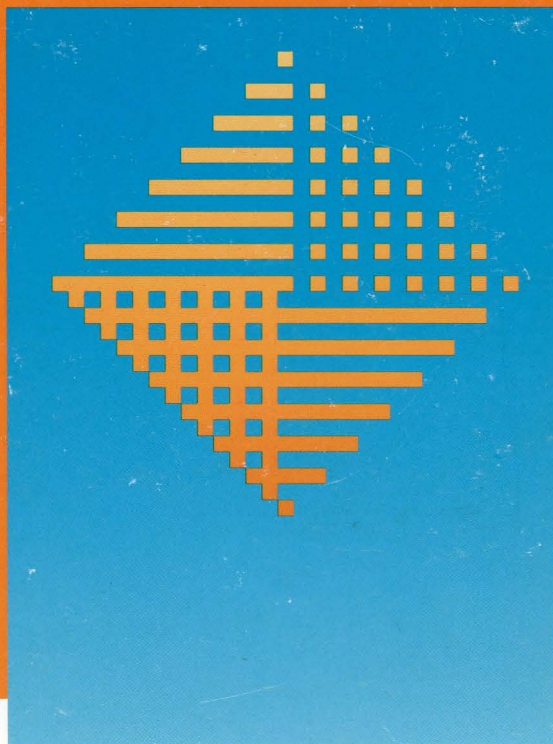
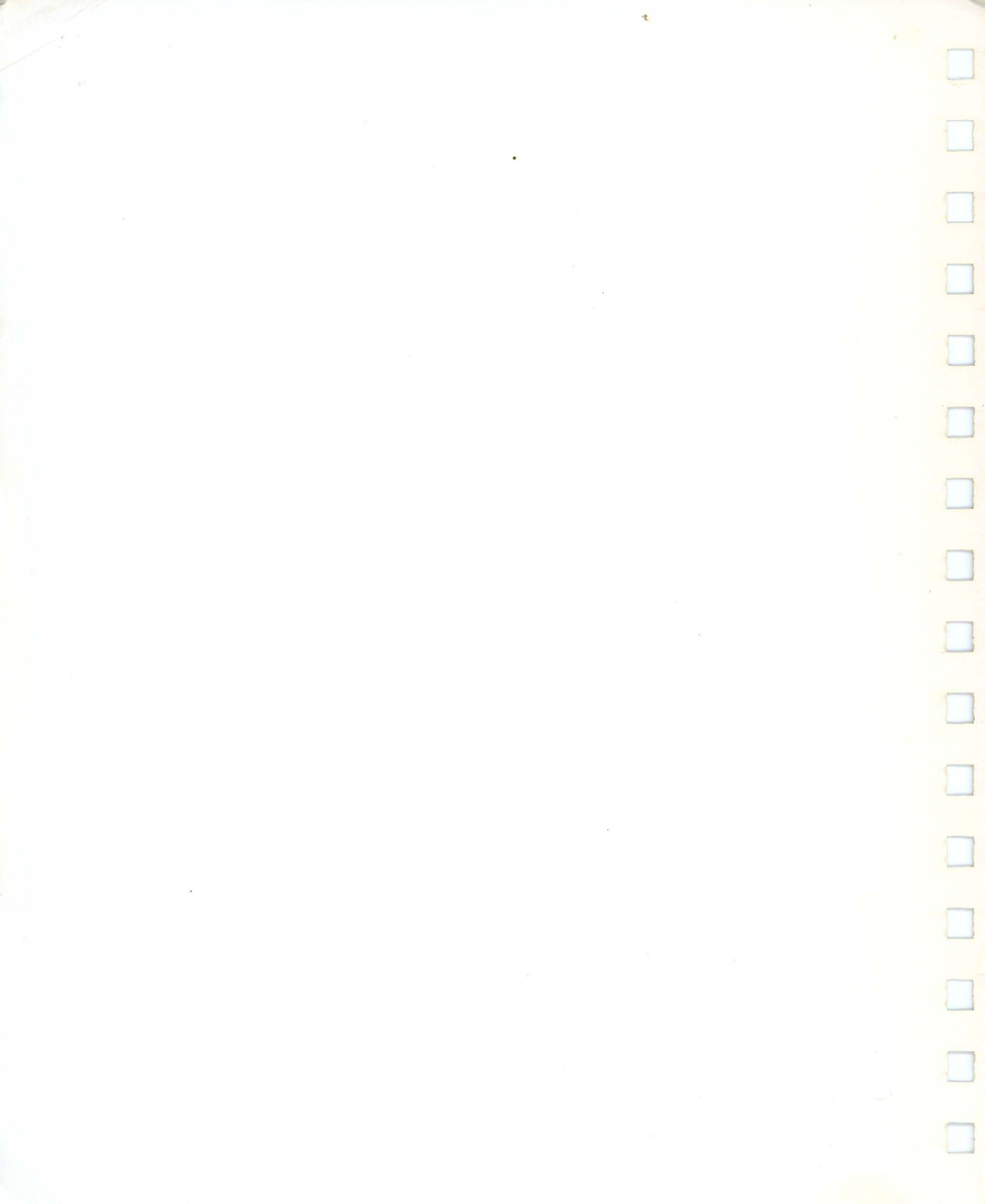


# BNR Prolog

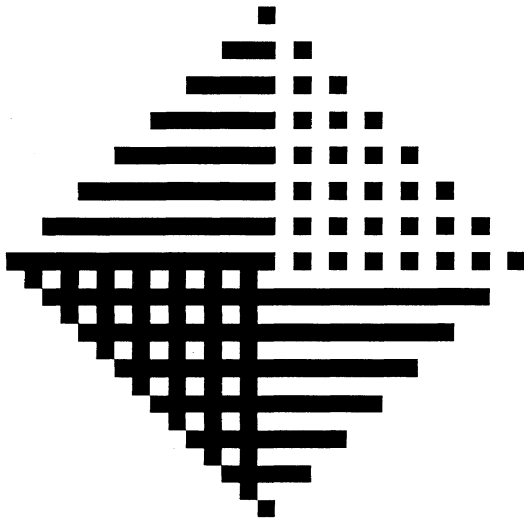
## User Guide











# **BNR Prolog User Guide**







# Table of Contents

---

<b>Part I Introduction</b>	<b>1</b>
<b>Chapter 1 BNR Prolog Product Information</b>	<b>3</b>
Inventory	3
System Requirements	3
Software	4
User Guide Overview	5
<b>Chapter 2 Logic Programming</b>	<b>7</b>
Prolog	7
Suggested Prolog References	8
<b>Chapter 3 The Prolog Model</b>	<b>11</b>
<b>Chapter 4 Using the BNR Prolog System</b>	<b>19</b>
 <b>Part II Prolog, a Programming Language</b>	 <b>29</b>
<b>Chapter 5 Pure Prolog</b>	<b>31</b>
BNR Prolog Terms	32
Unification	35
Call	39
Backtracking	40
Conjunction	47
Putting It Together	51
Characteristics of Pure Prolog Programs	53



<b>Chapter 6 Filters and Negation</b>	<b>59</b>
Monotone Filters	61
Persistent Filters	61
Constructed Filters	64
<b>Chapter 7 Passive Constraints</b>	<b>69</b>
Constraint Notation	70
Constraint Interpolation	72
Generalized Types	73
Sound Negation and Negative Knowledge	75
<b>Chapter 8 Control</b>	<b>81</b>
Eliminating Computation Branches	81
Directing Execution	83
Active Constraints	89
<b>Chapter 9 Operators</b>	<b>97</b>
Specifying an Operator	98
Predefined Operators	101
<b>Chapter 10 Cyclic Structures</b>	<b>103</b>
Cyclic Structure Support	103
 <b>Part III Arithmetic</b>	 <b>117</b>
<b>Chapter 11 Functional Arithmetic</b>	<b>119</b>
Evaluation	120
Extending Functional Arithmetic	121
Arithmetic with Constraints	125
<b>Chapter 12 Relational Arithmetic</b>	<b>131</b>
The Interval Type	133
Evaluating Interval Expressions	139



---

Using Relational Arithmetic	151
Summary	171
<b>Part IV Programming with Side Effects</b>	<b>173</b>
<b>Chapter 13 Text Input and Output</b>	<b>177</b>
Streams	178
Detecting I/O Errors	182
Read and Write Predicates	182
<b>Chapter 14 The Knowledge Base</b>	<b>189</b>
Contexts	190
Asserts and Retracts	197
<b>Chapter 15 State Spaces</b>	<b>201</b>
Global State Space	202
Local State Spaces	207
<b>Chapter 16 User Interfaces</b>	<b>211</b>
Events	213
Windows	219
Pictures	231
Menus	232
Dialogs	236
A Complete Program	238
<b>Part V Miscellaneous</b>	<b>245</b>
<b>Chapter 17 Foreign Language Interface</b>	<b>247</b>
Defining and Calling Externals	247
Writing an External Procedure	248
Pascal Examples	250

---



C Examples	257
<b>Chapter 18 System Information</b>	<b>261</b>
Application Structure	261
Monitoring the Environment	265
Building an Application	266
Managing Error Conditions	274
<b>Chapter 19 The Debugger</b>	<b>277</b>
The Box Model	277
Debugging a Program	280
Entering the Listener	287
Debugging Event Handlers	288
<b>Chapter 20 Prolog Compatability Issues</b>	<b>291</b>
Sequences	291
Sequences in Terms	296
Operators	302
<b>Appendices</b>	<b>305</b>
<b>Appendix A:</b>	<b>307</b>
🍏(Apple) menu	307
File menu	308
Edit menu	310
Find menu	311
Window menu	313
Contexts menu	315
Menu Command Shortcuts	317
Editing Keys	318
Execution Control Keys	319



Methods of Text Selection by Mouse	319
<b>Index</b>	<b>321</b>







# Part I Introduction







---

# Chapter 1

## BNR Prolog Product Information

---

BNR Prolog is a full featured implementation of Prolog for the Macintosh family of computers. It extends the power of earlier generations of Prolog by providing constraint management and relational arithmetic facilities that greatly enhance both the logical features of the language and its problem solving power.

### Inventory

The BNR Prolog package contains the following:

- BNR Prolog User Guide
- BNR Prolog Reference Manual
- Warranty / License
- Application Disk and Tutorial Disk
- Registration Card

### System Requirements

The BNR Prolog system runs on any Apple Macintosh computer with a minimum of 1 megabyte of memory. A system with a hard disk drive is recommended.

The versions of the Macintosh system software required are

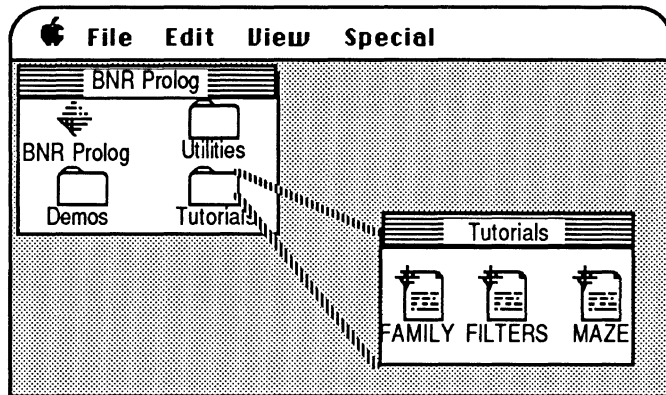
- version 4.1 or higher of the System file



- version 2.0 of the MPW Pascal , C, and assembly language interfaces (for those wishing to write their own externals)

## Software

The BNR Prolog disk should appear on the Macintosh desktop as



### BNR Prolog Release Disk

*BNR Prolog* is the BNR Prolog application

*Utilities* is a folder of files that provide extensions to the basic application

*Demos* is a folder of demonstration programs

*Tutorials* is a folder containing all Prolog examples and programs found in this user guide



---

## User Guide Overview

This document is intended as a guide to the use of BNR Prolog for users learning Prolog and for experienced users. New users are also encouraged to read one of the recommended textbooks on Prolog listed in the next chapter. The guide has been divided into five parts of related topics, and an appendix:

- Part I        discusses the philosophy of Prolog, and gives a short tutorial on the use of BNR Prolog.
- Part II       reviews the concepts of "pure" Prolog, describing deviations, and the ways in which filters and constraints can emulate the effects of pure Prolog. Operators and the support of cyclic structures are also described.
- Part III      provides detailed information about the use of arithmetic. Aside from functional arithmetic, which results in deviations from the pure Prolog model, intervals are provided as a foundation for relational arithmetic, allowing arithmetic relationships to be expressed logically.
- Part IV       describes methods of programming with side effects, another deviation from pure Prolog. This includes such topics as input and output, management of the knowledge base, state spaces, and support for effective user interfaces.
- Part V        provides additional information on developing, debugging and building applications using the BNR Prolog development environment. Compatibility with Edinburgh Prologs is also discussed.
- Appendix     summarizes the BNR Prolog desktop information, giving details of the menus and command shortcuts.



Users new to Prolog should read "Part I Introduction" and the chapter "Pure Prolog". All examples in this guide are provided on the tutorial disk to encourage experimentation. More advanced topics such as constraints and relational arithmetic may be left for later.

This user guide should be used in conjunction with the *BNR Prolog Reference Manual*. Whereas the user guide is designed to instruct and explain general concepts, the reference manual is organized to facilitate quick access to complete information on specific topics.

## Typographic Conventions

The following typographic conventions are used throughout this guide:

- |                |   |
|----------------|---|
| Typeface       | Examples are displayed in this typeface. It is used to display any text which appears on the screen or in a program listing. BNR Prolog built-in predicate names and their corresponding arguments also appear in this special typeface.  |
| <i>Italics</i> | Italics are occasionally used to highlight key words or concepts in the text, particularly the first time they are defined. As well, italics are used when specifying the use of certain keyboard characters, for example, press <i>return</i> , or when making explicit references to titles of books. |
| "Quote"        | Quotation marks are occasionally used to highlight words or statements in text. Quotation marks are also used when making cross-references to chapter titles.   |
| 'Symbol'       | The basic Prolog type symbol can be enclosed in either double or single quotation marks. Single quotation marks have been used with symbols throughout the text.  |



# Chapter 2

## Logic Programming

---

The cornerstone of logic programming is the idea that symbolic logic can be used as a programming language. Symbolic logic offers both a declarative method of expressing relations between objects, and through the automation of logical reasoning, a procedural method of interpreting these relations. Thus, logic programs are both specifications in the language of logic, and executable instructions for a computer.

The appeal of logic programming is that it frees programmers from the need to explicitly encode every task the computer must perform to solve a problem. Instead, explicit coding is replaced by a task better suited to human thinking: declaring one's thoughts in the language of logic.

### Prolog

Prolog is based on a subset of logic, and is currently the most successful implementation of a logic programming language. A Prolog program is a set of sentences that have the following structure:

```
P <- A1 & A2 & A3... & An
```

Declaratively this statement is read as P is true if all A1, A2, A3, ..., and An are true. This subset of logic, known as Horn clause logic, forms the *pure* or logical part of Prolog.

Extending pure Prolog to provide a practical programming language requires the introduction of language features that may result in "nonlogical" behavior. Nonlogical features in a logic programming language modify the declarative reading of programs and force the programmer to express his or her ideas



in a procedural way. Among these additions to pure Prolog are mechanisms for :

- controlling the flow of execution of a program
- communicating with input and output devices
- executing arithmetic efficiently
- manipulating Prolog programs

In recent years, several new techniques have been developed which isolate or remove the nonlogical effects of the additions to pure Prolog. Among these are logical constraints and relational arithmetic. In addition, some extensions of the BNR Prolog language permit programming tasks involving program manipulation to be logical. These features of BNR Prolog simplify many programming tasks and, in many cases, permit more efficient implementations.

## Suggested Prolog References

Readers who are new to Prolog may find it useful to accompany this user guide with one of the following text books:

- Bratko, I. *Prolog Programming for Artificial Intelligence*. Wokingham, England, Reading, Mass., Menlo Park, Calif., Don Mills Ont.: Addison-Wesley, 1986
- Clocksin, W. F., and Mellish, C. S. *Programming in Prolog*. 3rd ed. Berlin, Heidelberg, New York, London: Springer-Verlag, 1984.
- Covington, M. A., Nute, D., and Vellino, A. *Prolog Programming in Depth*. Glenview, Ill., London: Scott, Foresman and Company, 1988.



Pereira, F. C. N., and Shieber, S. M. *Prolog and Natural-Language Analysis*. CLSI Lecture Notes, 10. Stanford: Center for the Study of Language and Information, University of Chicago Press, 1987.

Sterling, L., and Shapiro, E. *The Art of Prolog*. Cambridge, Mass., London England: The MIT Press, 1986.







---

# Chapter 3

## The Prolog Model

---

The underlying model for conventional high level languages was established nearly three decades ago with FORTRAN, although many users of personal computers may be more familiar with this model through the medium of Basic. Examining some of the fundamental characteristics of FORTRAN helps to highlight the basic differences between Prolog and conventional languages.

FORTRAN distinguishes code from data. Data consists of numbers, while code consists of arithmetic expressions, assignment statements, and some control constructs. The link between code and data is the variable. A variable, which is referenced in arithmetic expressions, is the name of a cell holding changing data elements. Code is procedural in that it describes the step by step actions to be performed, and is written primarily as sequences of actions. Control constructs link the sequences.

The evolution of conventional languages, from FORTRAN through Algol and Pascal to Ada, has progressively enriched the repertoire of both control and data structures, but has not altered these basic characteristics. Some of the characteristic problems of conventional programs, such as variable name aliasing and side effects, are due to the variable concept.

This procedural model of computation is not the only one. At the time that FORTRAN was being formulated, a radically different model was proposed in the language LISP. In its pure form, LISP eliminates any intrinsic distinction between code and data, and dispenses entirely with variables. Instead, it focuses on a generic treatment of structures built from lists, a general notion of abstraction, and a mechanism of function evaluation. These are all coordinated by a powerful theoretical model. It is not



surprising that LISP and the conventional languages are seen as complementary, since one excels in precisely those areas where the other has the most difficulty.

Prolog arose from quite a different environment, the field of automatic theorem proving. Prolog has retained much of the flavor and the vocabulary of symbolic logic, but only recently has been recognized and developed as a useful programming language. As a consequence, Prolog employs two distinct vocabularies and interpretations, one based on the viewpoint of formal logic and the other on the art of programming. This dual nature may be a source of confusion for new Prolog programmers, but it is also one of Prolog's great strengths.

To describe the basic computational model of Prolog, we start with the schema

```
question + knowledge -> answer
```

Instead of the conventional distinction between code and data, we distinguish between questions and knowledge. Consider the example

```
question:      Was it cold on Saturday?
knowledge:      It was snowing on Saturday.
                It was cold if it was snowing.
answer:         It was cold on Saturday.
```

Although there is a distinction between questions and knowledge, the form of both is roughly the same, that of grammatically correct sentences. To be sure, in natural languages, grammatical details and inflections may vary between a question and the corresponding answer, but the essential relations expressed in them are the same.

The fundamental principle which led to Prolog is the realization that if one uses a somewhat austere relational language, then the forms of questions and statements of knowledge become identical. As a result, the process of combining the question and relevant knowledge becomes a mechanical operation. Thus, the previous example is rephrased formally as



```
question:      cold(saturday)?
knowledge:      cold(_x) :- snowing(_x).
                snowing(saturday).
answer:         cold(saturday).
```

The symbol ":-" means *if*.

```
cold(_x) :- snowing(_x).
```

means

```
cold(_x) is true if snowing(_x) is true.
```

The symbol `_x`, is a logic variable and is interpreted as *anything*. The expression `cold(_x)` is a *structure*: such structures are the basic units of information storage, and are analogous to records in Pascal. Sentences are called *clauses*, and consist of *rules* (`p :- q.`) and *facts* (`p.`).

The procedure known as inferencing matches the question/answer (*goal*) against the conclusion of each rule. This matching process, called *unification*, is described in more detail in the chapter *Pure Prolog*. The substitution of values for variables, called *instantiation*, plays the role of parameter passing in a conventional language. In this case

```
cold(saturday)
```

matches

```
cold(_x)
```

if `_x` is instantiated with `saturday`. Making this substitution everywhere in the rule produces the instance

```
cold(saturday) :- snowing(saturday).
```

which reduces the original question to

```
?- snowing(saturday).
```

Thus, each inference can be thought of as consisting of two steps: *unification* (matching) of the question against the rules in the



knowledge base, and *reduction*, which replaces the original question with a new one derived from the rule. The existence of such a rule in the knowledge base is a license to perform such an inference any number of times.

The reduction of a goal is generally composed of many inferences, each of which is analogous to a procedure call in a conventional language. The word *call* is therefore used to designate an inference. The reduction step, which makes the conditions of the rule the new question, is analogous to executing the body of a procedure in a conventional language. Thus, we refer to the condition(s) in a rule as the *body* of the rule.

One more inference step, this time matching

```
snowing(saturday)
```

with

```
snowing(saturday)
```

reduces the original goal to an empty question, thus terminating the process. If a sequence of calls leading to the empty question cannot be found, then we say that the question *fails*. Generally this means only that the knowledge base is unable to provide an answer to the question.

The answer is a consequence of the knowledge base, because it is the result of a finite number of calls. Furthermore, the answer is always an instantiation of the question; that is, it has the same form but may have some variables instantiated during the inference process. Thus the question, when sufficiently *narrowed*, becomes the answer. (It is convenient to regard the special case of a question that *fails* as an extreme case of narrowing.)

To express more complex knowledge, we must allow joint conditions to be associated with a conclusion. For example,

```
It was too cold for skiing  
if it was cold and the wind was gale force.
```



might be expressed as

```
too_cold_for_skiing(_x) :-  
    cold(_x), wind_force(gale, _x).
```

where the *","* represents *and then*. The precise conditions under which the more general *and then* interpretation reduces to the simpler *and* interpretation is discussed in detail later.

Intuitively one expects that a question might have alternative answers, all equally valid if not equally useful. This concept has no counterpart in conventional programming languages. To illustrate, consider a modified version of the `cold` example

```
question:      cold(_what_day)?  
knowledge:     cold(_x) :- snowing(_x).  
               snowing(saturday).  
               snowing(wednesday)  
answers:       cold(saturday).  
               cold(wednesday).
```

Each alternative answer is produced by the same inference procedure, but is the result of a different choice of which rule to apply at each step. In some cases there may even be, in principle, an infinite number of different answers.

A third characteristic of Prolog is the method, called *backtracking*, by which alternatives are explored. From a group of alternatives, one is chosen and pursued. If it fails, the system backtracks to the last choice made (undoing any instantiations made in the interim) and tries another alternative. The order in which alternatives are tried is determined by the order of the alternative clauses in the knowledge base.

Essentially, Prolog has two built-in general mechanisms for solving problems. One, *narrowing*, is efficient but relatively weak. The other, *exhaustive search*, is strong but inefficient. The art of Prolog programming is to combine these two mechanisms in a way that is both efficient and effective for a given class of problems. One strategy for combining the two is to emphasize *narrowing*. Narrow the problem as much as



possible, and then subdivide it. Alternatively, *exhaustive search* can be emphasized, with narrowing as a constraint on the search.

## Summary

The previous section introduced

- the question + knowledge → answer paradigm
- a uniform relational notation
- primitive inference step or call
- answers as narrowing of the question
- sequential conjunction
- alternative answers

These basic concepts are reflected in a number of specific implementation details.

The distinction between knowledge and goals is represented by dividing the system into two regions. The *clause space* holds the knowledge, while the *goal stack* holds questions and answers. The knowledge base in BNR Prolog can be structured into distinct layers or contexts, as discussed in the chapter "The Knowledge Base".

When not answering a question, the system waits for user input. The part of the system responsible for user interaction (key strokes, menu selections, mouse clicks) is called the *listener*.

Each sentence entered into the system by the user can be either a question or an addition to the knowledge base, and some convention is required to determine which case applies. The convention used is that all sentences end with a period, but questions begin with the question prefix "?-". For example,



---

```
?- cold(saturday).    % this is a question
cold(saturday).       % this is an addition
                      % to the knowledge base
```

Since questions are frequent, the listener supplies the "?-". In such cases, the question prefix need not be typed, providing the question is entered after "?-". The question/answer relationship is represented by echoing the question with the answer substitutions. If there is no answer, then NO is output. If there is more than one answer, then the first answer is displayed, and the system waits for input telling it to proceed as follows

- a semi-colon ";", generates the next answer if there is one
- a *return* prints all the answers without further prompting
- and any other key will terminate the question





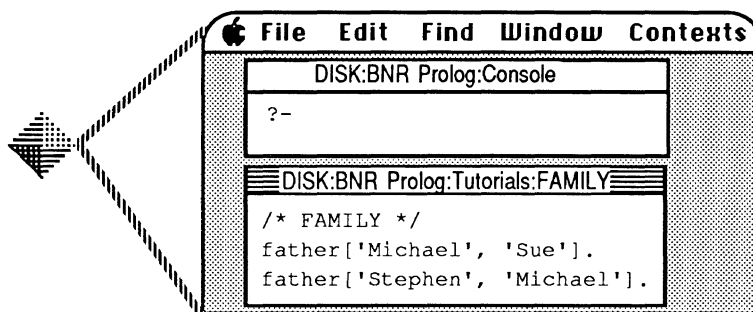




## Chapter 4

# Using the BNR Prolog System

The  icon represents the BNR Prolog application, and the  icon represents a document file created using that application. Opening either icon loads the BNR Prolog application, and creates an interactive text window labeled *Console* that is the default output stream. Opening a document file has the additional effect of opening a text window on the file. If several text documents are opened by the user each has a separate text window.



Opening a BNR Prolog Document

The BNR Prolog desktop is made up of a multi-window text editor and a menu bar. Text is input interactively to the underlying BNR Prolog system from any text window by selection (using standard Macintosh selection practices), and entry (using the *enter* key). If no text is highlighted, the line on which the cursor is currently positioned is the default selection. The part of the system that handles user interaction is called the *listener*.



Other than the special significance of the *enter* key, BNR Prolog provides a standard set of file handling, window handling and text edit commands through a menu/mouse interface. Shortcuts to many commands are also available through a *Command* key sequence. (See the descriptions of the Desktop and Editing Keys in the "Appendix" for further details)

The activity box to the left of the horizontal scroll bar in the active text window displays the current application state. For example, the listener may display **listening** when waiting for a question, or **running** when searching for answers. The message in the activity box is programmable, so other messages that depend on the activity in progress may be displayed.

It is possible to input from any part of an open document. Selecting part of the document and pressing the *enter* key causes just the selected portion of the text to be passed to the system. The *enter* key, which is always tied directly to the system, sends selected text to the system from any activated window. System responses to entered text (such as answers or confirmations) appear at the end of the Console window. In summary, default input is from the active text window; default output is to the end of the Console window.



## Tutorial: Using a BNR Prolog Document

This tutorial uses the FAMILY knowledge base to familiarize you with the BNR Prolog system on the Macintosh. It can be found in the Tutorials folder that is provided with the BNR Prolog application.

Instructions are provided to lead you through opening a text document, performing some window manipulations, selecting and entering text, and use of **Contexts** menu.

The clauses displayed in the following example provide a very basic set of facts and rules. (Comments are designated by "/\* \*/").

### Program FAMILY

```
/* FAMILY */

/* Part of a family tree*/

/* For "father", "mother", and "parent", the first
argument is the parent and the second is the son or
daughter */

father('Michael', 'Sue').
father('Stephen', 'Michael').
father('Stephen', 'Julie').
father('Harry', 'Stephen').
father('Pierre', 'Sarah').
father('Pierre', 'Odette').
father('Charles', 'Pierre').
father('Greg', 'Caroline').

mother('Sarah', 'Sue').
mother('Hazel', 'Michael').
mother('Hazel', 'Julie').
mother('Eleanor', 'Sarah').
mother('Eleanor', 'Odette').
mother('Odette', 'Caroline').
```



```
parent(_X, _Y) :- father(_X, _Y).
parent(_X, _Y) :- mother(_X, _Y).

/* queries for testing purposes */

/*
?- father(_Parent, _Child).
?- listing.
*/
```

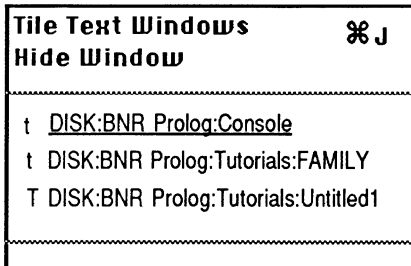
## Opening a Document



◇ Using standard Macintosh methods of opening applications and files, find the text document *FAMILY* in the *Tutorials* folder and open it. This provides you with two text windows, *Console* and *FAMILY*, the latter being the active window.

◇ Pull down the **File** menu and choose **New....** Create an empty text window called *Untitled1*. This is now the active window.

## Tiling Windows



◇ The **Window** menu displays the names of all open text windows, with that of the currently active window displayed in bold. Choose **Tile Text Windows**. This permits you to see all of the text windows at once. Notice that the listener has output the question prefix, "?-" to the *Console* window.



### Entering Sentences by way of Console

◊ Select the Console window, and individually enter the sentences

```
father('John', 'Mary').  
father('John', 'Michael').
```

Be sure to back space over the question prompt before typing. OK output after a sentence confirms the addition of the sentence to the knowledge base.

### Querying by way of Console

◊ Type the following query in the Console window and input it by typing *enter* at the end:

```
?- father(_Parent, 'Mary').
```

### Using a Question Window

◊ Select the window Untitled1, and type the following fact and query, using *return* to go to a new line:

```
father('John', 'Marsha').  
?- father(_Parent, _Child).
```

◊ Select and *enter* the query. After the first answer is displayed, type *return* to obtain the remainder. Only the original two facts are in the knowledge base. Note that the active window after a query is Console. Enter *Command-* if you wish to discontinue the output at any time.

◊ Select and *enter* the fact in the Untitled1 window, and then repeat the query.



## Loading from a Window

```
/*  
?- father(_Parent, _Child).  
?- listing.  
*/
```

◊ Select the window `FAMILY`, then pull down the **Contexts** menu and choose **Load Window**. This loads the contents of the file `FAMILY` on top of the existing knowledge.

◊ Select and *enter*

```
?- father(_Parent, _Child).
```

from the `FAMILY` window to query the predicate `father`. The listener outputs the first answer,

```
father('Michael', 'Sue').
```

and then waits to see if you want additional answers.

## Multiple Solutions

```
?- father(_Parent, _Child).  
father('Michael', 'Sue')  
father('Stephen', 'Michael')  
father(Stephen, 'Julie')  
father('Harry', 'Stephen')  
father('Pierre', 'Sarah')  
father('Pierre', 'Odette')  
YES
```

◊ Type `;` to indicate that another answer is desired. This may be repeated until no further answers are possible, or if *return* is typed, all answers are output without pause. Typing `q` (or some other key) ends output immediately. YES output after a query indicates that there was at least one answer to the question posed.

◊ Repeat the query. After the first solution, type *return* to obtain all possible solutions. Note the changes in the activity box in the lower left corner. Note that the sentences entered as knowledge before loading `FAMILY` are the last solutions.



## Listing a Predicate

◊ List all of the clauses that make up the predicate `parent` by typing in the Console window

```
?- listing(parent).
```

◊ Select and *enter* the query

```
?- listing.
```

from the FAMILY window. Note that all known clauses in FAMILY are displayed in the Console window.

## Multiline Input

◊ Type the following

```
ancestor(_Parent, _Child) :-  
    parent(_Parent, _Child).  
ancestor(_Parent, _Child) :-  
    parent(_Parent, _X),  
    ancestor(_X, _Child).
```

in the Console window. Select both rules and type *enter*.

◊ Now list or query the new predicate. It has automatically become part of the currently active FAMILY knowledge base.



## Listing by Menu

◇ Pull down the **Contexts** menu. The bottom section of the menu lists the existing contexts (see the chapter "Contexts"). The most recently loaded file, *FAMILY*, is the newest context. Choose *userbase*, then *FAMILY*, and note the differences in the list of predicates. If you select *father*, you will see both the clauses you entered into the Console and the Untitled1 windows, as well as the clauses from the file *FAMILY*, with the context of each displayed as a comment.

◇ Note that the predicate *ancestor* is part of *FAMILY*. Although it was not in the original file, it was automatically added to the top context, which is *FAMILY*.

## Reloading a Window

◇ Select the window *FAMILY*. Pull down the **Contexts** menu again, and choose **Load Window**. Use the menu to list the predicates in *FAMILY* again. The effect of this command is to reload the context *FAMILY* with the contents of the window. Since *ancestor* was not added to the window *FAMILY*, it was not reloaded.

## Exiting a Context

◇ Using the **Contexts** menu once more, choose **Exit Context** and select *FAMILY* from the submenu. The **Contexts** menu now has no item for *FAMILY*. (Exiting *userbase* will remove the context, but the system will create a new, empty *userbase*.)



## Saving Text

◊ Use the **Save as...** command in the **File** menu if you want to save your text in a disk file before you quit. This saves a copy of your version of `FAMILY` on disk. Note that the window has been replaced by the new file.

## Quitting

◊ When you are ready, select **Quit** from the **File** menu, and answer any questions about saving the updates. (You probably do not want to save the `Console`.)

For further information on the use of the menu/mouse interface, see the description of the Desktop in the "Appendix".







# Part II

## Prolog, a Logic Programming Language

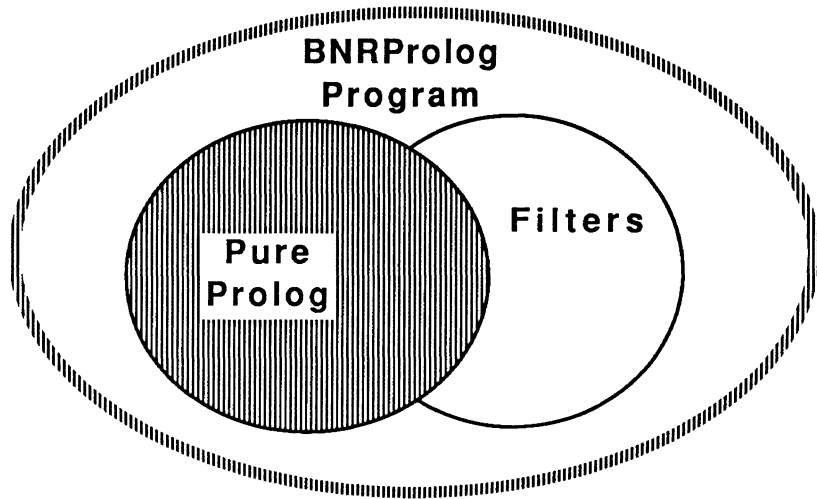






# Chapter 5

## Pure Prolog



Pure Prolog is the heart of the language. Historically it was the earliest part of the language to be developed, and its properties give the language its characteristic flavor. The pure Prolog subset was originally developed as an implementation of Horn clause logic, which is a subset of first-order predicate calculus. In this model the rule

$$p(\_X, \_Y) :- q(\_X, \_Z), r(\_Z, \_Y)$$

represents the predicate calculus expression

$$(X) (Y) (Z) \quad \neg q(X, Z) \vee \neg r(Z, Y) \vee p(X, Y)$$

Many Prolog text books take Horn clause logic, or even full first-order predicate calculus, as their starting point for describing pure Prolog. The connection between Horn clause logic and the formal properties of the pure subset will be discussed in more detail later.



Another approach, which we will follow, is to think of the pure subset as the language generated by *call*, *conjunction*, *backtracking*, and nothing else. People who are primarily interested in programming usually find this approach more natural. Before discussing the programming mechanisms of pure Prolog, it is important to cover some basic syntax issues.

## BNR Prolog Terms

Each sentence in a BNR Prolog program consists of a term followed by a period, ".". A term may be a simple constant (such as a number or a symbol), a logic variable, or a compound term. The following discussion briefly describes each type; for a more detailed description, refer to the *BNR Prolog Reference Manual*. Programs may also contain comments which are ignored on input. Comments have two forms. If enclosed in "/\* \*/", a comment may carry over more than one line. Comments of this form may be nested. Within a single line, any text between "%" and the end of the line is a comment.

### Numbers

Integers are whole numbers in the range of -268435456 (that is  $-2^{28}$ ) to 268435455 ( $2^{28} - 1$ ).

Floating point numbers are in the range  $-3.4e+38$  to  $3.4e+38$ , and may be specified in fixed point (for example 123.45) or scientific notation (for example 1.2345e2).

### Symbols

Symbols are case-exact, must not begin with an upper case letter or an underscore, "\_", and may be any arbitrary sequence of up to 255 alphanumeric characters. Symbols are used to represent operators, data, or clause names. Any sequence of characters may be used as a symbol by enclosing it in single or double quotes. Some examples of symbols are '123', 'Abner', boy, +.



## Variables

A variable name begins with an upper case character or with an underscore. A single underscore, "`_`", denotes an anonymous variable. Each occurrence of an anonymous variable is distinct from any other. Some examples of variables are `_21`, `_Mexico`, `_Book`, `_table`. Variable names are always output with a leading underscore to make them more readily distinguishable from quoted symbols beginning with uppercase letters.

## Lists

A list is a construct that groups zero or more terms into a single term. The elements in a list, an ordered sequence of terms, must be separated by commas, "`,`", and the entire list must be enclosed in square brackets, "`[]`". Available memory is the only limit to the number of terms in a list. Some examples of lists are `[a, 2, [carrot, radish]]`, `[[1, 2], []]`.

Often a list is broken into the initial elements, and the tail (the following sequence of elements). The syntax `_List..` specifies the tail of a list and is called a *tail variable*. A tail variable is only valid within a list or a structure (see below), and must be the last element before the closing bracket.

In any sentence using a tail variable `_List..`, the expression `_List` may be used to represent `[_List..]`. Some examples containing tail variables are `[a, b, _X..]`, `[_tail..]`, `[_First, _Rest..]`.

The common "bar" notation, as in `[_First, | _Rest]`, is accepted on input and coerced to `[_First, _Rest..]`.

## Structures

A structure consists of a symbol or variable immediately followed by a parenthesized list. Thus, all the features of lists also pertain to structures. Some examples of structures are `neighbor('James', 'Charles')`, `_Function(_Args..)`.



## Clauses

Clauses are terms which are asserted into the knowledge base. The head of a clause (the left hand side of ":-") is always a structure, and the body (the right hand side of ":-") is always a list.

A clause is recognized by the first symbol (hereafter called the clause name) in the head of the sentence. One or more clauses with the same name are grouped as a *predicate definition*, or *predicate*. If there are no arguments in the head of a clause, the structure may be written without parentheses. However, the clause head is coerced to a legitimate structure syntax with an empty parenthesized list.

A tail variable may be the last term in an argument list. This feature permits the definition of predicates which take a variable number of arguments. Such predicates are called *variadic*.

Rules are sentences that permit the inference of information from the truth of some other information. The key to rules is the concept *if*, represented by the symbol ":-", which separates the head and the body. Some examples of rules are

```
father(_X, _Y) :- parent(_X, _Y), male(_X).  
parent(_X, _Y) :- mother(_X, _Y).
```

The parser accepts rule definitions without the brackets that denote a list structure, but the clause body is coerced to a legitimate list.

Facts are the simplest type of clause, since the body of a fact is the empty list, "[]". They are used to express a primitive relation between the arguments. The meaning of the relation, and the ordering of the arguments depend entirely upon your convention. Some examples are

```
mother('Sue', 'Michael').  
year_end.
```



Facts are coerced on input to the standard clause form. `year_end` must have both the head coerced to a structure, and the tail coerced to a list:

```
year_end() :- [].
```

Facts are displayed without the clause body when listed.

## Unification

The pure Prolog call mechanism consists of unification followed by goal reduction. Using the built-in unification operator "=", helps to understand the meaning of unification. Questions of the form

```
?- _X = _Y.
```

succeed if and only if the terms `_X` and `_Y` unify. As the notation `_X = _Y` suggests, unification can be thought of as an equivalence relation, that is, a relation satisfying the following *equivalence laws*:

*reflexive* `_X = _X` for any term `_X`

*symmetric* `_X = _Y` if and only if `_Y = _X`

*transitive* if `_X = _Y` and `_Y = _Z`, then `_X = _Z`

## Unifying Ground Terms

Terms with no variables, *ground terms*, unify only if they are identical:

Succeed	Fail
?- fred = fred.	?- fred = george
?- 2 = 2.	?- 2 = 3.
?- 0.123 = 0.123.	?- 1 = 1.0.
	?- 2 = 1 + 1.



Unification of a number with an arithmetic expression does not succeed because the left argument is an integer while the right argument remains an unevaluated expression. (The operation "==" evaluates its arguments and compares the results; this is covered in the chapter *Functional Arithmetic*.)

Lists or structures unify if and only if their corresponding elements unify.

Succeed	Fail
<pre>?- [2, 1.1] = [2, 1.1]. ?- f(2, [a]) = f(2, [a]).</pre>	<pre>?- [2, []] = [2, 1.1]. ?- f(2) = f(2, b)</pre>

Thus, for ground terms (terms containing no variables) unification is just equality of terms.

## Unifying Variables

If we think of each variable as representing a definite but unknown term, then the interpretation of unification as equality can be extended to terms involving variables as well. Since a variable represents any term, it unifies with any term, as demonstrated in the following successful queries:

```
?- _X = fred.
?- fred = _Y.
?- _Z = 2.
?- _w = [a, b, c].
?- g(a, 2, []) = _t.
```

Unifying two variables causes them to become one variable, as the following query demonstrates.

```
?- _X = _Y.
?- (_X = _X)
YES
```

The substitution of terms that result from unification are seen in the answer. Although only one name is displayed, the result is



in fact symmetric. (This is a particularly striking example of the difference between logic variables and conventional variables.)

As with unification of ground terms, lists or structures containing variables unify if and only if their corresponding elements unify. Thus, in

```
?- [fred, 2, 1.1] = [fred, _Y, _Z].
```

unifying the pattern `[fred, _Y, _Z]` effectively checks for a list of length 3 with `fred` in the first position and extracts the second and third elements in one operation. It is important to be aware that each variable can take on only one value at a time. Thus,

```
?- [fred, 2, 1.1] = [_X, _Y, _X].
```

fails, since `_X` cannot be both `fred` and `1.1` at the same time.

The unification of structures is similar to that of lists, as demonstrated in the following successful queries:

```
?- _F(_X, _Y) = g(2, 3).
?- _F(_X, fred) = _G(3, _F).
?- _F(_X, _Y) = 2 + 3.
```

Note that this applies also for those structures which are syntactically represented by operators, as is demonstrated in the last example, where `_F` is bound to the operator `+`. Note that `2 + 3` is equivalent to `'+'(2, 3)`.

The effects of unification can easily become quite complex, as is demonstrated by the following queries and answers.

```
?- [_X, _Y, _Z] = [_Y, _Z, _A].
?- ([_X, _X, _X] = [_X, _X, _X])
YES
?- [_X, f(_X, _Z), _Z] = [2, f(_U, _V), _C].
?- ([2, f(2, _Z), _Z] = [2, f(2, _Z), _Z])
YES
```



## Unifying Tail Variables

A tail variable can be unified with a sequence of terms. For example, in

```
?- [_A, _X..] = [1, 2, 3, 4].
```

A unifies with 1, and [\_X..] becomes [2, 3, 4]. The term [\_X..] unifies with any list. For example

Succeed	Fail
?- [_X..] = [].	?- [_X..] = f.
?- [_X..] = [f].	?- [_X..] = f(2, 3).
?- [_X..] = [2, _Z..].	?- [_X..] = 2.
?- [_X..] = _v.	

A list which ends with an uninstantiated tail variable, known as an *indefinite list*, can be extended by instantiating the tail variable. In the query

```
?- [[a, _X..], [b, _Y..]] = [_Z, [_X..]].  
?- ([[a, b, _Y..], [b, _Y..]] =  
    [[a, b, _Y..], [b, _Y..]])
```

YES

the indefinite list [\_X..] is instantiated to the list [b, \_Y..], which contains another tail variable. Tail variables may also be used in structures, as seen in

```
?- _F(_X..) = g(2, 3, 4, 5).
```

where \_F is bound to g and [\_X..] to [2, 3, 4, 5], thereby splitting the structure into its *principal functor* g and its *argument list* [2, 3, 4, 5].



Experimentation with more complex examples of lists and structures reveals some exceptional cases. For example,

```
?- [_F(2, 3), _F] = [_G, 4].
```

produces an error condition, since it creates an invalid term, `4(2, 3)`. Methods of preventing such errors are discussed in the chapter "Filters and Negation".

Another phenomenon which might be puzzling at first is the creation of an infinite term or cyclic structure. For example,

```
?- _F = g(2, _F).
```

```
?- (g(2, g(2, g(2, g(2, g(2, g(2, [...]))))))) =
    g(2, g(2, g(2, g(2, g(2, g(2, [...])))))))
```

```
YES
```

```
?- [_X..] = [2, 3, _X..].
```

```
?- [2, 3, 2, 3, 2, 3, ...] = [2, 3, 2, 3, 2, 3, ...]
```

```
YES
```

Cyclic structures, also known as rational trees, are rarely used in Prolog programs, but it is important to know they can be created and used. They are discussed further in the chapter "Cyclic Structures".

## Call

Explicit unification, "=", is used infrequently, because implicit unification is performed with every call to a clause. To explain this concept, assume that the following fact is part of the knowledge base:

```
reduce(_A * _B + _A * _C, _A * (_B + _C)).
```

This fact corresponds to the distributive law of arithmetic, and is useful in simplifying arithmetic expressions. The formal arguments of this fact are implicitly unified with the actual arguments on each call, as demonstrated in the following queries:



```
?- reduce(_Y, a * (b + c)).  
    ?- reduce(((a * b) + (a * c)), (a * (b + c)))  
YES  
  
?- reduce(a * (b + 4) + a * c, _X)  
    ?- reduce(((a * (b + 4)) + (a * c)), (a * ((b + 4) + c)))  
YES
```

The query

```
?- reduce(2 + 3 * b, _x).
```

fails, since the form of the first argument does not unify with the first argument of the head of the clause `reduce`. In general, clauses simply fail if they are called with inappropriate arguments, rather than causing errors as frequently happens in conventional languages.

It is important to note that the two direct effects of a call are the instantiations, which are caused by the unification with a clause head; and the goal reduction step, which defines the next goal. With respect to variable instantiations, any sequence of calls is equivalent to some sequence of unifications with terms from the clause heads.

## Backtracking

If the head of more than one clause in a predicate matches a question, then each of these possibilities must be tried. Such a predicate is called *nondeterministic*, since more than one clause may succeed. This name is misleading since it suggests something random, which is not the case at all.

At a point where more than one possible choice exists, known as a *choicepoint*, the system chooses one possibility and pursues it to either success or failure. In the latter case, the system comes back to the choicepoint, resets all variable bindings to their values before any choice, and pursues another alternative. This strategy is called backtracking.



Although the choice of alternative could be arbitrary, sequential Prologs choose according to the order in which the clauses are stored in the knowledge base. Thus, the programmer has explicit control of the search order. This order can always be ascertained by using the *listing* predicate.

When the predicate `reduce` is extended by the following clauses

```
reduce(_A * _B - _A * _C, _A * (_B - _C)).
reduce(_A - _A, 0).
reduce(_A / _A, 1).
reduce(_A + 0, _A).
reduce(_A * 1, _A).
reduce(_A / 1, _A).
reduce(_A * 0, 0).
```

backtracking is demonstrated by the following queries and their results:

```
?- reduce(2 * 3 - 2 * 3, _X).
?- reduce(((2 * 3) - (2 * 3)), (2 * (3 - 3)))
?- reduce(((2 * 3) - (2 * 3)), 0)
YES

?- reduce(2 * 3 - 2 * 4, _X).
?- reduce(((2 * 3) - (2 * 4)), (2 * (3 - 4)))
YES

?- reduce(_X, _Y).
?- reduce(((A * B) + (A * C)), (A * (B + C)))
?- reduce(((A * B) - (A * C)), (A * (B - C)))
?- reduce((A - A), 0)
. . .
YES
```



## **member**

The three most basic elements of pure Prolog, unification, call, and backtracking are evident in the commonly used predicate `member`. The definition of `member` expresses the idea of a term being an element of a list. (Note the use of "bar" notation as an alternative input form.)

```
member(_X, [_X | _Ys]).  
member(_X, [_Y | _Ys]) :- member(_X, _Ys).
```

A declarative paraphrase of the definition is

```
_X is a member of a list  
if _X is the first element of the list  
or if _X is a member of the rest of the list.
```

Note the results of the following queries, and the order in which the solutions appear:

```
?- member(a, []).                % fails  
NO  
  
?- member(a, [a, b, c]).         % finds one  
?- member(a, [a, b, c]).  
YES  
  
?- member(a, [a, b, a, c]).      % finds two  
?- member(a, [a, b, a, c]).  
?- member(a, [a, b, a, c]).  
YES  
  
?- member(_X, [a, b,]).          % enumerates all  
?- member(a, [a, b]).  
?- member(b, [a, b]).  
YES  
  
?- member(w, [_A, b, c, _D]).    % updates list  
?- member(w, [w, b, c, _D]).  
?- member(w, [_A, b, c, w]).  
YES
```



```

?- member(_X, [_A, _B]).           % enumerates and
?- member(_X, [_X, _B]).           % updates
?- member(_X, [_A, _X]).
YES

?- member(a, [_X, _Xs..]).          % generates
?- member(a, [a, _Xs..]).           % indefinite list
?- member(a, [_X, a, _Ys..]).
?- member(a, [_X, _Y, a, _Ys..]).
?- member(a, [_X, _Y, _Y_1, a, _Ys..]).
. . .

?- member(_X..).                   % universal question
?- member(_X, [_X, _Ys..]).
?- member(_X, [_Y, _X, _Ys..]).
?- member(_X, [_Y, _Y_1, _X, _Ys..]).
. . .

