4.900

User Guide

by Rebecca Shalfield

WIN-PROLOG User Guide

The contents of this manual describe the products, version 4.900 of *LPA-PROLOG* for Windows (hereinafter called *WIN-PROLOG*), major toolkits (including flex for Windows and Prolog++ for Windows) and several miscellaneous utilities, and are believed correct at the time of going to press. They do not embody a commitment on the part of Logic Programming Associates (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Authors: Rebecca Shalfield, Clive Spenser, Brian Steel and Alan Westwood

Logic Programming Associates Ltd.
Studio 30
Royal Victoria Patriotic Building
Trinity Road
London
SW18 3SX
England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
web site: http://www.lpa.co.uk

LPA-PROLOG and *WIN-PROLOG* are trademarks of LPA Ltd., London England.

18 May, 2010

Contents

FIGURES

About This User Guide

This user guide provides the starting reference for LPA's set of tools for Windows - version 4.900 of ***LPA-PROLOG*** for Windows (hereinafter called ***WIN-PROLOG***), toolkits (including VisiRule, Flex and Prolog++ for Windows) and miscellaneous utilities.

This user guide describes how to install ***WIN-PROLOG*** and related toolkits onto your system. It describes how to use the core integrated development environment, IDE, provided by LPA, including the editor, debugger and compilers. It then goes on to describe how to use various utilities that accompany the core Prolog such as DDE and OLE Automation, and tools, such as the Dialog Editor and Call Graph. Finally, it covers how the basics of additional environments provided with the flex and Prolog++ toolkits, though some of these, such as VisiRule, are described more fully in their own dedicated manuals.

The goals of this manual are to enable you to install the LPA software, launch it and navigate your way round the menu system, and compile and run basic examples.

This user guide is divided into various parts:

WIN-PROLOG - describes how to install and run ***WIN-PROLOG*** and the use of its environment and menus. This section also describes how to install toolkits for use with ***WIN-PROLOG***.

Utilities - describes how to use the supplied utilities:

OLE Automation - allows ***WIN-PROLOG*** to control and communicate with other applications via OLE Automation.

DDE - allows ***WIN-PROLOG*** to communicate with other applications via DDE.

Call Graph - allows the graphical viewing of the code structure.

Dialog Editor - lets you generate code for dialogs using a graphical editor.

Cross Reference - shows the relationships between the Prolog programs currently loaded.

flex for Windows - how to run Flex and use its environment and menus.

Prolog++ for Windows - how to run Prolog++ and use its environment and menus.

Appendices - contains information on:

A - setting ***WIN-PROLOG***'s command line switches.

B - ***WIN-PROLOG*** and memory management.

C - Types of compilation.

D - Associating *WIN-*PROLOG files with the application.

Simple examples are provided throughout this user guide, showing you how to perform basic actions.

More detailed information about the Prolog system itself can be found in the *Technical Reference* and *Programming Guide*. There are also four Prolog tutorials supplied the LPA software.

More detailed information about the Flex system itself can be found in the *Flex Manual* and *Flex Tutorial*.

Detailed information about VisiRule can be found in the *VisiRule Manual* and *VisiRule Tutorial*.

Please enjoy reading this user guide, and have fun with ***WIN-*PROLOG** and related toolkits

LPA, 18 May, 2010

What's in *WIN-*PROLOG?

Welcome to version 4.900 of ***WIN-PROLOG,*** the latest release of LPA's acclaimed Prolog compiler and related toolkits.

For the dedicated Prolog programmer, ***WIN-PROLOG*** provides a complete development environment; including easy-to-use pull-down menus, support for multiple edit windows, rich edit (e.g. automatic syntax colouring, multiple fonts in a window, etc.), comprehensive text search and replace facilities, and source level debugging and true support for Unicode on all versions of Windows. ***WIN-PROLOG*** has incremental and optimised compilation, together with a hashed compilation mode which allows a matching clause to be found almost instantly. Being a genuine 32-bit application, ***WIN-PROLOG*** provides access to as much of your machine's memory as you want.

WIN-PROLOG allows you to create polished Windows applications; it provides an extensive range of graphics predicates, allowing convenient access to a large number of Windows Graphical User Interface (GUI) functions. All of the GUI features used by the environment, and more besides, are directly available to Prolog programs, allowing customised environments to be built and shipped as part of an application. There's even a Dialog Editor to draw the screens and generate the Prolog code.

For those wishing to develop the front-end to their application in another language, be it C, C++, Delphi, Java, Visual BASIC, C#, Pascal or any other Windows development language, ***WIN-PROLOG*** can be used to create the artificial intelligence-based back-end. The front-end and back-end can then communicate via the LPA Intelligence Server.

WIN-PROLOG conforms fully to the Edinburgh syntax, which is, itself, largely compatible with the ISO Prolog standard. In order to provide full compatibility with Quintus Prolog, a number of Clocksin and Mellish* predicates have also been augmented. A special string data type and associated predicates permit powerful file, window and other input/output (I/O) processing not normally possible in Prolog.

For the more advanced Windows programmer, ***WIN-PROLOG*** includes predicates for the calling of Dynamic Link Library (DLL) functions. Any DLL, written in C, C++, Delphi, Java, Pascal, Visual BASIC or any other Windows development language, can be loaded by ***WIN-PROLOG***. Utilising ***WIN-PROLOG***'s extensive range of data types and I/O functions, DLLs can exchange all types of data with Prolog programs. Furthermore, once loaded, DLLs can send messages to Prolog at any point, not just when they are being called, thereby allowing background processing, modeless dialogs and inter-process communication to be built in easily.

Chapter 1 - Installing *WIN-*PROLOG and Toolkits

This chapter describes how to install ***WIN-*PROLOG** and related toolkits, and how to set up your Windows environment to make best use of them.

Why Must *WIN-*PROLOG and Related Toolkits be Installed?

There are three main reasons why the ***WIN-*PROLOG** system (***WIN-*PROLOG** itself and any related toolkits) must be installed before it can be run. Firstly, the software is supplied on CD-ROM in a compressed form. Secondly, your name must be registered in the software: this helps to distinguish your copy of ***WIN-*PROLOG**, which you may have customised, from any other copies at your site. Thirdly, the ***WIN-*PROLOG** system consists of a large number of files, containing various modules and optional extensions, which need to be accessed by the system at different times.

The ***WIN-*PROLOG** system is installed onto your hard disk via use of the supplied industry-standard-based LPA Setup Program. This program, when run, decompresses the files, registers your name in the software, sets up the required directory structure on your hard disk and, finally, copies the relevant files into those directories.

Requirements for *WIN-*PROLOG

***WIN-*PROLOG** can be installed on a system running any of:

Microsoft Windows 7

Microsoft Windows Vista

Microsoft Windows XP

Microsoft Windows 98/ME

Microsoft Windows NT 4.0/2000/XP

***WIN-*PROLOG** requires at least 64Mb of free memory (RAM) over and above Windows itself, though more is recommended. An additional 32Mb should be allowed for if running either flex for Windows or Prolog++ for Windows.

The necessary support for ***WIN-*PROLOG**'s Rich Edit 3.0 features is included in later versions of Windows (2000, ME, XP, Vista, Windows 7). For your convenience, two of the latest versions of these Rich Edit 3.0 files may be included with ***WIN-*PROLOG** version 4.500 onwards:

RICHED20.DLL	version 5.30.23.1210
MSLS31.DLL	version 3.10.349.0

Please note: ***WIN-PROLOG*** may not run if the correct Rich Edit DLLs are not installed.

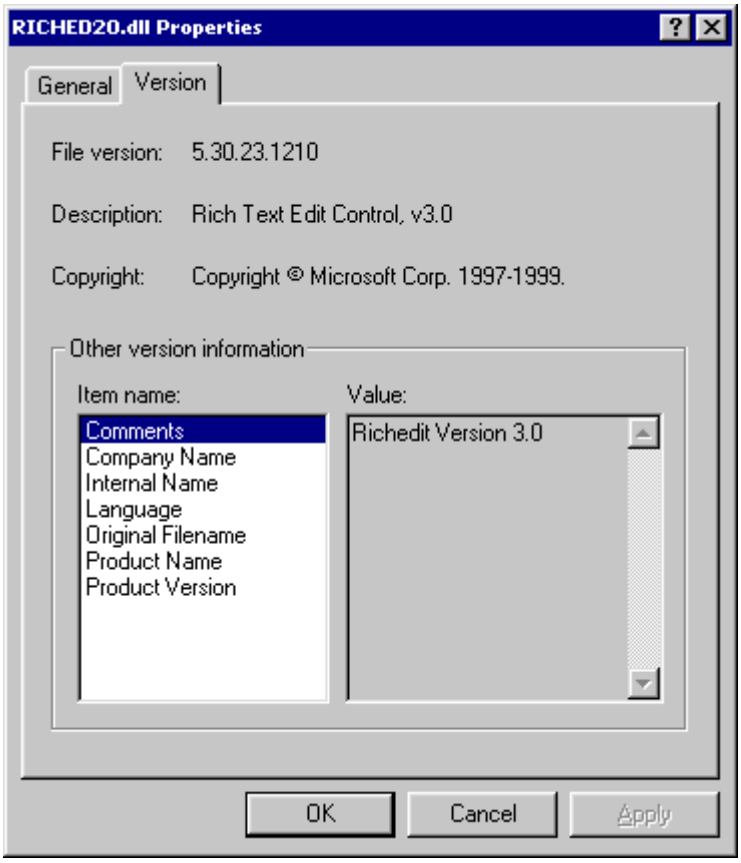


Figure 1 – A Properties dialog showing the version number of a DLL

Running the LPA Setup Program

Insert the LPA CD into your CD-ROM drive, and run the LPA Setup program (SETUP.EXE) on that disc. You can do this by simply double-clicking on the the .exe file or by using the Start/Run menu option.

In the example below, it is assumed that your CD-ROM drive is drive E:.

: Click on the Start/Run menu option to display the Run dialog.

Type the following command in to the "Open:" box:

e:\setup *<enter>*

In this, and all other examples, only type the characters shown in **bold** letters, and press the named key for anything bracketed in *<italics>*.

After a few seconds, the Setup program's welcome screen will be displayed, whereupon follow the on-screen instructions and read the relevant sections below.

Responding to the LPA Setup program

Most of the installation of ***WIN-PROLOG*** is automatic: at each stage, you will typically be presented with a dialog containing two buttons, labelled "Next >" and "Cancel". If you click on "Next >" or press *<enter>*, you will be taken on to the next stage of installation; otherwise, you can choose to abort installation at any point by clicking on "Cancel" or pressing *<esc>*. If you choose to abort, you can start the installation process again whenever you want to, by returning to the steps outlined in the previous section. Should you wish to return to an earlier screen to correct a previous entry, this can be achieved by clicking on the "< Back" button.

