sizeof(dom),
   write("\nSize=",A),
```

```
% when there are alternatives, the largest is returned
   B = sizeof(dom1),
   write("\nSize=",B),

% Find size of a single alternative
   C = sizeof(dom1,g),
   write("\nSize=",C),

% Find size of a term pointed to by a variable
   X = f(1,1,1),                                    % This is from dom1
   D = sizeof(X),
   write("\nSize=",D),

% Find size of a string pointed to by a variable
   Y = "hello there",
   E = sizeof(Y),
   write("\nSize=",E),

% Find size of a reference variable
   refint(Z),
   F = sizeof(Z),
   write("\nSize=",F).
```

Load and run this program. Try changing the domains and their alignment, and watch the results.

malloc and free

When writing functions in other languages, you often need to allocate dynamic memory. You've already seen ***MEM_AllocGStack,*** but this allocates memory on Prolog's Global Stack, which is released automatically. Permanent allocations should be done in the heap, and because Visual Prolog already has suitable memory allocation routines, it's generally preferable to use these. In fact, in DOS it's mandatory to use them, since a foreign memory allocation package would be allocating memory from the same physical memory as Visual Prolog. On other platforms, you can use C's **malloc** and **free,** but this would duplicate an amount of code and data, and the two packages would both be holding released memory in separate pools. Moreover, Visual Prolog's heap allocation system has a performance far superior to that supplied with most C compilers.

Therefore, on all platforms except UNIX, public routines for **malloc** and **free,** consisting of bindings to Visual Prolog's heap allocation routines, are provided in the initialization assembler and object files. These files are found in the subdirectories for the different platforms and compilers in the FOREIGN directory of your distribution. Note that when linking, it's essential that the

appropriate initialization file appears before the (C) library containing **malloc** and **free.**

Examples

List Handling

In this section we give a more useful example that shows how to convert a list to an array and back to a list again.

The C routine **ListToArray** takes a list of integers, converts this to an array placed on the Global Stack, and returns the number of elements. The conversion is done in three steps:

1. The list is traversed in order to count the number of elements.
2. The array with the needed number of elements is allocated.
3. The list is traversed again while the elements are transferred to the array.

The C routine **ArrayToList** takes an integer array and the size of the array as arguments, then converts these to a list of integers. This routine only makes one pass, building the list as it indexes through the array.

All of this is used in the C-coded predicate ***inclist.*** When given a list of integers, ***inclist*** first converts the input list to an array, increments the elements of the array by 1, then converts the array back to a list of integers.

```
                        /* Program lstar_p.pro */

project "lstar"

global domains
   ilist = integer*

global predicates
   inclist(ilist,ilist) - (i,o) language c

goal
   inclist([1,2,3,4,5,6,7],L), write(L).
```

Here is the C program defining the two C procedures **ListToArray** and **ArrayToList,** and the external Visual Prolog predicate ***inclist.***

```
                        /* Program lstar_c.c */

   #define     listfno  1
   #define     nilfno   2
   typedef unsigned char BYTE;

   void *MEM_AllocGStack(unsigned);

   typedef struct ilist {
      BYTE Functor;
      int Value;
      struct ilist *Next;
   } INTLIST;

   int ListToArray(INTLIST *List,int **ResultArray)
   {
      INTLIST *SaveList = List;
      int *Array, len;
      register int *ArrP;
      register int i;

/* Count the number of elements in the list */
      i = 0;
      while ( List->Functor == listfno ) {
        i++;
        List = List->Next;
      }
      len = i;

   Array = MEM_AllocGStack(i*sizeof(int));
      ArrP = Array;

/* Transfer the elements from the list to the array */
   List = SaveList;
   while ( i != 0 ) {
     *ArrP++ = List->Value;
     List = List->Next;
     i--;
   }