```

The last two queries have an infinite number of answers, only a few of which are shown in each case. Note that every other `member` question is a special case of the universal question posed in the last query. The predicate `member`, used frequently in applications involving lists, is also a useful model for writing more complex list processing predicates.

## Recursion

Recursion, that is having a predicate call itself directly or indirectly, is one of the most important features of list processing. For example, in the predicate `member` if `_X` unifies with the first element in the list, it is definitely a member of the list, and the predicate succeeds. However, if the first clause does not succeed, then `_X` does not unify with `_Y`, so `_X` and the remainder of the list, `_Ys`, are passed to `member` in a recursive call. Reversing the order of these clauses has the following effect:

- It functions the same as `member` with the exception of the nonterminating case, but the answers are produced in reverse order.



- The operation is more expensive since all the sublists are generated before any testing is done. The last case is the same as the worst case of the original `member`.

One of the most important points to remember when using recursion is that you must have a terminating condition. In the predicate `member`, the terminating condition is related to the ever decreasing size of the list that is being passed as an argument in the recursive calls. Eventually the list will be empty, but an empty list does not unify with either of the clauses in the predicate. Thus, the exhaustion of the list forces the predicate to terminate.

### **append**

Another classic Prolog predicate is `append`, which expresses the idea of a third list being the concatenation of the first two:

```
append([], [_rest..], [_rest..]).
append([_X, _Xs..], [_rest..], [_X, _rest2..]) :-
    append(_Xs, _rest, _rest2).
```

Note that because of tail variable coercion the second clause is represented internally as:

```
append([_X, _Xs..], [_rest..], [_X, _rest2..]) :-
    append([_Xs..], [_rest..], [_rest2..]).
```

The declarative paraphrase of `append` is:

```
appending the empty list to any list A gives A
appending list A to list B yields list C, where the
first element of C is the first element of A, and the
rest of C is the result of appending B to the rest
of A.
```

Some examples are:

```
% fails, 'Fred' is not a list
?- append([], 'Fred', _X).
NO
```



---

```

% base clause
?- append([], [2, 3], _X).
  ?- append([], [2, 3], [2, 3]).
YES

% concatenation
?- append([1, 2], [3], _X).
  ?- append([1, 2], [3], [1, 2, 3]).
YES

% prefix decomposition
?- append([_A, _B], _X, [1, 2, 3]).
  ?- append([1, 2], [3], [1, 2, 3]).
YES

% nondeterministic decomposition
?- append(_X, _Y, [a, b]).
  ?- append([], [a, b], [a, b]).
  ?- append([a], [b], [a, b]).
  ?- append([a, b], [], [a, b]).
YES

% appending to an indefinite list
?-append([a], [_Y..], _Z).
  ?- append([a], [_Y..], [a, _Y..]).
YES

% appending to an indefinite list
?- append([a, _X..], [b], _Z).
  ?- append([a], [b], [a, b]).
  ?- append([a, _X], [b], [a, _X, b]).
  ?- append([a, _X, _X_1], [b], [a, _X, _X_1, b]).
    . . .

% universal question
?- append(_X..).
  ?- append([], [_rest..], [_rest..])
  ?- append([_X], [_rest..], [_X, _rest..]).
  ?- append([_X, _X_1], [_rest..], [_X, _X_1, _rest..]).
    . . .

```

---



Consider the following mutation of *append*, formed by rearranging the arguments slightly:

```
rev_append([], [_X..], [_X..]).
rev_append([_X, _Xs..], [_Y..], [_Zs..]) :-
    rev_append(_Xs, [_X, _Y..], _Zs).

?- rev_append([_A, _B], _X, [1, 2, 3, 4]).
?- rev_append([2, 1], [3, 4], [1, 2, 3, 4]).
YES

?- rev_append([1, 2], [3, 4], _X).
?- rev_append([1, 2], [3, 4], [2, 1, 3, 4]).
YES
```

Repeating some of the queries to *append*, makes the effect of the change apparent. If the second list is empty, the effect is that of another common Prolog predicate, *reverse*.

## Ordering of Clauses

The order of clauses in pure Prolog does not affect the declarative meaning of a predicate, but may have profound consequences on its behavior. Some clause orders may not terminate properly, or may be very inefficient when compared with other equivalent orders. For example, it is usually better to put the termination clause first as in

```
append([], [_X..], [_X..]).
```

To illustrate ordering of clauses, backtracking and calling a variable, consider the definition of OR (predefined by the system). The operator ";" is defined as infix, and the meaning of the operation is defined as

```
(_P ; _Q) :- _P.
(_P ; _Q) :- _Q.
```



The arguments to ";" must be executable terms. For example, successive instantiation of `_X` occurs in the following:

```
?- member(_X, [a, b, c]) ; member(_X, [d, e, f]).  
?- ([member(a, [a, b, c]) ; member(a, [d, e, f])]).  
?- ([member(b, [a, b, c]) ; member(b, [d, e, f])]).  
. . .
```

## Conjunction

The idea of *and then* is expressed using list notation. For example,

```
?- P, Q, R.
```

can be read as

```
P and then Q and then R
```

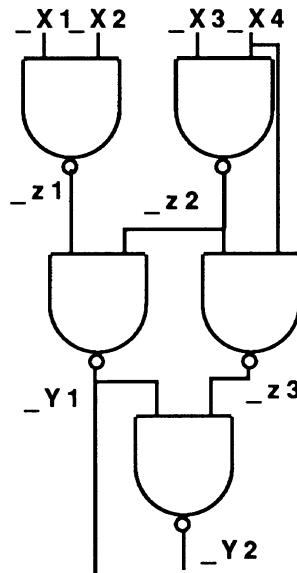
In the pure Prolog subset, however, the order of operations does not affect the meaning of the answer, that is, `[P, Q]` means the same as `[Q, P]`. This allows us to drop *then*, treating the comma as *and*. This is illustrated by the following queries:

```
?- _X = a, reduce(_X * b + _X * c, _Y).  
?- [(a = a), reduce((a * b) + (a * c), (a * (b + c)))]  
YES  
  
?- reduce(_X * b + _X * c, _Y), _X = a.  
?- [reduce((a * b) + (a * c), (a * (b + c))), (a = a)]  
YES
```



## Conjunction and Backtracking

A program using just conjunction and backtracking can be written to simulate the behavior of a Boolean switching circuit, such as is shown in the following figure.



Circuit

The components of the circuit are *nand* gates. With the variables representing *wires*, the definitions the circuit is described as follows:



```
% nand(_input1, _input2, _output)
    nand(0, 0, 1).
    nand(0, 1, 1).
    nand(1, 0, 1).
    nand(1, 1, 0).

circuit([_X1, _X2, _X3, _X4], [_Y1, _Y2]) :-
    nand(_X1, _X2, _z1),
    nand(_X3, _X4, _z2),
    nand(_z1, _z2, _Y1),
    nand(_X4, _z2, _z3),
    nand(_Y1, _z3, _Y2).
```

The results of queries would be similar to:

```
?- circuit([0, 1, 0, 1], [_t, _u]).
    ?- circuit([0, 1, 0, 1], [0, 1]).
YES

?- circuit([_a, _b, _c, _d], [1, 0]).
    ?- circuit([0, 0, 1, 1], [1, 0]).
    ?- circuit([0, 1, 1, 1], [1, 0]).
    ?- circuit([1, 0, 1, 1], [1, 0]).
    ?- circuit([1, 1, 0, 0], [1, 0]).
    ?- circuit([1, 1, 1, 0], [1, 0]).
    ?- circuit([1, 1, 1, 1], [1, 0]).
YES

?- circuit([0, _b, 0, _d], [_t, 1]).
    ?- circuit([0, 0, 0, 0], [0, 1]).
    ?- circuit([0, 0, 0, 1], [0, 1]).
    ?- circuit([0, 1, 0, 0], [0, 1]).
    ?- circuit([0, 1, 0, 1], [0, 1]).
YES
```



## Conjunction and Recursion

An example of conjunction and recursion together starts with a modified version of member called `choose` (a recursive predicate). This is used in a conjunction in another recursive predicate, `permutation`:

```
choose(_X, [_X, _Ys..], [_Ys..]).
choose(_X, [_Y, _Ys..], [_Y, _Zs..]) :-
    choose(_X, [_Ys..], [_Zs..]).

permutation([], []).
permutation(_List, [_X, _Rest..]) :-
    choose(_X, _List, _Xs),
    permutation(_Xs, _Rest).
```

Example queries are:

```
?- choose(_X, [a, b, c, d], _List).
?- permutation([a, b, c, d], _X).
```

A call to `choose` nondeterministically instantiates a member of `[a, b, c, d]` to `_X`, and `_List` becomes `[a, b, c, d]` with the chosen item deleted. A call to `permutation` generates all twenty-four permutations of the list `[a, b, c, d]`.

## Ordering of Subgoals

Although the meaning does not depend on the order, the order of subgoals should also be considered carefully because of termination and efficiency issues. There are instances where `[P, Q]` does not terminate, but `[Q, P]` does, as can be seen in the following:

```
?- member(_X, _Y), _Y = [a, b, c].
?- [member(a, [a, b, c]), ([a, b, c] = [a, b, c])].
?- [member(b, [a, b, c]), ([a, b, c] = [a, b, c])].
?- [member(c, [a, b, c]), ([a, b, c] = [a, b, c])].
    .
    .
    .
<Error> 5 Global stack full
```



```

?- [_Y = [a, b, c], member(_X, _Y)].
?- [([a, b, c] = [a, b, c]), member(a, [a, b, c])].
?- [([a, b, c] = [a, b, c]), member(b, [a, b, c])].
?- [([a, b, c] = [a, b, c]), member(c, [a, b, c])].

```

YES

## Putting It Together

Prolog encourages the construction of simple, but very general definitions, and then combining them to achieve the desired effect. Building on the example `reduce` (see "Call" in this chapter) illustrates this. Note that some obvious rules for the reduction of arithmetic expressions are missing, for example

```
reduce(1 * _A, _A).
```

Although this rule could be added directly to the knowledge base, it is a consequence of rules already existing and the law of commutativity for `"*`". Therefore, a general rule, `reduce_sym`, can be written that switches the arguments of a commutative operation in order to get a reduction. Since `"+"` is also a commutative operation, it will also be covered.

```

reduce_sym(_X, _Y) :- reduce(_X, _Y).
reduce_sym(_F(_X1, _X2), _Y) :-
    commutative(_F),
    reduce(_F(_X2, _X1), _Y).

commutative(+).
commutative(*).

```

The predicate `reduce_sym` makes use of the variable functor notation `_F(_X, _Y)` in its pattern matching. Note that `a + b` is really `'+'(a, b)`; the infix syntax only affects input and output of terms, not their actual structure. Insuring that the operation is commutative before calling `reduce` prevents a lot of unnecessary work.

For a term like `(a + 0) + 0`, a first reduction step produces `(a + 0)`, which can be further reduced. Reduction can be



continued as far as possible through recursion, as demonstrated in the predicate `reduces_to`.

```
reduces_to(_expl, _expl).
reduces_to(_expl, _exp2) :-
    reduce_sym(_expl, _X),
    reduces_to(_X, _exp2).

?- reduces_to(a + 0 + 0, _X).
?- reduces_to(((a + 0) + 0), a).
?- reduces_to(((a + 0) + 0), (a + 0)).
?- reduces_to(((a + 0) + 0), ((a + 0) + 0)).
YES
```

Terms like  $(a * 1 - 1 * a)$  still cannot be reduced without first reducing the arguments. It is necessary therefore to reduce each argument first, starting with the simplest terms. This can be generalized by making the reduction rule an argument, as demonstrated in the predicate `transform` that follows. Variables represent functors both in the pattern matching and as a call in the rule body. A valid predicate name must be substituted for `_RuleName` before the call, or an error occurs.

```
transform(_Op(_Left, _Right), _Result, _RuleName) :-
    transform(_Left, _Left1, _RuleName),
    transform(_Right, _Right1, _RuleName),
    _RuleName(_Op(_Left1, _Right1), _Result).
transform(_Op, _Result, _RuleName) :-
    _RuleName(_Op, _Result).
```

The following example specifies the operator `"==>"` as infix, with weak precedence, and also the definition of the related operation. The details of specification of operator syntax and precedence are covered in chapter *Operators*. Note that the rule `reduces_to` is passed as an argument in the call to `transform`. Reduction of some simple arithmetic expressions is demonstrated.

```
op(1000, xfx, '==>').
_X ==> _Y :- transform(_X, _Y, reduces_to).
```



```

?- a * 0 - 0 * a ==> _X.
  (((a * 0) - (0 * a)) ==> ((a * 0) - (0 * a))).
  (((a * 0) - (0 * a)) ==> ((a * 0) - 0)).
  . . .

?- a * 1 - 1 * a ==> _X.
  (((a * 1) - (1 * a)) ==> ((a * 1) - (1 * a))).
  (((a * 1) - (1 * a)) ==> ((a * 1) - a))
  . . .

?- (a * 1 + b * 0) / (1 / 1 + (a - a)) ==> _X.
  (((a * 1) + (b * 0)) / ((1 / 1) + (a - a))) ==>
    (((a * 1) + (b * 0)) / ((1 / 1) + (a - a))).
  (((a * 1) + (b * 0)) / ((1 / 1) + (a - a))) ==>
    (((a * 1) + (b * 0)) / ((1 / 1) + 0)).
  . . .

```

Although this still is not a complete program for simplification of arithmetic expressions, it can do some reductions that are quite complex. (Those with backgrounds in other languages might consider how they would implement this algorithm in their favorite conventional language, and how they would convince themselves that it was implemented correctly.)

## Characteristics of Pure Prolog Programs

In this chapter the pure Prolog subset has been described as the language generated by nondeterministic calls and conjunction. It was mentioned that the pure subset can also be described by its connection to Horn clause logic. In addition, it can be described in terms of certain formal properties. The differences between the three descriptions are of particular interest.

Prolog, as described above, differs from Horn clause logic in three basic respects. First, the language of classical logic is limited to finite terms. Unification, in the classical sense, therefore should not produce cyclic structures. However, Prolog implementations have usually omitted the check ("occurs check") required to detect cycle formation because of its



inefficiency. Many Prolog systems generate unpredictable errors when handling cyclic structures.

BNR Prolog belongs to a family of Prolog systems (which includes Prolog-II) which not only permits cyclic structure formation, but correctly handles unifications and other operations involving cyclic structures. Languages in this family are not, in fact, based on classical Horn clause logic at all, but have other semantic models which allow more general data structures. Since the majority of computations do not involve cyclic structures, they can still be viewed as if they were in Horn clause logic.

The second basic difference from Horn clause logic is due to the depth-first search strategy used in Prolog. If a search tree is infinite, as is frequently the case, Prolog pursues an infinite branch until memory is exhausted, never reaching other branches which possibly contain solutions. In such cases, Prolog is incomplete while the classical theoretical model is not. Therefore, Prolog programs derived from the classical model contain the implied condition that the program terminates without error. In fact, nontermination is not uncommon, and dealing with it is an important part of Prolog programming.

The third difference is the use of features that do not exist in Horn clause logic such as variable functors, tail variables with structures, and "metacalls" as used in the predicate `OR`. These features permit the finite encoding of what would otherwise require an infinite set of rules.

## Formal Properties of Pure Prolog

In order to introduce the formal properties of pure Prolog, consider an arbitrary query or call `p(X)` where `p` is any predicate that is free of side effects, and `X` stands for the entire argument list. If the call succeeds, the answer, denoted by `p(X1)`, is generally different from the call because of variable instantiations. The first formal property, *narrowing*, is universal and was mentioned in the chapter "The Prolog Model".



*Narrowing: the answer  $X^1$  is narrower than the initial  $X$ . The failure of  $p(X)$  is regarded as having the "empty" answer, which is narrower than anything.*

To discuss the remaining properties that hold for pure Prolog programs, that is the *monotone*, *persistent*, and *idempotent* properties, it is useful to take another look at unification.

Recall that unification was introduced previously as an equivalence relation. A different interpretation, described in mathematical notation, is useful for analyzing formal properties. Let each term  $T$  be represented by the set  $\{T\}$  consisting of all the ground terms produced by instantiating the variables in  $T$  in all possible ways. If  $T$  contains no variables at all, then this set consists of  $T$  alone. Thus, if  $T$  is the term  $[a, a]$ , then the set  $\{[a, a]\}$  is the set  $\{T\}$ . If  $T$  is the term  $[_X, a]$ , then  $\{T\}$  represents a set that includes all  $[_X, a]$  with all instantiations of the variable  $_X$ . One member of this set would be  $[a, a]$ .

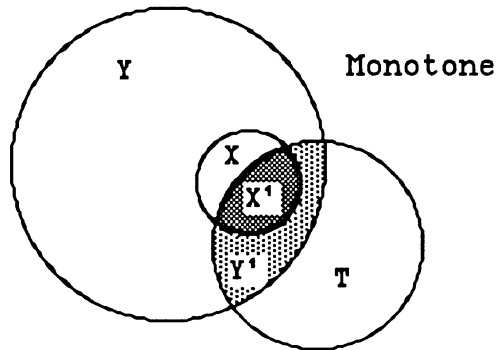
Unifying two terms  $X$  and  $Y$  instantiates both to the same term, where the resulting set of terms is the intersection of the sets  $\{X\}$  and  $\{Y\}$ . The *reflexive*, *symmetric*, and *transitive* laws of *equivalence* correspond respectively to the *idempotent*, *commutative*, and *associative* laws of *set intersection*. The concept of  $X$  being narrower than  $Y$  can then be expressed formally as  $\{X\}$  is a subset of  $\{Y\}$ .

Since this explanation of unification involves sets of ground terms rather than single terms, it is called a second-order interpretation. With this second-order interpretation, it is easy to see that if the term  $T$  is fixed, then the effect of  $X = T$  is to restrict  $X$  to correspond to the intersection of  $\{X\}$  and  $\{T\}$ . (The effects of unification with a fixed term is particularly important because calling a predicate involves unifying the question with a clause head.)

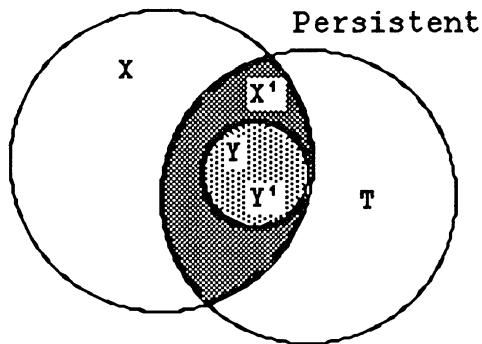
If  $Z$  denotes an arbitrary term before unification with  $T$ , the same term after unification is denoted by  $Z^1$ .



*monotone: if  $X$  is narrower than  $Y$  before unification then  $X^1$  is narrower than  $Y^1$  after.*



*persistent: if  $Y$  is narrower than  $X^1$ , then  $Y$  is the same as  $Y^1$ , since  $Y$  is already within the intersection.*



The persistent property essentially reflects the fact that logic variables, once instantiated, do not change until failure occurs. As a consequence, many operations, for example unification with a fixed term, have the property of not being disturbed by subsequent narrowing. The idempotent property is an important special case of persistence.

*idempotent:  $X^{11}$  is the same as  $X^1$ .*



Since unification has all of these formal properties, each call and each sequence of calls also has them. It therefore follows that any pure predicate, composed of nondeterministic calls and conjunctions, has these properties along every terminating branch. A predicate is monotone if narrowing the question narrows the answer, and it is persistent if a call that is syntactically identical to a previous call in the same goal sequence is always redundant. This is sufficient to prove that pure predicates *commute*, that is  $P, Q$  is the same as  $Q, P$  whenever both expressions terminate. (This is necessary if  $;$  is to be interpreted as "and".)

Any program that satisfies the *monotone* and *persistent* properties may, in principle, be written as a pure program. More practically, behaviorally pure programs can be written using "impure" constructs, and can be much more efficient than constructively pure ones. From a theoretical viewpoint, there are advantages in defining pure Prolog in terms of these behavioral properties rather than in terms of how they are constructed.

Because these formal properties refer only to the relationships between various questions and answers on terminating branches of the computation, they have nothing to do with completeness issues. Therefore, infinite branches of the computation can be pruned without invalidating these properties for the remaining branches. This is important in practical programs where it is frequently necessary to sacrifice completeness to ensure termination by using the nonlogical techniques discussed in the next few chapters.





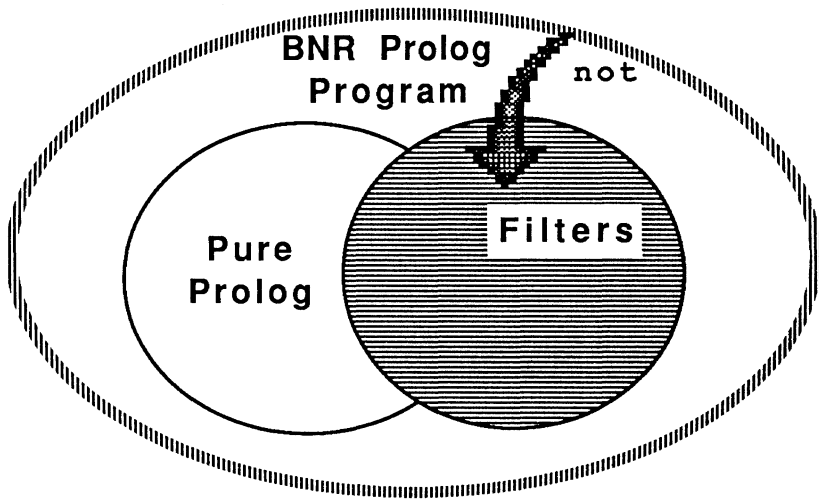


---

# Chapter 6

## Filters and Negation

---



To exercise control over a Prolog computation, it is often necessary to examine data structures without affecting them. Since unification may instantiate as well as test, it alone is not sufficient to perform these examinations. Those predicates which have no side effects and do not instantiate are called *filters*. Filters influence the computation solely by succeeding or failing. Two of the simplest filters are `true` which always succeeds; and `fail` which always fails, resulting in backtracking.



Several of the general properties of filters follow directly from these definitions:

- the disjunction of filters is a filter
- the conjunction of filters is a filter
- filters commute with each other, thus  $[f1, f2]$  is the same as  $[f2, f1]$
- filters are idempotent, thus  $[f, f]$  is the same as  $f$

Many filters fall into one of the two classes, *persistent* or *monotone*. Recall that the persistent property implies that a predicate "remains true". Thus, a later call to the same filter is redundant in any sequence of successful calls. For filters, the monotone property has the opposite significance. If a monotone filter is true at some point in a sequence, then it must have been true at previous points in the sequence as well.

In general, filters do not commute with unifications. More specifically, for each unification there is some filter with which it does not commute. Those filters which are both monotone and persistent do commute with unification, and thus are part of the pure Prolog subset and may be read declaratively.

The lack of commutative property between filters and unification may not seem significant in a small example. However, any procedures using such filters (or primitives requiring instantiated arguments) also do not commute with unifications. Such procedures lose the properties of pure Prolog, and therefore can only be understood procedurally. In fact, this is one of the major sources of "impurity" in Prolog.

One problem with such non-commutativity of primitives is that if they are used too soon (before instantiation) one risks rejecting what might be a valid answer. Deferring such tests until all critical instantiations are done (using the "generate and test" paradigm), helps to avoid this problem. However, it is generally



very inefficient since the test is frequently far from the decision point.

## Monotone Filters

There are only three primitive monotone non-persistent filters, `var`, `tailvar`, and `acyclic`. The filter `var`, for example, succeeds if and only if all its arguments are uninstantiated variables. `var` does not commute with unifications, as seen in the following table

Succeed	Fail
<code>var(_x)</code>	<code>var(7)</code>
<code>var(_x1, _x2, _Month)</code>	<code>var(2, _X)</code>
<code>var(_x), _x = 2</code>	<code>_x = 2, var(_x)</code>

Monotone filters can be created by combining other monotone filters using conjunction and disjunction.

Combinations of monotone filters and pure prolog are neither filters since they may instantiate, nor pure since they are not persistent. Thus, the declarative interpretation of the program is adversely affected. For example, using the clause

```
f(_X) :- var(_X), _X = 2.
```

`f(_X)` succeeds, but `[f(_X), f(_X)]` fails. This violates the idempotent nature of both filters and pure Prolog.

## Persistent Filters

Most primitive filters are persistent but not monotone. The typical persistent filter, `nonvar`, is true just when all its arguments are instantiated. `nonvar`, like `var`, does not commute with unifications. In the following, the first query fails because `_x` is an uninstantiated variable at the time tested, but the second succeeds because `_x` is instantiated before it is tested:



```
?- nonvar(_X), _X = 2.      % fails
?- _X = 2, nonvar(_X).    % succeeds
```

Each of the basic types in the language has its own primitive filter:

```
bucket
float
integer
interval
list
structure
symbol
```

All have analogous definitions and are mutually orthogonal.

Note the following examples:

Succeed	Fail
<pre>integer(1) float(1.0) symbol('Fred') structure(fred[]) list([]) nonvar(_x + 2)</pre>	<pre>integer(1.0) float(1) structure('Fred') symbol(fred[]) symbol([]) ground(_x + 2)</pre>



The persistent filter, `ground`, in the last example, is true only when its argument is a term containing no uninstantiated variables or tail variables.

Persistent filters can be created by combining other persistent filters using conjunction and disjunction. Combinations of persistent filters and pure prolog are also persistent. For example, the expressions

```
_x = a + _b, integer(_b)
nonvar(_x), list(_list), member(_x, _list)
```

are persistent. Although they are idempotent, they are not monotone, therefore not pure; they may instantiate, and thus are not filters.

To illustrate the application of filters, consider again the `reduce` example in the previous chapter. One of its most undesirable properties is the non-terminating behavior of `_X ==> _Y` when `_X` contains variables. One way to curb this exuberance is to use `ground(_X)` as a filter in the definition for `"==>"`. `_X ==> _Y` then fails rather than causing a stack overflow. The result is a persistent, but not monotone, predicate.

A less drastic fix is to change `reduce_sym` to use a `nonvar` test:

```
reduce_sym(_X, _Y) :- nonvar(_X), reduce(_X, _Y).

reduce_sym(_f(_X1, _X2), _Y) :-
    nonvar(_X),
    commutative(_f),
    reduce(_f(_X2, _X1), _Y).
```

This cuts out the non-terminating branches without actually prohibiting variables. The resulting `"==>"` is also persistent, but not monotone.

Some filters are both monotone and persistent, and thus can be considered part of pure Prolog. These include `true`, `fail`, and all executable ground terms. They are discussed in the chapter "Passive Constraints".



## Constructed Filters

Filters are traditionally considered to be non-logical because they do not generate their solutions (that is, they are non-constructive), and they do not commute with unifications. For example, `[integer(_X), _X = 2]` is not the same as `[_X = 2, integer(_X)]` if `_X` is initially uninstantiated.

There are four ways to create filters:

- combining other filters using conjunction and disjunction
- discarding the results of unifications
- using the predicate `not`
- grounding terms, as described in the chapter "Passive Constraints"

### Combining Filters

As noted before, filters that are monotone can be created from monotone filters, and filters that are persistent can be created from persistent filters. There are filters which are neither monotone nor persistent. One way to form them is to combine monotone and persistent predicates. The filter `weird` is an example:



```
weird(_X + _Y) :- integer(_X), var(_Y).  
?- weird(_u + _v).           % fails  
?- weird(2 + _v).           % succeeds  
  ?- weird((2 + _v)).  
?- weird(2 + 3).           % fails
```

Such filters cannot usually be given a declarative interpretation.

A set of primitives which fall into this class are the term comparisons:

$\_X @= \_Y$	$\_X @\backslash= \_Y$
$\_X @< \_Y$	$\_X @> \_Y$
$\_X @=< \_Y$	$\_X @>= \_Y$

These comparisons are frequently used in sorting lists of arbitrary terms. They can produce unpredictable results and must be used with caution, however, unless the terms compared are ground terms. The details of the comparisons and their ordering can be found in the *BNR Prolog Reference Manual*.



## Discarding Results

There are other ways to form filters besides combining primitive filters. From the viewpoint of the caller, if calling a predicate does not appear to instantiate any arguments passed to it, then that predicate is a filter. To accomplish this, the results of any unifications inside the predicate must be discarded.

Consider the problem of creating a filter that tests if a list is indefinite. The built-in predicate `termlength` (not a filter), takes a list (or structure) as the first argument, and returns the actual length, and either `[]` if the list is definite, or its terminating sublist if indefinite. An example is

```
?- termlength([a, b, _c..], _n, _t).  
?- termlength([a, b, _c..], 2, [_c..]).
```

Note that `termlength` is not a filter, since `_n` and `_t` are instantiated. However, by discarding the results of these unifications, an indefinite list filter can be formulated as follows:

```
indefinite(_list) :-  
    termlength(_list, _n, [_x..]),  
    tailvar(_x..).
```

## Negation by Failure

Negation by failure is the most common mechanism for building complex filters. The expression

```
not(P)
```

where `P` is some goal, succeeds if `P` fails and fails if `P` succeeds. A declarative reading is "`P` is not provable" which is not the same as "`(not P)` is true". (There has been much confusion over `not`, because in first-order predicate calculus `not(P)` means "`(not P)` is true".)

Provided `P` has no side effects, `not(P)` is a deterministic filter since all instantiations by `P` are undone by backtracking. If `P` is monotone, then `not(P)` is persistent. For example,



`not(var(_x))` is persistent since `var(_x)` is monotone. Similarly, negations of persistent predicates, such as `not(nonvar(_X, _Y))`, are monotone. Negations of pure predicates are both persistent and monotone.

`not` satisfies some of the formal requirements of the classical negation, such as:

- `not(P)`, `P` fails
- `P`, `not(P)` fails provided `P` is idempotent
- `not(P)`, `not(Q)` is the same as `not(P ; Q)`, where `P` and `Q` are executable goals

However, unless `P` is a filter:

- `not([P, Q])` is not always the same as `not(P) ; not(Q)`
- `P ; not(P)` is not always the same as `true`

If we define

```
pos(P) :- not(not(P)).
```

then `pos(P)` has the declarative meaning "P is possible", that is, it would succeed if called, where

- `not(pos(P))` is the same as `pos(not(P))`
- `pos(P)`, `not(P)` always fails (either order)
- `not([not(P), not(Q)])` is the same as `pos(P) ; pos(Q)`
- `pos(P) ; not(P)` is the same as `true`

Note that `pos(P)` is not the same as `P` unless `P` is a deterministic filter. It follows that if the multiplicity of solutions is neglected, filters form a Boolean algebra: *and*, *or*, and *not* with all their usual properties. `not` and `pos` map all of Prolog to the subset of filters.

In summary, filters are predicates that never instantiate variables. In general they are not part of pure Prolog because they do not commute with all unifications. Filters make it possible to determine the instantiation state of term, thus allowing for actions to vary according to what is or is not instantiated. A



basic set of primitive filters is provided by the system. Additional filters can be constructed by combining existing filters with logical connectives, by discarding the results of unification, and by using the `not` metapredicate (which makes any predicate into a filter).

Filters commute with each other and form a Boolean algebra under suitably chosen logical operations. Monotone and persistent filters are important subclasses of filters, since they have different characteristics and uses. Filters which are both monotone and persistent are also pure predicates, and therefore commute with all other pure predicates. One way to form such special filters is by the passive constraint mechanism, the subject of the next chapter.

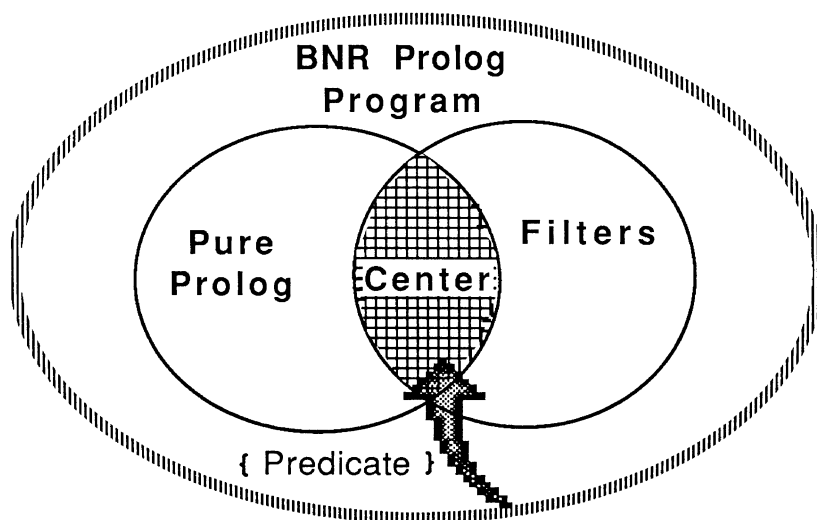


---

# Chapter 7

## Passive Constraints

---



There exists a group of filters which commute with all pure programs. These filters are both monotone and persistent, and form the class of predicates known as the *center*. Those combinations that are created from *central* predicates using conjunction, disjunction, and negation are also central.

In addition to `true` and `fail`, all ground terms belong to the center. Just as the `not construct` maps any program into a filter, the *passive constraint* construct maps any program into the center. The use of constraints provides a simple and uniform way to restore commutativity, and hence the declarative reading of programs. It also radically changes the way in which one formulates problems. The resulting code is not only logically cleaner, but usually runs much faster (sometimes by *orders of magnitude*).



## Constraint Notation

Syntactically, constraints consist of a goal enclosed in braces, "{}". Regarded as a data structure, a constraint is just a structure with name "{}". Specifying more than one goal, separated by commas,

`{A, B, . . .}`

is semantically equivalent to

`{A}, {B}, . . .`

When a constraint `{G}` is executed, it imposes the constraint `G` on the downstream execution; if there is backtracking across this point the constraint is removed. Thus in the predicate definitions:

`R :- P, Q.`

`P :- A, B, {C}, D, E.`

any constraints imposed by the execution of `{C}` are in force during the subsequent execution of `D` and `E`. Furthermore, if `P` succeeds and exports any constrained variables, then the constraints will be in force during the execution of `Q`, and so on.

Constraints are classed as

- *active constraints*, which are of the form `{P -> Q}`
- *data flow constraints*, which are of the form `{X is Expression}`
- *passive constraints*, which are all others

Active and data flow constraints have somewhat different properties and uses, which are discussed in later chapters.

Passive constraints are all arbitrary constructs of the form `{G}` where the goal `G` does not have either of the forms `P -> Q` or `X is Expression`. They never act as generators, and are



technically filters since they never cause instantiations (which is why they are called passive).

The passive constraint  $\{G\}$  defers the execution of  $G$  until it is ground. If and when  $G$  ever grounds, it will be interpolated into the stream of executing goals. If  $G$  succeeds, the constraint is satisfied and becomes dormant. If  $G$  fails, control backtracks through any choicepoints of  $G$  to the alternatives of the action that triggered  $G$ . Thus, the use of constraints ensures that tests are performed immediately after an instantiating choice, and the use of a backtracking strategy is more efficient. Because dormant constraints become "reactivated" whenever backtracking crosses their trigger instantiations, constraints apply to all subsequent computations.

## Declarative semantics

A passive constraint  $\{P\}$  has the formal declarative meaning of  $\text{ground}(P) \supset P$ , where  $\supset$  is logical implication. In other words, if  $P$  is not ground, then  $\{P\}$  is true; if  $P$  is ground, then  $\{P\}$  is true only if  $P$  is true. This may be different from the interpretation of other systems with otherwise similar constraint implementations, which may take the meaning of  $\{P\}$  to be equivalent to  $P$ . To understand the significance of this distinction, consider a predicate  $h$  with some internal variable  $\_X$  which is constrained as follows

```
h :- [{g(_X)}, . . .].
```

Suppose that  $h$  succeeds without ever instantiating  $\_X$ , which is thus an irrelevant variable. After returning to the caller,  $\_X$  would be inaccessible for instantiation (that is,  $\_X$  is not part of the answer in any way). According to the conditional interpretation,  $h$  is entitled to succeed; since  $\_X$  can never be instantiated, the implication  $\text{ground}(g(\_X)) \supset g(\_X)$  is true. However, with non-conditional interpretation,  $g(\_X)$  must be executed, and if it fails then  $p$  must also fail.

One consequence of conditional interpretation is that formally inconsistent constraints on an irrelevant variable, for example,



`{g(_X), not(g(_X))}` become formally equivalent to `not(ground(g(_X)))`, which is in accordance with the actual behavior. (Note that the problem of formally inconsistent constraints on irrelevant variables is a major problem for any non-conditional interpretation.)

## Constraint Interpolation

Interpolation of constraints into the stream of executing goals occurs right after unification instantiates the last variable. This usually corresponds to the `:-` in a clause definition. If several constraints are triggered by the same unification, they are processed in some unspecified order, as illustrated with the example `spyvar`.

### **spyvar**

Although passive constraints do not bind variables, they can have side effects. Consider the following

```
spyvar(_Name, _Var):-
    var(_Var), symbol(_Name), {disp(_Var, _Name)}.

disp(_Var, _Name) :- write(_Name: _Var), nl.
disp(_Var, _Name) :-
    write(unbinding, ' ', _Name), nl, fail.

p(cat).
p(dog).
```

In a query such as the following,

```
?- spyvar(fred, _fred), p(_fred).
(fred : cat)
?- [spyvar(fred, cat), p(cat)].
unbinding fred
(fred : dog)
?- [spyvar(fred, dog), p(dog)].
unbinding fred
YES
```



Each time `_fred` is instantiated, `disp` prints a message giving the instantiation, and another message records each unbinding. Some interesting variations on this technique can be used to provide graphics interfaces for programs without directly modifying their source. (See the *queens8* file in the *Demo* folder for an example.)

## Generalized Types

One of the simplest uses of passive constraints is to provide generalized "types". For example, `{integer(_X)}` enforces a type restriction on subsequent bindings to `_X`, and can be considered analogous to a type declaration in languages like Pascal. Apart from side effects, all passive constraints commute with each other and with all pure Prolog programs. For example,

```
{integer(_X)}, _X = 2
```

is equivalent to

```
_X = 2, {integer(_X)}
```

Many common constraint constructs are formed from the basic filters:

```
{float(_X)}  
{symbol(_X)}  
{list(_X)}  
{var(_X)}  
{nonvar(_X)}  
...
```

Note that `{var(_X)}` prevents `_X` from ever being instantiated, while `{nonvar(_X)}` does nothing at all semantically. One particularly useful example uses a `symbol` restriction in conjunction with variable functors, as in:

```
{symbol(_F)}, _X = _F(_A..).
```

This prevents `_F` from ever being improperly instantiated to a non-symbol.



## Test and Generate

It is usually easy to transform the standard technique of generate and test,

```
p :- generate(_X), test(_X).
```

into a constraint program of the form

```
pc :- testc(_X), generate(_X).
```

where `testc` is a constraint version of `test`.

## Eight Queens Program

For example, the traditional Prolog solution to the eight queens problem becomes

```
eight_qn([_X1, _X2, _X3, _X4, _X5, _X6, _X7, _X8]) :-  
    once(qn([_X1, _X2, _X3, _X4, _X5, _X6, _X7, _X8])),  
    permutation([1, 2, 3, 4, 5, 6, 7, 8],  
                [_X1, _X2, _X3, _X4, _X5, _X6, _X7, _X8]).  
  
eight_qn(_).  
  
qn([]).  
  
qn([_X, _Y..]) :- safe(_X, _Y, 1), qn(_Y).  
  
safe(_X, [], _Nb).  
safe(_X, [_F, _T..], _Nb) :-  
    {noattack(_X, _F, _Nb)},  
    _NewNb is _Nb + 1,  
    safe(_X, _T, _NewNb).  
  
noattack(_X, _Y, _Nb) :-  
    (_Y <> _X), (_Y <> _X - _Nb), (_Y <> _X + _Nb).  
  
permutation([], []).  
  
permutation(_Xs, [_Z, _Zs..]) :-  
    choose(_Z, _Xs, _Ys), permutation(_Ys, _Zs).
```

In this program, the changes consist of using "{}" on the `noattack` predicate, and reversing the order of the calls in the



mainline. The resulting code runs approximately five times faster, due to the more efficient backtracking strategy made possible by constraints.

## Sound Negation and Negative Knowledge

Recall that `not(P)` means that `P` is not provable, rather than that `P` is not true. Passive constraints provide a mechanism for expressing some instances of the second concept. Consider

```
{not(P)}
```

where `P` is any goal. This has the effect of deferring the evaluation of `not(P)` until `P` is ground, resulting in a Boolean negation. This is sometimes called *sound negation*.

One elementary use of this construct is to eliminate exceptions. For example,

```
{list(_X), not(_X = [])}.
```

eliminates the case of `_X` being an empty list in all subsequent computation. Another important construct, sometimes called `dif`, defined by

```
dif(_X, _Y) :- {not(_X = _Y)}.
```

ensures that `_X` and `_Y` are never fully instantiated to the same values. This is our first example of a *joint constraint*. It is, of course, easier to just write `{_X \= _Y}`.

In general, the representation and use of *negative knowledge* in Prolog applications has been difficult both conceptually and technically. In most cases, the use of constrained negation is found to provide the correct, as well as most elegant and efficient, solution.



### Dinner Party Program

To illustrate the use of constrained negation, consider the description of a dinner party attended by four couples:

The following is information about four couples named Smith, Jones, Brown, and White, and their comments regarding a dinner party they attended. Their first names are Joan, Mary, Alice, Kathy, Bob, Charles, John, and Ron. What is each person's full name, and what did that person say:

- Charles's wife did NOT say it was delicious.
- Mrs White said it was delicious.
- Joan did NOT say it was heavenly, and her husband did NOT say it was boring
- Bob said it was boring, and he is not married to Alice.
- Charles is NOT married to Kathy or Alice.
- Ron is NOT married to Alice.
- Mary Smith said it was fabulous.
- John said it was catered.
- Charles, whose last name is NOT Smith, said it was tasteful.
- Mr Jones said it was fattening.
- Mrs Jones did NOT say it was nouvelle cuisine.



Using a conventional Prolog solution, the negative information is checked last, after everything is ground:

```
/* list items contain title, first name, last name,
statement*/

dinner_party1(_Info) :-
    _Info =
        [['Mrs', _, 'White', delicious],
         ['Mr', 'Bob', _LnameB, boring],
         ['Mr', _, 'Jones', fattening],
         ['Mrs', 'Mary', 'Smith', fabulous],
         ['Mr', 'Charles', _LnameC, tasteful],
         ['Mr', 'John', _, catered],
         [_, _, _, heavenly],
         [_, _, _, 'nouvelle cuisine']
        ],

    % true list items
    member(['Mrs', 'Joan', _LnameJ, _], _Info),
    member(['Mrs', 'Alice', _LnameA, _], _Info),
    member(['Mrs', 'Kathy', _LnameK, _], _Info),
    member(['Mr', 'Ron', _, _], _Info),
    member(['Mrs', _, 'Jones', _], _Info),
    member(['Mrs', _, 'Brown', _], _Info),
    member(['Mr', _, 'Brown', _], _Info),
    member(['Mr', _, 'Smith', _], _Info),
    member(['Mr', _, 'White', _], _Info),

    % untrue list items
    not(member(['Mr', 'Charles', 'Smith', _], _Info)),
    not(member(['Mrs', _, 'Jones', 'nouvelle cuisine'],
               _Info)),
    not(member(['Mrs', 'Joan', _, heavenly], _Info)),
    not(member(['Mr', 'Ron', _LnameA, _], _Info)),
    not(member(['Mr', 'Charles', _LnameA, _], _Info)),
    not(member(['Mr', 'Charles', _LnameK, _], _Info)),
    not(member(['Mr', _, _LnameJ, boring], _Info)),
    not(member(['Mrs', _, _LnameC, delicious], _Info)),
    not(member(['Mrs', 'Alice', _LnameB, _], _Info)).
```



```
/*  
?- dinner_party1(_X).  
*/
```

A solution using sound negation would be

```
dinner_party2(_Info) :-  
    {_LnameC \= 'White', _NJoan \= 'Joan',  
     _LnameJ \= _LnameB, _LnameB \= _LnameA,  
     _LnameC \= _LnameK, _LnameC \= _LnameA,  
     _LnameR \= _LnameA, _LnameC \= 'Smith',  
     _NJones \= 'Jones'  
    },  
  
    _Info =  
    [  
        ['Mrs', _, 'White', delicious],  
        ['Mr', 'Bob', _LnameB, boring],  
        ['Mr', _, 'Jones', fattening],  
        ['Mrs', 'Mary', 'Smith', fabulous],  
        ['Mr', 'Charles', _LnameC, tasteful],  
        ['Mr', 'John', _cat, catered],  
        [_, _NJoan, _, heavenly],  
        [_, _, _NJones, 'nouvelle cuisine']  
    ],  
  
    % true list items  
    member(['Mr', _, 'White', _], _Info),  
    member(['Mrs', 'Joan', _LnameJ, _], _Info),  
    member(['Mrs', 'Alice', _LnameA, _], _Info),  
    member(['Mr', 'Ron', _LnameR, _], _Info),  
    member(['Mrs', 'Kathy', _LnameK, _], _Info),  
    member(['Mr', _, 'Smith', _], _Info),  
    member(['Mrs', _, 'Jones', _], _Info),  
    member(['Mrs', _, 'Brown', _], _Info),  
    member(['Mr', _, 'Brown', _], _Info).  
  
/*  
?- dinner_party2(_X).  
*/
```



---

In more complex cases of this kind, the constraint solution is far more efficient than traditional solutions, and also easier to express.

In summary, passive constraints convert Prolog programs into ones which are a monotone and persistent, thus making them part of pure Prolog. This improves the declarative reading of programs, and often significantly improves efficiency. Other control mechanisms for making programs more efficient are described in the next chapter.







# Chapter 8

## Control

---

As described in the chapter "Pure Prolog", programs that are pure are frequently nonterminating due to uninstantiated variables, or inefficient since questions containing uninstantiated components may generate an infinite number of answers. The chapter "Filters and Negation" covers the use of filters to fail branches of a computation where the instantiation state is insufficient for termination. The chapter "Passive Constraints" discusses the use of deferred evaluation in hopes that the instantiation state becomes sufficient for evaluation. This chapter describes other methods of eliminating nonterminating or unnecessary computation branches, as well as methods of directing execution.

### Eliminating Computation Branches

The basic primitive for eliminating branches of the search tree is the cut primitive. The standard cut, written as "!", removes all choicepoints created since the call of the parent goal. Generally, the parent goal, the nearest up the call chain, is the goal that calls the "!" primitive. Exceptions to this which are transparent to the "!" operation are the `or` operator, ";" and the conditional `if-then` operator, "`->`" described later in this chapter. For example, in

```
p :- [q, r, !, s, t].  
p :- [u, v, w].
```

execution of "!" removes any choicepoints for `q`, `r`, and `p`, and the second clause for `p` is never executed. The success of `p` now depends solely on the outcome of `s` and `t`. The computation is said to be committed to finishing the first clause. Note that the frequently used sequence `!, fail` causes the parent goal to fail completely.



"!" has three formal properties:

- it makes its parent goal deterministic up to the point of the cut
- it is idempotent, that is `!, !` is the same as `!`
- it commutes with all successful deterministic predicates, that is `P, !` is equivalent to `!, P` if and only if `P` is deterministic

The cut operation has some practical difficulties. Since it discards a portion of the computation, using "!" sacrifices completeness unless the discarded branches either are known to fail, or provide only redundant solutions. Furthermore, because the use of cut requires careful procedural reading to determine its precise semantic effects, programs employing it are much harder to understand. In many cases, these negative effects can be alleviated by using some of the higher level control constructs described below.

In short, the semantics of "!" depends on where it is executed rather than where it is written in a predicate. As well, "!" requires the creation of special cases. Otherwise, for example, `;!` would attempt to cut the `or` operation rather than the previous goal.

Frequently it is desirable to cut a goal prior to the immediate parent goal. To avoid the complications of restructuring code or introducing cumbersome workarounds, an ancestral cut of the form

```
cut (name)
```

is provided, where `name` is some ancestor goal. `cut` removes all choicepoints back to and inclusive of the named goal. Because ancestral cut explicitly specifies its target goal, it needs no special cases and can be used with constructs such as `foreach` and `not` without confusion. The predicate `failexit(name)` is the same as `cut(name), fail`, and unconditionally fails the named ancestral goal.