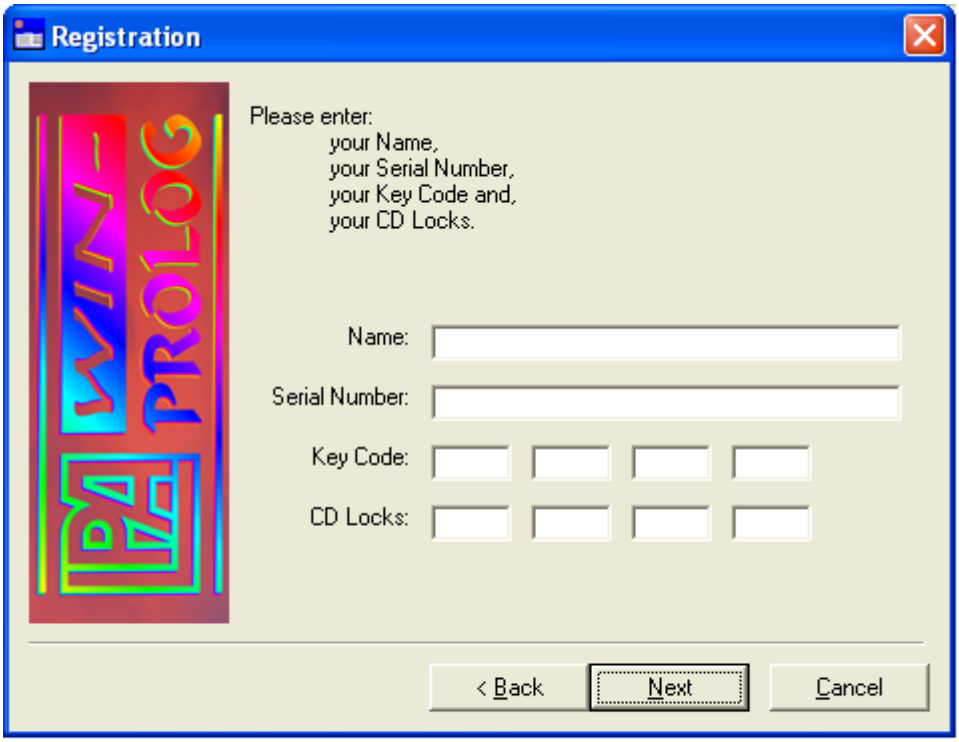


Figure 2 - Registration Screen

The "Registration" Dialog

The "Registration" dialog allows you to enter your registration details, consisting of a name, serial number, key code and CD code. You can use the cursor keys and mouse as required to edit any mistakes.

The name field can include anything you like, such as your name, your company's name or the department in which you work; The name field can be up to 38 characters inclusive. The name entered will form part of the ***WIN-PROLOG*** welcome banner and will be displayed whenever you run the system; Hint: if you want the welcome banner to be neatly left/right justified, then fill in all 38 characters in the box (or add extra spaces between the words!). Please note that only ASCII characters can be entered into the name field.

The **serial number** consists of 10 digits.

The **key code** and **CD Locks** code both consist of 20 digits split into four groups of five. Don't worry if one group has less than five digits - the missing zero(s) will be added automatically.

Click on "Next >" or press *<enter>* when you are ready. If any of the serial number, key code or CD code details are incorrect, you will be asked to enter the details again.

The "Components" Dialog

In addition to installing ***WIN-PROLOG*** itself, the LPA Setup program also allows you to install a number of toolkits:

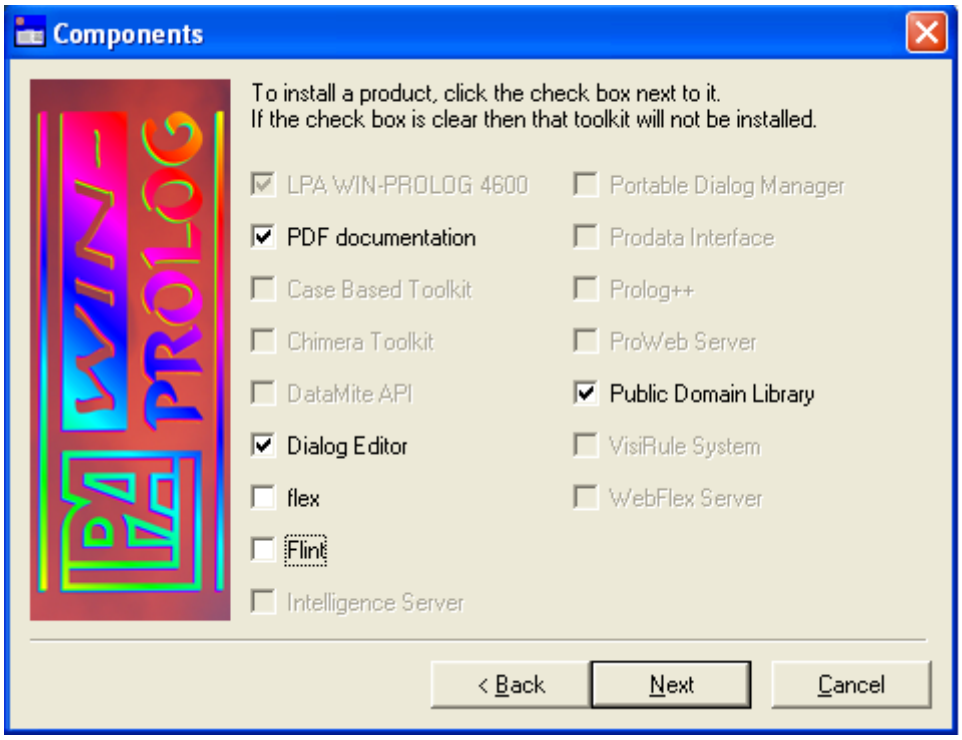


Figure 3 - Installation Screen

Note: the bold items are the ones you have successfully unlocked using the keycodes entered. You can choose to deselect them and not install them at this stage. Greyed out items indicate the item that are not available for installation or, in the special case of the core Prolog system, cannot be deselected.

Toolkit	Description
LPA WIN-PROLOG 4900	***WIN-PROLOG*** version 4.900
PDF documentation	Documentation in Adobe's Portable Document Format
flex	The flex expert system toolkit
Prolog++	The Prolog++ object-oriented toolkit
Flint	The Flint Uncertainty Handling toolkit
Intelligence Server	The Intelligence Server toolkit

Dialog Editor	The Dialog Editor
Public Domain Library	The Edinburgh Public Domain Library
Agent Toolkit	TCP/IP and Agent toolkit
Prodata Interface	Prolog to ODBC data source interface
Portable Dialog Manager	The Portable Dialog Manager
ProWeb Server	'Prolog on the Web' toolkit
DataMite API	Data Mining toolkit.
WebFlex Server	'flex for the Web' toolkit
Case Based Toolkit	Case-Based Reasoning toolkit
VisiRule System	"draw" the logic of your application as a decision flowchart

Note that the LPA Setup program treats ***WIN-PROLOG***, its help files, its PDF documentation and any of its toolkits as separate "components". Within the "Components" dialog, select each component that you wish to install and for which you have the requisite access. The ***WIN-PROLOG*** component itself has already been selected for you; this is selected, and hence installed, regardless of whether it is already installed on your hard disk.

The "Destination Location" Dialog

When ***WIN-PROLOG*** is transferred to your hard disk, it will need to be assigned a "home" directory in which to keep all its files. Within the "Destination Location" dialog, you will be asked for the name you want to use for this directory. The recommended name, *C:\Program Files\WIN-PROLOG 4900*, is pre-filled in the "Destination Directory" box: you can accept this by clicking on "Next >" or by pressing *<enter>*.

If you want to install ***WIN-PROLOG*** in another directory, click on the "Browse" button to view your hard disk. This will display the "Choose Directory" dialog from where an existing directory can be selected or a new directory specified by typing it in to the "Path:" box; use the cursor keys and mouse as required to edit any mistakes. Click on "OK" when you have finished; this will return you to the "Destination Location" dialog. Click on "Next >" or press *<enter>* when you are ready.

The "Confirmation" Dialog

All the details entered in the previous dialogs will now be displayed for confirmation. If the entered details are correct, click "Next >"; if not, click "< Back" one or more times to amend the details that are incorrect. Note that the contents of each completed dialog will be preserved.

Creating Directories and Copying Files

The Setup program will now copy all relevant files to the specified "home" directory on your hard disk, creating directories as required. Your name will be registered in the software. If you have selected one or more toolkits, the Setup program will automatically install these for you. Throughout this process, you will be shown the names of the files as they are being copied. If a file of the same name already exists in the specified "home" directory on your hard disk, you will be prompted prior to overwriting.

Completion of the LPA Setup program

Once all files have been successfully copied from the CD-ROM into your "home" directory, you will be presented with the "Installation is now complete" dialog. Since all automatic installation procedures have now been completed, there is no "Abort" button.

Simply press *<enter>* or click on "OK" once you have read the message. The Setup program will then terminate, leaving you in Windows. There may be a readme on the CD which you should read.

The LPA Setup program does not modify your *AUTOEXEC.BAT* or *CONFIG.SYS* at all, hence PATH will not include a pointer to the ***WIN-PROLOG*** directory.

The LPA Setup program does not modify the Windows Registry at all; you will, therefore, be unable to load a Prolog source file (which, incidentally, has the same file extension as a Perl script) into ***WIN-PROLOG*** simply by double-clicking on it.

Extending a Time-Limited Version

If you have a time-limited version of ***WIN-PROLOG*** (perhaps for evaluation purposes) that you would like to extend for a longer period, a new keycode (obtained from LPA) can be entered by executing the following goal:

```
?- ensure_loaded( system(keycode) ).<enter>
```

Installing Additional Toolkits

Should you wish to install one or more additional toolkits to an already existing installation, the same LPA Setup program that you must use the same LPA Setup program. That means finding the LPA CD.

When asked by the LPA Setup program to select the components required for installation during this session, you should de-select any components already previously installed, unless, of course, you wish to re-install them; ***WIN-PROLOG*** itself will be reinstalled automatically, so must not be already running. When asked by the LPA Setup program to enter the "home" directory for the toolkits now being installed, ensure that you give the "home" directory for your ***WIN-PROLOG*** system.