      *ResultArray = Array;
      return(len);
   }
```

```
void ArrayToList(register int *ArrP,register int n,
                 register INTLIST **ListPP)
{
   while ( n != 0 ) {
     *ListPP = MEM_AllocGStack(sizeof(INTLIST));
     (*ListPP)->Functor = listfno;
     (*ListPP)->Value = *ArrP++;
     ListPP = &(*ListPP)->Next;
     n--;
   }
   *ListPP = MEM_AllocGStack(sizeof((*ListPP)->Functor));
                                                  /* End of list */
   (*ListPP)->Functor = nilfno;
}

void inclist(INTLIST *InList,INTLIST **OutList)
{
   register int *ArrP, i, len;
   int *Array;

   len = ListToArray(InList,&Array);
   ArrP = Array;
   for ( i = 0; i < len; i++)
     ++*ArrP++;
   ArrayToList(Array,len,OutList);
}
```

This program belongs to the kind where memory alignment can be critical. If you intend to compile to several platforms, you're well advised to keep an eye on this. As a first step, check that the sizes of the structures shared by C and Prolog are the same; the padding applied when aligning on non-byte boundaries will make things a bit bigger. The ***sizeof*** function comes in handy here. You can write a small C function:

```
unsigned c_ilsize(void)
{
   return(sizeof(INTLIST));
}
```

returning the size of the *INTLIST* structure. This can then be used by a Prolog predicate to verify that the sizes of *INTLIST* and *ilist* are identical:

```
GLOBAL PREDICATES
   unsigned c_ilsize() language C
```

```
PREDICATES
   scheck

CLAUSES
   scheck:- ILSize = sizeof(ilist), ILSize = c_ilsize(), !.
   scheck:- write("ilist element sizes differ\n"), exit(1).
```

Calling Prolog from Foreign Languages

If you supply Prolog clauses for global predicates declared as being of foreign language, those predicates may be called from foreign languages. They will have parameter access and entry and exit code, including register preservation, as for the language specified.

Hello

This small project is hello-world, with a twist.

```
                        /* Program hello_p.pro */

global predicates
   char prowin_msg(string) - (i) language c
   hello_c - language c

clauses
   prowin_msg(S,C) :-
      write(S," (press any key)"), readchar(C).

goal
      prowin_msg("Hello from PDC Prolog"),
      hello_c.
```

The global predicate *prowin_msg* is now accessible from C and can be called just like any other C function:

```
                         /* Program hello_c.c */

   char prowin_msg(char *);

   void hello_c()
   {
     while ( prowin_msg("Hello from C (press 'C')") != 'C' )
       ;
   }
```

As is evident, values may be returned to foreign languages.

Standard Predicates

Most of Visual Prolog's ***standard*** predicates can be called from C, but their public names and exact functionality are subject to change without notice. It's therefore strongly recommended that you write a small set of interface routines if you want to call Visual Prolog standard predicates from C. The following illustrates bindings to a number of Visual Prolog's DOS Textmode I/O predicates:

```
                        /* Program spred_p.pro */

project "spred"

global predicates
   myfail language c as "_fail"
   mymakewindow(integer,integer,integer,string,integer,integer,
                integer,integer)
                           - (i,i,i,i,i,i,i,i) language c as "_makewindow"
   myshiftwindow(integer) - (i) language c as "_shiftwindow"
   myremovewindow language c as "_removewindow"
   write_integer(integer) - (i) language c as "_write_integer"
   write_real(real) - (i) language c as "_write_real"
   write_string(string) - (i) language c as "_write_string"
   myreadchar(char) - (o) language c as "_readchar"
   myreadline(string) - (o) language c as "_readline"

   extprog language c

clauses
   myfail:- fail.

   mymakewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols):-
      makewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols).

   myshiftwindow(WNO):- shiftwindow(WNO).

   myremovewindow:- removewindow.

   write_integer(I):- write(I).

   write_real(R):- write(R).

   write_string(S):- write(S).

   myreadchar(CH):- readchar(CH).

   myreadline(S):- readln(S).

goal
   extprog.
```

These may be accessed freely by C, as illustrated by *extprog:*