Because the action of "!" depends on where it is executed, it is very difficult to simulate with a Prolog meta-interpreter. However, since the semantics of the ancestral cut are specified explicitly and are independent of the point of execution. (It can be used to simulate itself.)

A special predicate, `block`, which allows the naming of a sequence of goals, provides a target goal for an ancestral cut used within the block. A call to `block` executes all arguments except the first, which is the block name. For example, in

```
a :- p, block(name, q, [r, s], cut(name), t).
```

`name` is the specified block name, and `cut(name)`, if executed, removes any choicepoints that exist in that block.

At the other extreme, "!" sometimes cuts too much and one is forced to introduce additional predicates merely to limit its scope. For this class of problems, the list cut cuts back to the last opening bracket, "[". Recall that clause bodies are always lists and have an implied set of brackets if none are written. Thus in

```
p :- [a, [b, cut], c, cut, e, f].  
p :- [h, i].
```

the first `cut` determinizes `b` only, while the second removes choice points for `a` and `c` but not for `p` itself. There is also a list `failexit` which is equivalent to `cut, fail`.

## Directing Execution

### **once**

The use of `cut` permits certain constructs to be defined that cannot be coded within pure Prolog. The simplest of these is the built-in predicate `once`, which is defined as

```
once(_P) :- _P, !.
```

and provides only the first solution for its argument goal.



If the objective is to obtain the first solution to a goal, and then prevent backtracking, the most intuitive operation is `once`. For example, the non-deterministic predicate `member` (defined in the chapter "Pure Prolog") is inefficient if just used for checking for membership, since it does not quit when the first occurrence of an item is found. If this is the intended use, then replacing the call with `once(member(_x, _list))` is a reasonable optimization. The purity of the `member` predicate is maintained, and one less predicate is defined compared with making another version of `member` that includes a `cut`.

### ***if-then***

Another common method of controlling the flow of execution is the *if condition then goal*, "`->`", with an optional *else goal*, "`;`". The if-then conditional operation is defined by

```
_P -> _Q :- _P, !, _Q.  
_P -> _Q.
```

and has the following general properties:

```
— true -> Q    is the same as      Q  
— fail -> Q    is the same as      true  
— P -> fail    is the same as      not (P)  
— P -> true    is the same as      once(P ; true)
```

A natural extension of the if-then conditional, the if-then-else construct is defined by

```
_P -> _Q ; _R :- [_P, !, _Q].  
_P -> _Q ; _R :- _R.
```

Note that this syntax overloads the meaning of "`;`", which is not quite "or" in this case. This control construct is only occasionally useful in Prolog, and the long cascades of if-then-elses so characteristic of conventional languages should be avoided.



Because the list separator, ",", has weaker precedence than any of the operators, square brackets must be used to group goals in a conditional expression. For example,

```
P :- Q -> R ; S, T.
```

executes either R or S, and then T. If the else expression is changed to

```
[S, T]
```

the predicate executes either R or S, T.

### **foreach**

In many programming languages, one method of controlling the path of execution is the iterative loop. Emulation of some such loops is possible with the built-in predicate `foreach`. Rather than looping on a counter, as is the case in many languages, BNR Prolog loops on execution of what is known as the loop generator.

The `foreach` construct is defined by

```
foreach(_P do _Q) :- _P, _Q, fail.
foreach(_P do _Q).
```

The generator is some nondeterministic goal, `_P`, whose failure ends the loop. As long as the generator continues to succeed, `foreach` executes the goal `_Q` within the loop's boundaries. For example, in the following

```
incomes(_Incomes) :-
    write('Income earned: '), nl,
    foreach(member([_Name, _X], _Incomes) do
        [write(_Name, ': ', _X), nl]).
```

`foreach` continues to test the incomes and display the names of the wage earners in the family, as long as `member` succeeds. Each loop finds the next alternative solution to `member` as result of backtracking. Because backtracking undoes any variable



instantiations from previous solutions, no running counts can be maintained within the `foreach` loop.

Often with iterative loops, there is a need to jump out of the loop if a certain condition occurs. In the previous example, an exit from the `foreach` loop occurs only if `member` fails. Normally, `member` only fails if the end of the list is reached, or an inappropriate list structure is discovered. Failure can be forced by using the built-in predicate `failexit`, which is described under `cut`. In the previous example, the following sequence would abort the `foreach` loop when a salary under 10000 was generated.

```
[(_X >= 10000) ->
  [write(_Name, ': ', _X),nl];
  failexit(foreach)
]
```

### **or**

Although disjunction has been discussed as part of the pure Prolog subset, it merits some mention as a control construct, since it has an impact on the path of execution in a program. The `or` construct is defined by

```
(_P ; _Q) :- _P.
(_P ; _Q) :- _Q.
```

When using the disjunction operator, `;`, if the first option fails, the second option, or branch, is attempted before backtracking takes place. Note that the sequence

```
P, Q ; R, S
```

is parsed as

```
P, (Q ; R), S
```



## Program MAZE

The following program uses such control constructs as `once`, `not`, `"->"`, `;"`, and `cut` to find a path through a maze. Each position in the maze is represented by a number, and its relationship to other positions is defined by the facts `neighbor`.

```

/* queries to turn display route on or off
?- assert(traceroute()). % enter to turn trace on
?- retract(traceroute()).% enter to turn trace off

    query to find a path through the Maze
?- traverse_Maze.
*/

/* MAZE : traverse Maze from start to finish, and
display the first route found */

traverse_Maze :-
    once(route([start], _Path)), write(_Path), nl.

/* found the way out */
route([finish, _Rest..],[finish]) :- cut(route).

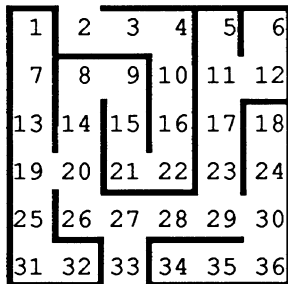
/* find the route by trying a neighbor, if that path
eventually leads to a dead end, then backtrack and
try another neighbor.*/

route([_LastPosn, _Rest..], [_LastPosn, _Path..]) :-
    traceroute -> [write(r, [_LastPosn, _Rest]), nl],
    adjoining(_NextPosn, _LastPosn),
    not(member(_NextPosn, _Rest)),          % avoid looping
    route([_NextPosn, _LastPosn, _Rest..], _Path).

adjoining(_X, _Y) :- % find two adjoining points
    neighbor(_X, _Y) ; neighbor(_Y, _X).

/* check for membership in list */
member(_X, [_X, _Y..]).
member(_X, [_Y, _Z..]) :- member(_X, _Z).
```





### The Maze

```
/* Table of adjoining points in maze */

/* vertical */
neighbor( 1, 7).
neighbor( 7, 13).
neighbor(13, 19).
neighbor(19, 25).
neighbor(25, 31).
neighbor(start, 2)
neighbor( 8, 14).
neighbor(14, 20).
neighbor(20, 26).
neighbor( 9, 15).
neighbor(15, 21).
neighbor(27, 33).
neighbor(33, finish).
neighbor( 4, 10).
neighbor(10, 16).
neighbor(16, 22).
neighbor( 5, 11).
neighbor(11, 17).
neighbor(17, 23).
neighbor(23, 29).
neighbor( 6, 12).
neighbor(18, 24).
neighbor(24, 30).
neighbor(30, 36).

/* horizontal */
neighbor(19, 20).
neighbor(31, 32).
neighbor( 2, 3).
neighbor( 8, 9).
neighbor(26, 27).
neighbor( 3, 4).
neighbor(21, 22).
neighbor(27, 28).
neighbor(28, 29).
neighbor(34, 35).
neighbor(11, 12).
neighbor(29, 30).
neighbor(35, 36).
```



## Active Constraints

Active constraints are conditional operations of the form

```
{condition -> goal}
```

where the declarative meaning is *if and when condition is true, then do goal*. Formally, the semantics of active constraints are defined by

```
{condition -> goal} :-  
    condition ->  
        goal ;  
    first_var(condition, _Variable) ->  
        freeze(_Variable, {condition -> goal}).
```

where `first_var(condition, _Variable)` returns the first variable found in `condition` with a breadth-first scan. `freeze(_V, _G)` succeeds while deferring the execution of `_G` until `_V` is instantiated. Note that if `condition` becomes ground without succeeding, then the constraint succeeds without executing `_G`.

The status of `condition` is retested only when the variable currently being monitored becomes instantiated. It is not affected by instantiations of unmonitored variables. Thus, it is difficult to know exactly when constraints on an arbitrary `condition` are tried, and unwarranted assumptions about the order or frequency of the tests should not be made.

Generally, conditions must be filters since they do no binding, and they must be free of side effects and persistent for the declarative reading to be accurate. Several categories of standard predicates meet all the restrictions on conditions.



Among the most useful are the simple instantiation checks, for example

```
{nonvar(_X, _Y) -> Q}
{integer(_I, _J, _K) -> Q}
{ground(_X) -> Q}
{acyclic(_X) -> Q}
{not(acyclic(_X)) -> fail}.
```

Note that a passive constraint, {P}, can be formally expressed as {ground(P) -> P}. In the last example above, not(acyclic(\_X)) could be used to trigger a warning message or special actions if and when \_X becomes a cyclic structure. Because constraint conditions may be executed many times, they should also be efficient.

Another useful category of predicates, that meet the criteria for conditional operations in active constraints, is that of arithmetic comparisons. They execute their expressions whenever the conditions become both ground and true, and succeed without executing the consequent expression otherwise. For example,

```
{_Distance < 20 -> apply_brake}
{_Distance < 10 -> {_Speed > 60 -> pray}}
```

More complex use of active constraints is illustrated by considering an extended example, which uses only the simplest form of active constraint,

```
{nonvar(_X, _Y, . . .) -> Q}
```

where Q is executed if and when all the variables listed become instantiated (not necessarily fully instantiated). Use of other instantiation checks would be similar.



## Functional Dependencies

A prime tool in the formal analysis of the structure of data bases is the concept of functional dependency. In a relational data base, a functional dependency

$$A \Rightarrow B$$

is said to exist between subsets of the set of attributes ("field names"), if the values of attributes in  $A$  determine those in  $B$ . The set of attributes on the left of " $\Rightarrow$ " is sometimes called a "key" to the set of attributes on the right. For example, an airline database might have functional dependencies such as:

```
[flight_number] => [departure_time, capacity]
[depart_airport, arrival_airport] => [fare]
[flight_number, depart_airport] => [arrival_airport]
```

In database theory, it is important to be able to compute all the functional dependencies derived from a given set of functional dependencies. This is called closure. For example, consider the following where " $\Rightarrow^*$ " represents transitive closure:

[a]	$\Rightarrow$	[b, e, f, g]
[a, c, d, i]	$\Rightarrow$	[h]
[c, d]	$\Rightarrow$	[j]
[c, d, f]	$\Rightarrow$	[k]
[b]	$\Rightarrow$	[g]
[c, f]	$\Rightarrow$	[a, b, e]
[a, c]	$\Rightarrow$	[d]
[a, d]	$\Rightarrow$	[c]
[e, g]	$\Rightarrow$	[b]



An example of transitive closure is

`[c, d, f]            =>*    [a, b, c, d, e, f, g, j, k]`

where	<code>[c, d, f]</code>	<code>gives c, d, f</code>
	<code>[c, d, f]</code>	<code>gives k</code>
	<code>[c, d]</code>	<code>gives j</code>
	<code>[c, f]</code>	<code>gives a, b, e</code>
	<code>[a]</code>	<code>gives b, e, f, g</code>

Some other examples of transitive closure are

<code>[a]</code>	<code>=&gt;*</code>	<code>[a, b, e, f, g]</code>
<code>[b]</code>	<code>=&gt;*</code>	<code>[b, g]</code>
<code>[c, d]</code>	<code>=&gt;*</code>	<code>[c, d, j]</code>
<code>[a, c, d, i]</code>	<code>=&gt;*</code>	<code>[a,b, c, d, e, f, g, h, i, j, k]</code>

If there are  $N$  attributes then computing closure is equivalent to computing a special kind of reachability relation in a graph of  $2 \times N$  nodes. This general problem is significant partly because of a number of special cases:

- If the left operand is always a single element list, and the right operand is interpreted as neighboring elements in a graph, then the closure is the reachability relation of the graph.
- If the right operand is always a single element, and the left operand is interpreted as "prerequisites", then the closure computation provides a test for acyclic directed graphs. This interpretation is useful in ordering data flow computations, for example.
- If the items are interpreted as facts (with lists being conjunctive) and the operator as "implies", then the closure implements an inference procedure for the Horn clause subset of propositional calculus (that is, a forward chaining inference system for a simple implicative logic). The use of active constraints for closure computations enables their completion in times proportional to the length of actual



inference chains and independent of the total size of the knowledge base. This technique is ideal for the very large, but shallow, knowledge bases often encountered in traditional expert systems.

### Closure Program

A conventional Prolog program for computing closures with respect to the above example, using an unordered list as its set representation, and representing " $\Rightarrow$ " as " $\rightarrow$ ", follows:

```
/* basic data */
closure(_In, _Out) :-
    [_FdList =
    . [
        [a] -> [e, f, g],
        [a, c, d, i] -> [h],
        [c, d] -> [j],
        [c, d, f] -> [k],
        [b] -> [g],
        [c, f] -> [a, b, e],
        [a, c] -> [d],
        [a, d] -> [c],
        [e, g] -> [b]
    ],
    clos(_FdList, _In, _Out)
].

/* if generate basic function def'ns _Fds for _I,
   then if _I is subset of _In, and _O is subset of
   _In,
       add _O to _In,
       then recurse,
       else return _In */
```



```
clos(_Fds, _In, _Out) :-
[
    member((_I -> _O), _Fds),    % if
    subset(_I, _In),
    not(subset(_O, _In)),
    union(_O, _In, _In1)
] ->
clos(_Fds, _In1, _Out) ;        % then
_Out = _In.                     % else

/* subset(X, Y) where X is a sublist of Y */
subset([], []).
subset([_X, _Xs..], _Y) :-
    member(_X, _Y), subset(_Xs, _Y).

/* union(X, Y, Z) where Z is concat of X, Y with
duplicates removed */
union([], _X, _X).
union([_X, _Xs..], _Y, _Zs) :-
    member(_X, _Y) ->
        union(_Xs, _Y, _Zs) ;
    union(_Xs, [_X, _Y..], _Zs).
```

Using the conventional approach, a calling pattern to read a set of subsets from a file and compute their closures is

```
[
    initialize(. . .),
    readsubsets(_S),           % generator
    closure(_S, _S_closed),    % compute closure
    writesubsets(_S_closed),   % output
    fail
].
```



The following is a solution to the same problem, based on active constraints. The data structure used to represent subsets is a collection of variables, where instantiated variables correspond to a subset, and  $P \Rightarrow Q$  is simulated by  $\{P \rightarrow Q\}$ :

```
/* closed(_varlist, _corresponding_names) */
closed([_A, _B, _C, _D, _E, _F, _G, _H, _I, _J, _K],
       [_a, _b, _c, _d, _e, _f, _g, _h, _i, _j, _k]) :-
{
    nonvar(_A) ->
        [_B, _E, _F, _G] = [_b, _e, _f, _g],
    nonvar(_A, _C, _D, _I) -> _H = _h,
    nonvar(_C, _D) -> _J = _j,
    nonvar(_C, _D, _F) -> _K = _k,
    nonvar(_B) -> _G = _g,
    nonvar(_C, _F) ->
        [_A, _B, _E] = [_a, _b, _e],
    nonvar(_A, _C) -> _D = _d,
    nonvar(_A, _D) -> _C = _c,
    nonvar(_E, _G) -> _B = _b
}.

```

The constrained approach might be

```
[
    initialize(. . .),
    closed(_S_closed, _names),          % constrain _S_closed
    readsubsets(_S),                    % generator
    _S = _S_closed,                     % compute closure
    writesubsets(_S_closed),            % output
    fail
].

```

The work of setting up the constraints can be done just once "universally", and the closure computation is triggered by unifying with the constrained to be closed object. Even for such a small problem, the constraint version is ten to fifty times faster (depending on the input data). On larger problems the difference is generally greater.



The various flavors of cut provide mechanisms for pruning unwanted (and possibly nonterminating) branches of the solution tree, but are often difficult to understand. `once` and `if-then`, `"->"`, `package cut` to make certain uses more understandable. Other prepackaged control constraints include `or`, `";"`, and `foreach`. The final control construct is the active constraint, which efficiently defers execution of the same goal until a specified condition is satisfied.



## Chapter 9 Operators

---

Use of operators in Prolog provides a notational or syntactic extension to the language. Without them, an arithmetic expression like

`2 + 3 * 4`

would be written in the much less readable functorial form

`'+'( 2 , '*'( 3 , 4 ) )`

Operators can often be used to increase program readability. Mathematical and logical operators in particular allow the program to resemble, as much as possible, the original problem formulation.

Using operators in expressions is only a convenience for reading and writing, although the word operator suggests that some operation or action is performed. Operators (except in a few special cases) are only used as functors to combine objects into structures, although the internal representation of expressions containing operators is in functorial form. The execution semantics of an operator, that is the operation, requires the definition of a clause with the operator symbol as the principal functor.

Syntactically, an arithmetic expression in which there are operators is like any other structure. However, no arithmetic is performed until execution of `is` or any of the arithmetic comparison operators.

Operators are not restricted to arithmetic. Any symbol composed completely of alphanumerics or completely of special characters can be declared as an operator. For instance, if the symbols



photograph, eats and roars are operators, a program could contain the facts

```
photograph elephants.      % photograph(elephants).  
monkey eats banana.        % eats(monkey, banana).  
lion roars.                 % roars(lion).
```

An operator composed of special symbols does not need blanks to separate it from its argument(s). Any other operator, for instance one that looks like a word, does require blanks separating it from its argument(s). Otherwise it would not be possible to determine the division between operator and argument. If there were no blanks in the first of the above examples, the sequence of characters, `photographelephants`, would be just one symbol.

## Specifying an Operator

To declare a symbol as an operator, its position, precedence and associativity must be specified. The positions in which an operator may appear are prefix, infix or postfix. A prefix operator such as `photograph` comes before its one argument. An infix operator comes between its two arguments, for example `eats`, and a postfix operator comes after its one argument, as with `roars`.

An expression may be composed of several subexpressions. Operator precedence is used to group subexpressions, thus establishing the order of evaluation. Values for precedence are integers in the range of 0 (low precedence) to 1200 (high precedence). The subexpression having the operator with the lowest precedence is evaluated first. Alternatively, the operator with the highest precedence is the principal functor of the entire expression. The expression `2 + 3 * 4` is grouped for evaluation as `2 + (3 * 4)` because `"+"` has higher precedence than `"*"`. Arithmetic conventions for evaluation order have been preserved by the appropriate assignment of precedence for the predefined arithmetic operators.



A parenthesized expression has a precedence value of 0. Any subexpression delimited by a set of parentheses is evaluated first. An unstructured object also has precedence value 0. If an argument is a structure, then its precedence is the precedence of the principal functor.

An expression may contain adjacent unparenthesized subexpressions, each with operators of the same precedence. In such a situation, operator associativity defines the order of evaluation, if there is one. Any other order of evaluation must be explicitly defined using parentheses. For example, the division operator, `/`, is left associative, which determines that the expression `32 / 4 / 4` is grouped as `(32 / 4) / 4`. The goal disjunction operator, `;`, is right associative, therefore `goal_1 ; goal_2 ; goal_3` is grouped as `goal_1 ; (goal_2 ; goal_3)`. The comparison operator, `<`, is not associative, so the expression `X < Y < Z` has no evaluation. Adjacent expressions for which there is no associativity must be explicitly parenthesized or a syntax error is returned.

An operator's type, a combination of its position and associativity, is a symbol used in the declaration of the operator. Prefix operators have type:

- `fx`, not associative
- `fy`, right associative

Infix operators have type:

- `xfx`, not associative
- `xfy`, right associative
- `yfx`, left associative

Postfix operators have type:

- `xf`, not associative



— `yf`, left associative

In these type symbols, "`f`" stands for the position of the operator and "`x`" and "`y`" represent the arguments for the operator. "`x`" represents an argument that does not allow associativity with the operator, and "`y`" represents an argument that does allow associativity.

More formally, a left associative operator must have the same or lower precedence argument to its left. Similarly, a right associative operator must have the same or lower precedence argument to the right. This is the interpretation of "`y`" in the type symbol. "`x`" in the type symbol means that the argument must have a strictly lower precedence.

In general, it is good programming practice to parenthesize expressions to enforce the expected evaluation order. The exception to this rule is when the operators used have well known precedence and associativity rules.

The format of an operator definition is

```
op(_Precedence, _Type, _Operator_symbol).
```

The operator declaration must be asserted before its symbol is used as an operator, even including the definition of the operator's semantics. Once it has been declared, all predicates handling Prolog syntax, for example `sread`, `swrite`, `load_context`, and `listener`, recognize expressions using the new operator.

An operator can have two definitions, where the types are either infix and prefix, or infix and postfix, providing the precedences match for each type. Operators cannot have three types, nor can the two types be prefix and postfix. "`-`" is an example of an operator that is both prefix and infix.

The semantics of an operator is the responsibility of the programmer. If there is more than one definition for the same operator, the semantics may be different for each definition.



## Predefined Operators

The list of predefined operators includes

```

op(1200, xfx, :-).      % if
op(1200, fx, :-).      % directive
op(1200, fx, ?-).      % query
op(1100, xfy, ;).      % disjunction, else
op(1050, xfy, ->).      % if-then
op(1000, xfy, &).      % conjunction
op( 950, xfx, where).% constraints
op( 950, xfx, do).      % foreach
op( 700, xfy, :).      % membership
op( 700, xfx, ==).      % arithmetic equality
op( 700, xfx, ==). % arithmetic equality
op( 700, xfx, <>).      % arithmetic inequality
op( 700, xfx, \=). % arithmetic inequality
op( 700, xfx, is).      % arithmetic evaluation
op( 700, xfx, =).      % unifiability
op( 700, xfx, \=).      % not unifiable
op( 700, xfx, <).      % less than
op( 700, xfx, <=).      % less than or equal to
op( 700, xfx, >).      % greater than
op( 700, xfx, >=).      % greater than or equal
op( 700, xfx, @=).      % literal identity
op( 700, xfx, @\=). % literal non-identity
op( 700, xfx, \==). % literal non-identity
op( 700, xfx, @<).      % literal less than
op( 700, xfx, @<=). % literal less than or identical
op( 700, xfx, @>).      % literal greater than
op( 700, xfx, @>=). % literal greater than or identical
op( 500, yfx, +).      % addition
op( 500, yfx, -).      % subtraction
op( 500, fx, -).      % negative
op( 400, yfx, *).      % multiplication
op( 400, yfx, /).      % division
op( 400, yfx, //).      % integer division
op( 300, yfx, **).      % exponentiation
op( 300, xfx, mod). % modulus

```







# Chapter 10

## Cyclic Structures

---

The chapter "Pure Prolog" introduced the concept of cyclic structures, or "rational trees" as they are sometimes called. A cyclic structure is formed when a variable is unified with a term containing that variable, such as in

```
_X = fred(_A, charlie, zebra(3, _X))
```

Cyclic structures can be useful in problems which are naturally described in terms of graphs or networks. Most Prolog systems permit the creation of cyclic structures, but unification of these structures usually results in a stack overflow. (The so-called "occurs check," which prevents cyclic structures from being created, is usually omitted for efficiency reasons.)

This chapter describes the support offered in BNR Prolog for cyclic structures and illustrates their use with examples drawn from the theory of finite state machines. Note that extensive use of large cyclic structures may require additional local stack space.

### Cyclic Structure Support

The support for cyclic structures is concentrated in a few critical operations: unification, the special predicates `acyclic`, `decompose`, and `spanning_tree`, and the general purpose predicates `variables` and `bind_vars`.

Unification is the most basic operation supported on cyclic structures. As demonstrated in later examples, some complex equivalence operations can be mapped directly to unification of cyclic structures, resulting in very simple and efficient implementations.



The monotone filter `acyclic(_Term)` is an inexpensive test for noncyclic structures. Therefore `not(acyclic(_Term))` is a persistent filter for detecting cyclic structures.

One strategy for handling cyclic structures is to decompose them into pieces that are trees, operate on the pieces, and then glue the results back together with unification. `decompose` and `spanning_tree` provide two different ways to decompose a cyclic structure into lists of trees which can be handled by conventional means.

### **decompose**

The syntax for `decompose` is

```
decompose(_Structure, _Tree, _List_of_Unifications)
```

where `_Structure` is the input argument. The output, `_Tree`, is a tree structure which becomes a copy of `_Structure` if `_List_of_Unifications` is executed, where a copy has the same structure but different variables. `_List_of_unifications` is a list of the form

```
[_Var1 = _Tree1, _Var2 = _Tree2, ...]
```

where each `_Varn` is a variable and each `_Treen` is an acyclic structure. The outputs of `decompose` are determined uniquely by the structure of the input, but may be different from the sequence of unifications by which the cyclic term was originally constructed.



An example is the use of `decompose` on the following cyclic structure

```
?- [(_X = fred(_A, charlie, zebra(3, _X))),
    decompose(_X, _T, _L)].

?- [(fred(_A, charlie, zebra(3, fred(_A, charlie,
zebra(3, fred(_A, charlie, zebra(3, [...]))))) =
fred(_A, charlie, zebra(3, fred(_A, charlie, zebra(3,
fred(_A, charlie, zebra(3, [...]))))),
decompose(fred(_A, charlie, zebra(3, fred(_A, charlie,
zebra(3, fred(_A, charlie, zebra(3, fred(_A, charlie,
zebra(3, fred(_A, charlie, zebra(3, [...])))))))),
_T, [(_T = fred(_1, charlie, zebra(3, _T)))]].
YES
```

where, after a certain depth has been reached, a cyclic structure is represented by "[...]".

The outputs of `decompose` are a minimal representation of the internal form of the term. `decompose` replaces each substructure which is referenced more than once with a variable and adds a corresponding unification term to the list. In practice, this is often, but not always, a logically minimal representation as well.

### **spanning\_tree**

The use of `spanning_tree` is similar to `decompose`. The difference between them is that `spanning_tree` performs the same operation as `decompose`, but only after the second occurrence of a substructure. The effect is that the outputs from `decompose` are "minimal" sized pieces, while `spanning_tree` returns "maximal" sized pieces. In particular, the second argument of `spanning_tree` is a maximal tree spanning the graph described by the input.



The following example illustrates the differences between the outputs from `decompose` and `spanning_tree`. Consider the cyclic structure `_X` formed by executing

```
[_X = fred(_X, _Y, _Z),  
  _Y = george(_X), _Z = harry(_Y)].
```

The arguments `_T` and `_L` as returned by `decompose(_X, _T, _L)` are

```
_T,  
[( _T = fred(_T, _1, harry(_1))),  
  (_1 = george(_T))  
]
```

while `spanning_tree(_X, _T, _L)` returns:

```
fred(_1, george(_2), harry(_3)),  
[( _1 = fred(_1, george(_2), harry(_3))),  
  (_2 = fred(_1, george(_2), harry(_3))),  
  (_3 = george(_2))  
]
```

Normal recursive Prolog predicates are meant to apply to trees and generally do not terminate when applied to cyclic structures. The cyclic structure decomposition/recomposition strategy mentioned earlier is implemented by using `decompose` or `spanning_tree` to convert a cyclic structure to a tree, perform the desired transformation, and execute the list of unifications that reconstructs the original structure. The basic techniques are

```
spanning_tree(_Graph, _MaxTree, _List),  
    % extract a maximal tree  
treepred(_MaxTree), % operate on the tree  
_List    % put back missing branches by executing  
    %unifications _Maxtree is a copy of _Graph  
    %on which treepred has operated
```



or

```
spanning_tree(_Graph, _MaxTree, _List),
    % extract a maximal tree
treepred(_MaxTree), % operate on the tree
_Graph = _MaxTree, % unify results back into _Graph
```

The `print` and `portray` predicates use techniques such as these to output cyclic structures in a readable form by means of predicates similar to the following:

```
cyclic_print(_Term) :-
    acyclic(_Term), !, writeq(_Term).

cyclic_print(_Term) :-
    not(not([bind_vars(_Term), cycprt(_Term)]))).

cycprt(_Term) :- decompose(_Term, _T, _List),
    writeq(_T where _List).
```

where is a predefined infix operator with no special meaning used to make printing of expressions involving cyclic structures or constraints easier to read. `bind_vars` binds any variables in the cyclic structure `_Term` to its own symbolic name so it won't be lost during the `decompose` operation. `not(not(. . .))` then undoes these bindings as well as discarding the auxiliary structures built in `cycprt`.

As well as the predicates mentioned previously, cyclic structures can be used with `assert`, `remember`, and `findall`, with one exception. Structures formed by looping a tail variable, such as

```
?- ([_X..] = [a, b, _X..]).
?- ([a, b, a, b, a, b, ...] = [a, b, a, b, a, b, ...]).
YES
```

cannot be copied and produce errors if encountered. However, such looped structures can be unified. For example, `_X` above unifies with `_Y` given by executing

```
?- [(_Y = [a, [_Z..]]), ([_Z..] = [b, a, _Z..])].
```



as seen in the following:

```
?- ([_X..] = [a, b, _X..]), (_Y = [a, _Z..]),  
   ([_Z..] = [b, a, _Z..]), ([_X..] = _Y).  
?- [[a, b, a, b, a, b, ...] = [a, b, a, b, a, b, ...]],  
   ([a, b, a, b, a, b, a, ...] = [a, b, a, b, a, b, a, ...]),  
   ([b, a, b, a, b, a, ...] = [b, a, b, a, b, a, ...]),  
   ([a, b, a, b, ...] = [a, b, a, b, a, b, a, ...])).  
YES
```

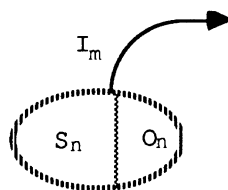
## Finite State Machine Problem

To illustrate the application of cyclic structures, consider some classical problems involving finite state machines. Finite state machines are not only an important topic in theoretical computer science and automata theory, but play an important practical role in both hardware engineering (for example, control circuits for bus protocols) and software engineering (parsing and searching algorithms). A finite state machine can be informally described as a finite set of states,  $S$ , an input alphabet,  $I$ , an output alphabet,  $O$ , the state transition function

$\text{next}(S, I) \rightarrow S$

and the output function

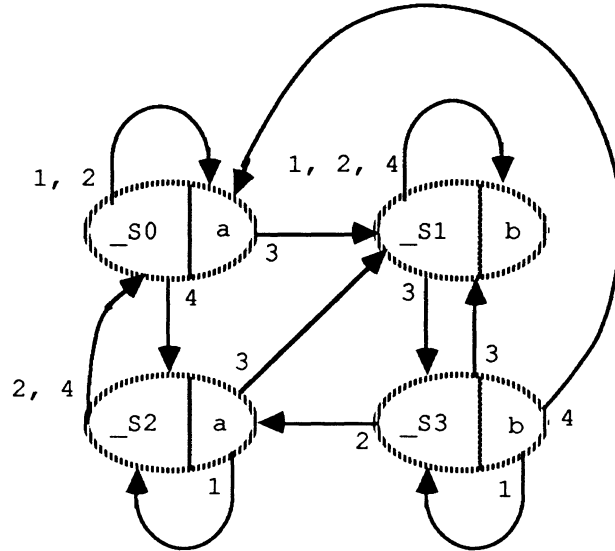
$\text{out}(S) \rightarrow O$



State Description

It is usual to specify one of the states as the initial state,  $S_0$ . A natural and frequently used representation is as a labeled graph called a state transition diagram in which nodes represent states and edges represent transitions.





machine1 State Transition Diagram

Such transition diagrams are usually represented in Prolog as facts if the only object is to simulate their behavior. However, in cases where one is concerned with issues such as testing equivalence of machines or producing implementations with the fewest states, a representation using cyclic structures is more useful. A cyclic structure representation of the above graph is constructed by the following program:

```

machine1(_S0) :-
    _S0 = a(_S0, _S0, _S1, _S2),
    _S1 = b(_S1, _S1, _S3, _S1),
    _S2 = a(_S2, _S0, _S1, _S0),
    _S3 = b(_S3, _S2, _S1, _S0).

```

The various states are represented as variables,  $S_n$ , the input set are integer numbers corresponding to argument positions, and the output map is encoded in the functor names, for example, a or b. By convention, the state machine input value of 1 acts as the identity transition for each state of the final state machine. Its inclusion ensures that `decompose` will return every reachable



state. The initial state is the argument to `machine1`. Each call on `machine1` constructs a new instance of a cyclic structure representing the machine and unifies it with the argument. Note that the cyclic structure representation returned only includes states reachable from `_S0`. Outputs may be left as variables representing "don't care" or "don't know" conditions.

The classic problems of finite state machine theory are

- calculating the behavior from the state description
- testing two machines or states for equivalence
- testing or constructing minimal machines
- calculating sequences for testing a machine

To simulate the behavior of a finite state machine such as `machine1`, we can use

```
/* behavior(_Inputs, _Initial_state,
           _Outputs, _Final_state)
*/

% end of input list
behavior([], _S, [], _S).

% state transition
behavior([_N, _Ns..], _S, [_F, _Fs..], _Final) :-
    arg(_N, _S, _F(_A..)),
    behavior(_Ns, _F(_A..), _Fs, _Final).

simulate(_Input, _Machine, _Output) :-
    _Machine(_M), behavior(_Input, _M, _Output, _).
```

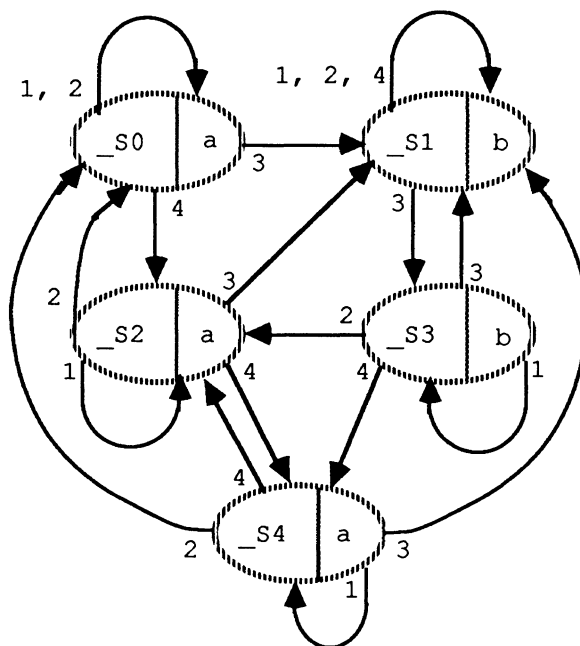
The following query generates the output of the state machine `machine1`, given the input sequence 2, 3, 3, 4.

```
?- simulate([2, 3, 3, 4], machine1, _Out).
   ?- simulate([2, 3, 3, 4], machine1, [a, b, b, a]).
YES
```



Note that this algorithm depends on the use of numbers for the input alphabet. (The use of `arg` makes the query nonlogical, but it requires only two logical inferences per transition.)

Consider the following state transition diagram and its corresponding cyclic structure representation.



machine2 State Transition Diagram

```
machine2(_s0) :-
    _s0 = a(_s0, _s0, _s1, _s2),
    _s1 = b(_s1, _s1, _s3, _s1),
    _s2 = a(_s2, _s0, _s1, _s4),
    _s3 = b(_s3, _s2, _s1, _s4),
    _s4 = a(_s4, _s0, _s1, _s2).
```



From the pictures it may be evident that this machine is equivalent to `machine1`, since states `_S0` and `_S4` are really the "same" state. To test this using the Prolog representations, it is only necessary to unify the two descriptions as follows:

```
% predicate to check for machine equivalence
equivalent(_Machine1, _Machine2) :-
    _Machine1(_M), _Machine2(_M).

?- equivalent(machine1, machine2).
?-equivalent(machine1,machine2).
YES
```

`equivalent` is written to use the names of machines as arguments in order to avoid the echoing of cyclic structures by the listener. Note that unification represents both equivalence of machines and equivalence of states. This is especially useful if either of the machines has some "don't care" conditions (either transitions or outputs).

`decompose` can be used in a utility predicate to extract the list of reachable states from the cyclic structure representation.

```
% bind the states
states(_M, _State_list) :-
    decompose(_M, _M1, _List),
    extract_state(_List, _State_list),
    _List,                % construct copy
    _M = _M1.             % unify copy with original

extract_state([], []).

extract_state([_S = _, _More..], [_S, _Ss..]) :-
    extract_state(_More, _Ss).
```



A state machine is minimal if it has the fewest possible states for a fixed behavior. Neither `machine1` or `machine2` is minimal. Since unification tests state equivalence, minimality can be tested simply by

```
distinct([_X]).  
  
distinct([_X, _Xs..]) :-  
    not(member(_X, _Xs)),  
    distinct(_Xs).  
  
minimal(_Machine) :-  
    _Machine(_M),  
    states(_M, _States),  
    distinct(_States).  
  
?- minimal(machine1).  
NO  
  
?- minimal(machine2).  
NO
```

The same technique provides a solution to another classic problem of automata theory, that of constructing a minimal machine equivalent to a given one. Given a cyclic structure representation, use `decompose` to generate a list of unifications which construct a duplicate machine and to extract its state list. Then use `member` to check that each additional state is distinct from the states already processed, otherwise merging it with its "partner".



```
/* state minimization-
   ( minimize machine in cyclic structure format)
*/
minimize(_Machine, _Minimal_machine) :-
    _Machine(_M),           % get machine definition
    states(_M, _States),    % extract list of states
    decompose(_M, _Minimal_machine, _List),
    construct(_States, _List, [], []).

/* construct(original_states, new_states,
               distinct_orig_states, distinct_new_states)
*/
construct([], [], _, _).    % all done
construct([_S, _Ss..], [_V = _Tree, _Xs..], _Old,
          _New) :-
    member(_S, _Old), !,    % redundant state
    member(_Tree, _New),    % amalgamate state
    construct(_Ss, _Xs, _Old, _New).
construct([_S, _Ss..], [_V = _Tree, _Xs..], [_Old..],
          [_New..]) :-
    _V = _Tree,             % construct machine
    construct(_Ss, _Xs, [_S, _Old..], [_V, _New..]).

display_min(_Machine) :-
    minimize(_Machine, _M), cyclic_print(_M).

?- display_min(machine2).
_T where [
  (_T = a(_T, _T, _1, _T)),
  (_1 = b(_1, _1, _2, _1)),
  (_1 = b(_1, _T, _2, _T)),
]
?- display_min(machine2).
YES
```

Another common problem when dealing with finite state machines is finding a set of test sequences to validate every transition. With the cyclic structure representation, `spanning_tree` can be used to create a maximal tree from the initial state to every other state. A nondeterministic version of



behavior can be applied to the result to generate a complete set of test vectors. The use of `spanning_tree` ensures that the generated sequences do not contain any unnecessary loops. The algorithm is written to stop after the first transition not in the spanning tree is reached. Thus, coverage is over all transitions and not just all states.

```

/* ndbehavior(_Inputs, _Initial_state, _Outputs,
               _Maxinputs)
/*
% first out of tree transition so stop
ndbehavior([], _O(_..), [], _) :- var(_O), !.
ndbehavior([_N, _Ns..], _O(_A..), [_F, _Fs..],
           _Maxinputs) :-
    integer_range(_N, 1, _Maxinputs), % try all outpaths
    arg(_N, _O(_A..), _F(_A1..)), % state transition
    ndbehavior(_Ns, _F(_A1..), _Fs, _Maxinputs).% recurse

test_sequence(_Machine, _Inputs, _Outputs) :-
    _Machine(_M), % get machine definition
    termlength(_M, _Maxinputs, []),
    spanning_tree(_M, _Mtree, _),
    ndbehavior(_Inputs, _Mtree, _Outputs, _Maxinputs),
    _M = _Mtree. % unify machines

?- test_sequence(machinel, _I, _O).
?- test_sequence(machinel, [1], [a]).
?- test_sequence(machinel, [2], [a]).
?- test_sequence(machinel, [3, 1], [b, b]).
?- test_sequence(machinel, [3, 2], [b, b]).
?- test_sequence(machinel, [3, 3, 1], [b, b, b]).
?- test_sequence(machinel, [3, 3, 2], [b, b, a]).
?- test_sequence(machinel, [3, 3, 3], [b, b, b]).
?- test_sequence(machinel, [3, 3, 4], [b, b, a]).
?- test_sequence(machinel, [3, 4], [b, b]).
?- test_sequence(machinel, [4, 1], [a, a]).
?- test_sequence(machinel, [4, 2], [a, a]).
?- test_sequence(machinel, [4, 3], [a, b]).
?- test_sequence(machinel, [4, 4], [a, a]).
YES

```



These examples illustrate the support for cyclic structures and show how such structures may be used to formulate problems with a natural graph representation, such as finite state machine theory. In cases where cyclic structure representations are appropriate, they can be much more efficient than the typical fact representations for handling problems involving the global structure of the graph.



# Part III Arithmetic







# Chapter 11

## Functional Arithmetic

---

Traditionally, Prolog has been oriented toward symbolic rather than numeric computation. Based on a simple recursive data structure, a purely logical form of integer arithmetic is possible in most Prolog systems, but it is not efficient. A system of functional arithmetic, similar to that described in this chapter, is available in most Prolog systems.

Functional arithmetic is so named because it is deterministic and requires full instantiation of the arithmetic terms in computations. For example, arithmetic comparison operators, such as "=", "<", ">=", evaluate successfully only if their arguments are both ground terms and valid arithmetic expressions. Such expressions are combinations of numbers, arithmetic operators, (+, -, \*, /, //) and such arithmetic functions as sin, cos, and min. For a complete list, see the *BNR Prolog Reference Manual*.

Arithmetic evaluations in programs may be a source of nonlogical behavior. For example

```
?- _X = 2, _X < 5.
```

succeeds but

```
?- _X < 5, _X = 2.
```

fails because `_X` is unbound when "<" is called.

The instantiation restriction in functional arithmetic causes not only unwanted failures, but also unidirectional programs. For example, the expression

```
_X is sin(_Y)
```



where `_Y` is a number, computes the `sin` of `_Y` and unifies the result with the variable `_X`. However, using the same expression, it is not possible to compute the `arcsin` of `_X` and bind it to the variable `_Y`.

This chapter examines the methods by which functional arithmetic expressions are evaluated, ways of taking advantage of failures, and ways of avoiding failures by using constraints to delay the execution of arithmetic expressions until the terms are ground.

## Evaluation

The operator `is` forces evaluation of the right operand, which must be an arithmetic expression. This fails if the operand contains any uninstantiated variables or otherwise cannot be evaluated. If the left operand is a number, the result of the evaluation is equated with it. However, if the left operand is a variable, then the result of the evaluation is bound to it. For example,

```
?- 5.0 is 2 + 3.      % succeeds, 5.0 equals 5
?- _A is 2 + 3.      % succeeds, _A is bound to 5
?- _A is 2 + _X.     % fails, _X is a variable
?- _A is 2 + x.      % fails, "x" cannot be evaluated
?- _A is sqrt(4).    % succeeds, evaluates sqrt(4)
?- _A is sqrt(-1).   % fails, cannot evaluate sqrt(-1)
?- _X = 4 // 2.      % binds _X to the term '//'(4, 2)
```

Arithmetic comparison operators force the evaluation of both the left and the right operands. The results of the evaluations are then compared arithmetically. For example,

```
?- (5 + 1) == (2 * 3). % succeeds, both evaluate to 6
?- (_X + 1) == (2 * 3). % fails, _X is not a number
?- sin( $\pi/3$ ) > cos( $\pi/3$ ). % succeeds
?- sin(40) > cos(_X). % fails, _X is not a number
?- 5 == 5.0.          % succeeds, values equal
?- 2 <> 3.             % succeeds, values not equal
?- 2 <> 2.             % fails, values equal
?- 2 <> _X.            % fails, expression not ground
```



```
?- not(2 == _X).           % succeeds, comparison fails,  
?- _X is (5.6 * 2) // 3. % succeeds, evaluates to 3
```

Note that the last query evaluates the right operand to 3 because the last operation performed is integer division, `//`. If the last operator is changed to `/`, `_X` binds to 3.73333.

Some Prolog systems produce error conditions when an invalid arithmetic expression is evaluated. The pragmatic reason for this is that an error message quickly informs the programmer of difficulty, simplifying the detection of these particular bugs. However, evaluation failures can be usefully exploited, as shown in the next section, or avoided completely through the judicious use of constraints.

## Extending Functional Arithmetic

With any language, the validation of arithmetic formulae in an arithmetically intensive application is a major problem. If writing a recursive procedure computing, for example, the evaluation of a polynomial, how does one ensure that the formula computed by the program is the correct one? When given symbolic or uninstantiated arguments, a function can compute symbolic expressions for validation against the specification by using the semantics of failure.

Consider the problem of writing a procedure `horner` that either computes the value of a polynomial or returns a symbolic expression if the polynomial cannot be evaluated. A typical set of questions and answers for this procedure might be

```
?- horner([a, b, c], x, _R).  
   ?- horner([a, b, c], x, (a + (x * (b + (x * c))))).  
YES  
  
?- horner([2, 4, 6], 5, _R).  
   ?- horner([2, 4, 6], 5, 172).  
YES
```



```
?- horner([2, _a, 3], 5, _R)
?- horner([2, _a, 3], 5, (2 + (5 * (_a + (5 * 3))))).
YES
```

One way to implement such a procedure is to define an infix operator ":-" that either performs an arithmetic evaluation, if it is possible to do so, or otherwise simply unifies its arguments.

```
op(700, xfx, ':-').
_X :- _Y :- _X is _Y, !.
_X :- _X.
```

The horner procedure for polynomials is then defined by

```
% horner(_Coefficients, _X, _Result)
horner([_A], _, _Result) :- !, _Result := _A.
horner([_A, _As..], _X, _Result) :-
    horner(_As, _X, _NewResult),
    _Result := _A + _X * _NewResult.
```

All occurrences of `is` used in computing values of a polynomial are replaced with `:-`. The resulting arithmetic functions, constructed with `:-`, compute numerically when given valid numeric arguments, but if the functions fail, they are forced to operate "symbolically".

## Vector Arithmetic Program

It is often useful to extend a language by providing user written arithmetic or nonarithmetic functions which are evaluated in a functional language. Consider a system that performs functional arithmetic on vectors, represented as lists of numbers, which can answer questions such as

```
% scalar arithmetic
?- (_X := 2 + 3).
?- (5 := (2 + 3)).
YES
```



```

% multiply vectors
?- _X := [2, 3, 4] * [3, 4, 1].
   ?- ([6, 12, 4] := ([2, 3, 4] * [3, 4, 1])).
YES

% add scalar to vector
?- _X := 3 + [1, 2, 3].
   ?- ([4, 5, 6] := (3 + [1, 2, 3])).
YES

% multiply vectors by scalars, then add vectors
?- _X := 4 * [1, 2, 3] + 2 * [7, 8, 9].
   ?- ([18, 24, 30] := ((4 * [1, 2, 3]) +
      (2 * [7, 8, 9]))).
YES

% inner product of vectors
?- _X := [1, 2, 3] ^ [2, 3, -1].
   ?- (5 := ([1, 2, 3] ^ [2, 3, -1])).
YES

```

A simple way to do this is to replace the definition of ":"=" above with

```

_Result := _ScalarExp :- _Result is _ScalarExp, !.
    % _ScalarExp is a scalar expression
[_Vector..] := [_Vector..] :- !
    % [_Vector..] is a vector
_Result := _VectorExp :- reduce(_VectorExp, _Result).
    % _VectorExp is a vector expression

op(700, xfx, ':=').      % assignment operator
op(500, xfx, '^').      % dot product operator

```

and provide a reduce procedure such as the one developed in the chapter "Pure Prolog". The first rule for this version of reduce handles evaluation of the inner products of vectors. The second rule handles all the other cases by insuring that \_Op is a valid vector operator, evaluating the operands individually, and performing the appropriate operation on the results, as can be seen in the following program:



```
% reduce arithmetic operations on vectors, scalars,
% or combinations thereof
reduce((_X ^ _Y), _Z):- !, inner(_X, _Y, _Z).
reduce(_Op(_X, _Y), _Z) :-
    vector_op(_Op),
    _X1 := _X,
    _Y1 := _Y,
    !,
    vector(_X1, _Y1, _Op, _Z).

% code for computing inner product
inner([], [], 0) :- !.
inner([_X, _Xs..], [_Y, _Ys..], _Z) :-
    inner(_Xs, _Ys, _Z0),
    _Z is _X * _Y + _Z0.

% list of valid vector operators
vector_op(+).
vector_op(-).
vector_op(/).
vector_op(*).
vector_op(min).
vector_op(max).

% case 1: last 3 arguments must be vectors, that
% is numeric lists, of equal length
vector([], [], _Op, []) :- !.
vector([_X, _Xs..], [_Y, _Ys..], _Op,
    [_Z, _Zs..]) :-
    !,
    _Z is _Op(_X, _Y),
    vector(_Xs, _Ys, _Op, _Zs).

% case 2: argument 2 must be a number
vector(_X, [], _Op, []) :- numeric(_X), !.
vector(_X, [_Y, _Ys..], _Op, [_Z, _Zs..]) :-
    !,
    numeric(_X), _Z is _Op(_X, _Y),
    vector(_X, _Ys, _Op, _Zs).
```



If the `is` operators in `vector` and `inner` are changed to `:=`, the components themselves can be vectors. Thus, the program can be significantly generalized.