Note: re-installing toolkits already previously instlled will cause the original files to get copied onto the Hard Disk. So, if you have edited any of the example programs for instance, do save them under a different name and they will stay unharmed.

Uninstalling LPA Products

There is no uninstall option for ***WIN-PROLOG***. ***WIN-PROLOG*** can be uninstalled simply by deleting its home directory using your usual Windows method. Note that all files in this directory will be deleted, including your own files and subdirectories! You will also need to remove the installed shortcuts from the Start Menu; on Windows NT 4, the 'WIN-PROLOG 4900' folder, as used by the Start Menu, will be found at something like (depending upon your version of Windows) 'C:\WINNT\Profiles\Administrator\Start Menu\Programs\WIN-PROLOG 4900'. ***WIN-PROLOG*** does not touch the Windows Registry at all. If any Microsoft DLLs were installed, these will, however, remain in place.

Technical Support

Should you experience any problems with either installing or using LPA products, please do not hesitate to get in contact - details are listed at the front of this user guide. In order to provide efficient suppor, you need to send the serial number and version number of your LPA product. It is best to contact LPA via email.

Trouble Shooting

Problem	Solution
"Error 34 occurred in LPA Setup. Please contact LPA Ltd"	A previously installed copy of ***WIN-PROLOG*** is already running. You are installing ***WIN-PROLOG*** into an existing directory on the destination drive and trying to overwrite a locked file. Try installing ***WIN-PROLOG*** into a new directory.
"Error 1123 occurred in LPA Setup. Please contact LPA Ltd"	The setup program is having trouble creating a folder with the name given; are you using Unicode characters or characters forbidden in a file name?

Chapter 2 - Running *WIN-*PROLOG and Toolkits

This chapter tells you how to launch ***WIN-PROLOG*** on its own or with a toolkit.

Introduction

There are many ways of launching an application such as ***WIN-PROLOG***:

- using the 'WIN-PROLOG 4900' entry on the Windows' Start Menu. This launches ***WIN-PROLOG*** using its default settings.
- double-clicking on the main executable file (PRO386W.EXE). This launches ***WIN-PROLOG*** again using its default settings.
- from a shortcut pointing to 'C:\Program Files\Win-Prolog 4900\Pro386w.exe' with or without additional command line switches.
- from a shortcut on the Start Menu and clicking on that
- executing 'C:\Program Files\Win-Prolog 4900\Pro386w.exe' (with or without additional command line switches) from the 'Run' dialog (reachable via the 'Run…' option on the Start Menu).

Entries on the Windows Start Menu

During the installation process, a number of entries for ***WIN-PROLOG*** are added to the Windows Start Menu as a group with the WIN-PROLOG 4900 name. Note, if you have simply copied a previously installed system from one computer to another, you probably will not have these shortcuts on your start menu.

This group can be accessed from the Windows Start Menu by clicking on "Start" to display the Start Menu and then ALL Programs, and navigating to and clicking on WIN-PROLOG 4900. Clicking on WIN-PROLOG 4900 launches ***WIN-PROLOG*** using its default settings.

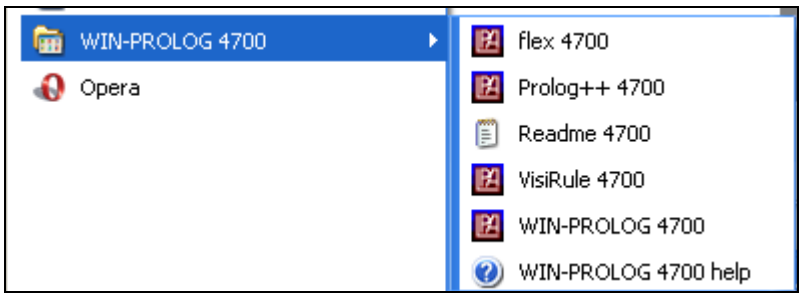


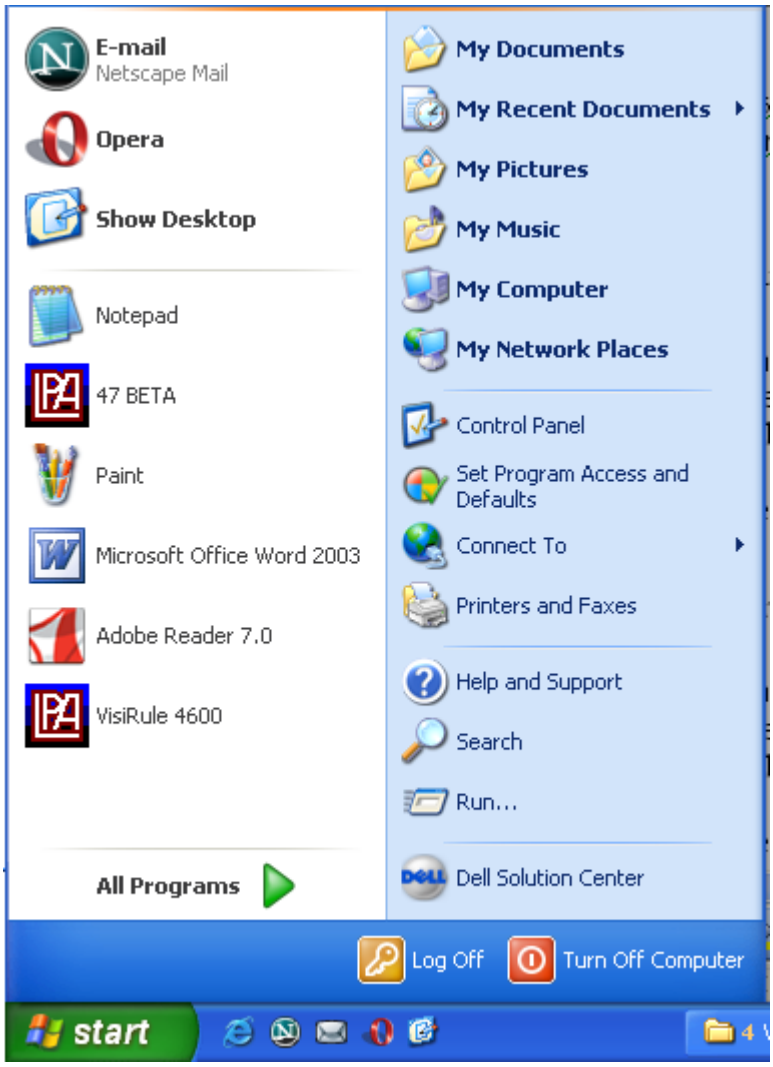
Figure 4 – A **WIN-*PROLOG*** *menu on All Programs of the Start Menu*

The ***WIN-PROLOG*** help file can be launched independently from the Start Menu and will be loaded into your default browser.

If you have installed VisiRule, flex and/or Prolog++, menu entries for some/all of these will also be present on the Start Menu as "Programs/WIN-PROLOG 4900/VisiRule 4900", "Programs/WIN-PROLOG 4900/flex 4900" and "Programs/WIN-PROLOG 4900/Prolog++ 4900" respectively.

You will also notice that the "WIN-PROLOG 4900" menu on the Start Menu has a "Documentation" submenu from which all the available LPA documentation files can be launched; even if you haven't installed a particular toolkit, its documentation is available for you to read. These files are in Adobe's Portable Document Format (PDF) and can be viewed via the Adobe Acrobat application.

Figure 5 - Start Menu with shortcut



Shortcut on your Start Menu

Remember: You can place shortcuts to frequently used applications on your Start menu for convenience's sake.

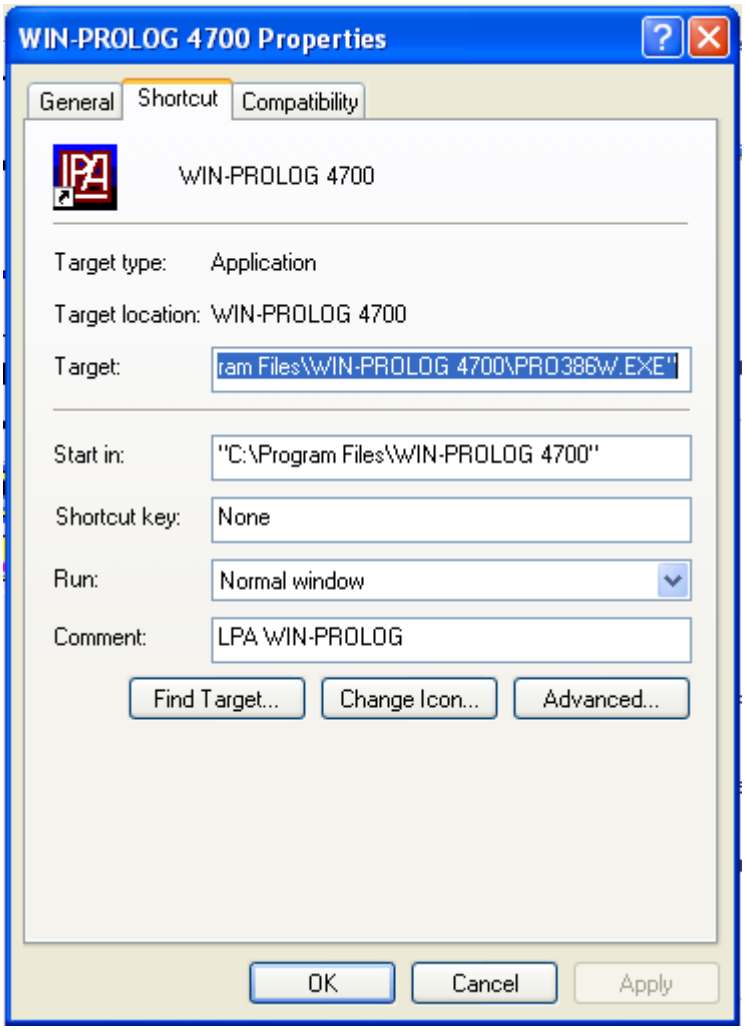


Figure 6- Properties Dalog

Shortcut properties

Target indicates the actual name of the executable file to run.

Start in indicates which directory to use initially.

Where Is Your *WIN-*PROLOG 'Home' Folder?

The LPA Setup program will have asked you where you wanted to create the ***WIN-PROLOG*** "home" folder in which to install the various ***WIN-PROLOG*** folders and files. By default, the ***WIN-PROLOG*** "home" folder is C:\PROGRAM FILES\WIN-PROLOG 4900\, but you may have changed this to somewhere more convenient for your particular computer setup.