```
                        /* Program spred_c.c */

void extprog(void)
   {
      char dummychar;
      char *Name;

      makewindow(1,7,7,"Hello there",5,5,15,60);
      write_string("\n\nIsn't it easy");
      readchar(&dummychar);
      write_string("\nEnter your name: ");
      readline(&Name);
      write_string("\nYour name is: ");
      write_string(Name);
      readchar(&dummychar);
      removewindow();
   }
```

Calling an Assembler Routine from Visual Prolog

You can also call assembler routines from Visual Prolog. The activation record is the same as for pascal (that is, parameters are pushed left to right), and the called routine should pop the stack itself. If you have a C compiler supporting inline assembler, things will be considerably easier than if you have to do everything yourself.

In any case there seems to be little point in using assembler since C handles most things, but a small example is included here for completeness. For obvious reasons, the code differs between 16 and 32 bit platforms.

Suppose you want to write a routine returning a 32-bit sum of the characters in a string, and also verifying that all characters are within a certain range, say A-Z.

The Prolog code for this could be:

```
                        /* Program csum_p.pro */

project "csum"

global predicates
   integer sum_verify(char,char,string,ulong) - (i,i,i,o) language asm

predicates
   uc_check(string)
```

```
clauses
   uc_check(S):-
      0 = sum_verify('A','Z',S,Sum), !,
      write('"',S,"\" OK, sum = ",Sum,'\n').
   uc_check(S):- write('"',S,"\" fails\n").

goal
   uc_check("UNIX"),
   uc_check("Windows").
```

where we have adopted the convention that a return value of 0 means the string was OK.

Here is the suitable 16-bit assembler code:

```
                        /* Program csum_a16.asm */

   ; 16-bit version

   CSUM_A16_TEXT  SEGMENT  WORD PUBLIC 'CODE'
   CSUM_A16_TEXT  ENDS
   _DATA          SEGMENT  WORD PUBLIC 'DATA'
   _DATA          ENDS
   CONST          SEGMENT  WORD PUBLIC 'CONST'
   CONST          ENDS
   _BSS           SEGMENT  WORD PUBLIC 'BSS'
   _BSS           ENDS
   DGROUP         GROUP    CONST, _BSS, _DATA
                  ASSUME  CS: CSUM_A16_TEXT, DS: DGROUP, SS: DGROUP

   CSUM_A16_TEXT      SEGMENT
      ASSUME          CS: CSUM_A16_TEXT

   PUBLIC         sum_verify
   sum_verify     PROC FAR
      push        bp
      mov         bp,sp

   lolim equ 16
   hilim equ 14
   string equ 10
   sum equ 6
```

```
   xor         dx,dx
   xor         bx,bx                              ; Do sum in dx:bx
   les         di,[bp+string]                    ; Pointer to string
   mov         cl,byte ptr [bp+lolim]              ; Low limit in cl
   mov         ch,byte ptr [bp+hilim]             ; High limit in ch
   xor         ax,ax

ALIGN 2
loopy:
   add         bx,ax                                          ; Add sum
   adc         dx,0
   mov         al,byte ptr es:[di]
   inc         di
   cmp         al,cl
   jb          end_check
   cmp         al,ch
   jbe         loopy

end_check:
   or          al,al
   jnz         go_home
   les         di,[bp+sum]
   mov         es:[di],bx
   mov         es:[di+2],dx
   inc         ax; ax: 0 -> 1

go_home:
   dec         ax                            ; ax: 1 -> 0, or 0 -> -1
   mov         sp,bp
   pop         bp
   ret         12
sum_verify     ENDP

CSUM_A16_TEXT  ENDS
END
```

When writing assembler code, take special care that the sizes of things on the stack follow the machine's natural word-size. This is 2 bytes on 16-bit machines and 4 bytes on 32-bit machines. A good first attempt is to compile a dummy C routine, with the correct parameters and local variables, to assembler, and then use the entry, exit, and variable access code generated by the C compiler.

It isn't necessary to preserve any of the usual registers when foreign language routines are called from Prolog, but if you're calling from C or assembler it's assumed that you preserve `si` and `di` (`esi` and `edi` on 32-bit platforms). On 32-bit platforms, `ebx` must also be preserved.

Index

—D—

—E—

—F—

—G—

—H—

—I—

—K—

—L—

—M—

—N—

—O—

—P—

—Q—

—R—

—S—

—T—

—U—

—V—

—W—