Techniques such as these take advantage of the failure semantics for invalid arithmetic expressions, providing an alternative interpretation if they cannot be numerically evaluated.

## Arithmetic with Constraints

Putting passive constraints on terms that evaluate arithmetic expressions guarantees that they will not fail because of unbound variables. Just as passive constraints restore the persistent or monotone properties to filters, they also alleviate some of the illogical behavior caused by the evaluation of arithmetic expressions.

*Passive constraints* are used with a simple arithmetic comparisons, where the execution of the constrained expressions are deferred until the variables in the expressions become ground. Some examples are

```
{_X > 0}
{_X >= _Y}
{_X <> _Y}
{_X + _Y == _Z}
```

In general, any arithmetic comparison can be the condition in an *active constraint* which has the form

```
{condition -> action}
```

For example,

```
{_T > 2000 -> shut_off_reactor}
```

is read as:

```
"if and when (_T > 2000) do shut_off_reactor".
```



Active constraints may also be nested, as in

```
{(_T > 4000) -> {(_Distance < 10000) -> flee_madly}}
```

which has the same meaning as

```
{[_T > 4000, _Distance < 10000] -> flee_madly}
```

*Data flow constraints* are used on *is* expressions, and have the form

```
{_X is expression}
```

Execution of *is* is deferred only until *expression* is ground. Data flow constraints restore the commutative properties that are lost by functional arithmetic. With the use of data flow constraints, both of the following examples succeed:

```
?- _X = 2, {_Y is _X + _X}.  
?- {_Y is _X + _X}, _X = 2.
```

### Critical Path Scheduling Problem

One application using data flow constraints solves the problem of computing a critical path using a Program Evaluation and Review Technique (PERT) chart. To define such a chart, one must determine if any activity precedes another, and the time required to complete each activity.

The time required to complete the activity, the activity time, is the difference between an early start and an early finish, or between a late start and a late finish. Slack time is the difference between the latest and the earliest times an activity can complete without disrupting a project. The critical path is the list of activities between start and finish which have no slack time. Consider the following example.



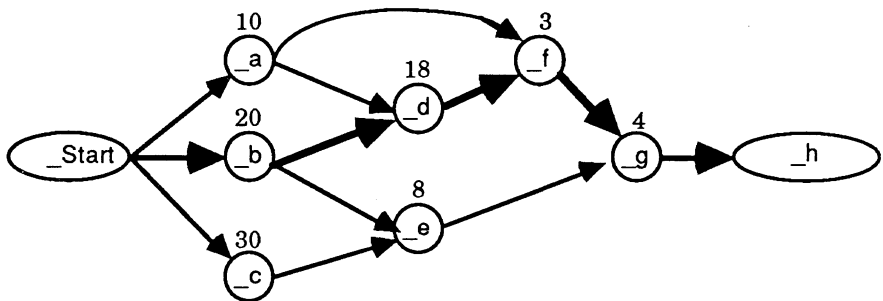
Given a start time, the program provides:

- a list indicating the activity time, the latest start time, the earliest finish time, and slack time for each activity
- the overall time required for the project.

The calling interface looks like

`plan(_Start, _List, _TotalTime)`, where

- `_List` consists of variables for each activity, which are bound to lists of the form `[ActivityTime, EarlyFinish, LateStart, Slack]` where `_Slack` is positive and defined to be  $(LateFinish - EarlyFinish)$ , or  $(LateFinish - (EarlyStart + ActivityTime))$
- `_Start` is a dummy first activity:  
`[0, _StartTime, _, _]`
- `_TotalTime` is the amount of time required to complete the project



Sample Project



```
/* Prolog program defining the "Sample Project" is: */
plan(_Start, [_a, _b, _c, _d, _e, _f, _g, _h], _Finish)
:-
    activity([_Start], _a, [_d, _f], 10),
    activity([_Start], _b, [_d, _e], 20),
    activity([_Start], _c, [_e, _g], 30),
    activity([_a, _b], _d, [_f], 18),
    activity([_b, _c], _e, [_g], 8),
    activity([_a, _d], _f, [_g], 3),
    activity([_e, _f], _g, [_h], 4),
    finish([_g], _h, _Finish).

activity(_Prereq, [_ATime, _EFinish, _LStart, _Slack],
        _Depend, _ATime) :-
    early_start(_Prereq, _EStart),
    {_EFinish is _EStart + _ATime},
    late_finish(_Depend, _LFinish),
    {_LStart is _LFinish - _ATime},
    {_Slack is _LFinish - _EFinish}.

finish(_Prereq, [0, _EStart, _EStart, 0], _EStart) :-
    early_start(_Prereq, _EStart) .

early_start([[_ , _EFinish, _ , _]], _EFinish) :- !.
early_start([[_ , _EFinish, _ , _], _Prereq..], _Max) :-
    {_Max is max(_EFinish, _EFinish2)},
    early_start(_Prereq, _EFinish2).

late_finish([[_ , _ , _LStart, _]], _LStart) :- !.
late_finish([[_ , _ , _LStart, _], _Subseq..], _Min) :-
    {_Min is min(_LStart, _LStart2)},
    late_finish(_Subseq, _Lstart2).

/* Display PERT schedule and total hours required*/
schedule :-
    plan([0, _Start, _ , _], _List, _Total),
    _Start is 0,
    nl, write('Total: ', _Total),
    nl, write('[ATime, EFinish, LStart, Slack   ]'),
    nl, print_schedule(_List).
```



```
print_schedule([]) :- nl.
print_schedule([_X, _Xs..]) :-
    write('['),
    row(_X),
    write(']'), nl,
    print_schedule(_Xs).

row([]) :- !.
row([_X, _Xs..]) :-
    _X < 10 ->          % adjust spacing
        write(' '),
    write(_X, ' '),
    row(_Xs).
```

Note that this program can be generalized if the activity times are not supplied as part of the program, but are supplied as arguments.

Although constraints restore some logical properties, the unidirectional behavior of functional arithmetic does not express true arithmetic relations. This issue is examined in the chapter "Relational Arithmetic".







# Chapter 12

## Relational Arithmetic

---

BNR Prolog introduces a new data type for numbers that is distinct from floating point or integer: the type *interval*. An interval is an object that represents a real number lying between a lower and an upper bound. The bounds of an interval are floating point numbers that define its *range*, which may vary over the life of the computation.

There are three principal characteristics of intervals that make them interesting and useful. Intervals

- allow the expression of arithmetic relationships in a fully logical and relational manner
- provide a means for proving statements about true *real* numbers without being restricted to the limitations of a particular floating point representation
- offer a mechanism for solving sets of linear and nonlinear equations and inequalities

In this chapter, the primitives for creating and manipulating intervals are explained, and several detailed examples using intervals are presented.

Intervals provide a foundation for relational arithmetic, as opposed to functional arithmetic. As the previous chapter points out, arithmetic in most Prologs cannot be read or understood declaratively; functional arithmetic obliges programmers to write procedural programs. For example, an ordinary arithmetic procedure such as

```
square(_X, _Y) :- _Y is _X ** 2.
```



can be used only one way: to determine  $\_Y$  when  $\_X$  is bound. The same equation cannot be used to determine the square root of  $\_Y$ . The query

```
?- square(_X, 4).
```

fails because  $\_X$  is unbound. For `square` to be a reversible procedure (assuming  $\_X$  is greater than 0), another rule is needed

```
square(_X, _Y) :- _X is sqrt(_Y).
```

IF  $\_X$  is not greater than 0, even this is insufficient. This limitation in the semantics of functional arithmetic is removed when arithmetic expressions operate on intervals, as explained later in this chapter.

Another advantage of intervals is that they offer a means of correctly performing arithmetic on real numbers rather than floating point numbers. This distinction is an important one. In most computer languages, so-called real number arithmetic is performed with floating point numbers. However, these floating point numbers are not strictly the same as the reals that they approximate, and the errors induced by arithmetic operations on them can quickly accumulate. Many numbers cannot be expressed in the finite representation supported by most computer systems. Thus,  $1.1 * 1.1$  is often 1.20999 rather than 1.21.

Because of rounding problems, floating point numbers do not obey all the axioms of real arithmetic, for example, the associative law of addition. Evaluation of logically equivalent expressions such as  $(X + Y) + Z$  and  $X + (Y + Z)$  yield different numerical results and the equality fails.

Operating on intervals permits the automatic tracking of the inherent worst-case imprecision of floating point arithmetic. As a consequence, such formal properties of real arithmetic are restored, and equations such as the above always succeed. Note that numerically induced errors in equality (or inequality) tests can be arbitrarily magnified when they are used to control the



flow of execution of a program, so restoring correctness to these elementary operations is not just a matter of a slight improvement in accuracy, but a qualitative change in the nature of numerical processing.

Interval arithmetic can also be used to solve sets of simultaneous linear or nonlinear equations, by systematically narrowing the intervals defined by those equations. These techniques are discussed later.

## The Interval Type

Intervals are different from any other Prolog data type. In some respects an interval is like a symbol: it has a unique identity and can unify only with itself or an unbound variable. In other respects, an interval is analogous to a variable: it narrows its value as computation moves forward and returns to a previous value by backtracking.

### **range**

An interval can be created or queried using the `range` predicate. If a goal of the form

```
range(_I, [_Lower, _Upper])
```

succeeds, it expresses the relation that `_I` is an interval with the floating point bounds `_Lower` and `_Upper`. The question

```
?- range(_I, [1.2, 4.5]).
```

binds `_I` to a system generated object with a name such as

```
_Interval_334260
```

Although it looks like a variable, this term should be thought of as the name of a real number that is initially between the bounds 1.2 and 4.5 inclusive. This is called the *numeric* interpretation.



An interval term can also be interpreted as the set of all real numbers that are in the closed segment between its lower and upper bounds. This is called the *regional* interpretation.

There are occasions when one interpretation is preferable to the other, but both serve only as mental models for understanding the behavior of intervals. There is an analogy here with the two interpretations of a logic variable. A variable can be interpreted as referring either to something specific (but unknown) or to the set of all its valid instantiations.

To illustrate some basic characteristics of intervals, consider the queries

```
?- range(_I, [1.2, 4.5]), _I = 2.6.
```

and

```
?- range(_I, [1.2, 4.5]), _I = 3.3.
```

Both of these succeed because 2.6 and 3.3 are points which exist in the interval between 1.2 and 4.5. However, the question

```
?- range(_I, [1.2, 4.5]), _I = 2.6, _I = 3.3.
```

fails because the same real number `_I` that lies in the range 1.2 to 4.5 cannot be both 2.6 and 3.3. The question

```
?- range(_I, [1.2, 4.5]), _I = 6.7.
```

also fails, this time because 6.7 lies outside the range for `_I`. An interval created without specifying the upper and lower bounds, for example

```
?- range(_I, [_ , _]).
```

or more simply,

```
?- range(_I, _).
```

means that `_I` lies between the largest positive and negative interval numbers represented by the system, (approximately  $-1e38$  and  $1e38$ ). These are called *indefinite intervals*.



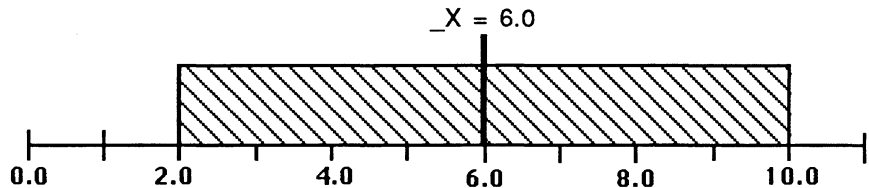
While the usual interpretation of an interval is the numeric one, there are occasions when the regional interpretation is more convenient. For example, it is sometimes necessary to find the midpoint of an interval, or the difference between the upper and lower bounds of an interval. These built-in functions, `midpoint` and `delta`, should be understood in terms of the regional interpretation of intervals.

### **midpoint**

The midpoint of an interval is obtained simply by calling the `midpoint` function on the interval. Thus,

```
?- range(_A, [2, 10]), _X is midpoint(_A).
```

yields the float `_X = 6.0`. In other words the midpoint of an interval is just the arithmetic mean of its upper and lower bounds.

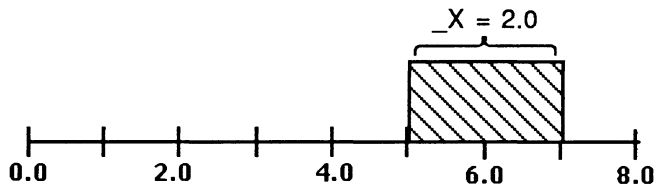


### **delta**

Similarly, the `delta` function returns the difference between the upper bound and the lower bound of an interval, so the query

```
?- range(_A, [5, 7]), _X is delta(_A).
```

returns `_X = 2.0`.





The midpoint and delta functions are used in exactly the same way as other built-in arithmetic functions.

### **range**

The predicate `range` can be used not only to generate bounded intervals but also obtain the upper and lower bounds of a given interval. Thus, the extent to which an interval has been narrowed by a constraint, such as an inequality or an arithmetic function, can be determined by the `range` predicate. The results of a range query are always outward rounded, so the result is always slightly larger and never smaller than the actual interval range. However, this may be lost by converting to decimal notation for printing. For example,

```
:- range(_I, _),           % set the interval
   _I =< 1/3,               % constrain upper bound
   _I >= 1/7,              % constrain lower bound
   range(_I, _R),          % what are bounds now?
   nl, write(_R).
[0.14286, 0.33333]
YES
```

### **print\_interval**

The predicate `print_interval` is used to write a term of type `interval`. When used with just one argument, `print_interval` writes the interval argument to the default output stream. If there are two arguments, it writes the interval argument to the stream specified by the first argument. For example,

```
?- range(_I, [0, 100]),   % set the interval
   _I =< 1/3,              % constrain upper bound
   _I >= 1/7,             % constrain lower bound
   nl, print_interval(_I).
[0.14285, 0.33334]
YES
```

Outward rounding also applies to `print_interval` even when the bounds may be integers, and is carried through the decimal conversions as well.



Note the effect of imposing two different range settings for the same interval:

```
:- range(_I, [2, 8]),           % set the interval
   range(_I, [-1, 3]),         % set it again
   nl, print_interval(_I).     % output range
[2.0, 3.0001]
YES
```

The first constraint imposes a lower bound of 2 and an upper bound of 8. The second further restricts the upper bound to 3. However, the lower bound (2) is already higher than that imposed by the the second constraint (-1), and therefore is not changed. Imposing two different range settings on the same interval is equivalent to constraining the upper and lower bounds with inequalities.

## Narrowing Intervals and Backtracking

As a computation proceeds forward, intervals can change their values only by narrowing. An interval is narrowed if its lower bound is raised, its upper bound is lowered, or both. Backtracking, may undo the narrowing of an interval in a way that is analogous to the unbinding of ordinary variables. The semantics of narrowing can be described as applying additional constraints to an interval. (This should not be confused with "{ }" described in the chapter "Control".)

If a goal narrows an interval such that the interval becomes empty, the goal fails. Such an empty interval results only from an inconsistent set of constraints. For example, the following predicate, *choice*, constrains its interval argument to be either the interval between 1 and 2, or the interval between 10 and 20.

```
choice(_I) :- range(_I, [1, 2]).
choice(_I) :- range(_I, [10, 20]).
```



Then the goal

```
?- choice(_I), _I >= 13, _I =< 15.
```

tries the first rule for `choice` and fails because 13 is greater than any real number between 1 and 2. Upon backtracking, `_I` is constrained by the second rule for `choice` and succeeds because it is possible to further constrain the real numbers between 10 and 20 to be between the range specified in the goal (13 and 15). On the other hand, the goal

```
?- choice(_I), _I >= 100, _I =< 1000.
```

cannot succeed for either rule in `choice`.

Unification does not narrow intervals; only arithmetic operations have this property. To find the extent to which a number of intervals have been narrowed after they have been constrained, it is helpful to write a variadic predicate that displays the current range of a sequence of intervals. In the following example, note the unification of intervals with unbound variables:

```
write_intervals() :- nl.  
write_intervals(_I, _Is..) :-  
    print_interval(_I),  
    write_intervals(_Is..).
```

Even though two intervals initially have the same range, downstream computation may affect the bounds of one interval differently than the other, depending on what further constraints are imposed on them. For instance, in the query

```
?- range(_A, [2, 3]),  
    range(_B, [2, 3]),  
    _A =< 2.5,  
    _B >= 2.5,  
    print_interval(_A, _B).
```

`_A` and `_B`, which initially have the same range, are narrowed to the ranges `[2.0, 2.5001]` and `[2.5, 3.0001]` respectively.



Interval arithmetic generated by equality, inequality, and basic arithmetic functions has all the properties of pure Prolog. Narrowing and backtracking have been discussed, as has the impact of failure on consistency. In addition, interval arithmetic is:

- monotone: If an interval has an initial range  $R_i$  and a final range  $R_f$ , then an initial range  $Q$  where  $R_i \supseteq Q_i$  narrows to a final range  $Q_f$  where  $R_f \supseteq Q_f$ . (This property is sometimes called "inclusion isotone" in interval arithmetic literature).
- idempotent: Applying an interval equation or inequality twice is the same as applying it once.
- commutative: The order of application of interval equations and inequalities does not affect the result. (Note: interval arithmetic does not commute with functions such as `range` query or `delta` which access the current value of the range.)
- persistent: to a degree. Like constraints, interval equations or inequalities can cause failure downstream, but failure does not occur as long as there might be a solution.

## Evaluating Interval Expressions

To make effective use of intervals, it is important to understand the behavior of intervals when arithmetic expressions that reference them are evaluated. Like floats and integers, intervals can be components of such arithmetic expressions as equations, inequalities or functions. There is a key difference between the results of such operations with ordinary numbers (which maintain constant values), and analogous operations with intervals. The ranges of intervals may be narrowed when an arithmetic expression is evaluated to maintain the truth of the expression. Such intervals are said to have been *constrained* by the arithmetic expression.

Interval arithmetic supports the evaluation of such operations as "+", "-", `sin`, `cos`, equalities and inequalities. With the exception



of exponentiation, "\*\*", where exponents are always integers, floating point and integer numbers are coerced to intervals by `is`. Constraints, "{ }", if applied, have the same semantics as described in the previous chapter.

## Equality

If two different intervals are intended to have the same values it is sufficient to constrain them with an equality constraint. For example,

```
:- range(_A, [2, 4]),
   range(_B, [1, 3]),
   _A == _B,
   nl, write_intervals(_A, _B).
[2.0, 3.0001] [2.0, 3.0001]
YES
```

constrains `_A` and `_B` to be the same henceforth. Any constraint on `_A` also applies to `_B`, just as any constraint on `_B` also applies to `_A`. Thus,

```
:- range(_A, [2, 3]),
   range(_B, [1, 5]),
   _A == _B,
   _B <= 2.4,
   _A >= 2.2,
   nl, write_intervals(_A, _B).
[2.1999, 2.4001] [2.1999, 2.4001]
YES
```

shows two intervals with the same range. Notice that the order in which the intervals are constrained does not matter.

The query:

```
:- range(_A, [2, 3]),
   range(_B, [1, 5]),
   _A >= 2.2,
   _B <= 2.4,
   _A == _B,
   nl, write_intervals(_A, _B).
```

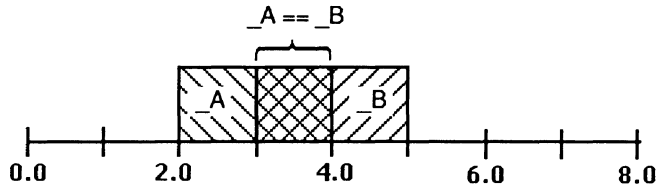


produces the same result.

Equating two intervals can be understood in either the numeric or the regional interpretation. The expression `_A == _B` can be read as either

- the real number denoted by the interval `_A` is the same as the real number denoted by the interval `_B`
- the region spanned by `_A` is the same as the region spanned by `_B`

Which declarative reading is most appropriate depends on what the intervals are intended to represent. No matter what the interpretation, the same narrowing effect takes place on the two intervals.



Note that the intervals `_A` and `_B` have now been narrowed to their intersection. The effect of `"=="` (equals) on intervals is analogous to that of `"="` (unification) in pure Prolog. In particular, equality, like unification, is an equivalence relation. The expression `not(_A == _B)` succeeds if `_A` and `_B` have no points in common and does no narrowing of either `_A` or `_B`. In effect, `not(_A == _B)` can be used to test that two intervals are disjoint.

Thus, the following:

```
?- range(_A, [2, 3]),
   range(_B, [4, 5]),
   not(_A == _B).    % succeeds
```



```
?- range(_A, [2, 4]),  
   range(_B, [3, 5]),  
   not(_A == _B).    % fails
```

### **Timetable Problem**

Equality on intervals is sufficient to solve the following nontrivial timetable problem:

Bob, Carol, Ted, and Alice are busy executives who need to have a meeting to discuss the filberflange shortage. Bob's secretary must schedule a meeting this week at a time when all four have no previous engagements. The meeting is expected to require at least 45 minutes, and must be held in a corporate conference room that has engagements of its own.

The schedules of the four executives and the room are coded by facts of the form

```
free_time(name, list-of-free-periods)
```

and each free period, where times are expressed in hours from Sunday midnight, is the structure of the form

```
period(start-time, end-time).
```



For example,

```
% People Schedule:

free('Bob', [
    period(9, 10), period(13.5, 14.25),      % Mon
    period(35.25, 35.75), period(39, 41),    % Tues
    period(56, 58.25),                       % Wed
    period(84, 84.5),                         % Thurs
    period(106, 107.25)]).                   % Fri

free('Carol', [
    period(10, 10.5), period(12.5, 13.25),   % Mon
    period(36.25, 37.75), period(39, 41),    % Tues
    period(57, 58.25),                       % Wed
    period(80, 85.5)]).                     % Thurs

free('Ted', [
    period(9, 11.5), period(12.5, 15),       % Mon
    period(33, 35), period(36, 42),          % Tues
    period(58, 59.25),                      % Wed
    period(80, 82), period(92, 93.5),        % Thurs
    period(105, 109.25)]).                  % Fri

free('Alice', [
    period(10.5, 15),                        % Mon
    period(32, 34), period(38, 40),          % Tues
    period(56, 59), period(62, 63.75),      % Wed
    period(81.75, 83.25),                   % Thurs
    period(104, 106.5)]).                  % Fri

% Room schedule:

free(room, [
    period(8, 12),                          % Mon
    period(38, 40),                         % Tues
    period(56, 57),                         % Wed
    period(84, 86),                         % Thurs
    period(104, 106)]).                    % Fri
```



To convert a list of periods to a list of intervals define:

```
period_interval([], []).
period_interval([period(_I, _J), _Ps..],
               [_Rij, _Rs..]) :-
    range(_Rij, [_I, _J]),
    period_interval(_Ps.., [_Rs..]).

free_time(_Person, _Ilist) :-
    free(_Person, _Plist),
    period_interval(_Plist, _Ilist).
```

To express the idea that one interval overlaps a member of a list of intervals, define the predicate `within` analogous to `member`:

```
within(_I, [_Int, ..]) :- _I == _Int.
within(_I, [_ , _Ints..]) :- within(_I, [_Ints..]).
```

To find the intervals (time-slots) during the week in which Bob, Carol, Ted and Alice can meet in the Room, it suffices to ask the question:

```
?- free_time('Bob', _Btimes),    % get free time lists
   free_time('Carol', _Ctimes),
   free_time('Ted', _Ttimes),
   free_time('Alice', _Atimes),
   free_time(room, _Rtimes),
   range(_Time, _),              % _Time is the period
   within(_Time, _Btimes),       %when they can all meet
   within(_Time, _Ctimes),
   within(_Time, _Ttimes),
   within(_Time, _Atimes),
   within(_Time, _Rtimes),
   0.75 <= delta(_Time),         % must be > 45 minutes
   nl, print_interval(_Time).
[39.0, 40.001]
YES.
```

which produces the hour 39-40. Notice that all the work in the question is done by `within` which in turn depends on the operation of `"=="` on intervals.



## Inequality

So far, we have used inequalities only to change the upper and lower bounds of an interval. But what does it mean to evaluate an inequality between two intervals?

"<=" and ">="

From the numeric perspective  $\_A \leq \_B$  simply means that the point represented by  $\_A$  is less than or equal the point represented by  $\_B$ . While this interpretation can always be applied, the question still needs to be answered from the regional point of view. There are three cases to consider:

- two disjoint intervals,  $\_A$ , and  $\_B$ , which do not affect each other, for example:

```
?- range(_A, [1, 2]),
   range(_B, [4, 6]),
   _A <= _B,
   nl,write_intervals(_A, _B).
[1.0,2.0001] [4.0,6.0001].
YES
```

- two intersecting intervals,  $\_A$  and  $\_B$ , where neither  $\_A$  nor  $\_B$  has narrowed, or where  $\_A$  and  $\_B$  narrow, for example:

```
?- range(_A, [2, 4]),
   range(_B, [3, 5]),
   _A <= _B,
   nl,write_intervals(_A, _B).
[2.0,4.0001] [3.0,5.0001].
YES
```

```
?- range(_A, [2, 4]),
   range(_B, [3, 5]),
   _A >= _B,
   nl,write_intervals(_A, _B).
[3.0,4.0001] [3.0,4.0001].
YES
```



- one interval, `_A`, contains the other, `_B`, which either lowers the upper bound of `_A` or raises the lower bound of `_A`, for example:

```
?- range(_A, [2, 5]),
   range(_B, [3, 4]),
   _A =< _B,
   nl,write_intervals(_A, _B).
[2.0,4.0001] [3.0,4.0001]
YES

?- range(_A, [2, 5]),
   range(_B, [3, 4]),
   _A >= _B,
   nl,write_intervals(_A, _B).
[3.0,5.0001] [3.0,4.0001]
YES
```

"<" and ">"

Since intervals are *closed*, that is they contain their end points, it is difficult to give the strict inequalities which define *open* intervals a clean numeric interpretation. The nearest approximation to an open interval that is expressible with floating point bounds is to treat the strict inequality

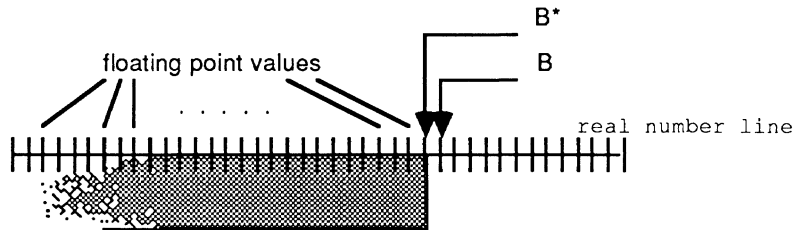
$$\_A < \_B$$

as a shorthand for

$$\_A \leq \_B^*$$

where `_B*` is `_B` with its lower bound moved to the next lower floating point number representable (this is machine dependent). For most users this is of little or no consequence. For those interested in formal real arithmetic, use of "<" and ">" should be avoided.





### Critical Path Scheduling Problem

In this section the critical path scheduling problem presented in the chapter "Functional Arithmetic" is formulated using interval equations and inequalities. Note that slack time maps into a delta on intervals, and the prerequisite time relation maps to a " $\leq$ " comparison.

```

schedule_int(_Deadline) :-
    range(_Start, [0, 0]), % make _Start an interval
    range(_Total, _),      % also _Total
    plan_int(_Start, _List, _Total,
              [10, 20, 30, 18, 8, 3, 4]),
    range(_Total, [_Early, _]),
    nl, write('Total time: ', _Early), nl,
    _Total <= _Deadline, % fails if infeasible
    nl, write('[Name, Start,      Finish,      ',
              'Duration  ]'),
    nl, print_sched_int(_List).

plan_int(_Start, _Activity_List, _Deadline,
         [_aTime, _bTime, _cTime, _dTime, _eTime,
          _fTime, _gTime]) :-
    _Activity_List = [[a, _aStart, _aFinish, _aTime],
                      [b, _bStart, _bFinish, _bTime],
                      [c, _cStart, _cFinish, _cTime],
                      [d, _dStart, _dFinish, _dTime],
                      [e, _eStart, _eFinish, _eTime],
                      [f, _fStart, _fFinish, _fTime],
                      [g, _gStart, _gFinish, _gTime],
                      ],
    activities(_Activity_List), % cont..

```



```
_Start =< _aStart,      _Start =< _bStart,
_Start =< _cStart,      _aFinish =< _dStart,
_bFinish =< _dStart,    _bFinish =< _eStart,
_cFinish =< _eStart,    _aFinish =< _fStart,
_dFinish =< _fStart,    _eFinish =< _gStart,
_fFinish =< _gStart,    _gFinish =< _Deadline.

activities([]) :- !.
activities([[_Name, _Start, _Finish, _Time], _List..])
:-
    symbol(_Name),
    real(_Start, _Finish),
    numeric(_Time),
    _Finish == _Start + _Time,
    activities(_List).

real().
real(_X, _Xs..) :- [range(_X, _), real(_Xs..)].

print_sched_int([]) :- nl.
print_sched_int([_X, _Xs..]) :-
    write(' '),
    row_int(_X), nl,
    print_sched_int(_Xs).

row_int([]) :- write(''), !.
row_int([_X, _Xs..]) :-
    [numeric(_X) & _X < 10] -> % create columns of output
    write(' '),
    prt(_X),
    symbol(_X) ->
        write(' ') ;
    write(' '),
    row_int(_Xs).

prt(_X) :- print_interval(_X), !.
prt(_X) :- write(_X).
```



**is**

As in functional arithmetic, the `is` operator is used to evaluate arithmetic expressions, but the expressions now contain intervals. The variable coercion rules related to the use of `is` with intervals are

- if the left side of an `is` expression is an uninstantiated variable, it will become instantiated
- if there is an interval on the right side of the expression, any variable on the left will be instantiated as an interval
- if the left side of the expression is instantiated, the expression becomes an equality test (like `"=="`)

The expression

```
_D is delta(_A)
```

computes a function on intervals (`delta`) and returns a floating point number `_D`. In this case `is` operates in the usual functional arithmetic way. However, the expression

```
_X is _A * _A
```

where `_A` is an interval, binds `_X` to an interval, not to a float.

**Point Intervals**

When a floating point number or an integer occurs in an expression containing an interval, that floating point number is coerced to a small interval containing the integer or floating point value. Since *point intervals* are usually intended to represent real numbers that may not be exactly representable as a floating point number, the region they span is of nonzero length.



Thus,

```
:- range(_A, [1.1, 1.1]), _D is delta(_A),  
   nl, write(_D).  
8.3447e_7  
YES
```

shows that `_A` is an interval with nonzero width. This feature makes it possible to evaluate expressions in real arithmetic correctly. For instance,

```
?- range(_A, [1.1, 1.1]), 1.21 == _A * _A.
```

is true in interval arithmetic, but not in functional arithmetic.

## Numerical Precision

Several kinds of numerical precision problems are treated elegantly in interval arithmetic. As seen previously, floating point round off errors (and their amplification after multiplication) are handled automatically by interval arithmetic. Another kind of precision problem that intervals handle naturally is generated by imprecise input data.

In most real applications, input data is accurate to within a few percent. When several such numbers are used as input data for a complex arithmetic formula, it can be quite difficult to estimate the accuracy of the results. One way of doing this is to perform a sensitivity analysis, varying every input parameter separately and observing the effect produced. However, this method not only neglects synergistic effects, but it is very time consuming.

Interval arithmetic provides a natural solution to this kind of problem: simply represent the input data as an interval. We can convert the exact values of a point to an interval that contains the error margin with the short program:

```
uncertainty(_Value, _PerCent, _Interval) :-  
    _Delta is _Value * _PerCent,  
    _Upper is _Value + _Delta,  
    _Lower is _Value - _Delta,  
    range(_Interval, [_Lower, _Upper]).
```



As a simple example, consider the equation

$$\_X * \_X + 3 * \_Y == \_Z$$

Suppose  $\_Z$  is the unknown quantity, while  $\_X$  is accurate to within 30% and  $\_Y$  is accurate to within 12%. If our first experiment produces values of  $\_X = 0.25$  and  $\_Y = 5.56$ , then our interval solution for  $\_Z$  is produced by

```
:- range(_Zint, _),          % _Zint is unknown
   uncertainty(0.25, 0.3, _Xint),
   uncertainty(5.56, 0.12, _Yint),
   _Xint * _Xint + 3 * _Yint == _Zint,
   _Delta is delta(_Zint) / 2,
   _Mid is midpoint(_Zint),
   _Percent is _Delta / _Mid,
   nl, write(_Percent), nl,
   print_interval(_Zint).
```

which shows that the worst case error in  $\_Z$  is slightly more than 12%. This reflects the fact that the main contribution to the error in  $\_Z$  came from  $\_Y$ . Now, suppose the measurement for  $\_X$  is 2.25. The resulting error in  $\_Z$ , obtained from a similar question, shows an error margin for  $\_Z$  of 22%. The increase in the error for  $\_Z$  reflects the greater contribution of the error in  $\_X$ , now that  $\_X$  is greater than 1.

For such a simple example, it is easy to estimate by hand the effect of an error in one variable on the result, but in general, the problem requires a great deal of additional calculation. With intervals, however, the effect of error propagation is built into the quantities that are being computed.

## Using Relational Arithmetic

Standard functional arithmetic in Prolog requires specific instantiation patterns in arithmetic expressions which obliges the programmer to think procedurally rather than declaratively. If a term in an arithmetic expression is not ground, most Prolog systems will either produce an error or fail.



The idea of an unknown arithmetic quantity can be expressed by an indefinite interval. The evaluation of any arithmetic expression containing such an interval attempts to narrow its upper and lower bounds in such a way that the expression is true.

### Temperature Conversion Problem

Consider a program that converts temperature measurements from degrees Celsius to degrees Fahrenheit:

```
cel_far(_C, _F) :- _F is _C * (9 / 5) + 32.
```

In functional Prolog arithmetic this procedure can succeed only if `_C` is bound to an integer or a float. Thus,

```
?- cel_far(_C, 98.6).
```

fails because `_C` is an unbound variable in the body of an arithmetic expression. If, however, the query is changed so that `_C` is indefinite, it imposes an arithmetic constraint on `_C` that can only be satisfied by shrinking `_C`.

```
:- range(_C, _),
   cel_far(_C, 98.6),
   nl, write_intervals(_C).
[36.999, 37.001]
YES
```

`cel_far` can also be used to compute `_F` from `_C`, as in:

```
:- range(_F, _),
   cel_far(37, _F),
   nl, write_intervals(_F).
[98.599, 98.601]
YES
```

In other words, the unknowns in `cel_far` are interchangeable, and it is a genuine relation.

The `cel_far` relation can operate not only on point intervals, but also on ranges of values for `_C` or `_F`. Thus, if the temperature



range in Florida in July is between 86 and 104 degrees Fahrenheit, this converts to the range [30, 40] in degrees Celsius in the query:

```
:- range(_C, _),  
   range(_F, [86, 104]),  
   cel_far(_C, _F),  
   nl, write_intervals(_C).  
[29.999, 40.001]  
YES
```

On the other hand, the temperature range in Ontario in January is between -34 and -10 degrees Celsius, which converts to the approximate range [-40, 14] in degrees Fahrenheit in the query:

```
:- range(_C, [-34, -10]),  
   range(_F, _),  
   cel_far(_C, _F),  
   nl, write_intervals(_C).  
[-34.001, 10.0]
```

The relational nature of interval arithmetic has other consequences. If both intervals `_C` and `_F` are declared to have definite ranges, the constraint imposed by `cel_far` narrows both variables simultaneously. For example, if the initial ranges for `_C` and `_F` only contain subranges for which the conversion equation is true, they are constrained to those subranges after the equation is evaluated, as in the following:

```
:- range(_C, [-34, 0]),  
   range(_F, [-40, 14]),  
   cel_far(_C, _F),  
   nl, write_intervals(_F, _C).  
[-29.201, 14.001] [-34.001, -9.999]  
YES
```



Even if the constraints are imposed after the call to `cel_far`, the narrowing of one interval immediately propagates to the other. For example:

```
?- range(_C, _),
   range(_F, _),
   cel_far(_C, _F),
   range(_C, [-34, 0]),
   nl, write_intervals(_F, _C),
   range(_F, [-40, 14]),
   nl, write_intervals(_F, _C).
```

If the initial ranges do not contain any subranges for which `cel_far` is true, the goal fails. Thus the following question fails:

```
?- range(_C, [-34, 0]),
   range(_F, [33, 64]),
   cel_far(_C, _F).
```

It is perhaps a bit misleading to use the `is` predicate when expressing arithmetic relations among intervals. The relation `cel_far` could be written differently by *equating* the left hand side with the right side, as in

```
cel_far(_C, _F) :- _F == _C * (9 / 5) + 32.
```

Note that `"=="` must compare the interval values of arithmetic terms, whereas `is` coerces its left hand side to be an interval if it is a variable. Thus, to compute the `cel_far` procedure defined with equality, `_F` must be declared as an interval before `cel_far` is evaluated. For example,

```
?- range(_F, _),
   range(_C, [32, 37]),
   cel_far(_C, _F),
   nl, write_intervals(_F).
```

If `_F` is not declared an interval, the execution of `"=="` attempts to evaluate a logic variable, which fails, as in ordinary functional arithmetic. The requirement that both terms in an arithmetic operation evaluate to intervals also holds for inequalities.



## Gas Law Problem

One of the advantages of relational arithmetic is the greater simplicity of programs. One interval equation can replace several expressions in functional arithmetic. For example, expressing the simple equation

$$P * V = N * 1.38e-23 * T$$

in functional Prolog arithmetic requires four rules (assuming `is` fails on nonground arithmetic expressions), one for each possible unknown:

```
gas_law0(_P, _V, _N, _T) :-
    _P is _N * 1.38e-23 * _T / _V.
gas_law0(_P, _V, _N, _T) :-
    _V is _N * 1.38e-23 * _T / _P.
gas_law0(_P, _V, _N, _T) :-
    _T is _P * _V / (1.38e-23 * _N).
gas_law0(_P, _V, _N, _T) :-
    _N is _P * _V / (1.38e-23 * _T).
```

With interval arithmetic, however, all that is required is

```
gas_law(_P, _V, _N, _T) :-
    _P * _V == _N * 1.38e-23 * _T.
```

and assurance that `_P`, `_V`, `_N` and `_T` are numerics when `gas_law` is called. Thus,

```
:- range(_P, _), gas_law(_P, 1, 2, 3),
    nl, write_intervals(_P).
[8.2799e-23, 8.2800e-23]
YES
```

finds a value for `_P` where `_V` and `_T` and `_N`, are given values, whereas

```
:- range(_V, _), gas_law(4, _V, 5, 6),
    nl, write_intervals(_V).
[1.0349e-22, 1.0350e-22]
YES
```



finds a value for `_V` from `_P`, `_T`, and `_N`. If the ranges for `_N`, `_P`, `_V` or `_T` are not point intervals, then they may narrow after the rule for `gas_law` is called. The question

```
:- range(_V, [1, 4]),
   range(_N, [1.0e+22, 5.0e+23]),
   gas_law(2, _V, _N, 4),
   nl, write_intervals(_N).
[3.6231e+22, 1.4493e+23]
YES
```

Similarly, if any one of the constrained intervals subsequently becomes more constrained, the other dependant variables will narrow accordingly. Thus the following

```
:- range(_V, [1, 4]),
   range(_N, [1.0e+22, 5.0e+23]),
   gas_law(2, _V, _N, 4),
   nl, write_intervals(_V, _N),
   _V >= 2, _V <= 2.5,
   nl, write_intervals(_V, _N).
[1.0, 4.0001][3.6231e+22, 1.4493e+23]
[2.0, 2.5001][7.2463e+22, 9.0580e+22]
YES

:- range(_V, [1, 4]),
   range(_N, [1.0e+22, 5.0e+23]),
   gas_law(2, _V, _N, 4),
   nl, write_intervals(_V, _N),
   _V >= 2, _V <= 2.5,
   nl, write_intervals(_V, _N),
   _N >= 8.1e+22, _N <= 8.2e+22,
   nl, write_intervals(_V, _N).
[1.0, 4.0001][3.6231e+22, 1.4493e+23]
[2.0, 2.5001][7.2463e+22, 9.0580e+22]
[2.2355, 2.2633][8.0999e+22, 8.2001e+22]
YES
```

If the ranges for `_P`, `_V`, `_N` and `_T` are ever constrained in such a way that `gas_law` does not hold, the query fails.



In this example:

```
:- range(_V, [1, 4]),
   range(_N, [1.0e+22, 5.0e+23]),
   gas_law(2, _V, _N, 4),
   nl, write_intervals(_V, _N),
   _V >= 2, _V <= 2.5,
   nl, write_intervals(_V, _N),
   _N >= 9e+29,
   nl, write_intervals(_V, _N).
```

the constraint on `_N` cannot be met and the entire question fails.

## Many to Many Relations

In contrast to functional arithmetic, relational arithmetic can express a many to many relationship.

### Square Problem

The evaluation of an arithmetic expression containing intervals constrains the intervals in such a way that the expression is true. This means that arithmetic relations which have many to one values constrain intervals to be in the range that contains all the values for which the relation is true. For example if the rule

```
square(_X, _Y) :- _Y is _X * _X.
```

is queried with either of the following,

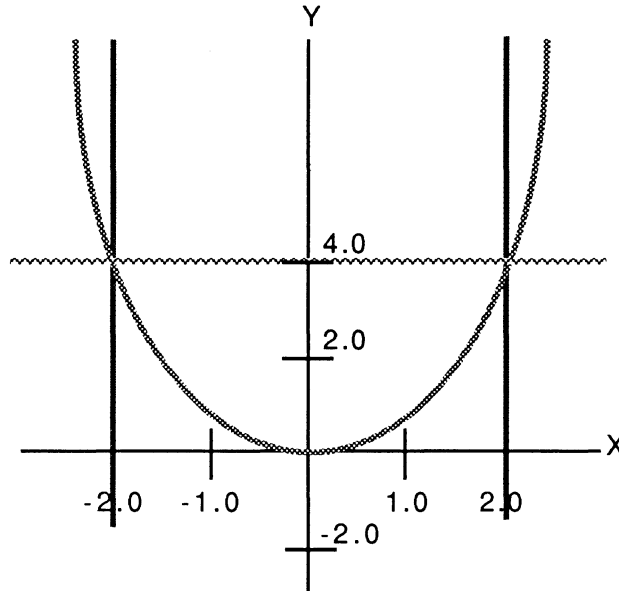
```
?- square(2, _Y).
?- square(-2, _Y).
```

`_Y` is bound to 4. Thus with the question

```
:- range(_X, _), square(_X, 4), print_interval(_X).
[-2.0001, 2.0001]
```

all solutions for the square root of 4 are obtained, and the range for `_X` becomes the interval containing *both* square roots.





```
range(_X, _), square(_X, 4)
```

This means that all solutions to `square(_X, 4)` lie between -2 and 2. If `_X` is further constrained to be either a positive number or a negative number, as in

```
?- range(_X, _),
   square(_X, 4),
   (_X >= 0 ; _X <= 0),
   print_interval(_X).
```

the range of `_X` narrows to the point interval `[2.0, 2.0001]` and backtracking for an alternative solution narrows `_X` to the point `[-2.0001, -2.000]`. Note that an attempt to narrow the interval `_X` to between 0 and 1 fails

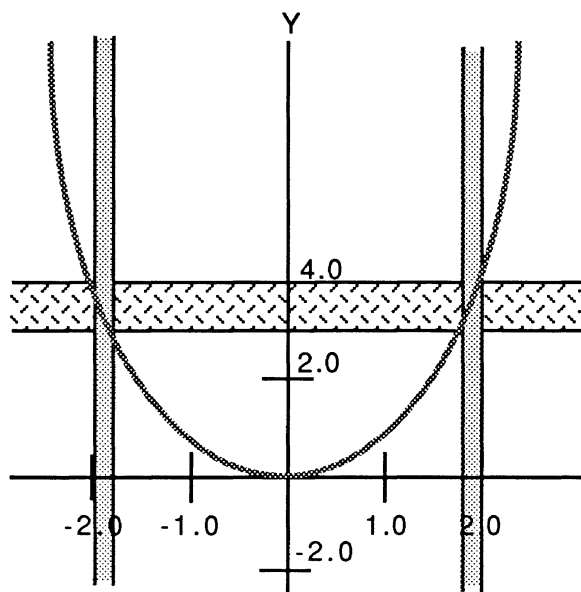
```
?- range(_X, _),
   square(_X, 4),
   _X >= 0,
   _X <= 1.
```



Now if we use this square relation with a `_Y` value that lies in the range `[3, 4]` rather than on the point `[4, 4]`,

```
?- range(_X, _),
   range(_Y, [3, 4]),
   square(_X, _Y),
   print_interval(_X).
```

we see that the range of solutions for `_X` is still between `-2` and `2`.



```
range(_X, _), range(_Y, [3, 4]), square(_X, _Y)
```

However, the solutions that lie in this interval are not point intervals, but rather, subranges of `[-2, 2]`. If `_X` is again constrained to be alternately positive and negative

```
?- range(_X, _),
   range(_Y, [3, 4]),
   square(_X, _Y),
   (_X >= 0 ; _X <= 0),
   print_interval(_X).
```



note that the range of `_X` has narrowed not to a point but to the intervals `[1.73, 2]` and `[-2, -1.73]` which are the positive and negative interval roots of the interval `[3, 4]`.

### Magnetism Problem

A more complex example comes from the theory of ferromagnetism. The magnetization,  $M$ , of a ferromagnet composed of  $N$  atoms with spin  $1/2$  and magnetic moment  $\mu$  can be described by the equation

$$M = N * \mu * \tanh(\mu * B / k * T)$$

where  $k$  is Boltzmann's constant,  $T$  is the temperature and in the mean field approximation,  $B = \lambda * M$  for some unknown  $\lambda$ . From the definitions

$$m = M / (N * \mu)$$
$$t = k * T / (N * (\mu ** 2) * \lambda)$$

the reduced equation

$$m = \tanh(m / t)$$

results. The Prolog program for this is simply

```
magnetization(_m, _t) :-
    _m > 0,
    _t > 0,
    tanh(_m / _t, _m).
```

A Prolog definition for `tanh` is

```
tanh(_X, _Y) :-
    _Y == (exp(_X) - exp(-_X)) / (exp(_X) + exp(-_X)).
```

but this is somewhat inefficient, both because it performs several transcendental evaluations, and because there are unnecessary occurrences of `_X` in the expression. In addition, the hyperbolic arctangent is known to always be in ranges between `[-1, 1]`, so a redundant condition, `range(_Y, [-1, 1])`, can be added to help in cases when the program is being used backwards.



A better version of `tanh` is :

```
tanh(_X, _Y) :-
    range(_Y, [-1, 1]),
    range(_W, _),
    _W == exp(-2 * _X),
    _Y == (1 - _W) / (1 + _W).
```

An example query is:

```
:- range(_m, [0.001, 1]),
   range(_t, [0.5, 0.55]),
   magnetization(_m, _t),
   nl, print_interval(_m).
[0.93552, 0.95751].
YES
```

## Equation Solving

Although some equations can be solved using only narrowing as in the above examples, most require the use of additional techniques. The inference mechanism for intervals is not an equation solver, and it is not sufficient on its own to solve even simple sets of linear equations unless they are in "triangular form".

For example the equations

$$\begin{aligned}\_A - \_B &= 1 \\ \_A + \_B &= 3\end{aligned}$$

which have the unique solutions  $\_A = 2$  and  $\_B = 1$  are not solved in interval arithmetic if the question is simply:

```
?- range(_A, _),
   range(_B, _),
   _A - _B = 1,
   _A + _B = 3,
   nl, write_intervals(_A, _B).
```

With this question alone, the ranges of  $\_A$  and  $\_B$  are correctly constrained by the equations but neither equation *by itself* is



sufficient to narrow the intervals further. The system is not designed to detect the fact that the two equations can be reduced to:

$$\begin{aligned}\_A &= \_B + 1 \\ \_A + \_B &= 3\end{aligned}$$

from which a term substitution yields

$$\_B + 1 + \_B = 3$$

which can be solved for  $\_B$  (the Gaussian elimination technique). This type of technique is applied automatically in systems that specialize in linear systems, but it is not effective for general systems of equations and inequalities involving nonlinear or nonfunctional relations.