WIN-PROLOG's principal executable file is PRO386W.EXE; you can, of course, double-click on this file directly to launch the application.

Installing on a network

In the file, PRO386W.INI, which is in the same folder as the version of PRO386W.EXE that is running, you can have the following heading and entry:

```
[pro386w]
 profile=...
```

where "..." is the name of another .INI file. If this field is present, WIN-PROLOG stops reading any further entries from PRO386W.INI, and instead opens "..." and uses that file in its place.

So, for example:

```
 [pro386w]
profile=c:\foo\bar.sux
```

would cause WIN-PROLOG to use the file, C:\FOO\BAR.SUX for its initialisation file. If every user's workspace had a C:\FOO folder, then each user would have his/her initialisation data saved in his/her own copy of the file, BAR.SUX, located within this folder.

Configuring *WIN-*PROLOG

The **WIN-PROLOG** system can be configured in two main ways, namely through an *"initialisation file" (.INI file)* and by a number of *"command line switches"*. In fact, the initialisation file can contain default settings for these switches, which can then be overridden as needed by the user. See the Appendix on Initialisation and Switches.

Configuring *WIN-*PROLOG Memory Usage

As a 32-bit program, ***WIN-PROLOG*** can use up to 2Gb (2048Mb) of memory. It is entirely up to you how much static memory you want to pre-allocate to ***WIN-PROLOG*** up front and how much you want to leave for other Windows applications (and Windows itself). By default, ***WIN-PROLOG*** uses around 3-4Mb of static memory, allocated amongst its six internal static data areas: you can change the size of one or more of these static data areas using command line switches.

In addition, ***WIN-PROLOG*** can be made to dynamically reallocate memory using the *xinit/9* predicate.

Memory allocated to a particular data area will only ever be used by that data area; use the *free/9* predicate to ascertain the minimum amount of memory required for each data area. If you need to change one or more of ***WIN-PROLOG***'s default settings, you will need to either add these to the PRO386W.INI file or create your own shortcut containing the required command line switches. An example set of command line switches required to load WordNet into ***WIN-PROLOG*** might be:

"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" /P60000 /T17775 /H16000 /B250 /L250

The whole issue of command line switches and memory allocation is discussed in the relavent Appendix. The Windows Task Manager dialog shows the amount of memory physically allocated to ***WIN-PROLOG*** but ***WIN-PROLOG*** also causes additional memory to be used; for example, loading a source code file into a rich edit window within the ***WIN-PROLOG*** development environment will use up memory which is allocated external to ***WIN-PROLOG***, as will loading a DLL.

The Windows Start Menu and Shortcuts

The Windows Start Menu is basically a collection of shortcut files. When a menu entry is clicked upon, the command line embedded in the shortcut is executed. In the case of the "WIN-PROLOG 4900" shortcut, the command line is "C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" (including the pair of double quotes as the pathname includes one or more space characters). The executed file, PRO386W.EXE, in the ***WIN-PROLOG*** "home" folder, is ***WIN-PROLOG***'s principal executable file.

Creating Your Own Shortcut to *WIN-*PROLOG

To create your own shortcut to ***WIN-PROLOG***, go to your ***WIN-PROLOG*** "home" folder and right click on the PRO386W.EXE file and select "Create Shortcut" from the menu which appears. A shortcut to PRO386W.EXE, appropriately named "Shortcut to Pro386w.exe" will be created in the ***WIN-PROLOG*** "home" folder. You can then copy or move this shortcut to anywhere you like, such as the "Desktop".

Another way to create a shortcut on the "Desktop" is to right click on the PRO386W.EXE file and, whilst holding down the button, drag and drop the file on to the "Desktop"; click on the "Create Shortcut(s) Here" entry from the menu which appears.

A shortcut, be it on the Start Menu, on the "Desktop" or in the ***WIN-PROLOG*** "home" folder, possesses a command line; in the case of ***WIN-PROLOG***, this might be "C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" (including the pair of double quotes as the pathname includes one or more space characters).

You will most definitely require a shortcut on your computer when launching a copy of ***WIN-PROLOG*** installed on another machine; the command line of such a shortcut might be "\\Rebecca\C\Program Files\WIN-PROLOG 4900\PRO386W.EXE" substituting 'Rebecca' for the name of your 'server'.

PRO386W.EXE's Optional Command Line

WIN-PROLOG differs from most Windows application in that it can be launched with an optional command line which will be executed on start-up. We shall look at the placing of switches and goals on the command line below.

What's Special About the VisiRule, Flex And Prolog++ Shortcuts?

The VisiRule, flex and Prolog++ menu entries on the Windows Start Menu differ from the one which launches just ***WIN-PROLOG*** in that they each include a goal for ***WIN-PROLOG*** to execute once up and running.

In the case of VisiRule, the command line (all on one line) of the shortcut is:

```
"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(
system(visirule) ).
```

In the case of flex, the command line (all on one line) of the shortcut is:

```
"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(
system(flexenv) ).
```

In the case of Prolog++, the command line (all on one line) of the shortcut is:

```
"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(
system(pppenv) ).
```

The pair of double quotes must be included in the command line of each of these shortcuts as the pathname includes one or more space characters.

The 'ensure_loaded(system(XXX)).' part is a Prolog goal (which must be terminated by a full stop) which ***WIN-PROLOG*** will execute once up and running; and in these cases loads the relevant toolkit. Alternatively, you can just launch Prolog on its own, and at any time run these goals from the query prompt in the Console window.

The call to *ensure_loaded/1* makes use of a logical path (i.e. system) which specifies where the file resides relative to ***WIN-PROLOG***'s home folder.

As you can see, you can have any number of shortcuts pointing to PRO386W.EXE, each with a different goal appended.

What About Appending Other Goals?

Any Prolog goal can be appended to a shortcut's command line. There are only two rules which need to be followed: (1) a space character must exist between the pathname and the goal, (2) you must include a full stop character after the goal:

<pathname><space><prolog_goal><full_stop>

In the same way you can put multiple goals on the ***WIN-PROLOG*** command line, so you can on the command line of a shortcut:

<pathname><space><prolog_goal><comma><prolog_goal><full_stop>

or indeed:

<pathname><space><prolog_goal><full_stop><prolog_goal><full_stop>

Changing a Shortcut's Command Line

The command line of a shortcut can be edited by clicking on the shortcut with the right mouse button and selecting "Properties" from the menu which appears. When the "Properties" dialog appears, click on the "Shortcut" tab.

You will see a "Target" edit field here that contains a command. You can enter any command here that can also be executed from the Windows command prompt.

The Working Directory

When creating a shortcut, it is possible to set a working folder for ***WIN-PROLOG*** which is different to its home folder. You may find this works differently on different versions of Windows. Please note that this may affect where ***WIN-PROLOG*** searches for any files that you include on the command line. In the flex and Prolog++ examples above, a change to the working folder will not affect the loading of FLEXENV.PC or PPPENV.PC because the call to *ensure_loaded/1* makes use of a logical path (i.e. system) which specifies where the file resides relative to ***WIN-PROLOG***'s home folder; ***WIN-PROLOG*** has a logical path for each of its examples, library and system folders so the goals 'ensure_loaded(examples(<filename>)).' and 'ensure_loaded(library(<filename>)).' are just as valid.

Creating and Executing Your Own "Loader" Program

A 'loader' program to load two or more of your own *WIN-PROLOG* programs can be created very easily. Create a .PL file with Prolog directives similar to the following:

```
:- ensure_loaded( examples(my_program1) ).
:- ensure_loaded( 'c:\program files\win-prolog 4900\examples\myprogram2.pl' ).
:- system_menu( , file, open('c:\program files\win-prolog
4900\examples\myprogram3.pl'(113,155,804,482) ) ).
```

Your 'loader' program can then be launched along with *WIN-PROLOG* itself by including it on the command line/shortcut:

```
"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(
examples(my_loader_program) ).
```

You may also like to look at *WIN-PROLOG*'s Project facility.

Associating Files with *WIN-PROLOG*

If you want to associate certain files (e.g. those having .PL or .KSL extensions) with *WIN-PROLOG*, and launch *WIN-PROLOG* whenever you double click upon one, you may like to read *Appendix D - Associating* WIN-*PROLOG Files With the Application*.

Trouble Shooting

There are about 30 points, during the startup sequence in *WIN-PROLOG*, when an error might occur, before its Main window has appeared: in these cases, it just exits "silently" without displaying any message. To help track these problems down, a simple utility, RUN386W.EXE, has been included. If you are finding that *WIN-PROLOG* simply appears to do nothing on startup, replace "PRO386W" with "RUN386W" in your shortcut or command line, for example:

```
pro386w /t1234 /p5678 write('hello').
```

becomes:

```
run386w /t1234 /p5678 write('hello').
```

The RUN386W.EXE file will attempt to run PRO386W.EXE as normal, with the given command line, but will display a simple message box upon completion of the Prolog session. This will contain either a number (normal exit) or alphanumeric code (error); please contact LPA Support for help if this happens.

Chapter 3 - Main Window, Console and Menus

This chapter describes the uses of the Main and Console windows of *WIN-PROLOG*. It covers basic input and output via the Main window's menu bar and the Console. The menu bar and the Console provide the fundamental interaction with the user.

The Main and Console Windows

On start up *WIN-PROLOG* briefly displays a banner, and then, when loaded, the Main window which includes the Console window. Initially the Main and Console windows appear as one window because the Console is maximised.