The following section shows the development of the equation solving predicate `solve`, which can be used to systematically solve such general nonlinear sets of equations and inequalities. In situations where the problems are of a form where conventional solution techniques can be used, they will usually be more efficient than this totally general purpose technique.

### **split**

A general way to find the roots of an equation is to systematically split the interval containing them with the predicate `split1`.

```
split1(_Range) :-  
    _Mid is midpoint(_Range),  
    (_Range =< Mid ; _Range >= Mid ).
```

The interval  $\_Range$  is narrowed to the bottom half by  $\_Range =< \_Mid$ , and upon backtracking,  $\_Range$  is narrowed to the upper half by the alternative  $\_Range >= \_Mid$ . With this, both solutions for `square` can be obtained by backtracking,



as in the query:

```
?- range(_X, _),
   range(_Y, [3, 4]),
   square(_X, _Y),
   split1(_X),
   nl, print_interval(_X).
```

In most cases, an equation with multiple solutions, such as the `square` program, only constrains an indefinite interval to be in the range where all the solutions lie. To get each solution separately, an attempt must be made to subdivide the interval result into subranges with a predicate like `split1`.

Of the two major limitations of `split1`, the most noticeable is that it partitions an interval into only two subranges. `split1` would be more useful if it could subdivide an interval into an arbitrary number of subranges using a recursive procedure, as in:

```
split2(0, _Range) :- !.

split2(_Depth, _Range) :-
    _D1 is _Depth - 1,      % decrement depth
    _Mid is midpoint(_Range),
    (_Range <= _Mid ;        % narrow UPPER bound or
     _Range >= _Mid),        % narrow LOWER bound
    split2(_D1, _Range).    % recurse
```

The `split2` predicate has an argument to control the depth of recursion. At each level, `_Range` is split one more time, first to the lower half of its range, or if that fails, to the upper half.

The second limitation of `split1` (inherited by `split2`) is that it subdivides the interval into alternative subintervals to the left or to the right of the midpoint of `_Range`. This subdivision is not an optimal one, since there may be more potential solutions in the one half of `_Range` than in the other. Although the alternative subintervals created by the use of `midpoint` contain the same number of real numbers, they usually do not contain the same number of floating point numbers.



**median**

Since floats are represented as binary numbers, there are as many floating point numbers between 512 and 1024 as there are between 1024 and 2048. One quarter of all floats lie between 0 and 1. The predicate `median` can be used to divide the intervals into two subintervals containing the same number of floats.

The median of an interval not containing 0 is approximately the square root of the product of its end points. Thus,

```
?- range(_I, [512, 2048]),
   _M is median(_I).
```

binds `_M` to 1024.0 whereas

```
?- range(_I, [512, 2048]),
   _M is midpoint(_I).
```

binds `_M` to 1280.0. If an interval `_I` contains the point 0 (that is has a negative lower bound and a positive upper bound), the median of `_I` is defined to be 0. Because intervals can be more precise than floats, `median` fails if the interval is too small to contain a floating point value.

A predicate like `split`, if defined using `median`, always first subdivides an interval into its negative and positive alternatives. To make `split` more efficient, substitute the `median` function for `midpoint` as follows:

```
split(0, _Range) :- !.
split(_Depth, _Range) :-
    _D1 is _Depth - 1,
    _Mid is median(_Range),
    (_Range=< _Mid ; _Range >= _Mid),
    split(_D1, _Range).
```

There are several limitations to `split` that have not yet been discussed. If, for example, the median of the interval being split is a solution, we might want to stop the splitting process so as not to subdivide the interval. Heuristic refinements such as this are built into the provided `solve` predicate.



**solve**

The predicate `solve` should be thought of as a control construct that systematically attempts to narrow intervals. Backtracking through `solve` produces alternative narrowings for intervals that have been constrained by a network of arithmetic constraints.

The query:

```
?- range(_A, _),
   range(_B, _),
   _A - _B = 1,
   _A + _B = 3,
   solve(_A),
   nl, write_intervals(_A, _B).
```

produces ranges for `_A` and `_B`, and the exact solutions are contained within those ranges. Note that narrowing `_A` automatically narrows `_B` as a result of the constraints imposed by the equations.

`solve` is not an efficient solution technique for this problem, which can be more easily solved in other, less general ways. However, consider the problem of finding roots for the polynomial equation

$$35 * X^{256} - 14 * X^{17} + X = 0$$

in the interval  $[0, 1]$ . The appropriate query is then simply

```
?- range(_X, [0, 1]),
   0 == 35 * _X ** 256 - 14 * _X ** 17 + _X,
   solve(_X),
   nl, print_interval(_X).
[0.0,0.0]
[0.84794, 0.84795]
[0.99584, 0.99585]
YES
```

This problem is considerably harder to solve by other methods.



Note that when using `solve`, failure implies that no solutions exist in the initial ranges, whereas success only implies that there may be solutions in the final ranges. The `solve` predicate is used to narrow intervals that have been constrained by sets of simultaneous equations and inequalities, both linear and nonlinear. Simultaneous equations may be arbitrarily complex, because `solve` is an algorithm (written in Prolog) for searching subintervals that satisfy the constraints imposed by the equations.

### **accumulate**

Certain common problems in mathematics normally require the use of recursion in Prolog. This can be a nuisance when using intervals, since it requires that each level of the recursion set up its own copy of the system of constraint equations. Recursion can be avoided in many cases by using the built-in predicate `accumulate`, which enables the transfer of interval values from one alternative branch of a computation to another. Such an operation is technically a side effect, producing changes not undone by backtracking, and therefore must be used with care. The predicate

```
accumulate(_Xin, _Exp)
```

where `_Xin` is an interval and `_Exp` is an arithmetic expression, evaluates `_Exp` and adds the result to `_Xin`. Since the value of `_Xin` is changed by the operation, not merely narrowed, it should not be constrained by any equations. There is no flow of information into any intervals used in `_Exp`.

The use of `accumulate` is best clarified by examination of several typical examples. One use of `accumulate` is to calculate the sum of a set of numeric values given by a generator, as in

```
total(_Var, _Number_generator, _Total) :-  
    var(_Total),                % convert _Var to an interval  
    range(_Total, [0, 0]),      % with initial value [0, 0]  
    foreach(_Number_generator do  
        accumulate(_Total, _Var)).
```



An example which sums a list:

```
:- total(_X, member(_X, [1, 1 / 2, 1 / 3, 1 / 4]), _Sum),
   nl, print_interval(_Sum).
[2.0833, 2.0834]
YES
```

The sum is based on the existing interval ranges at the time `total` is called. Therefore, it is not automatically updated by subsequent narrowing.

Another use for `accumulate` is as an efficient generator for subdividing intervals:

```
scan(_X, _X0, _Dx) :-
    interval(_X, _X0, _Dx),
    range(_Dd, _),
    _Dd is delta(_Dx),
    repeat,
    _X == _X0 + _Dx ->                % _X is [last,curr]
        accumulate(_X0, _Dd) ;        % update last
    failexit(scan).                   % exit when outside _X
```

Here `_X` is an interval that may or may not be constrained, `_X0` is an unconstrained "state" variable, and `_Dx` is an interval with a lower bound of 0. When `accumulate` is executed as a generator, `_X` is successively narrowed to nonoverlapping subintervals of size `delta(_Dx)`. The generator terminates as soon as `_X` fails to narrow, for example, when it reaches the end of its range or when it becomes inconsistent with its constraints.

A predicate such as `scan` is commonly used in graph plotting routines. Consider:

```
plot(_Name, _Y, _X, _Ddx) :-
    make_graph(_Name, [GXlo, GXhi], [GYlo, GYhi]),
    normalize(_X, _Nx, [GXlo, GXhi], 1),
    normalize(_Y, _Ny, [GYlo, GYhi], -1),
    range(_Dx, [0, _Ddx]),            % make step interval
    range(_Nx0, [Xlo, Xlo]),          % make initial interval
    foreach(scan(_Nx, _Nx0, _Dx) do
        graph(_Name, _Nx, _Ny)).
```

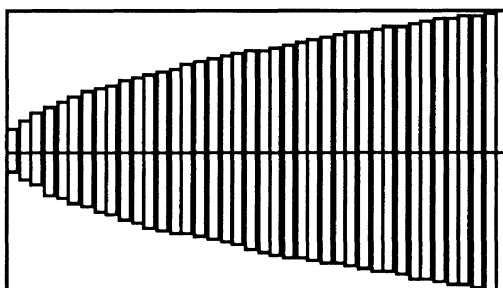


```
graph(_W, _X, _Y) :-
    dograf(_W,
        [fillpat(hollow),rectabs(_X, _Y, _Xl, _Yl)]).
```

where `_Name` is the title of the graph window, `_X` and `_Y` are the independent and dependent variables respectively, and `_Ddx` is the (floating point) step size. The predicate `make_graph` (not listed here) opens the graphics window, properly positioned and sized, and returns ranges scaled in terms of graphics coordinates. The intervals `_X` and `_Y` are assumed to be constrained before calling `plot`. Calls to `normalize` create interval variables that are scaled to fit the window, and couples them to `_X` and `_Y`.

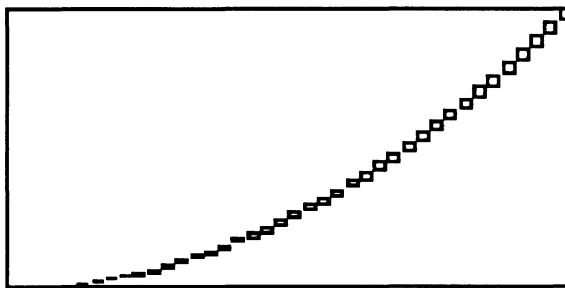
```
normalize(_X, _Nx, [_Lb, _Ub], _Sgn) :-
    range(_Nx, [_Lb, _Ub]),
    range(_X, [_Lbx, _]),
    _Mx is _Sgn * delta(_Nx) / delta(_X),
    _Sgn > 0 ->
        _Ox is _Lb - _Mx * _Lbx ;
        _Ox is _Ub - _Mx * _Lbx,
    _Nx == _Ox + _Mx * _X.
```

`scan` is used to generate the sequence of subintervals, using the narrowed `_Nx` and `_Ny` to plot the result. The following are pictures of relations plotted using `plot`:

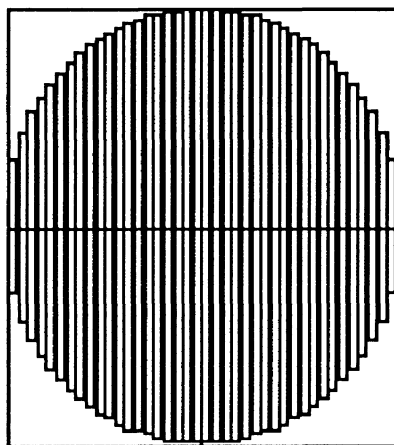


$$X == Y * Y$$





$$Y == X * X$$



$$X * X + Y * Y = 1$$

Another use of both `accumulate` and `scan` is the computation of generalized definite integrals, or quadratures, as intervals. For example, in the predicate

```
def_integral(_Y, _Y0, _X, _Dx) :-
    interval(_Y, _Y0, _X, _Dx),    % check parameters
    range(_X, [_Lb, _]),            % get lower bound
    range(_X0, _),                  % set up for scan
    _X0 == _Lb,
    foreach(scan(_X, _X0, _Dx) do
        accumulate(_Y0, _Y * delta(_X))).
```



\_X and \_Y denote the independent and dependent variables respectively, \_Dx the step size, and \_Y0 the computed value of the integral. The variable \_Y0 is an interval which may be assumed to have an initial value of 0, and a final value that is an estimate of the integral for \_Y with respect to \_X. As in the plotting example, explicit knowledge of the relation between \_X and \_Y is not required by `def_integral`.

This algorithm corresponds to the simplest possible conventional algorithm, and is not especially efficient. However, unlike its conventional counterpart, it computes strict conservative estimates of both lower and upper bounds for the integral. Unlike more sophisticated techniques it does not place restrictions on the class of relations it can deal with; the relation between \_X and \_Y need not have a classical integral at all and need not even be a function. More efficient techniques can be devised for more restricted problems in which this generality is not needed.

```
/* example of def_integral predicate */

integrate_normal(_Interval, _Step) :-
    range(_X, _Interval), range(_Dx, [0, _Step]),
    range(_Y, _), range(_F, [0, 0]),
    _N is sqrt(2 * pi),
    _Y == exp(-0.5 * _X ** 2) / _N,
    def_integral(_Y, _F, _X, _Dx),
    nl, print_interval(_F).

% between mean and standard deviation
:- integrate_normal([0, 1], 0.1).
[0.33329, 0.349]
YES

% between mean and 3 sigma, that is almost 0.5
:- integrate_normal([0, 3], 0.1).
[0.47891, 0.51837]
YES

% decreasing step size increases accuracy up to a point
:- integrate_normal([0, 3], 0.01).
[0.49667, 0.50063]
YES
```



## Summary

As a summary of the relational arithmetic system, the following table relates analogous features and properties of interval arithmetic and pure Prolog.

Features

Interval Arithmetic	Pure Prolog
relational narrowing interval integer, float expressions ==, <=, >= , (and) ; (or) range,delta,median,midpoint	relational instantiation variable ground term goals = , (and) ; (or) meta-primitives (var..)

Properties

Interval Arithmetic	Pure Prolog
idempotent inclusion isotone quasi-persistent	idempotent monotone persistent







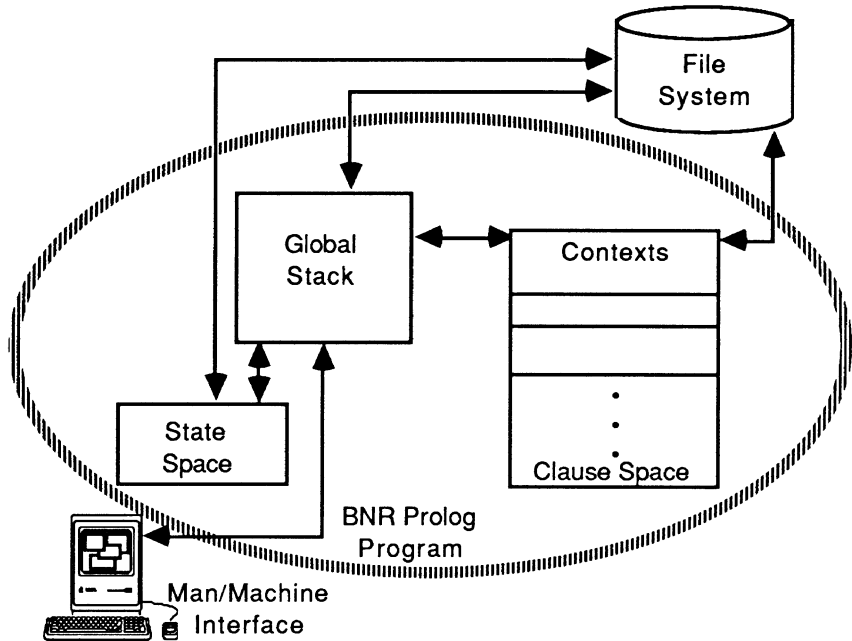
# Part IV Programming with Side Effects



Side effects are actions that are not undone on backtracking and therefore violate the logic programming paradigm. Goals which produce side effects often do not commute. Any predicate defined using side effects must be interpreted procedurally, and programs dependent on side effects can be hard to read and debug. Nonetheless, side effects provide practical solutions to many programming and efficiency issues. If used with care and properly contained, many of the undesirable characteristics of side effects can be minimized.

One of the more obvious uses of side effects is input and output of text data. It is hard to imagine a practical Prolog program which does not interact with the user and/or the file system to some extent. Other parts of the Prolog system subject to side effects include the knowledge base (programs which "learn" during their execution need to modify their program semantics) and the state space (an internal database provided to separate state information from program information). As the following diagram indicates, data flows between these "storage" mechanisms and the current execution state (state of variable bindings) by means of side effects.





The next chapters expand on support for input/output, accessing/modifying the knowledge base and the state space,s and support for the Macintosh user interface.

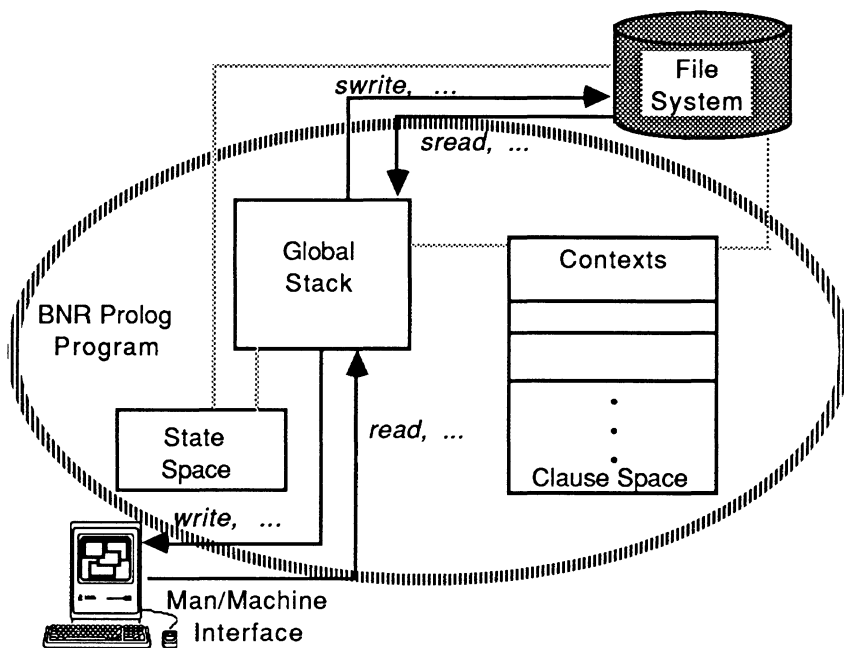






# Chapter 13

## Text Input and Output



One of the most frequent uses of side effects is the transfer of text data into and out of a program. In Prolog, input and output (I/O) predicates succeed by introducing side effects in their targets. If output is generated, it is not erased on backtracking. Similarly, if input is requested on a failing branch of a query, the position in the data at which the last read occurred is not restored on backtracking to allow a reread. Programs that are primarily concerned with processing external data are procedural in style, in contrast with the declarative style of logic programming programs.



## Streams

The targets for input and output predicates are streams and symbols. With a few exceptions, streams and symbols are used identically for input and output. Streams are essentially paths to external objects that are logically equivalent to a sequence of Prolog terms. Examples of such objects are text files, text windows and pipes.

A text window can be considered an open text file. Opening a window opens the corresponding file and provides a view of the file contents, automatically making the name of the text window the same as that of the file to which it corresponds. Input and output use the contents of the window in preference to the file, since the window is the most recent version of the file.

A pipe is a stream implemented as a FIFO (first in, first out) buffer in memory. Reading from a pipe succeeds only if valid text was previously written to the pipe. Pipes are always open for input and output.

### **open**

Before any stream input or output can occur, a stream must be opened. This requires the name of the file and the mode in which the file is to be opened. If `open` is successful, a unique integer identifier is assigned to the stream for use with other I/O predicates. The various permissible modes are shown in the following examples:

```
?- open(_stream_id, _file_name, read_only, 0).
   % an input-only stream for a file

?- open(_stream_id, _file_name, read_write, 0).
   % a read and write stream for a file

?- open(_stream_id, _window_name, read_window, 0).
   % an input-only stream for a window

?- open(_stream_id, _window_name, read_write_window, 0).
   % a read and write stream for a window
```



```
?- open(_stream_id, _any_name, read_write_pipe, 0).  
    % a read and write stream which is a pipe
```

(The use of 0 as the result code argument is to force failure if the open operation fails. Handling I/O errors is discussed later.)

At any one time, in addition to the two default I/O streams, there may be at most ten streams open. Only one stream may be open at a time for any given file, thus preserving a consistent single version of a window or a file.

### **close**

To avoid running out of streams, they should be closed when no longer needed, using the `close` predicate:

```
close(_stream_id, _error_code).
```

Closing the stream for a window does not preserve any changes in the associated file. The changes to a window are saved only by using one of the save commands in the **File** menu, or by calling the `savetext` predicate.

### **stream**

To verify that a particular stream is open, or to generate the list of all open streams, the `stream` predicate may be used:

```
stream(_stream_id, _full_name, _mode).
```

## **Stream Pointers**

The stream pointer represents the position in the file at which the next read or write is to occur. The first character in a stream corresponds to position 0, the second character position 1 and so on. When a stream is opened, its stream pointer is at position 0. Reading from a stream progresses from beginning to end, starting at the current stream pointer. Once the read is satisfied, the stream pointer is relocated one position beyond the last character read. Writing also progresses from the beginning



of the stream onwards. When the write is complete the stream pointer is positioned after the last character written.

A special value for the stream pointer is the symbol `end_of_file`.

### **at, seek**

`at` and `seek` can be used to support random access within streams. The predicate `at(_Stream_ID, _Ptr)` obtains the current pointer position. `seek(_Stream_ID, _Ptr)` may be used to position the stream pointer.

Use of `at` and `seek` are demonstrated in the following examples. `peek_char` is a predicate which 'looks ahead' at the next character in the stream. Such a predicate is often useful in parsing systems.

```
% peeks at next character in stream
peek_char(_Stream, _CH) :-
    at(_Stream, _N),
    get_char(_CH),
    seek(_Stream, _N).
```

`logical_read` is a read predicate which undoes the effect of the read on backtracking.

```
% undoes read on backtracking
logical_read(_Stream, _X) :-
    at(_Stream, _N),
    sread(_Stream, _X) ;
    [seek(_Stream, _N), fail].
```

Recall that opening any stream sets the pointer at the start of the stream. Data can be appended to a read/write stream if the stream pointer is first moved to the end as in the following sequence of goals:

```
% move pointer to end so new data follows old user code
open(_stream_id, _filename, read_write, 0),
seek(_stream_id, end_of_file),
append_data(_stream_id).
```



### **set\_end\_of\_file**

There are cases when new data should overwrite the old. This can be done by using the predicate `set_end_of_file` to position the `end_of_file` at the current stream pointer, after writing out the changed text. A segment of code that implements this is

```
write_revision(_stream_id),      % user code
set_end_of_file(_stream_id),
close(_stream_id, 0)
```

Unlike the stream pointer for a file, which moves through the file with subsequent reads or writes, the stream pointer for a pipe is always at the beginning, position 0. Advancing the stream pointer discards data up to the new pointer position. Setting the end of file marker for a pipe discards any remaining data in the pipe.

Data is always read from the beginning and always written at the end of a pipe. If there is not enough data in the pipe for a read to succeed, the read fails and whatever data exists is preserved. Once more data has been written to the pipe, a subsequent read succeeds. After a read, only the data representing the object read is discarded.

Closing a pipe also discards any remaining data.

### **Default I/O Streams**

Two default streams, one for input and one for output, are automatically opened when BNR Prolog is launched.

The default input stream (stream 0) is a pipe used to acquire interactive input from the user through the currently active window. A pipe is used so that syntax errors resulting from an incomplete query or assertion can be handled.

The default output stream (stream 1) is the `Console`, a read/write text window. A distinct feature of this window is that all text is output to the end of the window. This aids in keeping a log of interactive queries and responses.



The format of some predicates is such that if a stream identifier is not specified the appropriate default I/O stream is used.

## Detecting I/O Errors

Four of the I/O predicates, `close`, `get_term`, `open` and `put_term`, return integer result codes for the corresponding operation. Assuming the arguments are of the correct number and type, these predicates always succeed so the result codes can be returned as the last argument. Each code has a unique and fixed interpretation. (See the *BNR Prolog Reference Manual* for further information.)

A result of 0 means the operation was successful. Codes less than 0 indicate the operation is invalid, either due to some operating system detected error (-43 translates as an invalid file name) or restrictions within the Prolog system (-200 translates as maximum number of streams exceeded). Codes greater than 0 are associated with syntax errors (12 translates as unmatched right parenthesis).

If the goal is intended to fail when the operation fails, the call to one of these predicates should have the last argument bound to the value 0.

## Read and Write Predicates

Predicates which read terms assume that the input stream is structured as a sequence of one or more Prolog terms. Each term must be followed by a period "." and either a space or a newline character. For any other target structure, for example English sentences, the built-in predicates for reading characters must be used in combination with user defined predicates to inspect the structure of the input.

### `get_term`

The basic input predicate, which reads just one term, is:

```
?- get_term(_stream_id, _term, _error).
```



`get_term` is the only stream input predicate that returns an error code. If the code argument is a variable, the predicate always succeeds regardless of whether the actual read succeeds. If the error code is 0, `get_term` only succeeds if the read operation succeeds.

One of the more important syntax error codes is 2, which signifies that the term read is incomplete. When `get_term` is used with pipes, this code permits synchronization between readers and writers of pipes.

### **sread, read**

The `read` and `sread` predicates can read any number of terms from symbols, streams, or pipes. For example:

```
?- sread(_stream_id, _term1, _term2, _term3).  
    % read three terms
```

The predicates to read terms can be used to convert a symbol into a Prolog term. For example,

```
?- sread('[sym, two( \'in quotes\' , _var), 5].', _list).  
    ?- sread('[sym, two( \'in quotes\' , _var), 5].',  
            [sym, two('in quotes', _var), 5]).  
YES
```

unifies `_list` with a three element list, consisting of a symbol, a structure, and an integer. (`sread` fails if more than one term is read from a symbol.)

The predicate `read` is identical to `sread`, except that no target is specified. The read is performed on the default input stream.

### **write, write, swriteq, writeq**

The two primary write predicates, `write` and `swriteq`, write to streams, or symbols. The predicates `write` and `writeq` are identical to `swrite` and `swriteq`, respectively, except these predicates target the default input stream, and thus require no stream identifier. Any number of terms may be written, but no



special punctuation ("", for example) is output between terms. Structures and lists are written with the appropriate bracketing and with commas between elements. Expressions are output according to operator declarations, with blanks separating operators from their arguments, and parentheses to show grouping. The functorial form of operator expressions never appears. For example, the term

```
'is'(_x, '+'(_y, _z))
```

is always output as

```
(_x is (_y + _z))
```

Unlike `swrite`, `swriteq` delimits special symbols with single quote characters, and always writes a blank character after every term. Terms written using `swriteq` are in a format that is acceptable to the `read` predicates, save for the possible missing trailing punctuation. The same cannot be said for a term written by `swrite`. Consider:

```
:- swrite(l, 'The', 'cat in the hat', came, back).  
Thecat in the hatcameback  
  
:- swriteq(l, 'The', 'cat in the hat', came, back).  
'The' 'cat in the hat' came back
```

Just as `sread` can convert a symbol into a Prolog term, `swrite` and `swriteq` can convert a term into a symbol by directing their output to a symbol rather than a stream. The symbol is unified with the first argument, usually an unbound variable.

Because variable names are preserved with clause definitions and queries, the write predicates access and output variables by name whenever possible. Variable names are written with a leading underscore, "\_", to avoid confusion with capitalized symbols. If a term has different variables with the same name, a numeric suffix is added to the name to distinguish the variables. This is suffix convention applies only if the variables appear in the same term.



Given the fact

```
info(fact(_M, _N)).
```

note the difference in output for the following queries:

```
/* writing two terms
   variables are not distinguished */
:- info(X), info(Y), nl, writeq(X, Y).
fact(_M, _N) fact(_M, _N)

/* writing one term, a list
   variables are distinguished by number suffixes */
:- info(X), info(Y), nl, writeq([X, Y]).
[fact(_M, _N), fact(_M_1, _N_1)]
```

`swrite` can also be used to implement a deterministic variation of the `concat` predicate by specifying the target as a symbol. The following queries are equivalent:

```
?- swrite(_symbol, first_symbol, second_symbol).
?- concat(first_symbol, second_symbol, _symbol).
```

Because `swrite` can write any number of terms, it can be considered a variadic `concat` predicate. While `concat` is restricted to symbols, `swrite` and `swriteq` can concatenate any number of any type of term into one symbol.

### **put\_term**

`put_term`, which can write to streams or symbols, writes a single term with trailing punctuation, such that the output may be immediately read by a predicate that reads terms. This is the only output predicate that returns an error code.

```
:- nl, put_term(1, ['First', _x : [fast, slow]], 0).
['First', (_x : [fast, slow])].
YES
```



**print, portray**

The `print` predicate outputs a single term in accordance with the specification provided by the predicate `portray`. `print`, with one argument, writes the term to the default output stream. With two arguments, it writes the second argument to the stream specified by the first argument.

Any user implementation of `portray` must have two arguments, a stream identifier and the term to be written. If there is no user version of `portray`, `print` defaults to a version that specifies the format of `swriteq` and is also capable of writing a nonprintable term (either a cyclic or looped structure) in a printable form. For example:

```
:- _X = fred(2, george(_X)), nl, print(p(_X)).  
  (p(_1) where [_1 = fred(2, george(_1))])  
YES
```

**get\_char, put\_char**

A character is the smallest unit of data that can be read or written. However, as character is not a valid term type, characters are actually treated as symbols consisting of only one character. `get_char(_stream_id, _char)` reads a single character symbol from the specified stream; `put_char(_stream_id, _char)` writes a single character symbol to the specified stream. The stream argument may be omitted if the target is the default stream.

**nl**

A character or term following a newline character is shown at the beginning of the next line. Since this is a very common output format, the predicate `nl` (equivalent to `put_char('\n')`) is provided. A variation which takes a single argument specifying the target stream is also provided.



### **readln**

Reading a line of text is equivalent to reading a sequence of characters up to and including the next newline character, then returning the characters in a symbol without the newline. This action is performed by the `readln` predicate, which is provided primarily to optimize the processing of free form text.

`readln(_stream_id, _symbol)` unifies `_symbol` with next the line of text in the specified stream; `_stream_id` can be omitted if the default input stream is to be used.

In summary, predicates that employ side effects are essential in implementing practical Prolog programs. Text input and output predicates comprise the first set of such predicates, and are provided in some form in all Prolog systems. The next chapter discusses predicates affecting the internal structure of the program contained in the knowledge base.

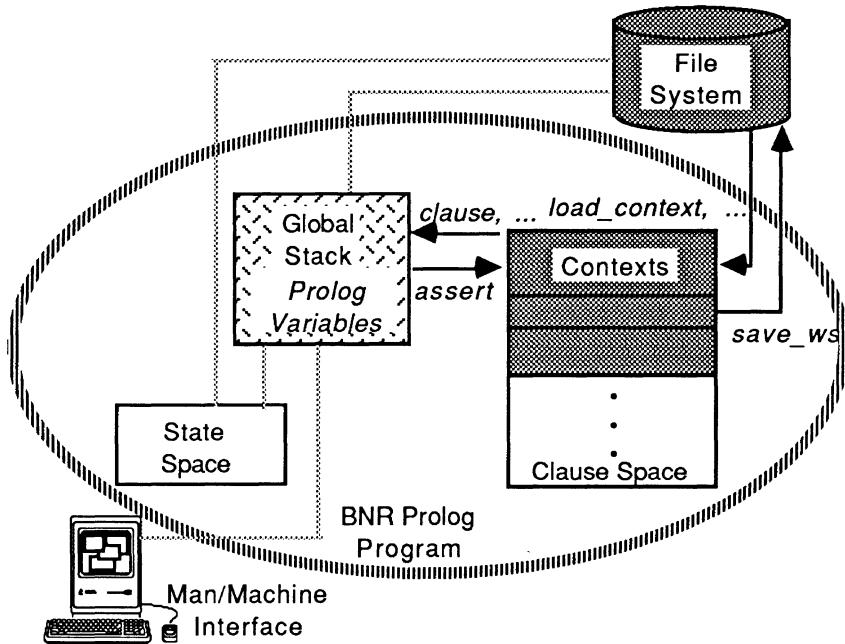






# Chapter 14

## The Knowledge Base



Unlike conventional languages, Prolog has traditionally permitted modification of the knowledge base (program) during execution. This is achieved through use of predicates such as `assert` and `retract`, which act on the knowledge base by means of side effects.

The BNR Prolog knowledge base is divided into modules called *contexts*. For large scale programming, some support for modularity is essential. On the pragmatic side, modules are units of software used for development and version control, and are the basis upon which software libraries are formed. Since the file is the most convenient unit of data storage and control,



BNR Prolog uses a one to one correspondence between files and modules.

Several considerations affect the possible forms of modularization. Modules should be as independent as possible, since dependencies between different modules create complex administrative problems and compromise independent development and library formation. In systems supporting compilation, these problems are magnified since both compile time and run time dependencies must be managed consistently.

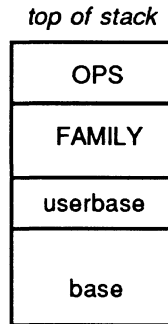
Modules should encourage information hiding. The proper use of a module should require much less information than is needed to duplicate its functionality. Further, the number of transformations (compiling, for example) that preserve a modules semantics should be maximized. In BNR Prolog, information hiding is provided by local names and state spaces.

Since correctness often depends on the enforcement of a controlled interface, mechanisms are provided to prevent the modification of predicate definitions.

## Contexts

A module in the knowledge base has a name, can vary in size, and is known as a context. Contexts exist in memory as a last in, first out (LIFO) stack, called the `world stack`. The first context is `base` which contains the definitions for the built-in predicates. This is followed by `userbase`, the default user context. One or more user contexts may follow, where the last context loaded is the current context on the top of the stack. In the following example, the current context is `OPS`, which was created after loading the context `FAMILY`.





Example of Clause Space

From the point of view of logic, this structure corresponds to the notion of extensions of a formal system. The contents of `base` above, regarded as a set of axioms and definitions, has been extended by the addition of further axioms and definitions in the other contexts. In this conceptual model, removing the axioms of an extension also removes all consequences of those axioms and restores the knowledge base to the state it was in prior to the extension.

Each context has a local state space in which to keep private state information. Local state spaces are discussed in the chapter "State Spaces".

## Creating and Removing Contexts

Contexts are usually created on the stack by loading a specified file. For example, the query

```
?- load_context('FAMILY').
```

creates a new context with the name `FAMILY` that contains the clauses read from the file *FAMILY*. If the new context has the same source file as one that already exists on the stack, the predicate `load_context` has no effect. (The load commands in the **Contexts** menu do a reload if the context already exists.)



Contexts not associated with files can also be created on the stack by using the `enter_context` predicate. For example, the query

```
?- enter_context('OPS').
```

creates a new empty context named OPS, which has no related file.

Context files contain none of the "directives" that are found in some Prolog systems. In BNR Prolog, operator definitions are handled simply as facts, and initialization is handled by means of the `$initialization` predicate. Whenever a file is loaded, this predicate is automatically executed once loading is complete. Any necessary initialization, including loading other contexts, can be specified in the body of `$initialization` clauses.

A binary context image is created for a source file and stored in its resource fork the first time the file is loaded from disk. Until the disk file is changed, whenever a call is made to load that file, the binary image is loaded instead of reparsing the text. There are no semantic differences between the source and its binary image, but the binary image loads considerably faster. Open text windows, which generally contain files undergoing significant (but uncommitted) changes, are always parsed each time; binary images are not constructed.

If the predicate `reload_context` is called to reload a specific file, all contexts are popped from the stack, starting with the current context down to the one specified. The popped contexts are then reloaded from open text windows or files, starting with the specified file. Open text windows are always checked first when reloading a file. Therefore, any changes to open files are effective with the reload. Since reload information is taken from files or open windows, any dynamic changes made either interactively or from programs, for example `asserts`, are removed. If a context on the stack was created with `enter_context`, it cannot be reloaded from a file. In such a case, reloading of that context is omitted.



A context is unloaded by exiting, which removes it from the stack. For example, the query

```
?- exit_context('FAMILY').
```

removes FAMILY. If the specified context is not at the top of the stack, all contexts above it are removed as well. If a nonexistent context is specified, there is no effect on the stack.

An `exit_context` restores the clause space to the state that existed at the time of the corresponding `enter_context` or `load_context`. In particular, all clause assertions and all operator definitions made in the interim are undone by the exit, and their storage is recovered. If any of the removed code is still being executed, or if there are any outstanding references to anything (including symbols) in a removed context, then "dangling reference" errors may be generated. For this reason, programs using `exit_context` or `reload_context` must ensure that there are no dependencies on the material being deleted. (The global state space may be used as a temporary storage while the context stack is changing. See the chapter "State Spaces")

## Symbols and Clauses in Contexts

Each context consists of a set of clauses. Symbols beginning with "\$", called local symbols, are scoped to the context, so \$fred in one context is distinct from \$fred in any other context. Thus several different contexts can each have their own distinct semantics for the same symbol. The use of local symbols minimizes the problem of name conflicts in contexts created by different programmers.

Local names entering the system for the first time always go into the current context. Thus, although a local symbol can be output from any context, the same abstract symbol can only be input into the current context. This feature can be applied to maintaining a symbol table for a scoped language, or in applications where it is necessary to keep a set of object language symbols distinct from the metalanguage symbols of Prolog.



If local symbols are exported from a context at runtime, it is easy to get confused as to which symbols are really the same if only the name of the symbol is output. For example, it is perfectly possible to have two distinct, nonunifiable symbols that both print as `$fred`. If two contexts are loaded, where `p($fred)` appears in one context, and `q($fred)` appears in another, then

```
?- p(_X), q(_Y).  
    ?- p($fred), q($fred)  
YES
```

but

```
?- p(_X), q(_X).  
NO
```

All symbols except those that start with "\$" are global to the system. Global name spaces may require some manual supervision to avoid unintended name clashes. Because global symbols have no ambiguity between name and symbol, their names can be freely input and output.

The scoping rules for the clauses in contexts vary somewhat from the rules for procedures within modules in other programming languages. Clauses are defined local to a context by using a local name. Clauses with the same local name may exist in different contexts and be totally independent. An example is the `$initialization` predicate mentioned above, which may be independently defined in every context.

Local predicates can only be called or referenced directly in the context in which they are defined. Thus, local predicates can be written (and compiled) with full knowledge of how they are to be used, which generally allows them to be written in a more specific and efficient manner, and with a wider range of possible compiler optimizations.

For global predicates, clauses may exist in more than one context. In such a case, the clauses in the more recent contexts are tried before the clauses in earlier contexts. This is called *overloading*.



A common use for overloading is to provide general clause definitions in lower contexts, and then overload these with more specific definitions in later contexts. When this organization is used, the lower contexts contain highly generalized abstract descriptions, which are particularized to restricted problem domains by overloading. Thus, a possible organization for a diagnostic expert system might be

*current context*

tentative hypotheses
specific problem symptoms
site specific knowledge
generic predicates for the problem domain
the abstract inference model

Overloading of global definitions can be prevented by using the `close_definition` predicate. A closed definition cannot be modified by later context loads or by program execution. It is the responsibility of the provider to close a definition if it is desirable to prevent overloading. A convenient place to close definitions is in the `$initialization` predicate of the defining context, although closing is possible at any time after the definition has been loaded.

The order in which contexts are loaded is generally not significant. One exception occurs when overloading is being used, in which case the proper order is determined by the desired order of predicate overloading. A second exception occurs when using local state spaces to hold local names from other contexts, in which case the semantics of local names may affect the order. (See the chapter "State Spaces".)



## Accessing Predicate Definitions

The definitions of predicates can easily be accessed from programs by using either the `definition` or `clause` predicates.

### **definition, clause**

`definition` uses the canonical form of clauses, as in the query

```
?- definition(_Head :- _Body, _Context).
```

`_Head` is always a structure and `_Body` is a list or variable. If the `_Context` parameter is instantiated, it causes the selection of clauses to be restricted to the specified context. Otherwise, it returns the context in which each particular clause occurs.

The use of the canonical clause form simplifies the retrieval of clauses which may have been originally entered in several different (but equivalent) forms. Since clause bodies are returned as lists, they may be manipulated by the usual list utilities or directly executed if desired. For example, the following predicate

```
esubset([], _).                % in the context OPS
esubset([_X, _Xs..], [_X, _Ys..]) :-
    esubset(_Xs, _Ys).         % in the context New OPS
esubset([_X, _Xs..], _Y) :-
    member(_X, _Y), esubset(_Xs, _Y).
                                % in context New OPS
```

can be queried to obtain all clauses,

```
?- definition(esubset(_A..) :- _B, _Context).
```

all clauses with two arguments,

```
?- definition(esubset(_A1, _A2) :- _B, _Context).
```

all facts,

```
?- definition(esubset(_A..) :- [], _Context).
```



all clauses in context OPS,

```
?- definition(esubset(_A..) :- _B, 'OPS').
```

and so forth. The clause predicate is similar but does not provide context information. For example,

```
?- clause(esubset(_A..) :- _B). % all clauses
?- clause(esubset(_A..) :- []). % facts only
```

### **hide**

The predicate `hide` can be used to hide a predicate so that `definition` and `clause` do not work on it. This facility can be used to prevent inspection of confidential algorithms.

### **listing**

The clauses of a predicate may also be displayed, in the order in which they will be tried, by using the `listing` predicate. Listing a predicate displays the name of the context (in comment form) for each clause of the predicate, as well as the internal representation of the clauses. For example, listing the predicate `father` gives:

```
:- listing(father).
   father('Aaron', 'Adam').      % Context: "OPS"
   father('Zachary', 'Zeke').    % Context: "OPS"
   father('Michael', 'Sue').     % Context: "FAMILY"
   father('Stephen', 'Michael'). % Context: "FAMILY"
YES
```

## **Asserts and Retracts**

`assert` is a predicate that adds a new clause to the knowledge base and asserts its truth, thus authorizing its unrestricted use in proofs. Every clause asserted is entered into the current context, and automatically disappears when the context is removed. `retract` is used to remove clauses in the current context. Confining the effects of `assert` and `retract` to the



current context provides a mechanism for containing knowledge base side effects.

### **assert**

The two natural forms of `assert`, called `asserta` and `assertz`, have identical interfaces based on the canonical form of clauses. `asserta` adds a clause to the top of the clause chain for its predicate. This is useful for overloading or extending predicate definitions, and also for dynamic fact tables. There is no ambiguity about its effect on concurrent executions of the overloaded predicate: existing executions are unaffected, while subsequent calls see the new clause.

`assertz` is principally used for loading clauses in the order in which they would be entered into a context file or interactively. When used to add facts to a dynamic predicate, `assertz` can be the cause of semantic problems if the predicate is being executed while being changed. (For this reason the `assert` predicate is defined to be equivalent to `asserta`.) Since clauses can be added only to the current context, if a predicate has clauses in lower contexts, `assertz` is not able to add a clause at the end of the clause chain for its predicate. (This may cause problems in porting from systems with a flat, unstructured clause space if there are global name clashes. To some extent these systems can be accommodated by restricting the clause space to a single user context. )

For example, if the current context, `OPS`, contains

```
father('Benjamin', 'Baxter').
father('William', 'Warren').
```

and the previously loaded context `FAMILY` contains

```
father('Michael', 'Sue').
father('Stephen', 'Michael').
. . .
```



and the following assertions are made

```
?- asserta(father('Aaron', 'Adam')).
?- assertz(father('Zachary', 'Zeke')).
```

a query of the predicate `father` causes a clause space search resulting in the following:

```
?- father(_X, _Y).
   father('Aaron', 'Adam').
   father('Benjamin', 'Baxter').
   father('William', 'Warren').
   father('Zachary', 'Zeke').
   father('Michael', 'Sue').
   father('Stephen', 'Michael').
   . . .
```

### **retract**

Clauses may be removed by using `retract`. Like `assert`, `retract` only applies to clauses in the current context. Because `retract` uses backtracking, it can be used to remove all clauses in the current context for the specified predicate. For example,

```
?- retract(father(_X..)), fail.
```

or

```
?- foreach(retract(father(_X..)) do true).
```

removes all clauses of the predicate `father` from the context OPS.

In summary, Prolog programs can be constructed from independent modules stored in separate files and loaded as contexts. Information hiding can be maximized by using symbols and predicates local to a context. The context stack discipline ensures that the knowledge base can be returned to a consistent state, and provides a mechanism for the containment of `assert/retract` side effects. State spaces, discussed in the next chapter, provide an alternative storage mechanism for Prolog data structures.

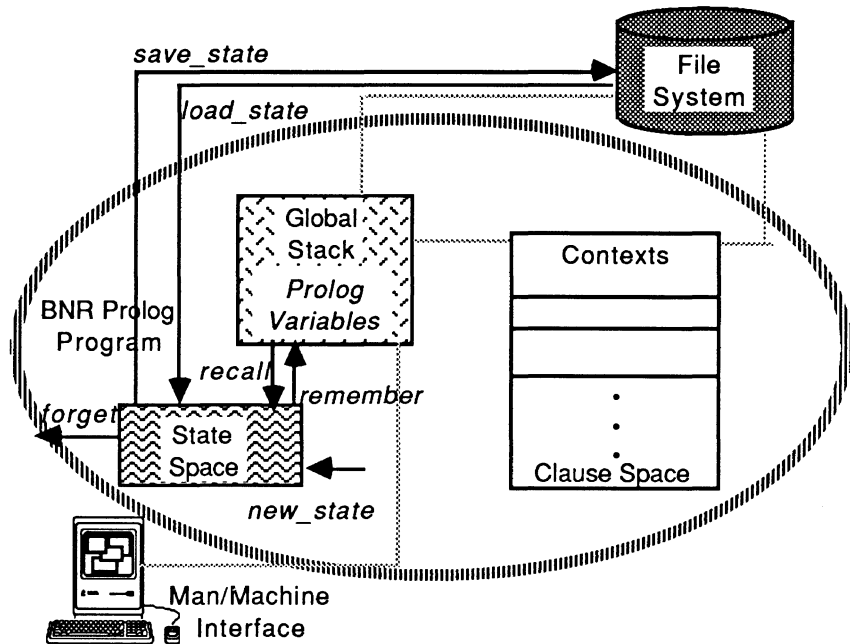






# Chapter 15

## State Space



Many applications require a place to store changeable data during a computation. Traditionally in Prolog, such data has been handled as dynamic predicates, using `assert` and `retract`. Asserting facts in the clause space authorizes their unrestricted use in proofs. However, it is often desirable to remember data without committing to its truth at all. In many cases it is desirable to be able to easily save the "state" of an application separate from the knowledge base of the application. This is difficult if the two are intermingled. Storage reclamation is also a problem if data changes occur frequently.



State spaces are memory resident data bases for Prolog structures which do not require that the data be "asserted", and offers more flexible data management. There may be private state spaces associated with each context, as well as a global state space. A convenient unification based access method is provided, similar to that used in the clause data base. This class of storage has attributes which are midway between external files and the knowledge base, and represents a different set of tradeoffs between accessibility, updateability, and data independence.

The contents of each state space is a collection of Prolog structures, grouped by functor name. Within each group, the structures are kept as an ordered set known as the *recall order*. The built-in predicates make it possible to add or remove terms from either end of this set, scan it from either end by pattern matching, or update it without affecting the order.

The interface to the state space is analogous to the handling of facts in the clause space, as can be seen in the following table:

State Space	Clause Space
remembera(_Structure)	asserta(_Fact)
rememberz(_Structure)	assertz(_Fact)
remember(_Structure)	assert(_Fact)
forget(_Structure)	retract(_Fact)
forget_all(_Structure)	retract_all(_Fact)
forgetz(_Structure)	
recall(_Structure)	clause(_Fact)
recallz(_Structure)	
update(_Struc1, _Struc2)	

## Global State Space

The global state space is visible from all contexts. It is not affected by loading, exiting, or reloading contexts. A global state space is created either by a call to `new_state`, or by the first call to `remember`. Any state space is dynamically extended as



required, memory permitting. However, there is less chance of encountering fragmentation problems if the size is preallocated.

### **new\_state**

The call `new_state(0)` discards the current global state space. If no state space currently exists, `new_state(size)` allocates `size Kbytes` for use as the global state space. Thus

```
:- new_state(0), new_state(50).
```

discards the existing state space and allocates a new space of 50 Kbytes. The call `new_state(_var)` can be used to query the size of the state space; `_var` is unified with 0 if no state space currently exists.

### **remember, recall**

The goal sequence:

```
remember(box(10, 10, 20, 20, red)),
remember(box(30, 20, 40, 50, green)),
remember(box(60, 80, 70, 90, green))
```

stores the specified structures in the global state space. They may then be retrieved by a subsequent nondeterministic call of the form:

```
?- recall(box(_A, _B, _C, _D, _Color)).
?- recall(box(60, 80, 70, 90, green)).
?- recall(box(30, 20, 40, 50, green)).
?- recall(box(10, 10, 20, 20, red)).
YES
```

Note that the terms are returned in last in, first out order.

`recall` works by unification and backtracking, as evidenced by:

```
?- recall(box(_A, _B, _C, _D, green)).
?- recall(box(60, 80, 70, 90, green)).
?- recall(box(30, 20, 40, 50, green)).
YES
```



Since retrieval keys on the structure name and the first argument if it is instantiated, it may be more efficient to rearrange the order of arguments to optimize the most frequent queries. The above example would be more efficient if `_Color` was the first argument in the structure `box`.

`recallz` is similar to `recall`, but accesses the items in the opposite order:

```
?- recallz(box(_A, _B, _C, _D, green)).
?- recallz(box(30, 20, 40, 50, green)).
?- recallz(box(60, 80, 70, 90, green)).
YES
```

### **forget**

`forget` and `forgetz` access the contents of the state space in a manner similar to `recall` and `recallz`, respectively, but they remove the items from the state space as they retrieve them. A last in, first out stack can be implemented using `remember` (which is equivalent to `remembera`) and `forget` together, or by using `rememberz` and `forgetz`. A first in, first out queue can be implemented with either a `remember` and `forgetz` combination, or with `rememberz` and `forget`. For example:

```
enqueue(_Queue, _Item) :- rememberz(_Queue(_Item)).

first(_Queue, _Item) :-
    recall(_Queue(_V)),
    !,
    _V = _Item.           % may fail if _Item does not unify

dequeue(_Queue, _Item) :-
    forget(_Queue(_V)),
    !,
    _V = _Item.           % may fail if Item does not unify
```

A common source of problems is forgetting that these operations are nondeterministic: an unexpected failure in subsequent calls can create a great deal of unnecessary work, or even damage the data base in the case of the `forget` predicate. Such problems are minimized by using `once(forget(_))`.



### **update**

The `update` predicate is useful for updating state space items without affecting their order. The call `update(_Old_term, _New_term)` replaces the first occurrence of `_Old_term` with `_New_term` in the same position in the recall order, but restores the original term on backtracking. A sequence like

```
[update(box(_A, _B, _C, red), box(_A, _B, _C, yellow)),
  update(box(_A, _B, _C, green), box(40, 40, 50, green)),
  cut]                                % commit changes
```

then acts as a transaction which is committed by the `cut`. In this transaction, either all or none of the updates in the sequence take place. Careful use of such transaction sequences can help maintain the internal consistency of the data base.

In many instances, the new structure must be computed from the old structure. In the following example, a message is addressed to an object instance that belongs to an object class. The sequence obtains a message from a message queue, then updates the object state and a history trace.

```
[forget(queue(message(_Instance, _Msg, _Msg_Data...))),
  [once(recall(map_object(_Instance, _Class))),
    once(recall(_Class(_Instance, _Old_Obj_state))),
    _Class(_Msg, _Old_Obj_state, _New_Obj_state,
            _Msg_Data...),
    update(_Class(_Instance, _Old_obj_state),
            _Class(_Instance, _New_obj_state)),
    remember(history(_Instance, _Msg, _Msg_Data)),
    cut          % commit update
  ]
]
```

The symbol represented by `_Class` is used both to group the states of object instances in the state space, and as the name of the predicate which specifies the state changes for objects of that class.



Operations that remove data from the state space, such as `forget` and `update`, reclaim the storage immediately. Therefore, data which is subject to frequent change is best kept in a state space if storage reclamation is an issue. However, because data is copied to the global stack with each `recall`, access to state space is slower than in the clause space and the need for speed must be balanced against the need for storage management.

### **inventory**

The call `inventory(_Name)`, where `_Name` is a variable, generates all the symbols that have entries in the state space. The following goal writes the contents of the state space to a stream:

```
foreach([inventory(_Name), recall(_Name(_A..))] do
    [nl(_Stream), put_term(_Stream, _Name(_A..))])
```

### **save\_state, load\_state**

The global state space is independent of the context stack. It can be saved to an external file using `save_state(_File_name)` and loaded into memory from a file using `load_state(_File_name)`. Loading a new global state space deletes any existing global state space. The binary state space files are a convenient medium for long term storage of Prolog data structures, as well as a communication medium between Prolog applications.

When local names that are stored in a global state space are later recalled, they become local to the topmost context at the time of recall. The effect is the same as writing the structures to a file and then reading them in later. This allows local names to be transferred from an old context to a new context "by name", simply by moving them in and out of the global state space. For example, in a Prolog source to source compiler or program transformer, if the original source contains local names, it can be loaded as a context. The clauses can then be extracted, transformed, and remembered into the global state space. The original source context can then be popped, and the transformed source moved from the state space into a new context. Any local



names in the original source, as well as any introduced during the program transformation, become local names in the newly created context.

## Local State Spaces

Each context has a single local state space referred to as `$local`. Like the global state space, local state spaces can be created explicitly by `new_state(Size, $local)`, or implicitly by the first `remember`. Note that since `$local` is a local name, it represents a different abstract symbol in each context.

The interface to the local state is the same as for the global state space, except that the identifying `$local` symbol is added to the end of each call as in

```
remember(box(10, 10, 20, 20, green), $local)
rememberz(box(10, 10, 20, 20, green), $local)
recall(box(_, _, _, _, green), $local)
recallz(box(_, _, _, _, green), $local)
forget(box(_, _, _, _, green), $local)
forgetz(box(_, _, _, _, green), $local)
```

Normally, only the code in a context has direct access to the associated local state space of that context. From the console, only the local state space of the current context is directly accessible. However, each `$local` symbol acts as a *capability* to access the corresponding local state space; if a context `C` dynamically exports its `$local` symbol, it can be used by code in other contexts to access `C`'s local state space. For example, it is possible to write a context that provides state space utilities (such as the queue utilities above), and make the local state space an explicit parameter.

```
enqueue(_Queue, _Item, _Space) :-
    rememberz(_Queue(_Item), _Space).

first(_Queue, _Item, _Space) :-
    recall(_Queue(_V), _Space),
    !,
    _V = _Item.           % may fail if _Item does not unify
```



```
dequeue(_Queue, _Item, _Space) :-  
    forget(_Queue(_V), _Space),  
    !,  
    _V = _Item.           % may fail if _Item does not unify
```

The call

```
:- enqueue(msg_queue, msg1(..), $local).
```

enqueues msg1 to whichever state space is referenced by \$local.

It is also possible to generalize this to include the global state space (only the enqueue predicate is shown, but the same method applies to first and dequeue as well):

```
enqueue(_Queue, _Item, _Space..) :-  
    rememberz(_Queue(_Item), _Space..).  
  
:- enqueue(msg_queue, msg1(..)).           % global  
:- enqueue(msg_queue, msg1(..), $local). % $local
```

Local states spaces differ from the global state space in that if local names from the same context, or from an older context (deeper in the stack), are saved in a local state space, they still refer to the same object when recalled later. This is necessary in applications such as building object oriented systems, where method names need to be "localized" to enforce uniqueness. On retrieval they should be the same object. If a program makes use of this behavior, it may restrict the order in which contexts are loaded.

Unlike global state spaces, local state spaces cannot be saved as files. The local state space is automatically discarded when the context is exited. This eliminates problems caused by leaving unwanted items in the state space to affect future executions. The global state space requires that the application explicitly perform this housekeeping function.

The characteristics of state spaces are incremental storage reclamation, unification based access, and independence from context structure. These make state spaces ideal for retaining



the changing information associated with graphics oriented interfaces, which is the subject of the next chapter.

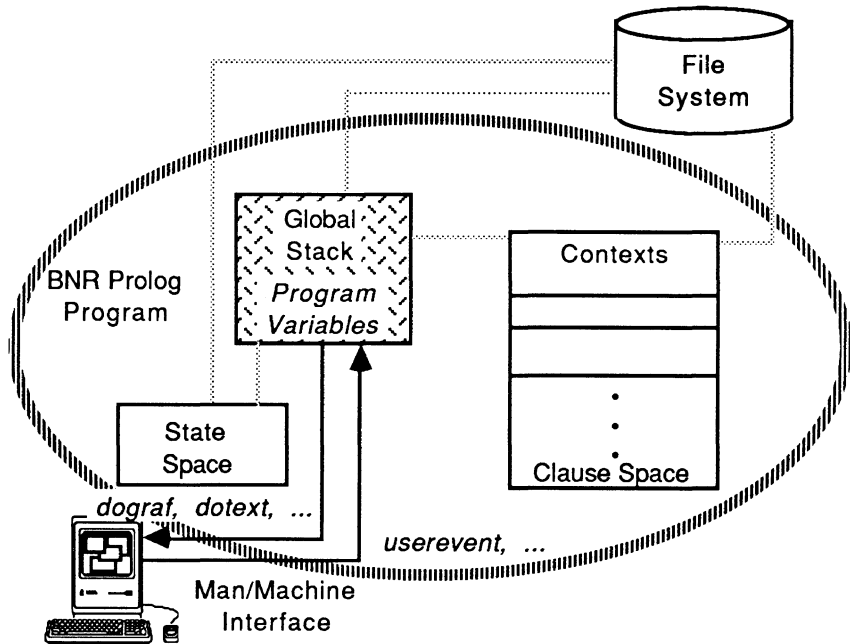






# Chapter 16

## User Interfaces



This chapter describes both the philosophical approach to the design of user interfaces for Prolog programs on the Macintosh, and the tools provided for their implementation. In general, user interfaces tend to be highly state dependent, and hence side effects play an important role in their construction.

Virtually every user interface can be modeled as a loop such as the following:

- display information to the user
- wait for user input



- process the user request
- return to the beginning of the loop.

The Macintosh provides two physical input devices, the keyboard and the mouse. These two devices are used in combination with output facilities to generate a range of input techniques, such as the mouse pointer, menus, windows and dialogs.

User input can be acquired in two ways, either synchronously using dialogs, or asynchronously by polling for events. Dialogs are predetermined mini-interfaces presented in pop-up windows that require a response from the user. While they provide a convenient input mechanism for some purposes, such as choosing a file or displaying a warning message, they restrict the user to operating in a specific *mode*.

Modal interfaces should be avoided whenever possible. It is preferable to give the user the freedom to perform any action allowed by the Macintosh, and then respond to it appropriately. All such actions (for example, clicking the mouse button or selecting a menu item) constitute *events*. The ability to detect and respond to asynchronous events makes it possible to build user interfaces that feel responsive and free form.

The basic context for input and output in a Macintosh user interface is the window. When a user event occurs, the active window provides a context for the interpretation of that event (for example, selecting **Close** from the **File** menu usually means close the active window). Every user event includes window information which can be used as a filter for routing the event to the appropriate program code.

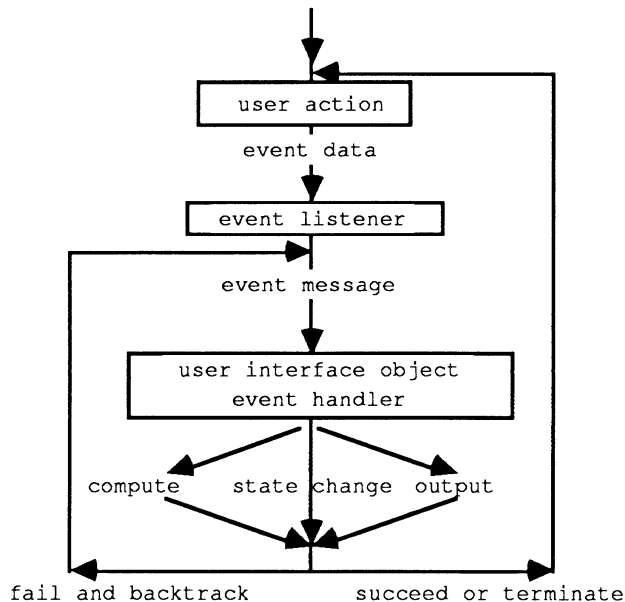
The text facilities provide for the manipulation of text windows at the level of characters or lines, as well as pattern matching, text selection, replacement, insertion, and deletion. In addition, the relationship between a text window and its associated disk file may be manipulated by loading, saving, redirecting, and determining whether the disk and window versions of the text match.



The graphics facilities allow access to most of the Macintosh Quickdraw toolbox routines. These include the drawing of lines, rectangles, ovals, text, icons and composite picture structures using integer, real, and interval coordinate specifications. A host of drawing attributes may also be set or queried.

## Events

The focal point of an application is an *event polling loop* in which events are detected and dispatched as messages to event handlers. The event detector and dispatcher is called an event listener. Event handlers are Prolog rules which execute when an appropriate event message is dispatched. These rules represent the primary mechanism by which one communicates with user interface objects.



Event Polling Loop

Although user interface objects can be any type of program construct, the fundamental user interface object in most

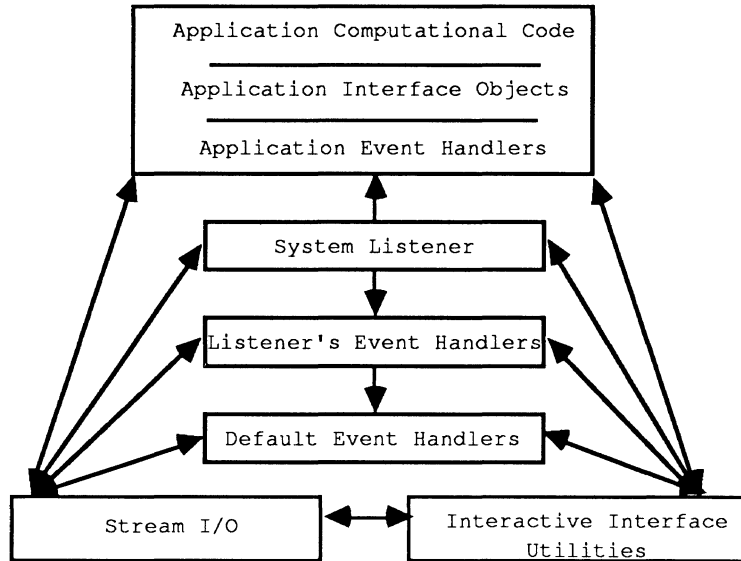


programs is the window. Once an event is detected, for example when the user has clicked on a menu item, the event type and the associated window can be used to trigger an action. This is analogous to sending a message (the event) to an object (the window). All events are associated with the name of a window (usually the active window) which is, by default, the intended receiver of the event message.

The actions triggered by any event message are determined by the event handlers. By succeeding they terminate the processing of the event. By deliberately failing after accomplishing some side effect, event handlers effectively resend event messages to other interested event handlers by means of normal Prolog backtracking.

The development environment is basically like any other Prolog application, in that its user interface is driven by an event polling loop written in Prolog. In the absence of user defined event handlers, a set of predefined handlers perform sensible default actions. Collectively, they implement the development environment. (Some of these handlers simply succeed without doing anything.) Because of this implementation, a considerable amount of functionality exists over and above that provided by the documented interface predicates.





User Interface Architecture

Since an application can be written purely as a set of event handlers, multiple applications may be active at the same time. The event handlers in each application receive only those event messages pertaining to its own user interface objects, preventing interference between applications. When no events are occurring, the system idles in the listener. As events occur, the listener sends messages to the appropriate handlers regardless of which applications are present. In this manner, all applications concurrently receive any and all event messages pertaining only to themselves.

Not every application can be coded as a set of event handlers. However, if an application periodically solicits user input, it can take on the role of the listener and distribute events fairly to all loaded handlers. In a standalone application this may not be a consideration, although even such an application may be coded as a set of independent processes running (apparently) concurrently.



## Event Listeners and Handlers

The Prolog system generates more than a dozen distinct user events based on its contextual interpretation of keyboard and mouse input. The basic event polling loop for any listener is driven by the `userevent` predicate which detects these events. Subsequent to the call to `userevent`, the appropriate event handler clause is invoked.

A skeletal listener might read

```
samplelistener :-  
    repeat,  
        userevent(_event, _window, _d1, _d2, noblock),  
        once(_event(_window, _d1, _d2)),  
    fail.
```

Note that since the listener loop is a failure driven backtracking loop, the handling of events must involve side effects to keep state information. For the same reason, there is no cumulative storage requirement on the global stack; only enough stack space to handle the single most complex event is required.

An application can combine system events with further context information to generate higher level events.

### **userevent**

The `userevent` predicate returns user events when they occur. Its calling format is defined as

```
userevent(_Event, _Window, _Data1, _Data2)
```

where the parameters contain the event that occurred, the window in which it occurred or the active window at the time it occurred, and two data fields whose contents depend on the event. `userevent` can also be called with a fifth argument, the symbol `noblock`, as in

```
userevent(_Event, _Window, _Data1, _Data2, noblock)
```



which forces immediate success, whether or not an event has occurred. Nonblocking calls return an idle event if no other event is pending.

The events generated by the system are, in order of report priority

```
userdeactivate, useractivate  
  
menuselect, usermousedown, usermouseup, userclose,  
userdrag, usergrow, userzoom, userkey  
  
userupdate  
  
userupidle, userdownidle
```

`userdeactivate` and `useractivate` events have the highest priority, and are generated when the active window is changed. First the window order is changed, then a `userdeactivate` is reported followed by a `useractivate`. These events are invaluable for tracking and responding to changes of the active window.

`usermousedown`, `usermouseup`, `menuselect`, `userclose`, `userdrag`, `usergrow`, `userzoom` and `userkey` are all of equal priority and are reported on a time ordered basis. If the mouse button is pressed inside an active graphics window a `usermousedown` event is generated. This, in time, is followed by a `usermouseup`. If a menu item is selected by means of the mouse or a command key equivalent, a `menuselect` event is generated. Windows have certain optional control regions in which a mouse click generates one of the `userclose`, `userdrag`, `usergrow` or `userzoom` events. If a key (other than a valid menu command key) is pressed, a `userkey` event is generated.

A `userupdate` event is generated when a portion of a graphics window requires redrawing by the application. This usually occurs when a hitherto covered portion of a graphics window is uncovered as the result of moving or reordering the windows.

The two idle events, `userupidle` and `userdownidle` are only generated when the nonblocking variant of `userevent` is invoked



and there are no other events available. These events are useful for tracking mouse movement (for example, dragging objects) and for enabling application background processing when no real user events are occurring.

It is important to note that if the `userevent` predicate is called with instantiated arguments and the next event does not make the call succeed, that event is discarded and lost. Thus, the query

```
?- repeat, userevent(userkey, _, _, _, noblock).
```

completely locks up the development environment until a key is pressed.

Events are handled by calling the goal

```
_Event(_Window, _Data1, _Data2)
```

where `_Event` is one of the event types generated by `userevent`. Therefore, event handlers are clauses of the form

```
_Event(_Window, _Data1, _Data2) :-  
    [< event handler body>].
```

where unification with the clause head provides the appropriate filtering of messages.

A typical example of a simple handler is the default `userkey` event handler supplied with the system. It simply echoes every key typed by the user back to the text window in which it was typed, filtering out command keys. This handler is normally executed for every key stroke typed in the development environment.

```
userkey(_Window, _Key,  
        [_control, _option, _capslock, _shift, 0, _mouseup]) :-  
    dotext(_Window, replace(_Key)).
```



## Windows

Windows are rectangular regions on the screen enclosed by frames. Every window has a unique name which identifies it, and its own relative coordinate system. Windows can be of any size and located at any position on the screen plane (with the coordinates between -32768 to 32767 along each axis). Multiple windows may be open at the same time, and may overlap each other on the screen. The window on the top of the stack is the active window. All other windows are ordered front to back, with the most recently used windows closest to the top of the stack.

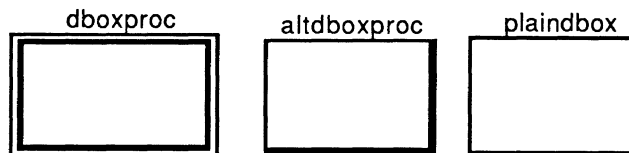
Once created, windows may be moved, resized, brought to the front of the stack (that is, made active) or closed, by using generic window predicates. Windows may also be hidden, or made invisible. Such windows maintain all their behavioral characteristics, but they cannot be seen and are placed at the bottom of the window stack.

### **openwindow**

Windows are opened by a call to

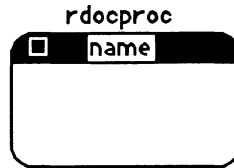
```
openwindow(_type, _name, pos(_left, _top),  
           size(_width, _height),  
           options(_options..))
```

Frames can provide various window manipulation controls in addition to visually displaying window boundaries. The type of a window frame is specified within the `options` argument of the `openwindow` predicate when the window is created. There are eight different window frames available. `dboxproc`, `altdboxproc` and `plaininbox` simply provide a visual window boundary.

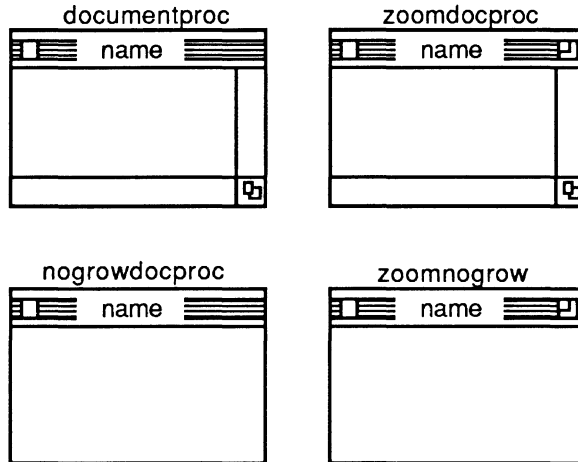




`rdocproc` is a round cornered frame with a title bar at the top containing the window name and optionally a close box. The title bar provides a convenient handle by which users can drag the window about, generating a `userdrag` event when the mouse button is pressed in it.



The four remaining types of frames are usually used for documents. In addition to the title bar and close box, `documentproc` and `zoomdocproc` provide a grow box in the bottom right corner of the window. A mouse click on the grow box returns a `usergrow` event. Growing a window means increasing or decreasing the size of the window by dragging its bottom right corner.



`zoomdocproc` and `zoomnogrow` provide a zoom box at the right end of the title bar. A mouse click on the zoom box returns a `userzoom` event. Zooming means toggling the window position



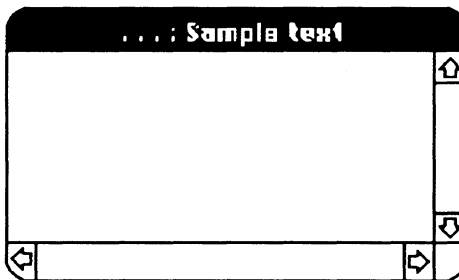
and size between those values set by the application or user and that of a full size window.

The events `userdrag`, `usergrow` and `userzoom` all have corresponding window predicates (`dragwindow`, `growwindow`, and `zoomwindow`) which are responsible for the dragging, growing and zooming of windows.

Two classes of windows are provided which differ in their uses and capabilities: graphics windows for graphics I/O and text windows for text I/O. Graphics windows can be used for any type of graphic I/O and provide limited higher level support. Text windows, on the other hand, may only be used for text I/O. They are explicitly connected to a text disk file, and actually represent a miniature text editing environment for that disk file. For example, the call

```
openwindow(text, 'Sample text', pos(0, _ty), size(_sw, _th),
           options(rdocproc, nomsgbutton))
```

creates a text window of the form



where the "...:" represents the path to the disk file.

The predicate `closewindow` is used to close an open window; its single argument is the window name, as in

```
closewindow('Sample text')
```



## Text Windows

Text windows are structured entities associated with text files, providing self contained text editing environments with high level routines to alter their structure and content. A text window is created by specifying the type `text` as an argument to the `openwindow` predicate. The window's name is the complete name of the associated disk file.

The content of a text window is organized both in terms of lines, and in terms of characters and the data is addressable in either format for the purposes of querying or altering it. The system maintains the window's contents and internally handles any mouse and update events for text windows.

In addition to the frame types discussed in the previous section, there are three other options that can be specified. These determine the presence or absence of vertical and horizontal scroll bars and the message button. The scroll bars allow users to quickly scroll through text vertically or horizontally when it does not all fit in the window. The message button displays a message and returns a `userkey` event when selected with the mouse. The message and the actual key returned in the user event can be both set and queried by using the `messagebutton` predicate.

### **`dotext`, `inqtext`**

The `dotext` predicate can provide text output, or set the attributes of the text window environment. Output descriptors are used to specify what to write, and the environment is set or queried through attribute descriptors. Multiple descriptor predicates may be included in a call to `dotext` by assembling them in a list of descriptors. (See the *BNR Prolog Reference Manual* for a complete description of available text output and attribute descriptors.)

Almost all text descriptors rely on the current text selection. The attribute descriptors set or query the text currently selected, and



the text output descriptors alter either the text actually selected, or the text at the position of the cursor. Relative text attribute descriptors operate with respect to the current character or line, and absolute text attribute descriptors selection operate with respect to the start of the text as a whole. Text descriptors that take or return text use either symbols or streams as the text source/target.

Text window attributes can be queried by a call to the `inqtext` predicate.

The following example uses `dotext` and `inqtext` to find and select the nearest query to the left of the current selection position. A query is assumed to be a sequence of characters beginning with "?-". The `select_query` predicate first searches for the prefix "?-" to the left of the current selection position in `_Window` and then sets the direction of search forward again. The `inqtext` goal determines the absolute character position of the prefix so that, once the end of the query is found with `find`, the entire query can be selected with `dotext`.

```
select_query(_Window) :-
    dotext(_Window, [scandirection(backward),
                     selection('?-'),
                     scandirection(forward)]),
    inqtext(_Window, selectcabs(_Bq, _)),
    once(find(_Window, '.', [' ', '\n', ''], _Eq)),
    dotext(_Window, selectcabs(_Bq, _Eq)).
```

A valid query must end with a period and be followed by either a space, a new line, or the end of the file. The main predicate, `find`, can be written as a general routine to search for instances of the character `_Char` that are followed by one of the characters in `_List`, and return the character's window position in `_EndPos`. If `_Char` is not found, `find` fails.



```
find(_Window, _Terminal, _List, _Next) :-
    repeat,
        (dotext(_Window, selection(_Terminal)) ->
            [inqtext(_Window, selectcabs(_, _EndPos)),
              _Next is _EndPos + 1,
              dotext(_Window, selectcabs(_EndPos, _Next)),
              inqtext(_Window, selection(_Char)),
              once(member(_Char, _List))
            ] ;
        failexit(find)).
```

## Handling Keyboard Events

One of the principal event handlers for text windows is `userkey`. For example, `select_query` could be made a menu selection or a control key sequence by adding the following clause to `userkey`:

```
userkey(_Window, '/', [1, 0, 0, 0, 0, 1]) :-
    once(select_query(_Window)).
```

This clause makes *control-/* the hot key for this new user interface feature. If *control-/* is to both select a query and enter it into the default input stream, the clause is

```
userkey(_Window, '/', [1, 0, 0, 0, 0, 1]) :-
    once(select_query(_Window)),
    inqtext(_Window, selection(0)).
```

The `userkey` event handler can also be used for other purposes. For example, to prevent the corruption of data in a specific text window, typing in that window can be disabled the presence of a clause for `userkey` that does nothing. To restrict the action of this handler to a particular text window it may be necessary to use `fullfilename` to match the file name obtained by the call to `userevent`. This technique may be used wherever there is a need to trigger on a text window name.

```
/* event handler to disable typing only in the text
window 'Sample text'*/

userkey(_window, _, _) :-
    fullfilename('Sample text', _window).
```



The next example uses a variation of this technique in combination with the `select_query` predicate defined above to change the the development environment so that queries can be entered with the *return* key. The `userkey` event handler checks that the *return* key was pressed in the Console (stream 1) and preceded by a period. Then it selects the query with `select_query` and submits it to the default output stream (stream 0). Otherwise the cursor is returned to its original position `Pos` and this clause for `userkey` is failed thus backtracking to the default `userkey` handler.

```
userkey(Console, '\n', _) :-
    stream(1, Console, _),
    inqtext(Console, selectcabs(_, Pos)),
    Prev is Pos - 1,
    cr_aux(Console, Prev, Pos).

cr_aux(Console, Prev, Pos) :-
    dotext(Console, selectcabs(Prev, Pos)),
    inqtext(Console, selection('.')),
    select_query(Console),
    inqtext(Console, selection(0)) .

cr_aux(Console, Prev, Pos) :-
    dotext(Console, selectcabs(Pos, Pos)),
    fail.
```

## Graphics Windows

A graphics window is created by specifying the type `graf` as an argument to the `openwindow` predicate. The inside of a graphics window is a cartesian plane with addressable points ranging from -32768 to 32767 along each coordinate axis. The window actually only displays the (+x, +y) quadrant with the origin at the top left corner of the window and coordinates increasing positively to the right and down.

Graphics windows provide basic self contained drawing environments, with low level routines for the manipulation of their contents, but little high level support. Applications are responsible for remembering the contents of these windows since no internal record is kept. This includes mouse movement as



well as content changes. Mouse activity is reported by means of `usermousedown` and `usermouseup` events from the active window.

When a portion of a window is freshly exposed (either when the window is first opened or when the windows on the screen are rearranged) the system reports a `userupdate` event indicating that a portion of the window is now blank and needs to be redrawn. The creator of the window must provide a `userupdate` handler to redraw the window as required. While this event is being processed, the system restricts any drawing to the affected area (the clipping region) to enhance update performance.

The `dograf` predicate can set the attributes of the graphics environment, or provide graphic output. Output descriptors are used to specify what to draw, and the attributes of the graphics environment are specified with attribute descriptors. Multiple descriptor predicates may be included in a call to `dograf` by assembling them into a list. These lists may also be nested, which essentially restricts the scope of any attribute alterations to the nesting level at which they occur. This makes it possible to localize the changes in the attributes of a graphics window without querying, remembering and then resetting the current attributes.

Coordinate specifications to output descriptors may be either relative to the current drawing position, or absolute with respect to the window origin. Coordinates values can be specified either as  $x, y$  pairs of integers or floats, or as intervals specifying  $x$  and  $y$  ranges. With intervals, if coordinate points are required the interval midpoints are used, and if rectangle edges are required the interval limits are used.

Querying of environment attributes is done by means of the `inqgraf` predicate.

The following demonstrates the use of some graphics predicates to clear, redraw and add to a drawing. These predicates are used for rubber lines in the next section.



`cleardrawing` deletes the state space records into which drawing data is remembered and then clears the window by drawing a rectangle the size of the window and filling it with the background pattern. Note that it uses double nested brackets for the graphics structure so as to make the fill pattern change only temporary.

```
/* implement basic drawing functions */
cleardrawing(_Window) :-
    forget_all(_Window(_..), $local),
    sizewindow(_Window, _w, _h),
    dograf(_Window, [[fillpat(clear), rectabs(0, 0, _w, _h)]]).
```

`redraw` backtracks through each item of data in the local state space and draws it in the graphics window.

```
redraw(_Window) :-
    foreach( recall(_Window(_data..), $local) do
        dograf(_Window, _data)).
```

`inrectangle` is a simple utility which succeeds if the specified point is within the specified rectangle.

```
inrectangle(_x, _y, [_xl, _yt, _xr, _yb]) :-
    _xl =<= _x, _x =<= _xr,
    _yt =<= _y, _y =<= _yb.
```

`addtodrawing` performs a verification and storage function. After verifying that the end point of the line is within the current window border, it draws the line and stores it in the state space. If the end point of the line is outside the window borders, `addtodrawing` beeps at the user.



```
addtodrawing(_Window, _x1, _y1, _x2, _y2) :-
    sizewindow(_Window, _w, _h),
    (inrectangle(_x2, _y2, [0, 0, _w, _h]) ->
        [_data = [moveabs(_x1, _y1),
                    lineabs(_x2, _y2),
                    circleabs(_x1, _y1, 1.5),
                    circleabs(_x2, _y2, 1.5)],
         dograf(_Window, _data),
         remember(_Window(_data..), $local)
        ] ;
        beep).
```

## Rubber Lines

The interactive trick of connecting the end of the mouse cursor to a fixed point on the screen with a line that grows, shrinks and moves about with the mouse is called rubber lining. This technique can be used to provide feedback for the user when drawing lines on the screen. The mechanics of rubber lining are demonstrated in the following program.

`startrubberlining` stores the initial coordinates of the line and draws the first line simply as a point.

```
startrubberlining(_Window, _x1, _y1) :-
    remember(_Window(rubber, _x1, _y1, _x1, _y1), $local),
    dograf(_Window, [penpat(gray), penmode(xor),
                    moveabs(_x1, _y1), lineabs(_x1, _y1)]).
```

`continuerubberlining` takes a new coordinate position and moves the rubber line from its previous location to the new location. If the new location is the same as the old location the drawing code is not executed. Omission of this causes a line which constantly flickers when the mouse isn't moving.



```

continuerubberlining(_Window, _xnew, _ynew) :-
    update(_Window(rubber, _x1, _y1, _xold, _yold),
           _Window(rubber, _x1, _y1, _xnew, _ynew),
           $local),
    not([_xold = _xnew, _yold = _ynew]),
    dograf(_Window, [moveabs(_x1, _y1), lineabs(_xold, _yold)]),
    dograf(_Window, [moveabs(_x1, _y1), lineabs(_xnew, _ynew)]).

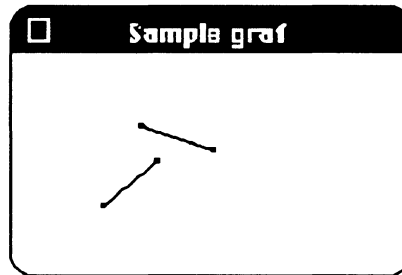
```

`stoprubberlining` cleans up the state space and erases the last rubber line.

```

stoprubberlining(_Window, _x1, _y1) :-
    forget(_Window(rubber, _x1, _y1, _xold, _yold), $local),
    !,
    dograf(_Window, [moveabs(_x1, _y1), lineabs(_xold, _yold),
                    penmode(or), penpat(black)]).

```



Graphics Window with Rubber Lines

The basic idea of rubber lines is to draw and undraw lines rapidly between a fixed point and the position of the mouse cursor. In this example, the drawing and undrawing of the line are implemented by using the `xor` drawing mode, which inverts all screen pixels in the drawing path. The first time a line is drawn it is visible. If another line is drawn in exactly the same place, the first one disappears.

## Handling Mouse Events

In the following example, event handlers for the mouse are used to drive the rubber line program.



Pressing the mouse in the graphics window starts a new line segment and in response to a `usermousedown`, rubber lining begins. When the user releases the mouse button, a `usermouseup` event signals to stop rubber lining. If the user has defined a valid line segment then it is added to the drawing.

```
/* event handlers for mouse clicking in the graphics
window 'Sample graf'*/

usermousedown('Sample graf', _x1, _y1) :-
    startrubberlining('Sample graf', _x1, _y1).

usermouseup('Sample graf', _x2, _y2) :-
    stoprubberlining('Sample graf', _x1, _y1),
    addtodrawing('Sample graf', _x1, _y1, _x2, _y2).

userdownidle('Sample graf', _xnew, _ynew) :-
    continuerubberlining('Sample graf', _xnew, _ynew).

userupidle('Sample graf', _x2, _y2) :-
    stoprubberlining('Sample graf', _x1, _y1),
    addtodrawing('Sample graf', _x1, _y1, _x2, _y2).
```

While the user is holding the mouse button down `userdownidle` events are received. Therefore, rubber lining is continued. Very occasionally, the system seems to lose a mouse up event. Just in case this happens, the mouse up idler tries to stop rubber lining. If a mouse up event is missed and rubber lining continues, the predicate `userupidle` stops it and saves the resultant line segment in the drawing.

The following handler is invoked when the sample listener detects that the mouse has been clicked on the close box of the 'Sample graf' window. This closes the window, deletes the menus (discussed in the section on Menus below) and exits from `samplelistener`.

```
userclose('Sample graf', _, _) :-
    cleanup,
    failexit(samplelistener).

cleanup :-
    closewindow('Sample graf'),
    stoppedmenus.
```



The response to a `userupdate` event is to redraw the window contents.

```
/* event handler for the graf update event */  
userupdate(_Window, _, _) :- redraw(_Window).
```

## Pictures

Pictures are the mechanism by which graphic information is transferred between BNR Prolog and other Macintosh applications (for example, MacDraw, MacWrite). Pictures may be stored in files, in which case they are saved as Macintosh resources of type `PICT` in the resource fork of the specified file. Pictures may also be stored in the clipboard scrap. Any picture may copied to the scrap and any data in the scrap may be loaded in the form of a picture. If the scrap currently contains text, the text is transformed into a picture before it is loaded.

In this discussion, the term picture refers to a special data structure consisting of a sequence of low level graphic drawing and attribute commands. Once created, a picture cannot be disassembled into individual graphics descriptors again. Any drawing created using the `dograf` predicate may be transformed into a picture. Once created, this structure may be saved to and loaded from a file, displayed in a graphics window, copied to and from the clipboard, and of course deleted.

A picture is created by calling `beginpicture`, performing one or more `dograf` calls and then calling `endpicture`. `beginpicture` expects the specification of a picture frame as an input argument. A picture frame is a logical boundary which, ideally, should surround the picture's contents. The frame specification is used later when displaying the picture to control the position and scale at which the picture is drawn.

`dograf` supports a picture graphics descriptor which also requires a frame specification. The originally specified frame and all of the picture's contents are mapped to this second frame when the picture is drawn. When a picture is created it is given



an identification number by the system which is returned as an output argument from `beginpicture`.

If a picture is to be stored in a file, the resource number and optionally a resource name may be specified in the `savepicture` predicate. Pictures that are stored in the Clipboard scrap can be distinguished from text by using the `scrapcontents` predicate.

The following code saves a drawing in a graphics window on the Clipboard for import to other applications. First the drawing is converted into a picture by calling `beginpicture`, telling it which window to monitor, drawing the picture and then calling `endpicture`. Then a call to `picttoscrap` transfers a copy of the picture, and the original can be deleted since it is not needed any more.

```
clip(_Window) :-  
    sizewindow(_Window, _w, _h),  
    beginpicture(_Window, frame(0, 0, _w, _h), _pictid),  
    redrawdrawing,  
    endpicture(_pictid),  
    picttoscrap(_pictid),  
    deletepicture(_pictid).
```

Pictures can be attached to windows (1 picture per window) using `attachpicture`. Once a window has a picture attached to it, the Macintosh operating system assumes responsibility for updating the window: no `userupdate` events are by the Prolog system. Pictures can be detached from windows using `detachpicture`.

## Menus

Second only to windows, menus are a fundamental user interface mechanism on the Macintosh. They provide a simple command interface, which, with the appropriate addition of command key equivalents appeal to users of both the mouse and the keyboard. In addition, they enforce a logical grouping of items and supply an online guide to the available and active application capabilities.



The menu bar at the top of the screen is the standard place for menus. Special menu variations called hierarchical and pop-up menus can appear elsewhere. (See the *BNR Prolog Reference Manual* for more information.)

Each installed menu must have a unique integer identification number. In addition, each item in a menu is individually numbered from the top down, starting from 1. Menu selection information is available either in the form of names or identification numbers. Although the `menuselect` event returns names, all menu predicates expect numbers.

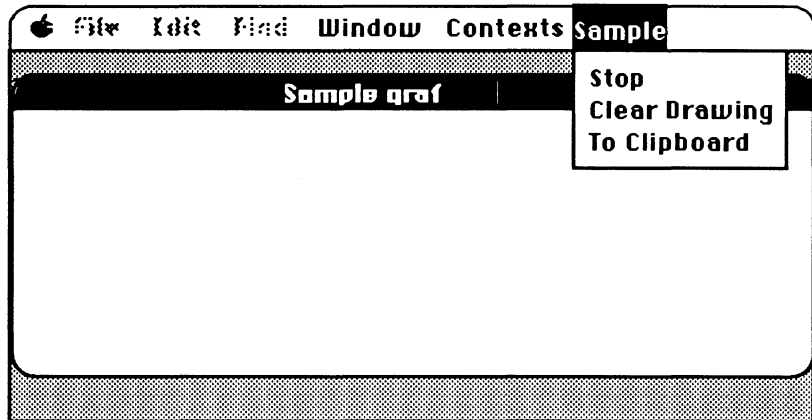
The creation, deletion, item content and attributes of menus are all under the application's control. The system handles the visual representation, viewing, selecting and event notification of user menu selections.

Menus may be created as Macintosh resources and installed at run time. It is also possible to create, install, and modify menus dynamically. The menus presented by the development environment are a mixture of these types. The system's **File**, **Edit**, **Find** and **Windows** menus are all resources that are available for use by applications. The **Contexts** menu, on the other hand, is created in Prolog.

Whenever menus or menu items are added or deleted, the item position is specified in terms of menu and item identifiers. This allows the placement to be fully controlled. The decoding of menu selections is largely independent of menu and item ordering since in general decoding is based on names rather than identifiers.

The following example creates a single menu in the menu bar. When the application is loaded but not running the menu contains one item: **Start**. While running it contains three items: **Stop**, **Clear Drawing** and **To Clipboard**. As seen in the figure that follows, to properly indicate that certain of the development environment's menus are not always relevant, some are disabled and enabled according to the ownership of the frontmost window position by monitoring activate events.





`installmenus` puts up a menu **Sample** at the right end of the list of existing menus and gives it a menu identifier of 100. It then installs the single item **Start** into this menu.

```
installmenus :-  
    addmenu(100, 'Sample', 0),  
    additem('Start', ' '), 100, end_of_menu, _id1).
```

Once **Start** has been selected, the application begins to run, changes the existing first item, **Start**, to **Stop**, and then adds the two new items at the end of the menu.

```
runningmenus :-  
    menuitem(100, 1, 'Stop', ' '),  
    additem('Clear Drawing', ' '), 100, end_of_menu, _id2),  
    additem('To Clipboard', ' '), 100, end_of_menu, _id3).
```

When the program stops running, the first item is changed back to **Start** and the last two items are deleted. A call to `enablemenus` is included here because when the windows are deleted at program completion, there is no final `userdeactivate` event to trigger the enabling.



```
stoppedmenus :-  
    menuitem(100, 1, 'Start', ''),  
    deleteitem(100, 3),  
    deleteitem(100, 2),  
    enablemenus.
```

The menus that are enabled and disabled are the **File**, **Edit** and **Find** system menus. Although their menu identifiers are unknown, they can be referenced by predefined symbols.

```
enablemenus :-  
    menuitem('File', 0, _, ''),  
    menuitem('Edit', 0, _, ''),  
    menuitem('Find', 0, _, '').  
  
disablemenus :-  
    menuitem('File', 0, _, '('),  
    menuitem('Edit', 0, _, '('),  
    menuitem('Find', 0, _, '(').
```

If the window 'Sample graf' defined by the program is active, the system menus should be disabled.

```
/* event handlers for the activation and deactivation  
of our windows */  
  
useractivate('Sample graf', _, _):- disablemenus.  
userdeactivate('Sample graf', _, _):- enablemenus.
```

## Menu Event Handlers

When the user picks the **Start** menu item, the application is started by calling `sample`. This sets up the windows, adjusts the menus, and enter the program's listener loop.

When **Stop** is picked, a call is made to `cleanup` which failexits out of the listener loop, readjusts the menus, and closes the program's windows.

The menu item **Clear Drawing** provides access to the `cleardrawing` function.



The handler for **To Clipboard** simply calls the clip predicate defined in the section on pictures.

```
/* menu handlers for Sample menu */
menuselect(_, 'Sample', 'Start') :- sample.
menuselect(_, 'Sample', 'Stop') :-
    !,
    cleanup,
    failexit(samplelistener).
menuselect(_, 'Sample', 'Clear Drawing') :-
    cleardrawing('Sample graf').
menuselect(_, 'Sample', 'To Clipboard') :-
    clip('Sample graf').
```

## Dialogs

Dialogs present the user with a preconfigured interface that solicits specific input information. A modest set of modal dialogs are provided. These dialogs, which require a response before activity resumes, are presented when some form of user input or confirmation is required before a particular activity can continue. At times the desired response may be simply to acknowledge the dialog.

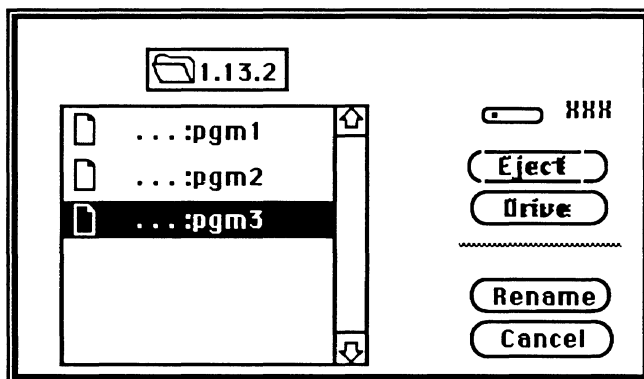
The system provides the following types of dialogs:

- |                        |  |
|------------------------|--|
| message                | to notify users of important information or status                             |
| query                  | to solicit the answer to a question via a user typed response                  |
| confirm                | to obtain a user's agreement to perform a certain action                       |
| select, selectone      | to acquire a user's choices from a selection of alternative items              |
| selectafile, nameafile | to let a user specify a filename and directory location for a filing operation |

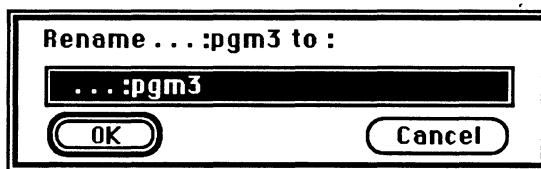


Dialogs are prepackaged pieces of code which contain all of the software to control a particular window. To some extent many of the capabilities of dialogs may be programmed in Prolog using graphics windows and the `dograf` and `inqgraf` predicates. Such typical dialog mechanisms as labeled buttons, radio buttons, checkboxes, static text, graphics fields and the like, may be programmed using the existing window and graphics facilities. Text editing fields and scrolling fields, however, are more difficult to program.

The **Rename** item in the **File** menu uses two dialogs, calling `selectafile` to get the full path name of the file to be renamed,



and `query` to get the file's new name.





The code for the event handler for this menu selection is:

```
menuselect(_frontwindow, 'File', 'Rename...') :-  
    ! ,  
    selectafile('', 'Rename', _name) ,  
    swrite(_prompt, 'Rename ', _name, ' to :') ,  
    query(_prompt, _name, _newname) ,  
    (renamefile(_name, _newname) ->  
        write('Renamed file ', _name, ' to ', _newname, '\n') ;  
        [beep,  
            write('Unable to rename ', _name, ' to ',  
                _newname, '\n')  
        ]  
    ).
```

Many more examples using the selection and file specification dialogs can be viewed by listing the `menuselect` handlers for the **File** and **Contexts** menus in the development environment's base context.

Applications in need of additional dialogs can write their own and access them by means of the external language interface.

## A Complete Program

### Program sample

The various pieces of the program `sample` that have been developed throughout the last few sections are collected and presented here in their entirety.

`sample` is a simple program that uses mouse and menu interface techniques to enable a user to draw graphics shapes in a window. The basics of window manipulation, graphics I/O, event handling (including menus), and the exporting of graphics pictures are presented. Note the strong bias toward a procedural interpretation of the code due to its reliance on side effects.

For demonstration purposes, `sample` provides its own main listener loop, although it works equally well with the system



listener. The listener loop, like the one presented previously, allows `sample` to do everything except perform the parsing of text which is handled by the system listener. The `$initialization` clause automatically sets up menus when `sample` is loaded. The **Sample** menu then appears and the user can select the **Start** item to start the program.