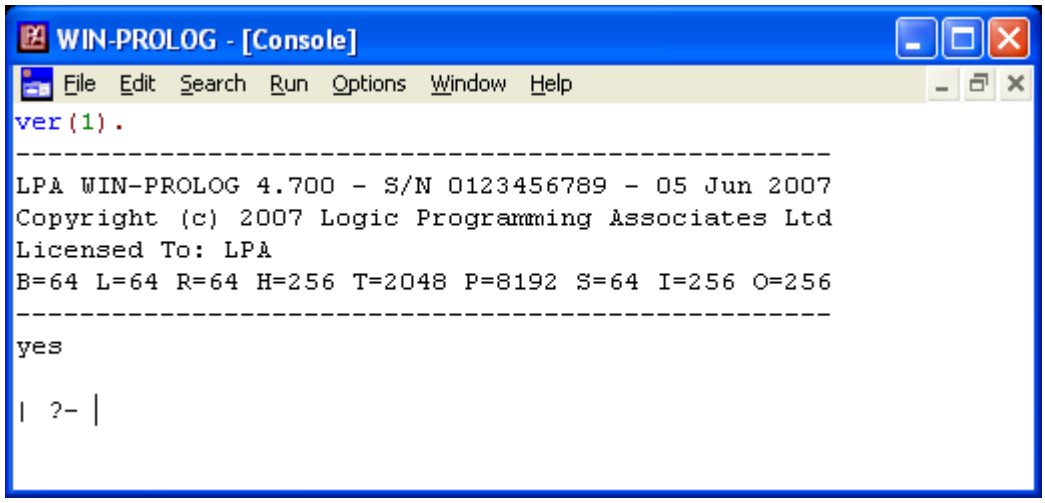


Figure 7 – WIN-PROLOGs Main window with Console window maximised

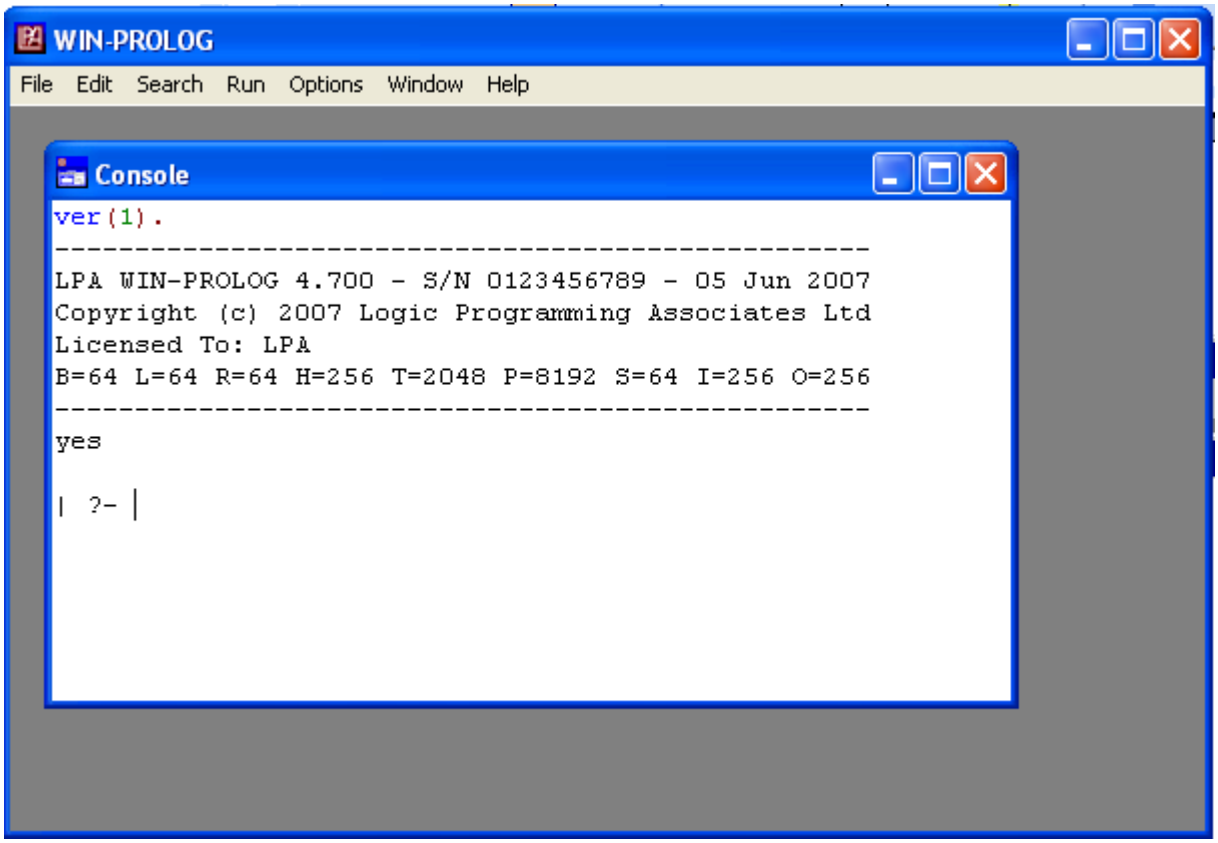


Figure 8 – WIN-PROLOGs Main window with Console window's size restored

The Main and Console windows form *WIN-PROLOG*'s command centre: operations are launched from the Main window's menus or entered by hand in the Console window. The Main window's menus provide the usual collection of file, edit and window options together with a number of *WIN-PROLOG*–specific options.

For those wishing to know, *WIN-PROLOG* is an MDI (Multiple Document Interface) application; any open windows, including the Console window and any program edit windows, are the Main window's child windows.

The *WIN-PROLOG* Console Input Model

This section explains the Console Input Model, first implemented in *WIN-PROLOG* 4.300, in conjunction with the adoption of Rich Edit 3.0.

Introduction

Background note: In versions of *WIN-PROLOG* prior to 4.300, the whole of the Console window was editable at all times. This allowed the user to modify not just previous input, but also output, and had the overall result that at any given time, the console probably did not contain a true record of what had happened.

In *WIN-PROLOG* 4.300 and beyond, editing is confined to a special region, called the "Input Zone", which effectively consists of the portion of the Console window that follows the most recent prompt or program output. Once entered, keyboard input becomes locked along with any subsequent output, as the Input Zone is re-established at the very end of this output.

While any part of the Console can be copied into the input zone for re-editing and re-entry, a much more useful mechanism for recalling commands is available. *WIN-PROLOG* has a powerful *command history* mechanism, that automatically stores up to 255 previously entered commands, even between *WIN-PROLOG* sessions.

Command History

These can be re-called and edited simply by using the *<CTRL-PGUP>* and *<CTRL-PGDN>* key combinations. No need to re-enter your queries.

Some Basic Terminology and Concepts

The Console Window can be thought of as being divided, invisibly, into two zones:

The "Fixed Zone" contains all console ("user") output, as well as previously entered commands, and cannot be edited.

The "Input Zone" is the bottom part of the console, into which new commands and other input can be typed, edited and entered.

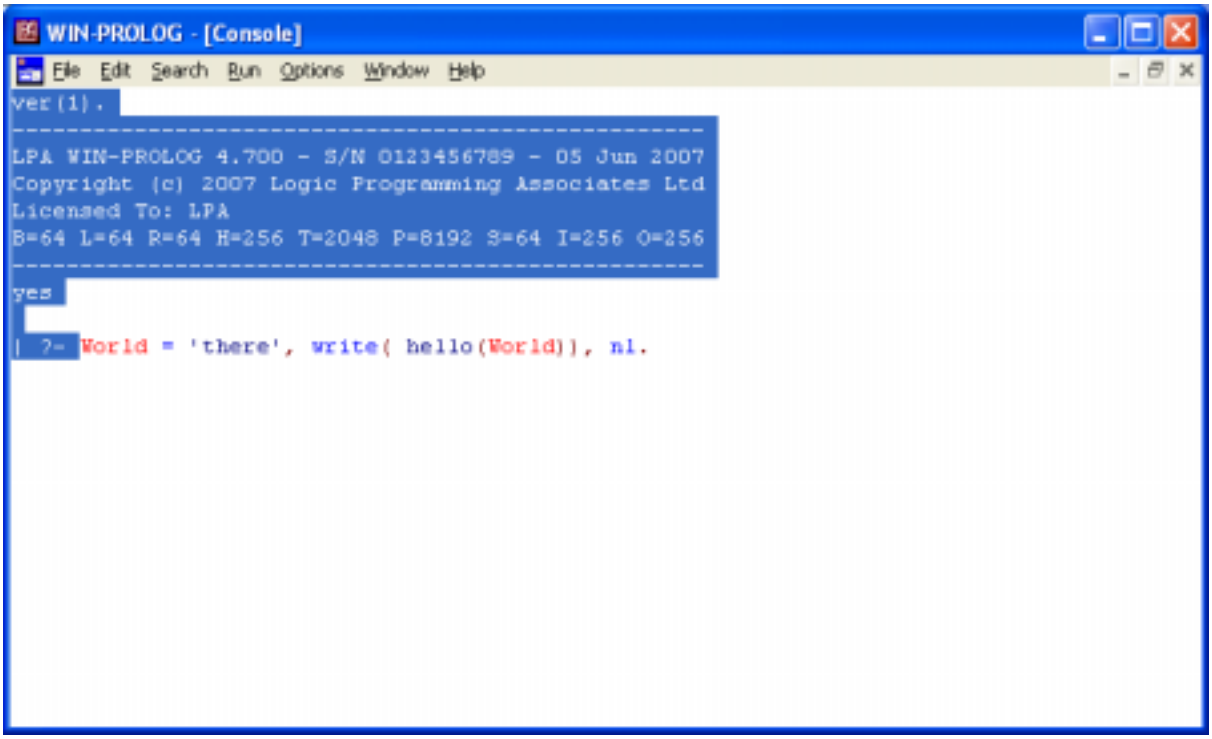


Figure 9- Console Window

In the above diagram, the area marked in white represents the Input Zone, and everything before it (the selected area) corresponds to the Fixed Zone. As a command is being typed into the Input Zone, it can be edited and modified at will; once *<ENTER>* is pressed to submit the command, the Fixed Zone expands to include the command, as well as any subsequent output, and once ***WIN-PROLOG*** is ready for more input, a fresh Input Zone is defined:

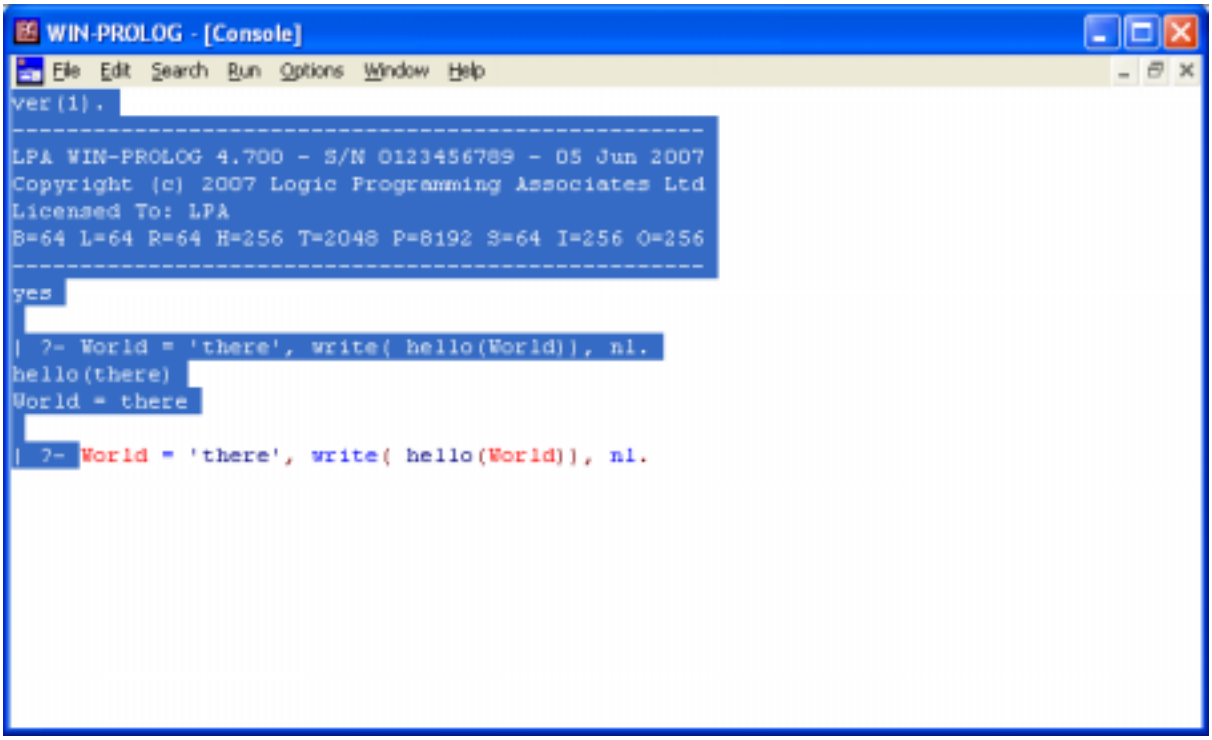


Figure 10- Fixed Zone Expanded

Three Ways to Re-enter Commands

There are three ways in which you can re-enter previous commands, with or without edits and modifications, using the "Console History", "Highlight and Enter" or simply "Copy and Paste" features.

Console History Mechanism Again – VERY IMPORTANT

By far the simplest and most powerful is to use the "Console History" mechanism, which records up to 255 of your commands in a special buffer.

You can review these commands with two key combinations:

<CTRL-PGUP>	Copy the previous (next earlier) command into the Input Zone
<CTRL-PGDN>	Copy the following (next later) command into the Input Zone

When you've found the command you want to re-enter, just press <*ENTER*>, or optionally make modifications first. You can review the console history at any time you are performing input: you can always return to the last line you were actually editing simply by pressing <*CTRL-PGDN*> repeatedly until it appears.

A key feature of the Console History mechanism is that your commands can be saved between ***WIN-PROLOG*** sessions, in a special file in ***WIN-PROLOG***'s "CACHE" directory. When you start a future ***WIN-PROLOG*** session, <*CTRL-PGUP*> and <*CTRL-PGDN*> are ready immediately to review and re-enter your commands from the previous session.

Highlight and Enter

The second way in which to re-enter a command is simply to highlight its text using the mouse, and then press <*ENTER*>. At any time text is highlighted, pressing <*ENTER*> simply copies that text directly into the Input Zone, ready for editing, and eventual re-entry. The rules about the <*ENTER*> key are simple:

1. Pressing <*ENTER*> when NO text is highlighted submits the entire Input Zone to WIN-PROLOG
2. Pressing <*ENTER*> when one or more characters of text is/are highlighted copies that text to the Input Zone, replacing its previous contents

Note that the Highlight and Enter model allows any console text, not just previous commands, to be selected and copied to the Input Zone. Furthermore, the area highlighted is not limited to the Fixed Zone: it can be entirely Fixed Zone, part Fixed and part Input Zone, or even just Input Zone. The last case is useful if, for example, you want to trim a command before entering it: simply highlight the bit you want to keep, and press <*ENTER*> twice - the first <*ENTER*> will replace the overall Input Zone with the stuff you highlighted (rule 2, see above); the second <*ENTER*> will submit it to WIN-PROLOG (rule 1, see above).

Copy and Paste

The third, and most familiar way in which to re-enter text is simply to use "Copy" and "Paste". These are available either through the "Edit" menu, or the usual Windows shortcut keys (<*CTRL-C*> for "Copy", <*CTRL-V*> for "Paste". You can also use "Cut" (<*CTRL-X*>) on the Input Zone, but not on the Fixed Zone (that's why it's called the "Fixed" zone!). Remember also that you can only "Paste" into the Input Zone (for the same reason).

The only advantage of Copy/Paste is that you are not limited to the Console window for your source material: you can grab example queries from Prolog program windows, or even PDF documentation files, for pasting into the Input Zone.

Related Keys and Menu Options

You can quickly identify and highlight the Input Zone by selecting the "Edit/Select Query" menu option; this has a matching keyboard shortcut:

<CTRL-Q>	Select (highlight) the entire Input Zone

Because the Fixed Zone is "Fixed", you cannot modify anything within it: moreover, neither can you delete text from it, with the result that the contents of the Console continue to grow throughout a ***WIN-PROLOG*** session. A special menu option, "Edit/Empty Console", allows you to clear everything from the console and start over.

While emptying the console means that you will lose your ability to "Highlight and Enter" or even "Copy and Paste" previous commands, it does not affect the "Console History" mechanism: *<CTRL-PGUP>* and *<CTRL-PGDN>* can still recall up to 255 of your previous commands.

Changing the Console History Depth

Although the Console History can store up to 255 of your previous commands, this would require a fairly large chunk of memory (each command is stored as 16-bit Unicode, and could be up to 4096 characters long, meaning that 8Kb of RAM are reserved per desired command. By default, ***WIN-PROLOG*** allocates space for 64 saved commands, using 512kb of memory. If you want to keep fewer, or more, commands, you can change the "depth" of the Console History using the menu option, "Options/Console History", to any value between 1 and 255.

The same menu option lets you choose whether or not to save the Console History between sessions: you might decide not to save commands in this way if, for example, you have been entering passwords or encryption keys, and don't want to risk someone else finding them subsequently.

Dynamic Syntax Colouring

One benefit of Rich Edit in ***WIN-PROLOG*** is the introduction of "Dynamic Syntax Colouring", which highlights the Prolog (or flex) syntax, distinguishing (for example):

Numbers	integer (dark green) or float (mid green)
Atoms	normal (dark blue), system predicates (royal blue), user compiled (light blue), etc…
Variables	single use/anonymous (light brown), multiple use (red)

as well as strings, char lists, punctuation, comments, etc., and even errors:

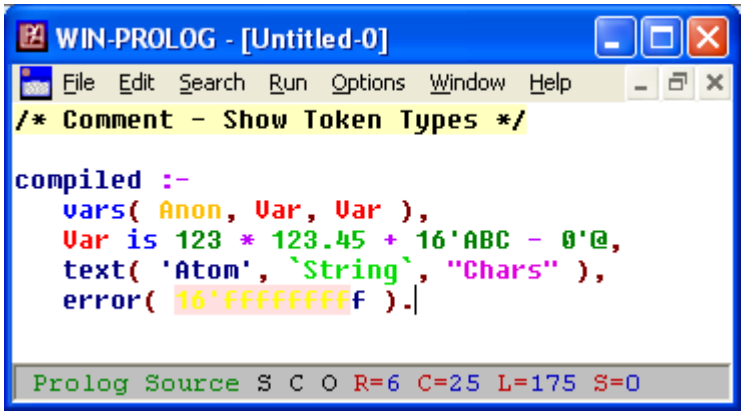


Figure 11- Coloured Syntax

Single use variables (i.e. Anon) are shown coloured light brown/yellow; should you mis-type a variable name causing it to inadvertently become a single use variable, its colour will ensure the error is easily spotted and duly corrected. You can also clearly see syntax errors such as an unterminated string. Rich Syntax Colouring is supported in:

- Prolog windows

- Flex windows

- Console Window (input zone only)

- Source Level Debugger

- Printouts via the File/Print menu option

If you are unhappy with the default choice of colour settings, you can choose you own via the Options/Colour Settings... menu option.

Syntax colouring takes place as you type, so you will see words change their colour as you move to the next word. This immediate feedback helps you detect simple typing errors – the cause of many a bug. No need to compile the window. This is especially helpful for students who are just starting out on their logic programming journey and old hands who type too quickly.

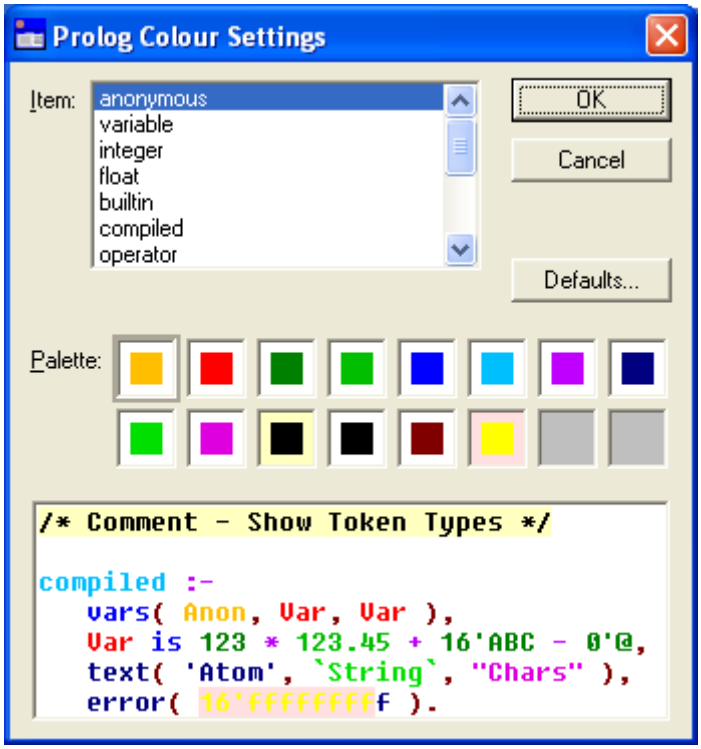


Figure 12- Prolog Colour Settings dialog

"Tuning" Automatic Syntax Colouring

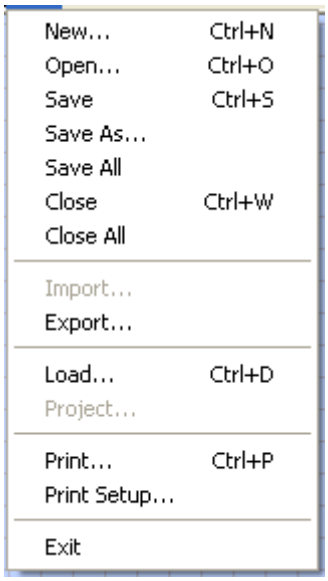
There is an intentional delay (user-definable) before the automatic syntax colouring takes place, to allow machines to be "tuned" for best results. The colouring process takes a finite time (*WIN-PROLOG* has to read the whole window, skip down to the start of the first term which is wholly or partly visible, then analyse and colour the visible window from that point). In large program windows, or on slower machines, this can take too much time to want it to happen between every keystroke. To stop the system getting sluggish, *WIN-PROLOG* delays the start of colouring until a finite period after input (mouse or keyboard) stops. By default, this is 250ms, but can be changed with the *rich_syntax/1* predicate to any value from 0 (off), 1 (immediate), or N milliseconds.

```
?- rich_syntax( 250 ). <enter>
yes
?- rich_syntax( N ). <enter>
N = 250
```