Some suggested user modifications for experimentation are:

- 1) Comment out the program's listener to observe how it continues to run driven from the system listener. Note that the program appears to run in parallel with the development environment. Queries can be executed by the system listener without affecting the `sample` program.
- 2) Add a Pick capability. This refers to the ability to point at an existing shape in the drawing with the mouse and to locate that shape in the database for editing purposes. This is primarily a matter of matching the approximate coordinates of the mouse click with the shape whose coordinates come closest to it in the database.
- 3) Add the capability to draw more than just one shape. Add shapes like circles, rectangles, and ovals. You'll also have to modify the rubber line techniques to accommodate these shapes.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This sample program illustrates the
% use of event loops, event handlers, menus,
% window management, and simple graf utilities.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      code to implement basic drawing functions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cleardrawing(_Window) :-
    forget_all(_Window(_.), $local),
    sizewindow(_Window, _w, _h),
    dograf(_Window, [[fillpat(clear), rectabs(0, 0, _w, _h)]]).

redraw(_Window) :-
    foreach(recall(_Window(_data..), $local) do
        dograf(_Window, _data)).

inrectangle(_x, _y, [_x1, _yt, _xr, _yb]) :-
    _x1 =< _x, _x =< _xr,
    _yt =< _y, _y =< _yb.

addtodrawing(_Window, _x1, _y1, _x2, _y2) :-
    sizewindow(_Window, _w, _h),
    (inrectangle(_x2, _y2, [0, 0, _w, _h]) ->
        [_data = [moveabs(_x1, _y1),
                    lineabs(_x2, _y2),
                    circleabs(_x1, _y1, 1.5),
                    circleabs(_x2, _y2, 1.5)],
         dograf(_Window, _data),
         remember(_Window(_data..), $local)
        ] ;
        beep).
```



```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       code to implement rubber lines
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

startrubberlining(_Window, _xl, _yl) :-
    remember(_Window(rubber, _xl, _yl, _xl, _yl), $local),
    dograf(_Window, [penpat(gray), penmode(xor),
        moveabs(_xl, _yl), lineabs(_xl, _yl)]).

continuerubberlining(_Window, _xnew, _ynew) :-
    update(_Window(rubber, _xl, _yl, _xold, _yold),
        _Window(rubber, _xl, _yl, _xnew, _ynew),
        $local),
    not([_xold = _xnew, _yold = _ynew]),
    dograf(_Window, [moveabs(_xl, _yl), lineabs(_xold, _yold)]),
    dograf(_Window, [moveabs(_xl, _yl), lineabs(_xnew, _ynew)]).

stoprubberlining(_Window, _xl, _yl) :-
    forget(_Window(rubber, _xl, _yl, _xold, _yold),
        $local),
    !,
    dograf(_Window, [moveabs(_xl, _yl), lineabs(_xold, _yold),
        penmode(or), penpat(black)]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       code to export pictures to the clipboard
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clip(_Window):-
    sizewindow(_Window, _w, _h),
    beginpicture(_Window, frame(0, 0, _w, _h), _pictid),
    redraw(_Window),
    endpicture(_pictid),
    picttoscrap(_pictid),
    deletepicture(_pictid).

```



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           event handlers (including menus)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

userclose('Sample graf', _, _) :-
    cleanup,
    failexit(samplelistener).

userupdate(_Window, _, _) :- redraw(_Window).

useractivate('Sample graf', _, _):- disablemenus.

userdeactivate('Sample graf', _, _):- enablemenus.

usermousedown('Sample graf', _x1, _y1) :-
    startrubberlining('Sample graf', _x1, _y1).

usermouseup('Sample graf', _x2, _y2) :-
    stoprubberlining('Sample graf', _x1, _y1),
    addtodrawing('Sample graf', _x1, _y1, _x2, _y2).

userdownidle('Sample graf', _xnew, _ynew) :-
    continuerubberlining('Sample graf', _xnew, _ynew).

userupidle('Sample graf', _x2, _y2) :-
    stoprubberlining('Sample graf', _x1, _y1),
    addtodrawing('Sample graf', _x1, _y1, _x2, _y2).

menuselect(_, 'Sample', 'Start') :- sample.

menuselect(_, 'Sample', 'Stop') :-
    !,
    cleanup,
    failexit(samplelistener).

menuselect(_, 'Sample', 'Clear Drawing') :-
    cleardrawing('Sample graf').

menuselect(_, 'Sample', 'To Clipboard') :-
    clip('Sample graf').
```



```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               setup code
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
enablemenus :-
    menuitem('File', 0, _, ' '),
    menuitem('Edit', 0, _, ' '),
    menuitem('Find', 0, _, ' ').

disablemenus :-
    menuitem('File', 0, _, ' '),
    menuitem('Edit', 0, _, ' '),
    menuitem('Find', 0, _, ' ').

setupwindows :-
    scrndimensions(_sw, _sh),
    _gy is 2 * _sh // 3,
    _gh is (_sh - _gy),
    openwindow(graf, 'Sample graf', pos(0, _gy),
               size(_sw, _gh), options(rdocproc)).

runningmenus :-
    menuitem(100, 1, 'Stop', ' '),
    additem('Clear Drawing', ' '), 100, end_of_menu, _id2),
    additem('To Clipboard', ' '), 100, end_of_menu, _id3).

setuptorun :- setupwindows, runningmenus.

installmenus :-
    addmenu(100, 'Sample', 0),
    additem('Start', ' '), 100, end_of_menu, _id1).

```



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%                                cleanup code
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

stoppedmenus :-
    menuitem(100, 1, 'Start', ''),
    deleteitem(100, 3),
    deleteitem(100, 2),
    enablemenus.

cleanup :-
    closewindow('Sample graf'),
    stoppedmenus.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%                                listener loop
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

samplelistener :-
    repeat,
        usevent(_Event, _Window, _D1, _D2, noblock),
        once(_Event(_Window, _D1, _D2)),
    fail.

/* mainline */
sample :- setuptorun, samplelistener.

sample :- [].

$initialization :- installmenus.
```



# Part V

# Miscellaneous







# Chapter 17

## Foreign Language Interface

---

Procedures can be written in other languages, for example, Pascal or C, and defined as Prolog clauses. To provide the interface between languages, procedures must be compiled as code resources, and then linked as a resource of type PEXT (Prolog EXternal). Code resources are single pieces of self contained code for which the entry point is at the beginning. The *Macintosh Programmer's WorkBench* and the *Lightspeed* family of languages are examples of development environments which provide tools for creating code resources.

### Defining and Calling Externals

External procedures are defined to the Prolog system by using the `defexternal` predicate. This predicate causes a code resource to be loaded into memory and asserts a Prolog clause in the current context to call it. The code resource remains in memory until the context, in which it is defined, is removed.

The form of `defexternal` is:

```
defexternal(_Predicate_name(_Variables..),
           _Filename,
           _Segment_name,
           [<input variable names and types>],
           [<output variable names and types>])
```

`_Predicate_name(_Variables)` defines the clause head of the external procedure. `_Filename` is the pathname of the file containing the code resource of type PEXT named `_Segment_name`, which contains the code for the external procedure. If `_Filename` is specified as `""`, then the current applications file is searched for the specified resource. The input and output lists are used to define which arguments are inputs, which must be



instantiated, and which are outputs, which must be variables, as well as the expected type. The order in which they appear in the lists corresponds to the the external procedure order. Allowable types are integer, float, symbol, and bucket.

The following example loads the code resource `freemem` from the current application file and creates a clause named `free_memory` which unifies its arguments with two integer values, since the arguments are defined as outputs.

```
?- defexternal(free_memory(_Largest, _Total),  
              '',  
              'freemem',  
              [],  
              [_Total : integer, _Largest : integer]).
```

Each call to `defexternal` results in one clause for the specified predicate. Multiple calls to `defexternal` can be made for the same predicate name. Thus, a predicate definition may be composed of a mixture of externally defined clauses and clauses written in Prolog. For calling and backtracking purposes, the clause order is defined by the assert order; externals are asserted with `assertz` semantics.

The call

```
?- free_memory(_LargestFree, _TotalFree).  
   ?- free_memory(132088, 143740).  
YES
```

unifies `_LargestFree` and `_TotalFree` to the first and second values returned by `free_memory` respectively.

## Writing an External Procedure

External procedures must be code resources, which are single pieces of self contained code for which the entry point is at the beginning. When called, each external procedure is passed a pointer to a structure that contains the result, a user definable handle, the input parameters, and the output parameters.



External procedures have a single parameter, the pointer to the structure. In Pascal, the procedure header has the form:

```
PROCEDURE Proc(VAR P : StackFrame);
```

while in C it has the form:

```
void Proc(P)

struct StackFrame *P;
```

The structure passed to the procedure for `free_memory` described above is as follows (Pascal interface):

```
StackFrame = RECORD
    Result      : LONGINT;
    Reserved    : LONGINT;
    UserHandle  : Handle;
    TotalFree   : LONGINT;
    LargestFree : LONGINT;
END;
```

The value of `Result` is initially 0, which indicates a failure. A non-zero result indicates a successful call. `Reserved` is a field saved for future use. The value of `UserHandle`, initially `NIL`, may be updated by the external procedure. It is maintained by the system between calls, and passed to the procedure with every call. The user handle can be used by a procedure to maintain its "global" data. When the code resource containing the procedure is removed, the user handle is freed.

External procedures need not be concerned with popping parameters off the stack on return.

## Parameter Interface

In an external procedure called from a Prolog program, all parameters are passed using 32-bit values. Parameters of type `integer` or `bucket` are of type `longint` in Pascal, or type `long` in C. However, integers must be representable in 29 bits including the sign, which restricts values to the range -268435456 and 268435455. A `float` is passed as a pointer to a `SANE`



extended floating point value, represented as `^extended` in Pascal, or `*extended` in C. Internally, floats are represented as a 20-bit mantissa with an 8-bit exponent, which may mean a loss of accuracy when results are passed around. A `symbol` is passed as a pointer to a Pascal string represented as `^string[255]`, or as a pointer to an array of 256 characters, `*char[256]`, in C.

In the case of both input and output parameters, the space for all strings and floating point numbers is allocated before calling the procedure.

## Restrictions

External procedures cannot declare global data. Any data that is intended as global should be defined and manipulated through the user handle in the stack frame. The user handle must be used as a handle. Handles may be passed between externals by using buckets.

Care should be taken in manipulating resources which are managed by the Prolog system, for example windows and menus. As well, pointer parameters should not be modified. Such action may leave unrecoverable space in memory.

## Pascal Examples

### QueryMem

The following is an example of a Pascal unit containing a procedure that obtains the total free memory and the largest block of free memory.



```
{ $$ freemem } { segment name }

UNIT QueryUnit;

INTERFACE

USES                                { if in MPW }

    Memtypes, Quickdraw, Osintf;

TYPE Stackframe = RECORD
    Result :      LONGINT;
    Reserved :    LONGINT;
    Userhandle :  Handle;    { predefined MPW type }
    TotalFree :   LONGINT;
    LargestFree : LONGINT;
END;

PROCEDURE QueryMem(VAR p : Stackframe);

IMPLEMENTATION

    PROCEDURE QueryMem(VAR p : Stackframe);

        VAR
            OldZone :      THz;
            Grow :        LONGINT;

        BEGIN
            { save current zone for restore later }
            OldZone := GetZone;
            { check application zone }
            SetZone(ApplicZone);
            { get total and largest free memory }
            p.TotalFree := FreeMem;
            p.LargestFree := MaxMem(Grow);
            { return successful result and restore zone to
              what it was before }
            p.Result := 1;
            SetZone(OldZone);
        END; { QueryMem }

    END.    { QueryUnit }
```



Assuming the source is in a file called `FreeMem.p`, the following commands should be executed to create a code resource using *Macintosh Programmer's WorkBench*.

```
Pascal FreeMem.p
```

to compile the unit, and

```
Link -o MemOut -rt PEXT=234 -sg freemem -m QUERYMEM  
FreeMem.p.o "{Libraries}"Interface.o
```

to link the unit, where

```
-o MemOut           specifies the name of linked file  
-rt PEXT=234       specifies the resource type and id (id is  
                   not used )  
-sg freemem        specifies the name of the segment and  
                   resource  
-m QUERYMEM        specifies the module is to contain this  
                   procedure, and any procedures it calls  
FreeMem.p.o "{Libraries}"Interface.o  
                   specifies the object files to be linked,  
                   that is the compiled output and the  
                   library files
```

To build and save the unit using *Lightspeed Pascal*, specify that the file is a code resource of type `PEXT`, with any resource number (code resources are located by name), and a name `freemem`.

To define `QueryMem` as a clause specify the following

```
?- defexternal(queryMem(_Largest, _Total),  
               'MemOut', 'freemem', [],  
               [_Total : integer, _Largest : integer]).
```

An example of a query using `freemem` is:

```
?- queryMem(_L, _T).
```



## Count

The procedure `Count` returns an integer indicating how many times it has been executed, beginning at 1 and adding 1 to the count for every call.

```
{ $S countseg }           { segment name }

{ * ***** Counter.p ***** * }
UNIT CountUnit;

INTERFACE

USES                        { if in MPW }
    Memtypes, Quickdraw, Osintf;

TYPE

    Space = RECORD
        CurrentCount : LONGINT;
    END;

    SpacePtr =           ^Space;
    SpaceHandle =        ^SpacePtr;

    StackFrame = RECORD
        Result :          LONGINT;
        Reserved :        LONGINT;
        UserHandle :      SpaceHandle;
        Value :           LONGINT;
    END;

{ define external availability of procedure Count }
PROCEDURE Count (VAR SF : StackFrame);
```



## IMPLEMENTATION

```
{ declaration of procedure Count }
PROCEDURE Count(VAR SF : StackFrame);
BEGIN
    { note use of ^^, as UserHandle is ptr to ptr }
    { if the first call then allocate handle
      and initialize count to 0 }
    IF (SF.UserHandle = NIL)
    THEN BEGIN
        SF.UserHandle :=
            SpaceHandle(NewHandle(sizeof(Space)));
        SF.UserHandle^^.CurrentCount := 0;
    END { IF };
    { increment count }
    SF.UserHandle^^.CurrentCount :=
        SF.UserHandle^^.CurrentCount + 1;
    { return current count value }
    SF.Value := SF.UserHandle^^.CurrentCount;
    { return a successful result value }
    SF.Result := 1;
    END { Count };
END. { CountUnit }
```

Using the *Macintosh Programmer's WorkBench* to compile and link the unit `CountUnit` in a file `Counter.p`, specify

Pascal Counter.p

to compile the unit

```
Link -o CountOut -rt PEXT=235 -sg countseg -m COUNT
Counter p o "{Libraries}"Interface.o
```

to link the unit. To build and save the unit using *Lightspeed Pascal*, specify that the file is a code resource of type PEXT, with any resource number, and a name `Counter`.

To define counter, specify the following

```
?- defexternal(counter(_Count), 'CountOut', 'countseg',
               [], [_Count : integer]).
```



then call it as follows:

```
?- counter(_X).  
    ?- counter(1).  
YES  
?- counter(_X).  
    ?- counter(2).  
YES
```

## Combine

This example shows how to manipulate strings, by taking two strings and combining them into a third. Note that although the input and output parameters can be in any order, the parameters passed to the procedure are in the order specified by the input and output parameter lists, with the inputs coming first.

```
{ $S segComb } { segment name }  
  
UNIT Comb;  
  
INTERFACE  
  
USES                                     { if in MPW }  
  
    Memtypes, Quickdraw, Osintf;  
  
TYPE  
  
    StrPtr = ^Str255;                  { Str255 = STRING[255]; }  
  
    Stackframe = RECORD  
        Result :          LONGINT;  
        Reserved :        LONGINT;  
        Userhandle :      Handle; { predefined MPW type }  
        In1 :             StrPtr;  
        In2 :             StrPtr;  
        Out :             StrPtr; { combined string }  
    END;  
  
    { define external availability of procedure }  
PROCEDURE Combine(VAR SF : Stackframe);
```



## IMPLEMENTATION

```
PROCEDURE Combine(VAR SF : Stackframe);  
  
BEGIN  
    { will the result be less than 255 chars ? }  
    IF (LENGTH(SF.In1^) + LENGTH(SF.In2^)) <= 255  
    THEN BEGIN  
        { combine the strings, return success }  
        SF.Out^ := CONCAT(SF.In1^, SF.In2^);  
        SF.Result := 1;  
    END  
    ELSE { string too big, fail }  
        SF.Result := 0;  
    END; { Combine }  
  
END.      { Comb }
```

To create the code resource if the source is in the file `Combine.p`, use the following commands in *Macintosh Programmer's WorkBench*.

```
Pascal Combine.p
```

```
Link -o OutComb -rt PEXT=236 -sg segComb -m COMBINE  
Combine.p.o "{PLibraries}"PasLib.o
```

To build and save this unit using *Lightspeed Pascal*, specify that the file is a code resource of type PEXT, with any resource number, and a name `Combine`.

Once created, this file can be used by specifying the following:

```
?- defexternal(combine(_Out, _A, _B),  
    'OutComb',  
    'segComb',  
    [_A : symbol, _B : symbol],  
    [_Out : symbol]).
```

An example using the code resource is

```
?- combine(_X, 'The little ', 'brown fox').
```



which returns "The little brown fox". To demonstrate that input order is based on the parameter lists and not the order of the clause head:

```
?- defexternal(combine(_Out, _A, _B),
  'OutComb',
  'segComb',
  [_B : symbol, _A : symbol],
  [_Out : symbol]).
```

In this case, the same query returns "brown foxThe little ". However, if both the first and the second definition exist at the same time, then there are two possible answers to the query, and backtracking causes both to be displayed.

In the following definition and query, the ordering of the external procedure's parameters has been changed. Note also that the name of the procedure has been altered. This has no impact on the Pascal resource, since the name is only meaningful within the context of the Prolog program.

```
?- defexternal(combined(_A, _B, _Out),
  'OutComb',
  'segComb',
  [_A : symbol, _B : symbol],
  [_Out : symbol]).

?- combined('The little ', 'brown fox', _X).
```

## C Examples

Beep is a procedure in C which beeps for the duration specified.

```
/* ***** Beeper.c ***** */

#include <types.h>
#include <memory.h>
#include <osutils.h>

#define TRUE 1
#define FALSE 0
```



```
/* type stackframe */
struct stackframe
{
    long Result;
    long Reserved;
    long MyHandle;
    long Duration;
};

/* procedure Beep */
void Beep(p)
struct stackframe *p;
{
    if (p->Duration > 0)
    {
        SysBeep(p->Duration);
        p->Result = TRUE;
    }
    else
        p->Result = FALSE;
}
```

To build the code resource Beep, the following compile and link commands are necessary under the *Macintosh Programmer's WorkBench*

C Beeper.c

to compile the unit, and

```
Link -o BeepOut -rt PEXT=100 -m Beep -sn Main=BeepSeg
Beeper.c.o
```

to link the unit, where

-o BeepOut	specifies the output file
-rt PEXT=100	specifies the resource type and id although resource id is not used
-m Beep	specifies to only use function Beep and any procedures it calls
-sn Main=BeepSeg	rename segment as BeepSeg
Beeper.c.o	specifies the object file name



To define `myBeep`, specify the following

```
?- defexternal(myBeep(_In), 'BeepOut', 'BeepSeg',  
               [_In : integer], []).
```

and to use it:

```
?- myBeep(10).
```







# Chapter 18

## System Information

---

For the most part, BNR Prolog users need not be concerned with the internal structure of the system. However, serious developers are often concerned fine tuning applications to minimize execution time and memory requirements, and packaging applications for distribution. These activities require additional insight into the internal structuring of memory and the use of performance monitoring predicates.

Derived applications are structurally identical to the BNR Prolog development system, with the exception of a small number of development facilities that have been removed. These include the ability to create the binary forms of contexts, to save work space files, and to insert spy points on predicate definitions. In their place are the Prolog application code and additional resources such as externals, menus, and icons, that implement the particular application. Most of the discussion that follows applies to both the development environment and derived applications. Any differences are explicitly noted.

### Application Structure

All Macintosh applications divide the available memory into an execution stack, and a heap which contains dynamically allocated blocks of memory. On startup, BNR Prolog applications preallocate three chunks of memory for the world, global and local stacks. (The use of these stacks is described below.) The remainder of the heap holds other runtime data structures such as control blocks associated with files and windows, state spaces, and such resources as code, pictures, icons, and menus. Many of the predefined predicates are written in Prolog; these are contained in the base context of the world stack.



Preallocation of the three stacks permits these memory areas to be managed in an optimal fashion according to their function, as well as guaranteeing a minimum amount of memory for each function. When the stacks overflow, the current goal execution is aborted. Automatic extension of these stacks is not attempted.

The *world stack* holds all clause definitions and is structured as a stack of named contexts. The lowest context is *base*, which contains the system predicates written in Prolog. The first user accessible context is *userbase*, followed by any other loaded and dynamically created contexts. (The **Contexts** menu provides feedback on the current state of the context stack.) The world stack must be preconfigured to hold the base context as well as any user defined contexts required at any given time.

The *global stack* holds the state of the current computation. Call activation records and variables are kept in the execution space. (In many ways this is analogous to the execution stack of a Pascal or C program.) The size of this stack is dependent on the execution behavior of the application.

The *local stack* holds the list of choice points used on backtracking, and provides temporary storage for unification, copying, parsing, printing and miscellaneous predicates. The size requirements are also dependent on the execution behavior of the application, but are usually smaller than the *global stack*.

The default amount of memory required to run BNR Prolog under MultiFinder is set to 1 Mbyte. (The default memory allocation may be found using **Get Info** in the finder's **File** menu.) BNR Prolog configures itself to run with less memory on smaller systems.

### **configuration**

The `configuration` predicate is used to set or query the initial goal, as well as the allocation sizes of the three stacks. These size values, specified in Kbytes, are saved in the initial configuration information for use when an application is launched or restarted; `configuration` does not change the



current allocations. If a size value is 0 (specifying use of a default value) or if a value is inappropriate (too large or too small), space is allocated from available memory using a default minimum size and a percentage of available memory. Failure to allocate the stacks causes the application to abort.

The configuration predicate also specifies the initial goal to be executed, saving the information with the initial configuration. The initial goal argument is a symbol containing a single Prolog term which is executed deterministically (as in :- \_initial\_goal.). Multiple goals can be collected in a list, for example

```
'[load_context(set_up), load_context(more_definitions)]'
```

The following query

```
?- configuration(0, 0, 20, _).
```

results in the sizes for the world stack and the global stack being based on the amount of memory available when the application is launched or restarted. The local stack is specified at 20 Kbytes, and the initial goal remains unchanged. (In this example, the current initial goal is unified with the anonymous variable.)

The purpose of the initial goal in derived applications is somewhat different from its purpose in the development environment. In the former the initial goal is the application goal; when the call to the goal returns, the application is terminated. In the development system, it is primarily used to permit the user to customize the environment by defining additional contexts to be loaded, additional files to open, and so on.

### **restart**

Execution of the `restart` predicate returns the environment to the state of the application immediately after launch. The three stacks are reallocated, and the initial goal is re-executed. All contexts are restored to their initial state and state spaces are



deallocated. Text windows stay open over a restart, but graphics windows are closed. `restart` is primarily used to put any configuration changes into effect, but is sometimes useful to return the development environment to a known state before continuing.

### **`quit, halt`**

To terminate execution of an application, the `quit` predicate (or its synonym `halt`) is used. Text windows are closed, with prompts to save the files whose contents differ from their windows.

## **State Spaces**

State spaces are allocated on the application heap by the program. Growth by incremental amounts is automatic. Although the memory internal to state spaces is dynamically recovered, a state space never shrinks in size. The `new_state` predicate can be used to explicitly control the allocation of memory to a state space. If a program requires a large state space, it is often more efficient to reserve the memory with a call to `new_state`, rather than to rely on incremental growth.

See the chapter "State Spaces" for more information on the use of state spaces.

## **Work Spaces**

During program development, it is often desirable to save an intermediate step in a development session, or to snap shot a problem for subsequent investigation. The state of the development environment is captured by using the `save_ws` predicate, which saves the binary format of the clause data base and the initial configuration data in a work space file. State spaces are not saved.

Opening a work space file in the finder launches the BNR Prolog application and loads the specified work space in preference to the one preserved in the application file. Only one work space



file should be opened; if multiple work space files are selected, the last work space in the list built by the Finder is used.

## Externals

An external is a clause whose the body is a code resource, written in an external language such as Pascal or C. To simplify the programming model for externals, arguments are constrained to be simple types (symbol, integer, float or bucket), and either input or output (but not both). External code resources, like other resources, are kept in the resource fork of either the application file or a separate user managed file.

See the chapter "Foreign Language Interface" for a complete description of externals.

## Monitoring the Environment

The environment monitoring predicates have several uses. Inspection of the amount of space used in the various stacks and the global state space may indicate how the environment can best be configured. A sequence of calls to the monitoring predicates around critical portions of a program can help identify the areas where there might be excessive consumption of time and/or space.

### **memory\_status**

The `memory_status` predicate returns space usage information for the three system stacks and the global state space. For each of these areas, there is a list of three numbers, the size of allocated space, the amount currently in use, and the measurement of the largest amount used up to the current time (a high water mark). All sizes are expressed in the number of bytes. The predicate `stats` resets all high water mark values.

### **cputime**

The predicate `cputime` returns the amount of time, in milliseconds rounded to the nearest 1/60th of a second, since



powerup of the Macintosh. The predicate `timer`, listed below, shows how `cputime` can be used in calculating the execution time of `_Goal`:

```
timer(_Goal, _Time) :-  
    _T1 is cputime,  
    _Goal,  
    _Time is cputime - _T1.
```

### **stats**

The `stats` predicate returns the number of logical inferences, the number of primitive calls (primitives are built-in utilities, not clauses), the number of interval operations and narrowing iterations, and the time (in units of 1/60 second) since all the counters for the above values were zeroed. When `stats` is called without arguments, the counters and the high water marks for the three stacks and the global state space are zeroed (see `memory_status` above).

The predicate `lipsrate` listed below calculates the logical inferences and primitive calls executed per second for `_Goal`:

```
lipsrate(_Goal, _Lips) :-  
    stats,  
    _Goal,  
    stats(_Inf, _Prim, _, _, _Ticks),  
    _Lips is (_Inf + _Prim) / (_Ticks / 60).
```

The iterations and interval operations information measures the amount of work done using interval arithmetic. The iteration count is the number of times the interval arithmetic engine has been invoked, while the interval operations is the number of atomic interval operations executed, for example, addition and multiplication. An atomic interval operation corresponds to a few equivalent floating point operations.

## **Building an Application**

All Macintosh files (also known as documents) have an application signature and a file type, each specified by a



sequence of four characters. The signature is used to associate applications and their documents; file type is used to distinguish between various document formats for a given application.

BNR Prolog documents have the following creator signatures and types:

Document	Signature	Type
BNR Prolog	APRO	APPL
source	APRO	TEXT
work space	APRO	APWS
state space	????	APSS

BNR Prolog may be launched through the finder by opening the application, a source document, or saved work space document.

An application built using the development environment should be given its own signature, so that relevant documents may be associated with it. Document types and creator signatures are registered with Apple to guarantee uniqueness.

An application is a single document containing the Prolog runtime system, a base work space, binary contexts, external code resources, other Macintosh resources and the initial configuration information. The work space resides in the data fork; all others reside in the resource fork of the file. An application has a file type of APPL, and a signature that is specified when it is built. State spaces are not part of application files. In fact, they are not normally associated with an application as their creator is initially specified as ????.

An application other than the development environment terminates after execution of the initial goal, since that is what drives it. Responsibility for initializing any resources that may be required, such as windows, menus, and files, rests with the application.



**build\_application**

An application is built by calling `build_application`, which has the following form:

```
build_application(_filename, _signature, _stack_sizes,  
                 _initial_predicate, [<contexts>])
```

`_filename` is the pathname of a file to be created. If the file already exists, then the predicate fails.

`_signature` is a symbol representing the creator of the application. It must be exactly four characters in length, padded with blanks if necessary. The application is automatically given the file type APPL.

`_stack_sizes` is a list of three numbers specifying the size in Kbytes of the world stack, the global stack, and the local stack respectively. If this argument is unbound, then the current development environment configuration is used. A size of 0 for a stack causes BNR Prolog to allocate the stack based on the amount of free memory available to the application. For example, if you only want a 20 Kbyte local stack and want the others to be as large as possible, `_stack_sizes` is `[0, 0, 20]`.

`_initial_predicate` is a term specifying the initial goal list. Upon completion of this term, the application exits.

`<contexts>` is a list of context file names or external file names to be included in the application file. "Current" binary images are created if necessary. If the empty list is specified, no contexts are added.

To create a simple application composed of nothing more than the initial goal that simply brings up a dialog with a message and then quits, specify:

```
?- build_application(silly, 'SAMP', _X,  
                    'message(\'Hi there\')', []).
```



Launching the program `silly` results in a dialog with the words `Hi there`. Clicking on `OK` causes the program to terminate. (If running under `MultiFinder`, you need to exit `BNR Prolog` only if it is in the same folder as the created application. This is necessary because applications try to open and close the file `Console` that is already in an open state for `BNR Prolog`.)

Note that an application cannot use the built-in predicates `build_application`, `save_ws` or `set_trace`, since they are available only in the development environment.

When the initial goal is executed, no contexts other than the default contexts included in the work space are loaded. This means that the initial predicate must be used to load all user defined contexts. The search algorithm for loading contexts first looks for a file (or window) before looking for the binary context inside the application. This allows the user to override an embedded context by having a file with the same name as the embedded context accessible in the current working directory. Choosing uncommon names for your contexts helps to prevent accidental overrides.

Certain operations, such as loading a context, result in a message being written to the *Console* file. This file is initially hidden from view. However, upon exiting `Prolog`, the user is prompted to save this file and any other open windows if changes have occurred. The application can avoid the prompt for the *Console* (which the user may have never seen) by closing (and possibly saving) the *Console* window immediately prior to quitting.

## Supporting the Macintosh Interface

Applications built using `build_application` do not contain any icons and appear on the desktop as the default system icon for an application. If the application requires icons for its files, they must be created outside of `BNR Prolog` and added to the application after it is built. Icons (resource type `ICON` or `ICN#`) consist of a small bitmap representing an image, and a mask to transform it when it is selected. Icons can be created by




specifying the bitmap, although it is generally easier to use specialized tools like *ResEdit* to create the icon. While an application does not require icons, they may be created to customize an application. (For example, BNR Prolog has four associated icons.) Each file type should have a different appearance when displayed by the finder.

In addition, applications do not initially contain a bundle resource defining the relationship between the application and the files associated with it. A bundle resource (resource type 'BNDL') specifies the application's signature, its version resource identifier, and the mappings between icons and file references that are contained in the application. The Macintosh system uses a bundle resource to find the icons associated with an application.

The version resource contains a string describing the application. The resource type of the version is the same as the application's signature. The example *silly*, has a resource of type *SAMP*. The version resource is displayed by the **Get Info** command from the finder.

A resource of type *FREF* is required by the Macintosh for each different type of document created by an application (unless the Macintosh defaults are used). A file reference contains the file type, a local file reference that is also used by the bundle, and a file name. BNR Prolog contains four such resources, one for each type of file (and icon).

For new applications, the  menu contains an item displaying **About BNR Prolog...** This can be changed by editing the application's menu resource, *SysApple*, to contain the appropriate string.



## Program silly

To illustrate these concepts, assume there is an application that only uses and creates text files. This means that two icons are required; one for the application and one for text files as follows:



This example uses the *Macintosh Programmers Workshop* tools, specifically Rez. (The format of the file is applicable to version 2.0 of MPW.) The code adds the necessary resources to the application so that the application itself, and any text files it creates, appear as the icons above. Double-clicking such text files launches the application. (Other tools, such as ResEdit and RMaker can be used to accomplish the same thing.)

```
#include "Types.r"

type 'SAMP' as 'STR ';
resource 'SAMP' (0)
    {"BNR Prolog sample application" /* version data */};

resource 'BNDL' (128)
{
    'SAMP',          /* application's signature */
    0,               /* resource id of version data */
    {'ICN#',
        {0, 128,      /* application icon is 128 */
         1, 129,      /* text file icon is 129 */
        },
        'FREF',
        {0, 128,      /* application file ref is 128 */
         1, 129,      /* text file ref is 129 */
        },
    }
}
};
```



```
resource 'FREF' (128, "Sample Application")
{
    'APPL',          /* file type */
    0,               /* local ref from BNDL */
    "Sample Application" /* file name */
};

resource 'FREF' (129, "Sample Text")
{
    'TEXT',          /* file type */
    1,               /* local ref from BNDL */
    "Sample Text"    /* file name */
};

resource 'ICN#' (128, "Sample Appl icon")
{
    { /* original image */
        $"0001 0000 0002 8000 0004 4000 0008 2000"
        $"0010 1000 0020 0800 0040 0400 0080 0200"
        $"0100 0100 0200 0080 0400 0040 0800 0020"
        $"118F 7A10 2249 4A08 4249 4A04 83CF 7A02"
        $"4668 4204 2428 4208 1428 43D0 0800 0020"
        $"0400 0040 0200 0080 0100 0100 0080 0200"
        $"0040 0400 0020 0800 0010 1000 0008 2000"
        $"0004 4000 0002 8000 0001",

        /* reversed image when selected */
        $"0000 0000 0001 0000 0003 8000 0007 C000"
        $"000F E000 001F F000 003F F800 007F FC00"
        $"00FF FE00 01FF FF00 03FF FF80 07FF FFC0"
        $"0FFF FFE0 1FFF FFF0 3FFF FFF8 7FFF FFFC"
        $"3FFF FFF8 1FFF FFF0 0FFF FFE0 07FF FFC0"
        $"03FF FF80 01FF FF00 00FF FE00 007F FC00"
        $"003F F800 001F F000 000F E000 0007 C000"
        $"0003 8000 0001"
    }
};
```



```

resource 'ICN#' (129, "Sample Text icon")
{
    { /* original image */
        $"0FFF FFF8 0800 0008 0800 0008 0800 0008"
        $"0800 0008 087F 3F08 0808 2008 0808 2008"
        $"0808 2008 0808 3C08 0808 2008 0808 2008"
        $"0808 2008 0808 3F08 0800 0008 0800 0008"
        $"0800 0008 0800 0008 0841 3F88 0822 0408"
        $"0814 0408 0808 0408 0808 0408 0808 0408"
        $"0814 0408 0822 0408 0841 0408 0800 0008"
        $"0800 0008 0800 0008 0800 0008 0FFF FFF8",

        /* reversed image when selected */
        $"0000 0000 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0 07FF FFF0"
        $"07FF FFF0 07FF FFF0 07FF FFF0"

    }
};

```

If the above text is placed in the file `silly.r`, then the *MPW* commands:

```

rez -a -o silly silly.r
setfile -a B silly

```

add the resources into the file `silly`, and set the bundle bit for the application since there is now a bundle. This bit causes the finder to examine the resource fork of the application and extract the icons for display.



## Managing Error Conditions

During the execution of a program, such things as execution of an integer or execution stack overflow can cause system errors. A mechanism is provided to enable recovery from both system and program errors through the predicate `capsule`.

The recommended technique is to encapsulate a program defined error handler within that portion of a program in which there is a possibility of error. For example, if the program portion is packaged as the goal code and the error handler is the predicate `error_handler`, the occurrence of code is replaced by the encapsulation

```
capsule(code, error_handler)
```

`capsule` succeeds as the program portion succeeds. If the program portion fails on a error, the interpreter recovers (instead of aborting) and the error handler is executed. `capsule` is defined as:

```
capsule(_Code, _Handler) :-  
    recovery_unit(_Code) ;  
    _Handler.  
  
recovery_unit(_Code) :- _Code.  
recovery_unit(_Code) :- failexit(capsule).
```

Since `failexit` applies to the innermost goal of the named predicate, capsules can be nested.

The error recovery mechanism is based on the special goal `recovery_unit`. When a system or program error is detected, `failexit(recovery_unit)` is executed if `recovery_unit` is present on the goal stack. Otherwise, the program aborts. To avoid confusion, the Prolog program should not use `capsule` or `recovery_unit` as internal predicate names.



The general strategy for a handler is to inspect the integer error code returned by the predicate `get_error_code`, and take appropriate alternate or recovery action, as in

```
error_handler :-  
    get_error_code(_Error_code),  
    error_recovery(_Error_code).
```

`error(_error_code)` triggers the error condition specified by its integer argument. Predefined system values of this code are found in "Appendix B" of the *BNR Prolog Reference Manual*.

Consider a simple handler which outputs an error message and aborts execution:

```
handler:-  
    get_error_code(_E),  
    nl, write('**** ',_E,' ****'),nl,  
    failexit(listener).      % not very clever
```

This handler may be tested with the following queries:

```
?- capsule(error(22), handler).  
  
% an attempt to execute a variable produces error 16  
?- capsule(error(_X), handler).
```

Using *Command-* is a second way of interrupting program execution. When this occurs, a call to the predicate `attention_handler` is interpolated into the normal execution stream. If the call to `attention_handler` succeeds, normal execution is resumed. The system provided `attention_handler` provides an optional traceback, aborts the current computation, and returns to the listener. An example of an `attention_handler` which optionally aborts or continues program execution follows:

```
attention_handler :-  
    cut(attention_handler),    % remove other choices  
    confirm('Abort execution?', '', 'YES', _Reply),  
    _Reply = 'YES' -> failexit(listener).
```







# Chapter 19

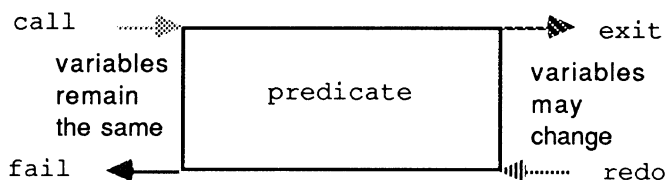
## The Debugger

In procedural programming languages, once a variable is assigned a value, it continues to have that value until either another is assigned or scoping rules make the variable inaccessible. By means of debugging facilities, the programmer can always examine the value of an accessible variable, as well as trace the path of the program as it moves forward in execution.

Not only does a Prolog program not always move forward, but a value that is bound to a variable through unification may be unbound by backtracking. Monitoring backtracking requires the ability to display the system's attempts to resatisfy a goal. Thus, a Prolog debugger must show the unbinding of variables as well their unification.

### The Box Model

Prolog debuggers are based on the four port box model of a predicate. (We use the term predicate here to refer not only to the logical relationship, but also to its implementation as a set of clauses with the same principal functor.) In this model, a Prolog predicate is treated as a black box with four ports: *call*, *exit*, *redo* and *fail*. The ports represent the various states in which a predicate may be found during execution.



Four Port Box Model



When a state change occurs, the debugger displays the current goal with the variable bindings, the execution port, and the unique invocation identifier. The identifier aids in correlation of the various state changes of the predicate. Whenever a predicate is called with a new set of arguments, a new invocation identifier is generated. Calls subsequent to the initial goal are distinguished by both a depth number and an invocation identifier.

Only those predicates that are visible from the current context are accessible to the debugger. This excludes local names in other than the current context.

The *call* port for a predicate represents the initial invocation of a Prolog goal. Execution of the predicate has been initiated by a call, and the variable bindings at the time of the call are displayed.

The *exit* port represents the state a predicate reaches upon the success of a call or redo. Execution of the predicate is complete, and the variable bindings at the time of completion are displayed. This includes the instantiated values of those variables that were unbound at the time of the corresponding call.

Like the call port, the *redo* port represents a state for which the predicate is the current goal, but in this case alternative solution for this goal is being attempted. Some subsequent goal has failed and backtracking has occurred. The redo port bears the same call number as the initial invocation. The variable bindings displayed are those of the corresponding exit.

The *fail* port indicates a failure state for the predicate after either a call or a redo. The variable bindings displayed are those of the corresponding call.

If a call is made to a predicate that is not in the knowledge base, the debugger signifies this by displaying a call and a fail with "???" to the left of the invocation number.

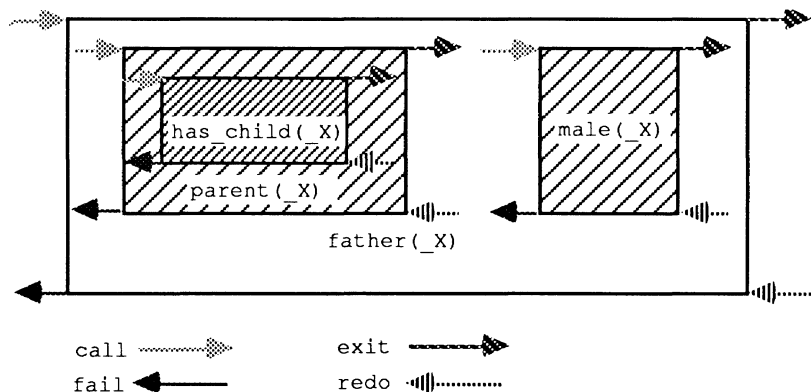


Using the following program and query :

```
/* program FFamily */
father(_X) :- parent(_X), male(_X).
mother(_X) :- parent(_X), female(_X).
parent(_X) :- has_child(_X).
has_child('Krystyna').
has_child('Jean_Jaques').
male('Jean_Jaques').
female('Krystyna').

?- father(_X).
```

the succession of predicate state changes that occur in the attempt to satisfy the goal is shown diagrammatically as follows:



### State Changes

The call port is entered only once with any given set of arguments. A redo port is entered only as the result of a previous failure or exit. A predicate passes through an exit or fail port only if it has entered a corresponding call or redo port.

To enable users to see the state changes according to the predicate box model, there are three modes of debugging available. *Creeping* displays every state change for every predicate executed. Each goal is executed by a second level



interpreter in this mode, which makes the global stack requirement much higher. *Leaping* displays every state change for those predicates specifically marked with *spypoints* by the user. *Skipping* displays every state change for the currently active predicate. A group of single character commands are available in all modes.

## Debugging a Program

Debugging is enabled by the predicates `debug` or `trace`, and disabled by `nodebug` or `notrace`. `Leap` mode is automatically enabled with the `debug` command. No break in execution of the program occurs until a *spypoint* is reached. Use of the `trace` command automatically causes a pause in execution at the call port for the first goal of the program.

Once debugging is specified and execution of a program is initiated, debug information is displayed, and the system waits for the next single character command from the user. In alphabetical order, a list of commonly used commands are:

- a* — abort execution of the current program and disable debugging
- b* — break execution to the listener (use `continue` to resume execution)
- c* or *return* — creep to the next port and display predicate information
- f* — fail the current goal
- g* — display ancestor history of current goal
- h* or *?* — display help menu
- l* — leap to the next port for a predicate marked with a *spypoint*.
- n* — disable debugging



- `r` — unbind variables and retry the call to the current predicate
- `s` — skip to the next port for the current predicate is reached. If the exit of that predicate is reached, revert to creep mode
- `+` — mark the current predicate with a spypoint
- `-` — remove a spypoint from the current predicate

After execution of commands that provide information rather than continue execution of the program, the debug information for the current predicate is repeated, and a further port command is awaited.

During execution of a program for which debugging is enabled, use of *Command-* causes an execution interrupt. Control is passed to the debugger, which is initially in interactive creep mode.

## Creeping

Creeping single steps the program from the current position to the next valid port for any callable predicate and displays debug information. For example, if tracing the program *FFamily*, as in

```
?- trace.  
?- father(_X).
```

the debugger interrupts the execution of `father` and displays the call port

```
(1) 0 call: father(_X) ?
```

At this point the debugger will accept interactive debug commands. Using creep to provide debug information pertaining to the program's execution, a complete step by step trace of all the ports is:



```
?- father(_X).
(1) 0 call: father(_X) ? c
(2) 1 call: parent(_X) ? c
(3) 2 call: has_child(_X) ? c
(3) 2 exit: has_child('Krystyna') ? c
(2) 1 exit: parent('Krystyna') ? c
(4) 1 call: male('Krystyna') ? c
(4) 1 fail: male('Krystyna') ? c
(2) 1 redo: parent('Krystyna') ? c
(3) 2 redo: has_child('Krystyna') ? c
(3) 2 exit: has_child('Jean_Jaques') ? c
(2) 1 exit: parent('Jean_Jaques') ? c
(5) 1 call: male('Jean_Jaques') ? c
(5) 1 exit: male('Jean_Jaques') ? c
(1) 0 exit: father('Jean_Jaques') ? c
?- father('Jean_Jaques').
```

If the user chooses to search the knowledge base for more than one solution to a query, the debugger attempts to redo those goals for which there are choicepoints, as in

```
(1) 0 redo: father('Jean_Jaques') ? c
(5) 1 redo: male('Jean_Jaques') ? c
(5) 1 fail: male('Jean_Jaques') ? c
(2) 1 redo: parent('Jean_Jaques') ? c
(3) 2 redo: has_child('Jean_Jaques') ? c
(3) 2 fail: has_child(_X) ? c
(2) 1 fail: parent(_X) ? c
(1) 0 fail: father(_X) ? c
```

YES

Note the port numbering format (n) m where (n) is unique for every predicate call and m indicates the depth of the current goal in relation to the main goal. Port numbering aids in distinguishing predicate calls and nesting levels within the call hierarchy.

When in creep mode, providing input for all ports of all predicates may be tedious. To reduce the amount of prompting by the debugger, leash can be used to create a leash list



specifying at which ports prompting is desired. For example, the command

```
?- leash([call, redo]).
```

results in a display of all ports, but pauses for input occur only at call and redo ports. Thus, the goal

```
?- mother(_X).
```

produces the output

```
?- mother(_X).
(1) 0 call: mother(_X) ? c
(2) 1 call: parent(_X) ? c
(3) 2 call: has_child(_X) ? c
(3) 2 exit: has_child('Krystyna')
(2) 1 exit: parent('Krystyna')
(4) 1 call: female('Krystyna') ? c
(4) 1 exit: female('Krystyna')
(1) 0 exit: mother('Krystyna')
?- mother('Krystyna').
(1) 0 redo: mother('Krystyna') ? c
(4) 1 redo: female('Krystyna') ? c
(4) 1 fail: female('Krystyna')
(2) 1 redo: parent('Krystyna') ? c
(3) 2 redo: has_child('Krystyna') ? c
(3) 2 exit: has_child('Jean_Jaques')
(2) 1 exit: parent('Jean_Jaques')
(5) 1 call: female('Jean_Jaques') ? c
(5) 1 fail: female('Jean_Jaques')
(2) 1 redo: parent('Jean_Jaques') ? c
(3) 2 redo: has_child('Jean_Jaques') ? c
(3) 2 fail: has_child(_X)
(2) 1 fail: parent(_X)
(1) 0 fail: mother(_X)
```

YES



The command

```
?- leash([]).
```

disables leashed debugger output without changing the mode of debugging. Thus, the debugger only pauses for spypoints, until such time as creeping is resumed.

## Leaping

Spypoints on predicates are enabled with `spy` or `spyall`, and removed with either `nospy` or `nospyall`. Predicates for which spypoints are enabled are displayed with "\*\*\*" to the left of the invocation number. When leaping, the debugger displays port and variable information only for the predicates marked with spypoints, although the complete ancestor history is available. For example,

```
?- debug.  
?- spy(parent).  
  
?- mother(_X).  
** (1) 1 call: parent(_X) ?
```

displays the call port for `parent` bypassing the port for `mother` and prompts the user for action. After a creep followed by a leap, the output is

```
?- mother(_X).  
** (1) 1 call: parent(_X) ? c  
   (2) 2 call: has_child(_X) ? 1  
** (1) 1 exit: parent('Krystyna') ?
```

Note that the exit port for the call to `has_child` is not shown. The leap command causes the debugger to go straight to the port of the next spied predicate, which in this case is the exit from `parent`.



To change the amount of prompting that the debugger demands, `spy` can specify a leash list for a predicate, as well as mark it for spying. For example,

```
?- spy([call, exit], parent).
?- spy([redo], female).
```

sets `spypoints` for both `parent` and `female`, and sets prompting only for the specified ports for those predicates. At the unleashed ports for the spied predicates, the debugger displays information, and moves on to the next port without pausing. Thus, the following

```
?- mother(_X).
** (1) 1 call: parent(_X) ? 1
** (1) 1 exit: parent('Krystyna') ? 1
** (2) 1 call: female('Krystyna')
** (2) 1 exit: female('Krystyna')
?- mother('Krystyna').
** (2) 1 redo: female('Krystyna') ? 1
** (2) 1 fail: female('Krystyna')
** (1) 1 redo: parent('Krystyna')
** (1) 1 exit: parent('Jean_Jaques') ? 1
** (3) 1 call: female('Jean_Jaques')
** (3) 1 fail: female('Jean_Jaques')
** (1) 1 redo: parent('Jean_Jaques')
** (1) 1 fail: parent(_X)
```

When creeping, step by step tracing is performed on all but those spied predicates for which leash lists have been specified. If the `spypoints` are set as follows,

```
?- spy([], parent, female).
```

the same output is produced by the query, but there is no pause at any port.



## Skipping

Skipping creates a temporary spypoint for the current predicate. Thus, when skipping, the debugger displays port and variable information only for the predicate at which the skip command is given. For example,

```
?- trace().  
  
?- father(_X).  
  (1) 0 call: father(_X) ? s  
  (1) 0 exit: father('Jean_Jaques') ? s  
    ?- father('Jean_Jaques').  
  (1) 0 redo: father('Jean_Jaques') ? s  
  (1) 0 fail: father(_X) ?
```

Any spypoints that may exist are ignored when skipping. A skip command at an exit of fail port is interpreted as a creep.

```
?- spy(female).  
  
?- mother(_X).  
  (1) 0 call: mother(_X) ? s  
  (1) 0 exit: mother('Krystyna') ? s  
    ?- mother('Krystyna').  
  (1) 0 redo: mother('Krystyna') ? c  
** (2) 1 call: female('Jean_Jaques') ? s  
** (2) 1 fail: female('Jean_Jaques') ? s  
  (1) 0 fail: mother(_X) ? c
```



## Entering the Listener

It is possible to enter the listener and execute some other call in the middle of the debugging session. When `break` is called or the `b` command is submitted to the debugger, the Prolog listener is called. For example,

```
?- mother(_X).  
  (1) 0 call: mother(_X) ? c  
  (2) 1 call: parent(_X) ? b  
  
*** Debugger temporarily turned off ***  
  
Break (level 1)  
  
?-
```

enters the listener at the time of the call to `parent`. At this point, all the facilities of the Prolog system are available. However, it is important to keep in mind that the debugging session is still on the goal stack, and that the main goal (`mother`) is still being executed. The execution break causes a new instance of the listener to be executed as a subgoal of `mother`. The debugging session resumes with a call to `continue`:

```
?- continue.  
Exit Break (level 1)  
  
*** Debugger turned back on ***  
  
(2) 1 call: parent(_X) ?
```

The break mechanism should be used with caution. Any changes made during a break in program execution are in effect as soon as debugging resumes.



## Debugging Event Handlers

Event handlers (as described in the chapter "User Interfaces") pose special debugging problems since all interactions with the system are through events, including those in the debugger. Debugging event handlers (with ports leashed) does not work unless the debugger's event loop runs with debugging off. Similarly, general interaction with the system becomes difficult unless the system listener disables debugging.

A special predicate called `trace_event(_Event_List..)` provides an event loop with the debugger enabled. `_Event_List` is a list of either event names (for example `userkey`) or event usage patterns (such as `_Event(_W, _D1, _D2)`) which can be used to filter out the relevant event sequence. Only events matching an element of `_Event_List` are output to the console, although all are executed.

Note that if the debugger stops at any port, then subsequent events may be consumed by the debugger. If any ports are leashed, user interaction with the debugger can perturb both event sequences and timing considerations. One way to debug event sequences is to use `trace_event` to generate an event sequence for output. This output can then be passed as an argument to `replay_events(_Events..)` as many times as required while debugging the handlers. This method of debugging is limited to handling a single event loop at a time.

The following definitions for `trace_event` and `replay_events` closely resemble their built-in counterparts.

```
trace_event :- $trace_event(F(_, _, _)).  
trace_event(_Xs..) :- $trace_event(_Xs..).  
$trace_event(_Xs..) :-  
    nodebug,  
    $adjust_events(_Xs, _Ys),  
    repeat,  
    userevent(_E, _W, _D1, _D2, noblock),  
    $trace_event_aux(_E(_W, _D1, _D2), _Ys).
```



```
$trace_event_aux(_E, _Events) :-
    not(not(member(_E, _Events))),
    write('\n'), writeq(_E), writeq(', '),
    debug,
    fail.

$trace_event_aux(_Event, _) :-
    [_Event, cut],
    fail.

$trace_event_aux(_, _) :-
    nodebug,
    fail.

replay_events(_Events..) :-
    debug,
    member(_Event, _Events),
    [_Event, cut],
    fail .

replay_events(_..).

$adjust_events([], []).

$adjust_events([_F(_Args..), _Xs..], [_F(_Args..), _Ys..]) :-
    $adjust_events(_Xs, _Ys).

$adjust_events([_F, _Xs..], [_F(_, _, _), _Ys..]) :-
    $adjust_events(_Xs, _Ys).
```

The "*BNR Prolog Reference Manual*" provides further information about debug commands, and a description of some predicates that are available to those who wish to tailor the debugger to suit their own needs.







# Chapter 20

## Prolog Compatibility Issues

---

This chapter is primarily aimed at the reader who is familiar with other Prolog systems, particularly those belonging to the Edinburgh family (such as C-Prolog, Quintus Prolog, Arity/Prolog or ALS Prolog). Those characteristics of BNR Prolog that are different from Edinburgh Prologs are highlighted, and the rationale behind the differences is provided.

For the most part, BNR Prolog is a superset of Edinburgh Prologs, and therefore programs written in Edinburgh dialects can be ported without difficulty. However, those programs dealing with the structure of clauses, terms, and lists, so called metaprograms, require an awareness of some of the underlying semantic differences.

Most of the built-in predicates and operator declarations found in *Programming in Prolog* by Clocksin and Mellish are provided. Those few predicates that are not built-in, such as `functor`, `"=.."`, and `call`, are unnecessary, but they are defined in the file **Edinburgh** that is provided with the software. This file should be loaded to facilitate porting programs written in an Edinburgh dialect.

## Sequences

Argument sequences, lists, and clause bodies are all expressed in a uniform notation. Since these structures play such an important role, it is helpful to review the idea of a list as it is presented in other Prolog systems, and to explain the design considerations that move BNR Prolog away from that model.



## Edinburgh Lists

In most Prolog implementations, a list is represented by the Prolog structure

```
. (CAR, CDR)
```

where "." is a binary functor, and CAR (the first element) and CDR (the remainder) are any Prolog terms. This is a straight forward representation of the LISP concept *cons cell*. Usually, the functor "." is defined as an infix operator so the expression above may also be written as

```
CAR . CDR
```

If the CDR of a cons cell is itself a structure of the same form, the result is a nested sequence of structures that make up the list. For example,

```
. (apple, . (orange, . (kiwi, [])))
```

is a typical list. Note that the last CDR in this nested structure is the special atom [], a conventional notation for terminating the last cons cell in a list. Lists that terminate in [] are called *proper lists*. The proper list may be written more succinctly by an alternative notation that shows only the CAR elements of the structure, for example:

```
[apple, orange, kiwi | []]
```

The special symbol, "|", which serves as the separator between a sequence of list elements and the list of remaining elements, may be omitted if [] is the last remainder, as in:

```
[apple ,orange ,kiwi]
```

To denote an extensible list, use a logic variable in the CDR. Thus,

```
[CAR | CDR]
```

represents a list with one or more elements.



The proper termination of lists with the empty list, [], is usually not enforced. Since lists are constructed out of pairs, the structure

```
[apple, orange | kiwi]
```

is a perfectly acceptable list that might well have been constructed by unifying a variable, for example,

```
[apple, orange | Fruit], Fruit = kiwi
```

However, most predicates which recursively operate on lists, such as the ubiquitous `append`, may in fact fail if the lists are improper. For example, with the usual definition of `append`

```
append([], L, L).  
append([H | T], L, [H | R]) :- append(T, L, R).
```

the question

```
?- append([apple | orange], [kiwi], L).
```

fails, but

```
?- append([kiwi], [apple | orange], L).
```

succeeds with `L = [kiwi, apple | orange]`. For the same reason,

```
?- append(banana, [kiwi], L).
```

fails whereas

```
?- append([kiwi], banana, L).
```

succeeds with `L = [kiwi | banana]`. Furthermore, there seems to be no way to protect from this type of behavior.



## BNR Prolog Lists

BNR Prolog addresses these problems by building the notion of a sequence of terms into the basic semantic model. The bracket notation

```
[apple, orange, kiwi]
```

is used to represent a proper list of elements and the notion of a proper list (that is, a true *sequence*) is a primitive construct in the language.

*A tail variable, which unifies only with a sequence of list elements, is used to refer to an arbitrary sequence of elements in a list.* Since this variable can refer only to the contents of a proper list, it has a special notation: a variable name followed by an ellipsis. For example, the list

```
[_first, _second, _rest..]
```

refers to a proper list with two elements or more elements. Compare this with the expression

```
[First, Second | ERest]
```

which, in Edinburgh Prolog, refers to a list containing two or more elements. Although `_rest..` appears to be simply an alternative notation for `| ERest`, this is not the case. The tail variable `_rest..` refers only to a sequence of zero or more Prolog terms, whereas `ERest` can refer to any kind of term.

Because tail variables refer to the sequence of elements in a list, it is possible to express the idea that `_L` is a proper list simply by writing

```
_L = [_X..]
```

There is no apparent way to express this idea in Edinburgh Prolog.



To ease the portability of Edinburgh programs which do not rely on the underlying semantics, the following coercions are performed on input:

- `[_A, _B..]` can be expressed as `[_A | _B]`
- `[_B..]` can be expressed as `_B`

Thus, the last clause of the standard predicate `append` can be written as

```
append([H | T], L, [H | R]) :- append(T, L, R).
```

which is coerced to

```
append([H, T..], L, [H, R..]) :-
    append([T..], L, [R..]).
```

Note that the list `[H | symbol]` is flagged as a syntax error, ensuring that improper lists cannot be input; the internal semantics insures they can never be created.

To illustrate the behavior of tail variables, consider the following unifications.

Goal	Result
<code>[a, _X..] = [a, b]</code>	<code>[_X..] = [b]</code>
<code>[_X..] = [a, b]</code>	<code>[_X..] = [a, b]</code>
<code>[_X..] = []</code>	<code>[_X..] = []</code>
<code>[_X, _Y..] = [a]</code>	<code>_X = a, [_Y..] = []</code>
<code>[a, _C..] = [_B, _D..]</code>	<code>_B = a, [_C..] = [_D..]</code>
<code>[_a, _b..] =</code>	<code>_a = _f,</code>
<code>[_f, _g, _h..]</code>	<code>[_b..] = [_g, _h..]</code>

In summary, *BNR Prolog terms are constructed from lists rather than lists from terms*. The notable characteristics of lists are summarized as follows:

- lists are always proper lists, that is the last remainder of a list is always the list with zero elements, `[]`



- all conventional list manipulation techniques apply to BNR Prolog lists
- while the "|" notation is supported, it is simply shorthand for the tail variable notation
- tail variables (variables of the form `_Var..`) may occur only as the last term in an explicitly written list
- the list with an indefinite number (zero or more) of elements is written `[_Var..]`

The added expressive power offered by tail variables is further apparent in the sections that follow.

## Sequences in Terms

Sequences may be used to represent lists, argument parameters and executable goals in a uniform manner. Moreover, they provide a mechanism for expressing variable functors and variadic predicates.

### Structures

The general form of any BNR Prolog structured term can be expressed as

```
_Functor(_Args..)
```

where `_Functor` is the principal functor and the list of arguments to `_Functor` can be expressed by `[_Args..]`. Thus the unification

```
_F(_Args..) = father(jupiter, vulcan)
```

succeeds with `_F` bound to `father`, and `_Args..` bound to the sequence `jupiter, vulcan`.



Variable functors and argument lists make it possible to express generic classes of structured terms:

```
f(_X, _Y)      % structure f with 2 arguments
f(_X, _Y...)   % structure f with 1 or more arguments
f(_X..)        % any structure f
_F(_X, _Y)     % any structure with 2 arguments
_F(_X, _Y...)  % any structure with 1 or more
               % arguments
_F(_X..)       % any structure
```

This ability to unify structured terms with variable structures makes the Edinburgh term examination predicate `functor` and `=..` redundant. Every occurrence of

```
functor(_Term, _Functor, _Arity)
```

can be replaced by the unification

```
_Term = _Functor(_Args..)
```

and the arity of the term is simply the length of the list `[_Args..]`, which can be obtained by

```
termlength([_Args..], _Arity, _)
```

One advantage of this representation is that `_Term` has a very general structure, without either `_Functor` or `_Args..` necessarily being bound. The predicate `functor`, on the other hand, requires either that `_Term` be a non-variable, or that `_Functor` and `_Arity` be bound to an atom and an integer respectively.



In BNR Prolog the structure of a term is explicit in its syntactic form. If a term does not have an argument list, for example, it will not unify with the expression `_Functor(_Args..)`. Thus, the following

Succeed	Fail
<code>test() = _F(_Args..)</code>	<code>test = _F(_Args..)</code>
<code>'I'(a, _C..) = _F(_Args..)</code>	<code>1.35e15 = _F(_Args..)</code>
<code>(3 + 5) = _F(_Args..)</code>	<code>1988 = _F(_Args..)</code>

Note the distinction between the 0-arity functorial structure `test()` and the symbol `test`, which is its principal functor.

The use of lists as arguments to predicate names simplifies and increases the efficiency of some common programming tasks. For example, consider the predicate `build`, which constructs a list of binary terms from a pair of lists (the arguments) and a principal functor.

```
build(_, [], [], []).
build(_F, [_A1, _A1s..], [_A2, _A2s..],
      [_F(_A1, _A2), _Rest..]) :-
    build(_F, _A1s, _A2s, _Rest).
```

A superficially equivalent program in Edinburgh Prolog is

```
build(_, [], [], []).
build(_F, [_A1 | _A1s], [_A2 | _A2s], [_T | Rest]) :-
    _T =.. [_F, _A1, _A2],
    build(_F, _A1s, _A2s, _Rest).
```

In either case, a query of the form

```
?- build(in, [_P1, _P2], ['Ohio', 'Iowa'], _T).
```

binds the variable `_T` to the list

```
[in(_P1, 'Ohio'), in(_P2, 'Iowa')]
```



Suppose the principal functor of the term is not known at the time of the call to `build`, but can only be determined sometime after the call. In such a case, the Edinburgh program fails (since `=` fails if `_F` is not an atom) whereas the BNR Prolog program succeeds with the same result. For example,

```
?- build(_F, [_P1, _P2], ['Ohio', 'Iowa'], _T),
   _F = state.
```

binds `_T` to

```
[state(_P1, 'Ohio'), state(_P2, 'Iowa')]
```

There is also an efficiency advantage here. To alternately build this list with `state` and `in` as the principal functors of the structure, not only must those atoms be bound prior to calling `build`, but the structures must be rebuilt from scratch. Thus, in Edinburgh Prolog the only way to produce alternative lists is with the question

```
?- (F = state ; F = in), build(F, [P1, P2, P3],
   ['Ohio', 'Iowa', 'Texas'], T).
```

which requires twice as much processing as the BNR Prolog question

```
?- [build(_F, [_P1, _P2], ['Ohio', 'Iowa'], _T),
   (_F = state ; _F = in)] .
```

The reason is that `_F` need not be instantiated, in BNR Prolog, before the call to `build`, whereas it must be instantiated in Edinburgh Prolog.

The capacity to unify structures with variable predicate names also makes metalogical programs fully backtrackable and therefore more declarative.



## Clauses

The general notion of a sequence suggests a uniform representation of clauses in which the body is a list of terms. With this representation, meta programming is simply a type of list processing. The general form of a BNR Prolog clause is

```
_Functor(_Args..) :- [_Body..].
```

where `_Functor` is a symbol (the predicate name), whose argument list is `[_Args..]`.

The body of the clause is the list `[_Body..]`. Since the body of a clause is always a list, it follows that lists are executable terms. The execution semantics of the sequence of elements in a list is simply that of logical conjunction, making the Edinburgh Prolog comma operator, `,` unnecessary. Executing the empty list `[]` is equivalent to executing the atom `true`.

A fact that is entered as

```
brother(groucho, harpo).
```

is represented internally as a clause with an empty body:

```
brother(groucho, harpo) :- [].
```

Edinburgh syntax is accepted by permitting clause bodies to be written without brackets. Thus, the clause

```
f :- a, b, c.
```

is coerced upon input to

```
f() :- [a, b, c].
```

All the sequences in BNR Prolog, including argument sequences to predicates and clause bodies, may be manipulated with all the usual list handling programs such as `member`, `append`, `length` and so on. The list nature of clause bodies means that all programs that examine the structure of clauses should be modified to handle list structures instead of terms with the `,`



operator. For example, the simple metacircular interpreter, `interpret`, written in Edinburgh Prolog as

```
interpret(true) :- !.
interpret((GoalA, GoalB)) :-
    !,
    interpret(GoalA),
    interpret(GoalB).
interpret(Goal) :-
    clause(Goal, Body),
    interpret(Body).
```

must be modified to read

```
interpret([]) :- !.
interpret([_Goal, _Goals..]) :-
    interpret(_Goal),
    interpret([_Goals..]).
interpret(_Goal(_X..)) :-
    clause(_Goal(_X..) :- [_Body..]),
    interpret([_Body..]).
```

(Note that the BNR Prolog predicate `clause` takes one argument while the Edinburgh version takes two.) The cut in the second clause in the Edinburgh program is unnecessary in the BNR Prolog version.

The use of BNR Prolog lists to represent the sequence of arguments to a predicate allows for the definition of predicates with a variable number of arguments. For example, it is possible to write `interpret` so that it takes a variable number (zero or more) of arguments, in much the same way as the recursive list definition is written. Note the changes in the argument lists from the previous definition.

```
interpret() :- !.
interpret(_Goal(_X..)) :-
    clause(_Goal(_X..) :- [_Body..]),
    interpret(_Body..).
interpret(_Goal, _Goals..) :-
    interpret(_Goal),
    interpret(_Goals..).
```



The convenience of variadic predicate definitions should become evident when using the BNR Prolog input/output primitives as well as the basic type filters.

The possibility that a predicate may have any number of arguments also lessens the importance of the concept of arity in BNR Prolog. Predicates are therefore referred to by name rather than by name and arity.

In BNR Prolog, for the sake of consistency, predicates that take no arguments should be written with an empty argument list. For convenience, however, the parser accepts 0-arity predicate definitions without the empty argument list. Thus,

```
test :- write('hello'), nl.
```

is coerced to

```
test() :- [write('hello'), nl].
```

Observe that only the clause head `test` is coerced to be the 0-arity functor, the symbol `nl` in the body of the clause remains unchanged. It is only when symbols like `nl`, `test`, `true` or `fail` are called as goals that they are interpreted as calls to the 0-arity predicates `nl()`, `test()`, `true()` or `fail()`, respectively.

## Operators

Operator declarations in BNR Prolog are facts in the clause space and may be asserted and queried like any other predicate. A symbol may only have one operator precedence number associated with it. For example, the infix operator

```
op(1000, xfy, '&').
```

cannot also be declared prefix with a different precedence number, such as

```
op(900, fx, '&').
```

The only other restriction is that an operator cannot be defined as both prefix and postfix.



The increased importance of arithmetic in BNR Prolog has resulted in a minor adjustment in operator definitions. The Edinburgh arithmetic equivalence operator, "=:=", has been changed to "==", as in the C programming language. This has a small ripple effect on some of the other operators, as summarized in the following table:

Function	Edinburgh Prologs	BNR Prolog
literal identity	==	@=
literal non-identity	\==	@\=
arithmetic equality	:=	==

The operators "=:=", "=\<=", and "\==" are defined in the Edinburgh compatibility file as equivalent to "==", "<>", and "@\<=" respectively. Therefore only Edinburgh "==" must be changed to "@=". Other operators which are predefined in BNR Prolog but not often found in other Prolog systems are

```

op(1000, xfy, '&')           % conjunction
op( 950, xfx, 'where')       % constraints
op( 950, xfx, 'do')          % used with foreach
op( 700, xfy, ':')           % external parameter typing

```

Note that there are no operator declarations for "."(cons) or "=". (univ) or ", "(and).

## Comma as an Operator

One of the consequences of using the list structure for clause bodies is that the comma is not an operator. Therefore, all operators, regardless of their relative precedence, bind tighter (have higher precedence) than comma. For example, the following expression, which has no precedence enforcing parentheses,

```

breakfast :-
    english -> tea ; coffee,
    oj,
    milk.

```



is interpreted by the BNR Prolog parser as:

```
breakfast() :-  
    [  
        ((english -> tea) ; coffee),  
        oj,  
        milk  
    ].
```

Parentheses around terms separated by ", " are not necessary, because operator precedences need not be overridden.



# Appendices








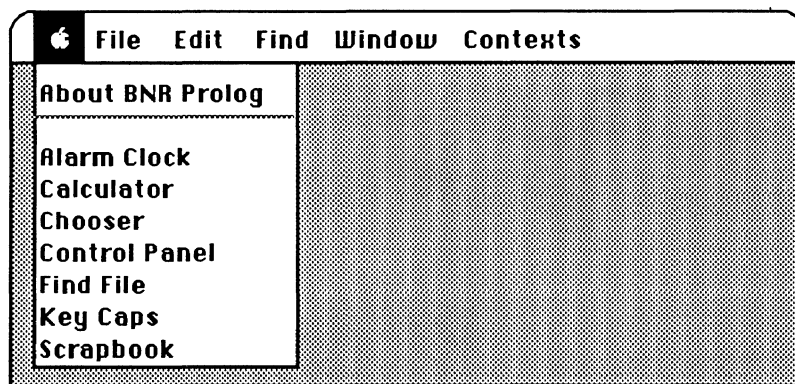
# Appendix A: The Prolog Desktop


---

The desktop provides a set of predefined pull down menus for access to the desk accessories, file handling, editing, managing windows, and consulting files. In addition to these menus, users can take advantage of the man/machine interface capabilities to define their own menus or modify the standard ones. Some menu options may be invoked by predefined Command keys specified with  $\mathcal{K}$  in the right hand side of each option.

## (Apple) menu

The  menu provides access to your Macintosh desk accessories and information about the installed version of Prolog.

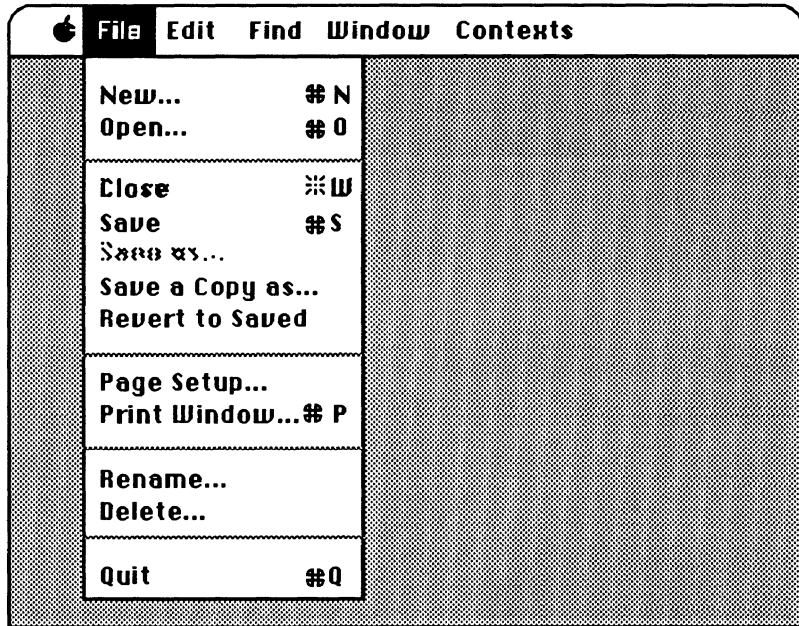


 menu



## File menu

The **File** menu provides several options for handling files and specifying printing parameters.



**File menu**

The **File** menu options are:

- New...** creates a new text window for a new text file. The file has default attributes. A dialog is presented where the default name and folder can be changed.
- Open...** provides a dialog that permits selection of an existing text file to be opened with default attributes. The newly opened file becomes the currently active text window. Note that opening a file does not add the contents of the file to the knowledge base.



- Close** closes the currently active text window and activates the previous active window. A dialog provides the option of updating the file on disk if there are any outstanding changes. If the file has a foreign creator, an additional dialog is invoked to confirm the intent to overwrite the original file.
- Save** updates the disk version of the file of the currently active window. This option is enabled only if there are outstanding changes. If the file has a foreign creator, an additional dialog is invoked to confirm the intent to overwrite the original file.
- Save as...** stores a version of the file of the currently active window on disk. A dialog permits the user to specify both the file name and the folder. The original window is closed without updating, and a new window is opened for the saved file.
- Save a Copy as...** stores a version of the current text window on disk. A file dialog permits the user to specify both the file name and the folder. The current window is not affected.
- Revert to Saved** replaces the contents of the currently active text window with the version of the file that is stored on disk. This option is enabled as soon as the currently active window diverges from the saved version.
- Page Setup...** allows specification of printing parameters for the current Prolog session. This option has no effect on the setup of the disk version of open text windows. Default parameters are  
US Letter, Font Substitution, Smoothing,  
Fasterbitmap, Portrait, 100%  
reduction/enlargement.
- Print Window...** prints the contents of the currently active text window, presenting a dialog permitting specification
-



of copies, pages, cover page and paper source information.

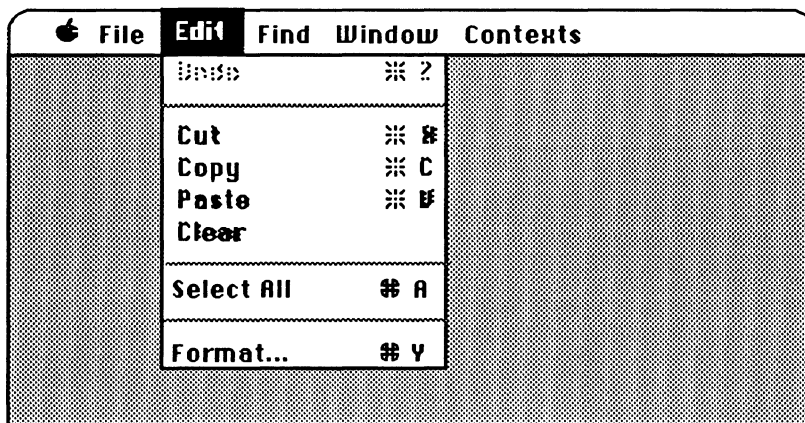
**Rename...** permits renaming of any disk file, and its associated text window, if any.

**Delete...** permits deletion of any disk file. If the file is currently open in a text window, a dialog queries for verification of the closure and deletion.

**Quit** provides an exit from the Prolog system. Dialog boxes are provided to confirm disk updates for text windows with outstanding content changes.

## Edit menu

The **Edit** menu provides selection, formatting, and cut and paste options. Selected strings can be transferred between windows



**Edit menu**

The **Edit** menu options are as follows:

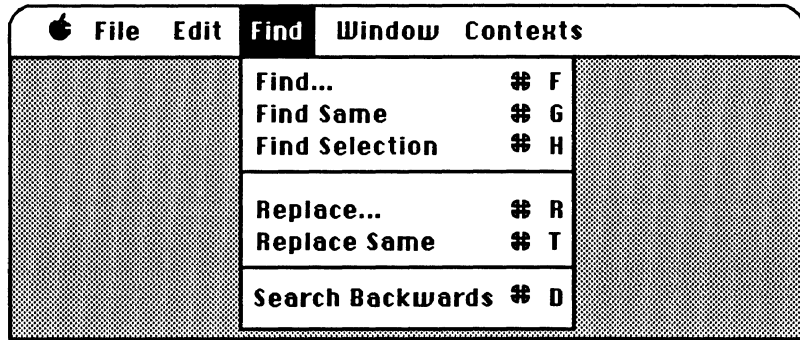


- Cut** deletes the current selection from a text window and places it on the Clipboard. This option is enabled once a piece of text is selected.
- Copy** copies the current selection on the Clipboard. This option is enabled once a piece of text is selected.
- Paste** copies the contents of the Clipboard into a text window at the position of the cursor, replacing any existing selection. This option is enabled immediately after a cut or a copy operation.
- Clear** deletes the current selection from a text window.
- Select All** selects all text in the currently active text window.
- Format...** provides a dialog that permits font selection, as well as specification of font size, tab size, and tab/space conversion. If a file is created elsewhere, tab characters that exist in the text appear as spaces. Use of the *tab* key generates the specified numbers of space characters.

## Find menu

The **Find** menu provides several searching and editing options. All options operate on the currently active window based on three search parameters: search string, search direction, and case sensitivity. The **Find...** and **Replace...** commands provide dialogs in which the search parameters can be specified. The **Replace...** command also permits specification of a replacement string. A search always starts from the current position of the cursor in the specified direction.





### Find menu

The **Find** menu options are as follows:

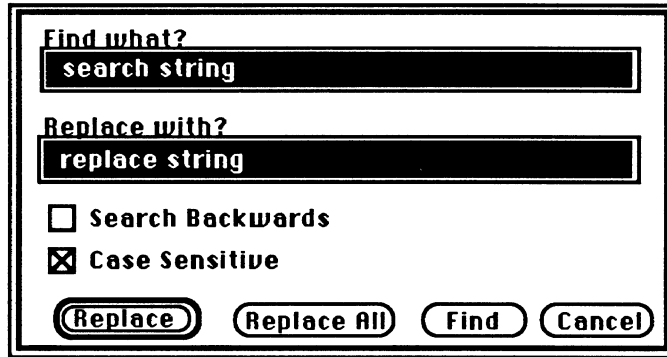
**Find...** displays a dialog box to allow specification of the search parameters. It then locates the next occurrence of the specified search string from the current position of the cursor.

**Find Same** locates the next occurrence of the previously specified search string.

**Find Selection** replaces the existing search string with the currently selected text, and locates the next occurrence of the new search string.

**Replace...** displays a dialog box to allow specification of the search parameters, the replacement string, and the desired operation. The operation is then performed, starting from the current position of the cursor.





**Replace** dialog box

**Replace Same** replaces the existing search string with the replacement string.

**Search Backwards** is a toggle command that is used to specify the search direction parameter. A backward search is enabled if the menu item is check marked. The default is a forward search. This parameter is also toggled with the Search Backwards option in **Find...** or **Replace...** dialogs.

## Window menu

The **Window** menu provides control of currently open windows. Windows can be selected by using this menu or by clicking in a visible portion of the window. The current default output window, can also be accessed through use of the *Command -K* sequence. The menu consists of three sections: the window commands, the text window list, and the graphics window list. Window commands are as follows:

**Tile Text Windows** reorganizes text windows into a set of tiles that do not overlap. This provides window visibility quickly without shifting all the windows and readjusting their sizes individually. Desk



accessories and graphics windows are not affected by this command.

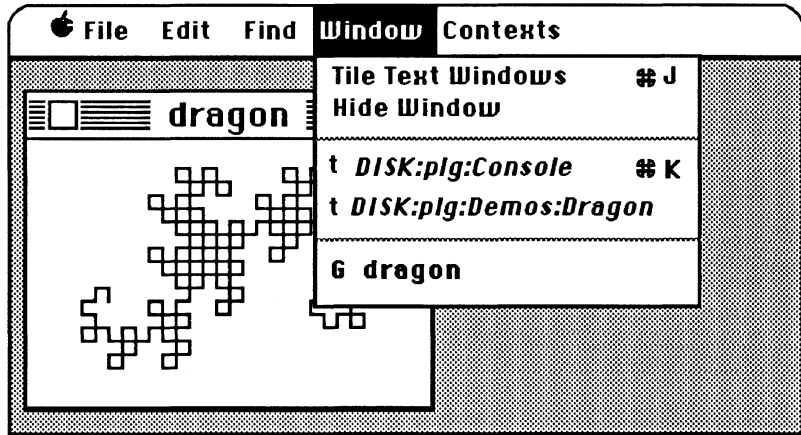
**Hide Window** makes the currently active window invisible, and the window beneath it becomes active. Selecting hidden windows from the menu reveals them.

The window lists display information about all open windows. The following notation is used to represent window attributes:

- underlined window name signifies outstanding changes
- **t** to the left of a window name signifies a text window
- **g** to the left of a window name signifies a graphics window
- the window name in bold font signifies the currently active window
- the window name in italic font signifies the window is hidden

In the following example, `Console` is the default output window, and `Dragon` exists as both a graphics and a hidden text window. The `Dragon` graphics window is the currently active window.





Window menu

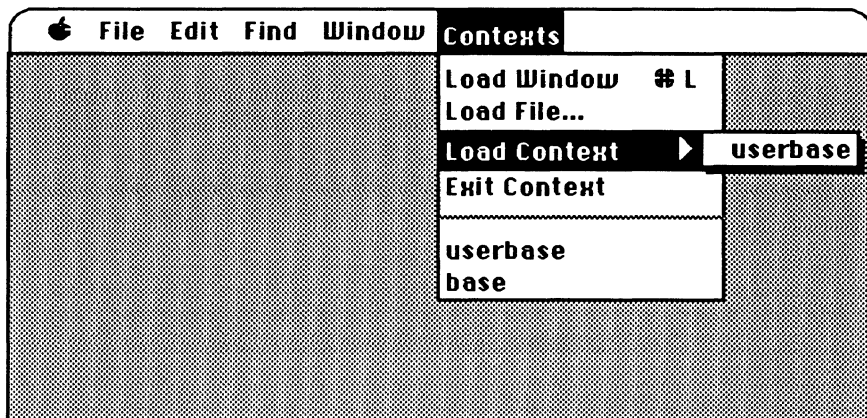
## Contexts menu

Prolog stores clauses (facts or rules), in memory in a stack like structure of modules called contexts. The current context is the top of the stack. Contexts are created either by loading a window or a file, or by creating a dynamic context.

For example, in the following picture, selecting the family context from the **Load Context** submenu causes the deletion of the existing contexts for both family and Picture, followed by the reloading of first family and then Picture.

When loading a file or a context for which there is an open text window, the window version of the file is loaded. Reloading a context whose original window was created by an `enter_context` command is not possible. See the chapter "Contexts" for more detailed information on contexts.





### Contexts menu

The Contexts menu consists of two sections, the context commands and the contexts list. The commands are as follows:

**Load Window** loads the contents of the currently active window.

**Load File...** loads the contents of the currently active window into a new context, or reloads an existing context from the currently active window.

**Load Context** permits selection and reloading of existing contexts from a submenu.

**Exit Context** permits selection and removal of existing contexts from a submenu..

The context list is an ordered list of all existing contexts starting with the current context. Clicking on a context name displays a dialog box that permits selection of clauses in that context for listing in the output window.



## Menu Command Shortcuts

The following is a list of the Command key shortcuts that are available in BNR Prolog, along with comments on their use.

<i>Command-A</i>	<b>Select All</b> , from the <b>Edit</b> menu
<i>Command-C</i>	<b>Copy</b> , from the <b>Edit</b> menu
<i>Command-D</i>	<b>Search Backwards</b> , from the <b>Find</b> menu
<i>Command-F</i>	<b>Find...</b> , from the <b>Find</b> menu
<i>Command-G</i>	<b>Find Same</b> , from the <b>Find</b> menu
<i>Command-H</i>	<b>Find Selection</b> , from the <b>Find</b> menu
<i>Command-J</i>	<b>Tile Text Windows</b> , from <b>Window</b> menu
<i>Command-K</i>	current default output window, from <b>Window</b> menu
<i>Command-L</i>	<b>Load Window</b> , from the <b>Contexts</b> window
<i>Command-N</i>	<b>New</b> , from the <b>File</b> menu
<i>Command-O</i>	<b>Open</b> , from the <b>File</b> menu
<i>Command-P</i>	<b>Print Window</b> , from the <b>File</b> menu
<i>Command-Q</i>	<b>Quit</b> , from the <b>File</b> menu
<i>Command-R</i>	<b>Replace</b> , from the <b>Find</b> menu
<i>Command-S</i>	<b>Save</b> , from the <b>File</b> menu
<i>Command-T</i>	<b>Replace Same</b> , from the <b>Find</b> menu



<i>Command-V</i>	<b>Paste</b> , from the <b>Edit</b> menu
<i>Command-W</i>	<b>Close</b> , from the <b>File</b> menu
<i>Command-X</i>	<b>Cut</b> , from the <b>Edit</b> menu
<i>Command-Y</i>	<b>Format...</b> , from the <b>Edit</b> menu
<i>Command-Z</i>	<b>Undo</b> , from the <b>Edit</b> menu

## Editing Keys

BNR Prolog supports a number of keys for editing the contents of text windows. The following is a list of the key commands that are not shown on the **Edit** menu. A list of the menu command shortcuts can be found in the chapter "Command Shortcuts".

*delete*            removes the character to the left of the cursor when there is no selection. Otherwise the selected text is removed.

*Command-delete* deletes the text from the cursor to the top of the file. If text is selected, the cursor is assumed to be to the left of the selection.

*clear*            deletes the character in front of the cursor when there is no selection. Otherwise the selection is removed.

*Command-clear* deletes the text from the cursor to the end of the file. If text is selected, the cursor is assumed to be to the right of the selection.

*Command—* displays the first page and places the cursor at the beginning of the file.

*Command-+* displays the last page and places the cursor at the end of the file.



*Command-*↓ places the cursor at the start of the last line of text in the window. If already on the last line, the file is paged ahead, and the cursor is placed on the last line of the next page of the file.

*Command-*↑ places the cursor at the start of the first line of text in the window. If already on the first line, the file is paged back, and the cursor is placed on the first line of the previous page of the file.

*Command-*← moves the cursor to the beginning of the current line.

*Command-*→ moves the cursor to the end of the current line.

## Execution Control Keys

*Command-*. aborts execution if request confirmed by a dialog

## Methods of Text Selection by Mouse

*double click* to select the word on which the mouse is positioned.

*double click on a left or right parenthesis, brace, or bracket* to select all text up to the matching parenthesis, brace, or bracket. Nested occurrences of these characters are properly handled.

*double click on the leftmost single or double quote* to select all text up to the next quote.

*triple click* to select the line on which the mouse is positioned.

*shift click* to extend or shorten an existing selection to the current position of the mouse.







# Index







## \$

\$initialization 192, 195  
\$local 207

## A

accumulate 166  
acyclic 60, 104  
Algebra  
    Boolean 67, 74  
altdboxproc 219  
append 44, 293  
Application  
    concurrent 215  
    exiting 27  
Argument  
    actual 39  
    formal 39  
    instantiation of 40  
Arithmetic  
    evaluation of 120, 139  
    functional 119-125  
    operators 97  
    relational 131, 157-171  
        solving equations 161  
        summary 171  
    validation of formulae 121  
assert 107, 197, 201  
asserta 197  
assertz 197  
at 180  
attention\_handler 275

## B

Backtracking 15, 32, 40-43, 85

    prevention of 84  
    with intervals 137  
base 191  
beginpicture 231  
Behavior  
    nonlogical 119  
block 83  
BNR Prolog  
    development environment :  
    icons 19  
    using 19-27  
bucket 61  
build\_application 268

## C

Call 32  
    mechanism of 14, 35  
    reduction step of 14, 35, 40  
    unification step of 35, 39-40  
capsule 274  
Choicepoint 40  
    removal of 81  
choose 50  
circuit 48  
Clause 13  
    body of 34  
    definition access 196  
    external 265  
    head of 34  
    local 194  
    name of 34  
    ordering 41, 46, 194  
    overloading 194  
    syntax of 34, 300  
Clause space 16  
close 179, 182  
close\_definition 195  
closewindow 221  
closure 93



Code Resource 247

Command Key 307

  + 318

  - 318

  . 23, 319

  A 317

  C 317

  clear 318

  D 317

  delete 318

  F 317

  G 317

  H 317

  I 317

  J 317

  K 313, 317

  L 317

  N 317

  O 317

  P 317

  Q 317

  R 317

  S 317

  T 317

  V 317

  W 317

  X 317

  Y 317

  Z 317

  ← 318

  ↑ 318

  → 318

  ↓ 318

Comment

  nested 32

  syntax of 32

Comparisons

  arithmetic 90

Conditions

  efficient 90

  ground 90

  joint 14

configuration 262

confirm 236

Conjunction 32, 47

  of passive constraints 69

Constraint

  active 70, 89-95, 125

  data flow 70, 126

  passive 69-79, 125

    backtracking with 71

    common constructs 73

    interpolation 72

    joint 75

  syntax of 70

Context 190

  creation of 315

  current 190

  dynamic 315

  exiting 26, 193

  lists 316

  loading of 269

  new 191

  predicate local to 194

  reloading of 192, 315

  userbase 262

continue 287

cputime 265

cut 81

  ancestral 82

  list 83

  negative effects of 82

## D

dbxproc 219

debug 280

Debugging 277-289

  break 287

  commands 280



- creep 279, 281
- depth number 278
- invocation identifier 278
- leap 279, 284
- port
  - call 277-279
  - exit 277-279
  - fail 277-279
  - redo 277-279
- skip 279, 286
- spypoint 279, 284
- decompose 104, 112
- definition 196
- delta 135
- Demos 4
- Dependency
  - functional 91-95
- Dialog 236-238
  - types of 236
- dinner\_party 76
- Disjunction 46
  - of passive constraints 69
- Document
  - saving of 27
- documentproc 220
- dograf 226, 231
- dotext 222

## E

- enablemenus 234
- end\_of\_file 180, 181
- endpicture 231
- enter\_context 191, 315
- Error
  - management of 274-275
- Event 213-218
  - asynchronous 212
  - blocking 216
  - handler

- debugging of 288
- handler of 213-218, 236
- idle 217
- listener for 218
- listener of 213
- mouse 229
- nonblocking 216
- polling for 212, 213-218
- priority of 217
- synchronous 212
- Execution
  - deferral of 81
- exit\_context 193
- External Procedure
  - definition of 248
  - interface to 247
  - parameters of 249

## F

- Fact 13
  - arguments of 34
  - syntax of 34
- fail 302
- failexit
  - list 83
- failexit(ancestral) 82
- Failure 86
  - call 14
  - goal 81
- File
  - binary image of 192
- Filter 59
  - complex 66
  - general 64
  - monotone 60
  - persistent 60, 89
- findall 107
- float 62, 73
- Floating point



- accuracy of 132
- syntax of 32
- foreach 85
- Foreign Language 247
- forget 204, 207
- forgetz 204
- fullfilename 224
- Functor
  - arity 297, 302
  - principal 38, 97, 298
  - syntax of 297
  - variable 52

## G

- Generate
  - and test 74
- Generator 70
  - loop 84
- get\_char 186
- get\_error\_code 275
- get\_term 182
- Goal 13
  - deterministic 82
  - ordering 50
  - parent 81
- Goal stack 16
- ground 63, 125

## H

- halt 264
- hide 197
- horner 121

## I

- if-then 81-84

- Inference 14
  - procedure 15
- Input
  - coercions on 33, 34, 35, 295, 30
- Input and Output 174
  - error detection 182
  - stream directed 178-182
    - default 181, 183
    - pipe 178, 181
    - pointer 179
  - symbol 178
  - symbol directed 184, 185, 186
  - target 178
  - term 182-187
  - text 177-187
- inqgraf 226
- inqtext 223
- Instantiation
  - process of 13
- Integer 62
  - syntax of 32
- Interface
  - C 249
  - dialog 212
  - graphics 73
  - menu 212
  - menu/mouse 20, 22, 27
  - modal 212, 236
  - parameters 255
  - Pascal 249-257
  - resource name 257
  - user 211-244
  - window 212-213
- Interval
  - coercion 139, 149
  - constraint on 140
  - disjunction of 145
  - equality of 140
  - indefinite 134
  - inequality of 146



- intersection of 145
- narrowing of 137-141
- numeric interpretation of 133, 141
- open 146
- precision of 150
- range of 131
- characteristics of 131
- regional interpretation of 133, 141
- the data type 133
- unknown as 151

interval 62

inventory 206

is 120, 149

## J

## K

Key

- Command 20
- enter 19, 20, 23
- return 24

Knowledge

- negative 75

Knowledge Base 13, 16, 23, 39, 41, 174, 189-199, 300

- context in 190

## L

Law

- associative 55, 132
- commutative 51, 69-73
- distributive 39
- equivalence 35
- idempotent 55
- reflexive 35, 55

- symmetric 35, 55
- transitive 35

lipsrate 266

List 62, 73

- argument 38, 300
- as clause body 300
- element of 42
- empty 34, 293
- indefinite 38
- proper 292
- syntax of 33
- tail of 33

Listener 16, 24

- activity box 20

Listing 41

- by menu 26
- by query 25

listing 197

load\_context 191

load\_state 206

Logic

- formal 12
- Horn clause 7, 31, 53
- programming 7
- symbolic 12

## M

Macintosh

- file
  - characteristic of 266

Quickdraw 213

resource 261, 267

- BNDL 270
- FREF 270
- ICN# 269
- ICON 269
- menu 233
- PICT 231



Macintosh Programmer's  
Workshop 271

maze 87

median 164

member 42, 113

Memory

    allocation of 261

memory\_status 265

Menu 232-236

    Contexts 24, 26, 315

        load file 316

    Edit 310

        clear 311

        copy 310

        format 311

        paste 310

        select all 311

File 22, 27, 308

    close 309

    delete 309

    new 308

    open 308

    page setup 309

    print window 309

    quit 309

    rename 309

    revert to saved 309

    save 309

    save a copy as 309

    save as 309

Find 311

    find 312

    find same 312

    find selection 312

    replace 312

    replace same 313

    search backwards 313

heirarchical 233

identification of 233

item in 233

pop-up 233

Window 22, 313

    hide window 314

    lists 314

    tile text windows 313

Apple 307

menuselect 217, 233

message 236

messagebutton 222

midpoint 135

Module

    characteristics of 189

## N

namefile 236

Negation

    by failure 66-67

    classical 67

    sound 75

new\_state 202, 207, 264

nl 186, 302

nogrowdocproc 220

nonvar 61, 73

not 64, 75, 107

numeric 73

## O

occurs check 53, 103

once 83

op 52, 302

open 178, 182

openwindow 219, 225

Operation

    conditional 84-86, 89-90

Operator 97-101

    arithmetic 119-125

    comparison 119-125



associativity of 99  
 comma as 303  
 declaration of 52, 98-100, 302  
 infix 98-100  
 postfix 98-100  
 precedence of 52, 302, 303  
 predefined 101  
 prefix 96-98  
 restrictions on 100, 302  
 type of 99  
 or 81

## P

Parameter  
   passing of 13  
   search 311  
 permutation 50  
 PEXT 247  
 PERT plan 126  
 picttoscrap 232  
 Picture 232  
   frame of 231  
   storage of 231  
 plaindbox 219  
 portray 107, 186  
 print 186  
 print\_interval 136  
 Predicate 34  
   Calculus 31  
   central 69  
   nondeterministic 40  
   side effect of 59, 66, 72  
   variadic 34, 302  
 print 107  
 Problem  
   critical path scheduling 126, 147  
   finite state machine 108  
   gas law 155  
   magnetism 160

silly 271  
 square 157  
 Temperature Conversion 1  
 timetable 142  
 Program  
   branch  
     elimination of 81-83  
   nonterminating 81  
   Pascal 250-257  
 Programming Model 11  
   procedural 11  
 Prolog  
   computational model of 12  
   Edinburgh 291  
   extensions of 8  
   formal properties of 54  
   fundamental principle of 1  
   pure 7, 31, 46-57, 61, 69, 81, 1  
   sequential 41  
   text books 8  
   versus procedural languag  
 Property  
   commutative 82, 126, 139  
   idempotent 56, 82, 139  
   monotone 53, 56-57, 69, 139  
   narrowing 15, 54  
   persistent 53, 56-57, 69, 139  
 put\_char 186  
 put\_term 182, 185

## Q

query 236  
 Question 14  
   narrowing of 15  
   prefix 16  
 quit 264



## R

- range 133, 136
- rdocproc 220
- read 183
- readln 187
- Real
  - accuracy of 132
- recall 203, 207
- recallz 204
- recovery\_unit 274
- Recursion 43
  - stopping 44
- reduce 39, 41, 51, 122
- reduce\_sym 51, 63
- Reduction
  - process of 13
- remember 107, 203, 207
- replay\_events 288
- Requirements
  - hardware 3
  - software 3
- restart 263
- retract 199, 201
- retract\_first 199
- Rule 13
  - arguments of 34
  - body of 14
  - syntax of 34

## S

- save\_state 206
- save\_ws 264, 269
- Search
  - depth first 54
  - exhaustive 15
  - tree 54

- seek 180
- select 236
- selectafile 236
- Selection
  - of file 24
  - of text 19, 20, 23, 25
  - of window 23
- selectone 236
- Sequence
  - as a structure 296
- set\_end\_of\_file 180
- set\_trace 269
- Side Effect 11, 73, 174, 211
  - freedom from 54, 59, 89
- solve 165
- spanning\_tree 105, 114
- split 162
- spy 285
- spyvar 72
- sread 183
- Stack
  - configuration of 262
  - global 261-263, 265
  - local 261-263, 265
  - world 190, 261-263, 265
- state 265
- State Space 174, 201-208, 264
  - creation of 203
  - global 202-206
  - loading of 206
  - local 207-208
  - recall order in 202
  - removal from 204
  - storage in 203
  - storage management 206
  - updating of 205
- Strategy
  - backtracking 71
- stats 266
- stream 179



Structure 13, 62, 296  
  cyclic 39, 54, 103-116  
  representation 105  
  syntax of 33  
swrite 183  
swriteq 183  
Symbol 62, 73, 193  
  as an operator 97  
  syntax of 32

## T

tailvar 61  
Term  
  comparison of 65  
  ground 35, 55, 64, 71  
  infinite 39  
termlength 66, 297  
Test  
  and generate 74  
timer 265  
trace 280  
trace\_event 288  
transform 52  
true 302  
Tutorials 4  
Typographic Conventions 6

## U

Unification  
  cyclic structure 103  
  filters and 60  
  implicit 39  
  of ground terms 35  
  of lists 36, 37  
  of structures 37  
  of tail variables 38, 295  
  of variables 36

  operator 35, 39  
  passive constraints and 69  
  process of 13  
  properties of 57  
update 205  
useractivate 217  
userbase 190  
userclose 217  
userdeactivate 217  
userdownidle 217, 230  
userdrag 217, 220  
userevent 216  
usergrow 217, 220  
userkey 217, 218, 222, 224  
usermousedown 217, 225, 230  
usermouseup 217, 225, 230  
userupdate 217, 226  
userupidle 217, 230  
userzoom 217, 220

## V

var 61, 73  
Variable  
  anonymous 33  
  binding of 40  
  constraint 71  
  functor 297  
  leading underscore of 33  
  syntax of 33  
  tail 38, 294, 295  
  syntax of 33

## W

Window 219-231  
  Console 22  
  current 314  
  frame 219



- options of 222
- graphics 225-231, 314
  - coordinates of 226
  - descriptor
    - nesting of 227
  - descriptor of 226
  - rubber lines in 228
- query through 23
- text 22, 178, 222, 314
  - descriptor of 222
  - file associated with 222
  - loading of 24
  - reloading of 26
  - selection by mouse 319
  - tiling of 22
- text entry through 23
- type 221

- Work Space 264
- write 183
- writeq 183

## **X**

## **Y**

## **Z**

- zoomdocproc 220
- zoomnogrow 220





User Guide

© Bell-Northern Research Ltd. 1988

All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language by any means without the written permission of Bell-Northern Research Ltd.

Printed in Canada