The Menu Bar

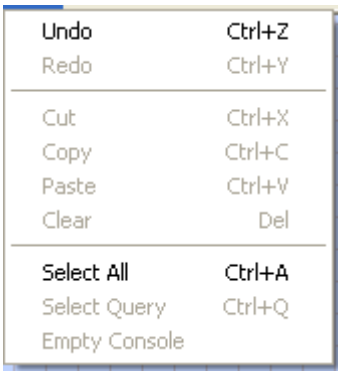
WIN-PROLOG's Main window contains a menu bar with several options, most of which will be described very briefly now. Your particular *WIN-PROLOG* setup may be slightly different as *WIN-PROLOG*'s menu bar is dynamically altered by the loading of one or more toolkits. Most of the menu options are described in more detail below.

the File Menu



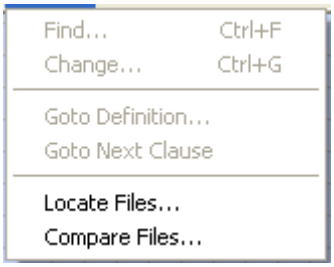
The "File" menu provides a selection of file-related functions. These are discussed in detail in ***Chapter 5 - the File Menu*** on page *66* and conform largely to those operations you would expect in any Windows application such as the loading and saving of programs, and the creation of new program windows. You can also use this menu to exit from *WIN-PROLOG*.

the Edit Menu



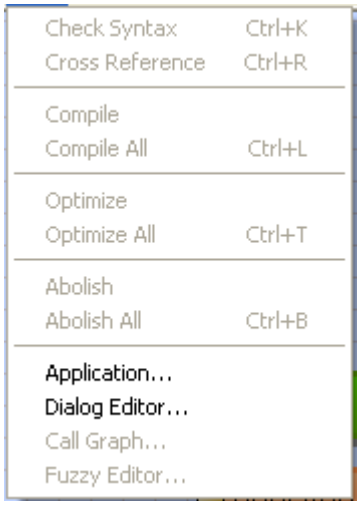
The "Edit" menu provides a range of program editing features, such as cut, copy, paste, clear together with undo and redo. These options, which are described in **Chapter 6 - the Edit and Search Menus** on page *75*, allow you to perform editing on the currently focused window.

the Search Menu



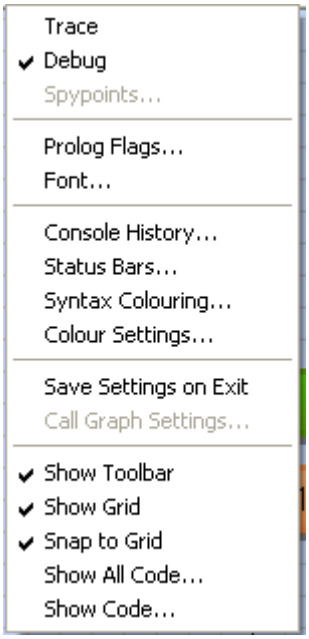
The "Search" menu contains higher level functions related to program editing. These are described further in ***Chapter 6 - the Edit and Search Menus*** on page *75*, and include find and change functions, including the ability to perform searches and changes across many program windows in a single command, as well as a special "go to" hot-link which lets you find the start of a predicate definition.

the Run Menu



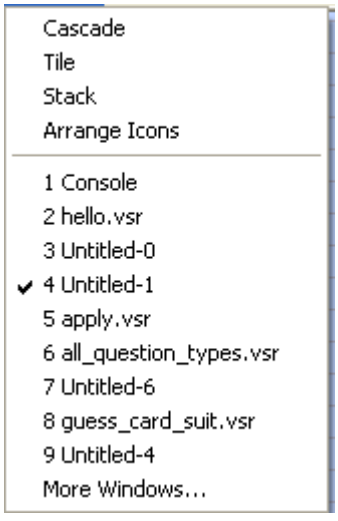
The "Run" menu provides the commands to compile, debug and run programs. These operations are described in ***Chapter 7 - the Run Menu*** on page *82* and are all directly related to Prolog programming. A number of additional entries may appear at the bottom of this menu depending upon what toolkits you have loaded.

the Options Menu



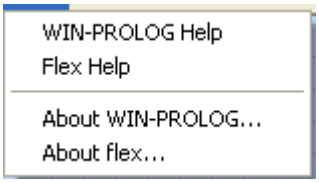
The "Options" menu provides options that allow you to tailor your ***WIN-PROLOG*** environment to suit your own particular needs. You can change the settings for the compiler, reporting of messages, debugger and system font. This menu also allows you to save the environment settings when you exit from ***WIN-PROLOG*** so that environment customisations can persist between sessions. These operations are described in ***Chapter 8 - The Options Menu*** on page 85, and are all related to the ***WIN-PROLOG*** environment.

the Window Menu



The "Window" menu provides the standard Windows window handling facilities. These include tiling and stacking the windows associated with ***WIN-PROLOG*** and also switching focus from one window to another quickly and easily. These operations, which are described in ***Chapter 9 - The Window Menu*** on page 89, enable you to maintain control of your ***WIN-PROLOG*** windows.

the Help Menu



The "Help" menu provides access to the ***WIN-PROLOG*** on-line help document using the Microsoft help application. The help file includes alphabetical and logically grouped listings of the ***WIN-PROLOG*** predicates and listings of error messages. The help menu, which is described in ***Chapter 10 - The Help Menu*** on page *90*, enables you to get quickly to the technical information you need. You can also use this menu to display the "About ***WIN-PROLOG*** " box.

Breaking Into a Query – VERY IMPORTANT

While you are getting to know your way around ***WIN-PROLOG***, you may execute a command that loops forever, such as:

?- **repeat, write('Hello World!'), fail.** *<enter>*

To break into its execution, press *<Ctrl>+<Break>* together.

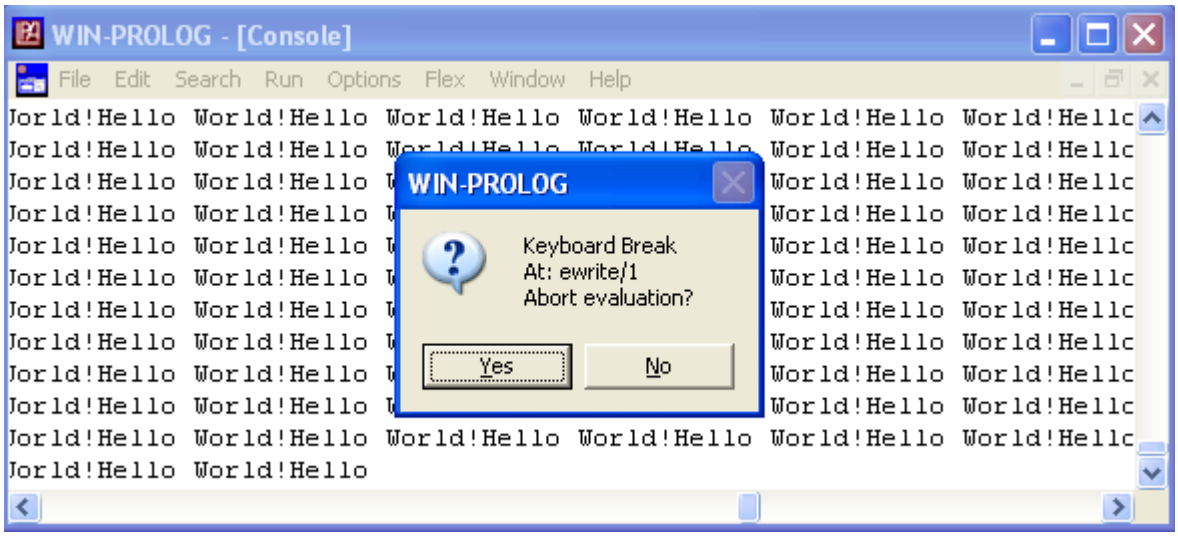


Figure 13- Breaking into a looping query

As Prolog is a recursive language, it's quite easy to inadvertently set off a computation which doesn't terminate and goes round and round and round …

*<Ctrl><Break>*gives you the chance to interrupt and abort this computation.

In a number of cases, this key combination will not work and then you have to press *<Ctrl><Right_Shift><Break>* together instead.

Otherwise, you may have to zap the process from the Task Manager.

Chapter 4 – Running a Prolog Example

This chapter steps you through the creation, compilation, running and saving of a simple Prolog example and the opening, compilation, running and saving of one of the supplied examples.

Creating a New 'likes' Program

With *WIN-PROLOG* up and running, click on the "File" menu option on the Main window's menu bar and then click on the "File" menu's "New..." option. A dialog will appear asking you to select the class of edit window required for your first Prolog program; select "Prolog Source Files (*.pl)".

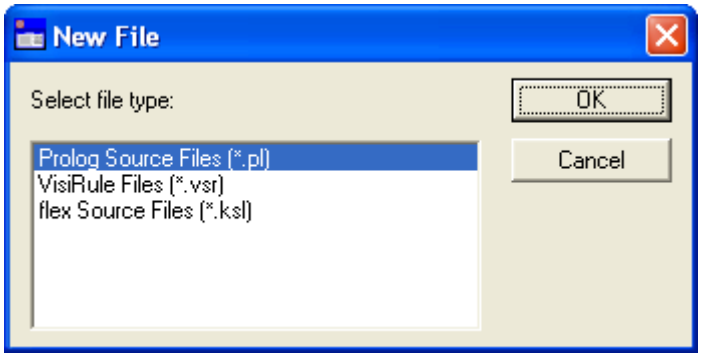


Figure 14 – Selecting the class of a new edit window

As soon as you click on the "OK" button, the dialog will disappear, to be replaced by a new rich edit window, ready for you to type in your first Prolog program.

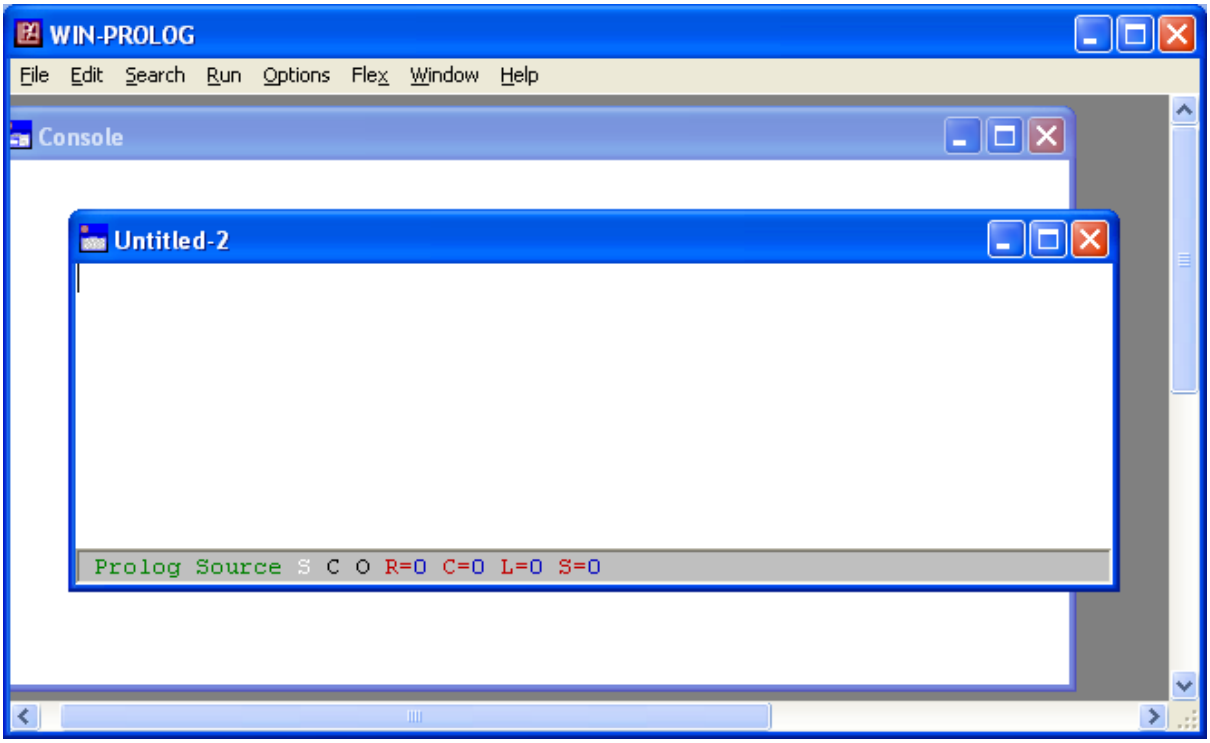


Figure 15 – An Untitled-<n> edit window

Enter the following simple example program into the "Untitled-0" rich edit window:

likes(brian, wine).		*<enter>*
likes(brian, X) :-		*<enter>*
alcoholic(X).		*<enter>*
		<enter>
alcoholic(beer).	*<spacebar>*	*<enter>*

In this, and all other examples, only type the characters shown in **bold** letters, and press the named key for anything bracketed in *<italics>*.

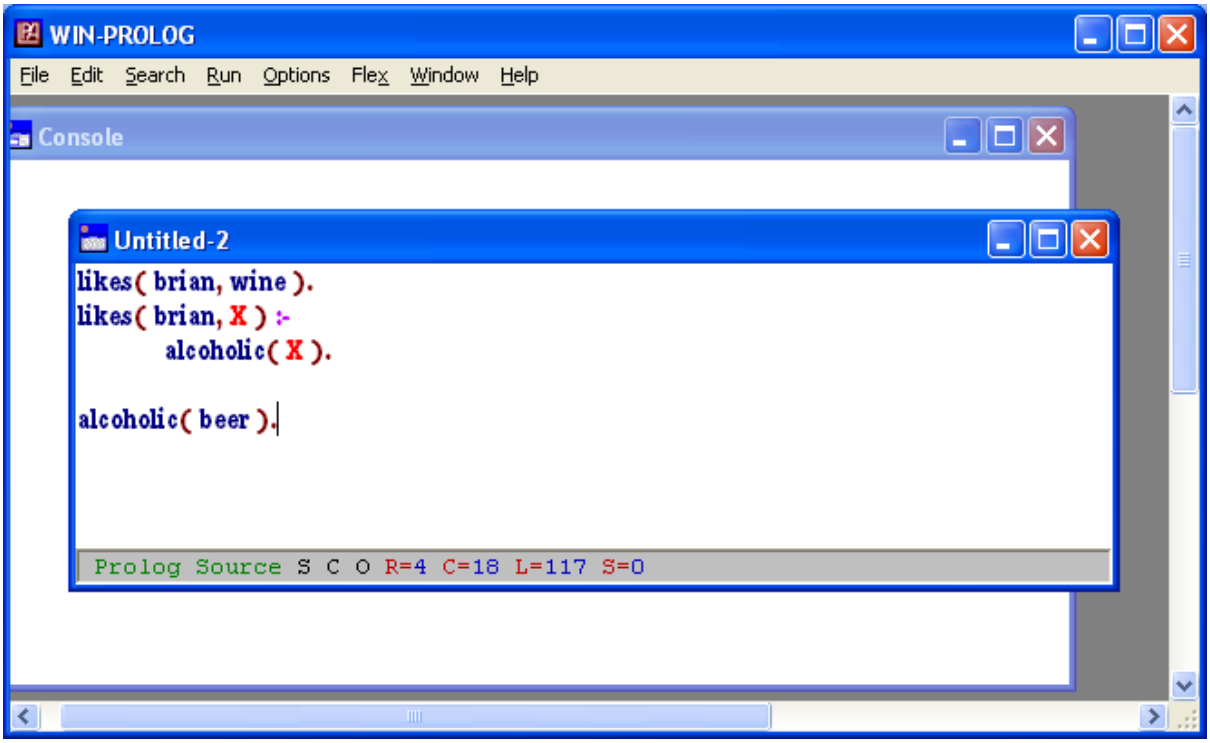


Figure 16 – Entering the simple example

Compiling your 'likes' Program

After entering your simple 'likes' example program, you need to compile it. Switch focus to the "Untitled-0" edit window and then select the "Run/Compile" menu option. After you click the mouse or press <*enter*>, the Console window (the window used for standard input and output) will display a message giving the information of how many seconds it took to consult the "Untitled-0" window. Note also that the "C" in the status bar at the bottom of the "Untitled-0" window changes colour.

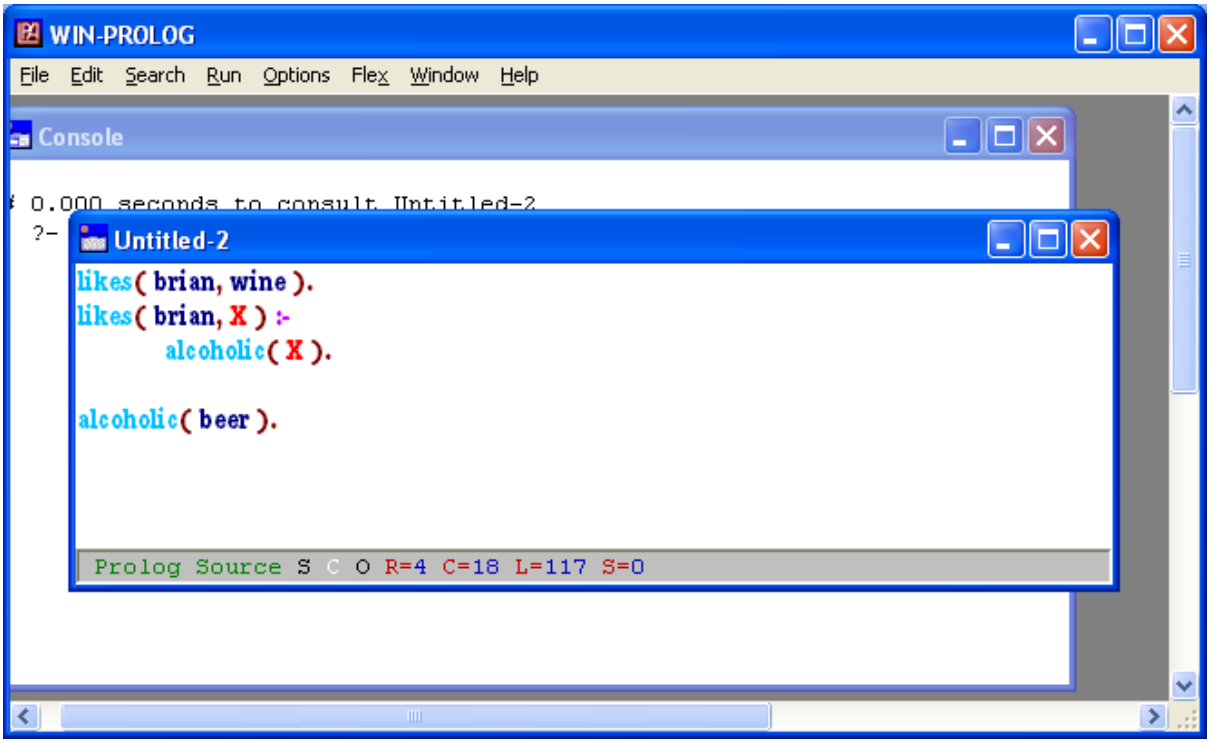


Figure 17 – After compiling your simple 'likes' example

Querying your 'likes' Program

Now change focus to the Console window: you can use the usual Windows methods to switch between *WIN-PROLOG*'s windows, such as <*ctrl-tab*> or simply moving the pointer over the window and clicking with the mouse. Let's now try out your 'likes' program by typing the following query (**bold** characters only) at *WIN-PROLOG*'s ?- prompt (to be found at the bottom of the Console window):

?- **likes(Who, What).** *<enter>*

The first solution should be displayed:

```
Who = brian,
What = wine;
```

If you press the <*space bar*>, or click the left mouse button, the next solution will be displayed:

```
Who = brian,
What = beer
```

Notice that *WIN-PROLOG* does not wait for another <*space bar*> or mouse click - when it knows there are no more solutions to a query, it automatically returns to the "?-" prompt.

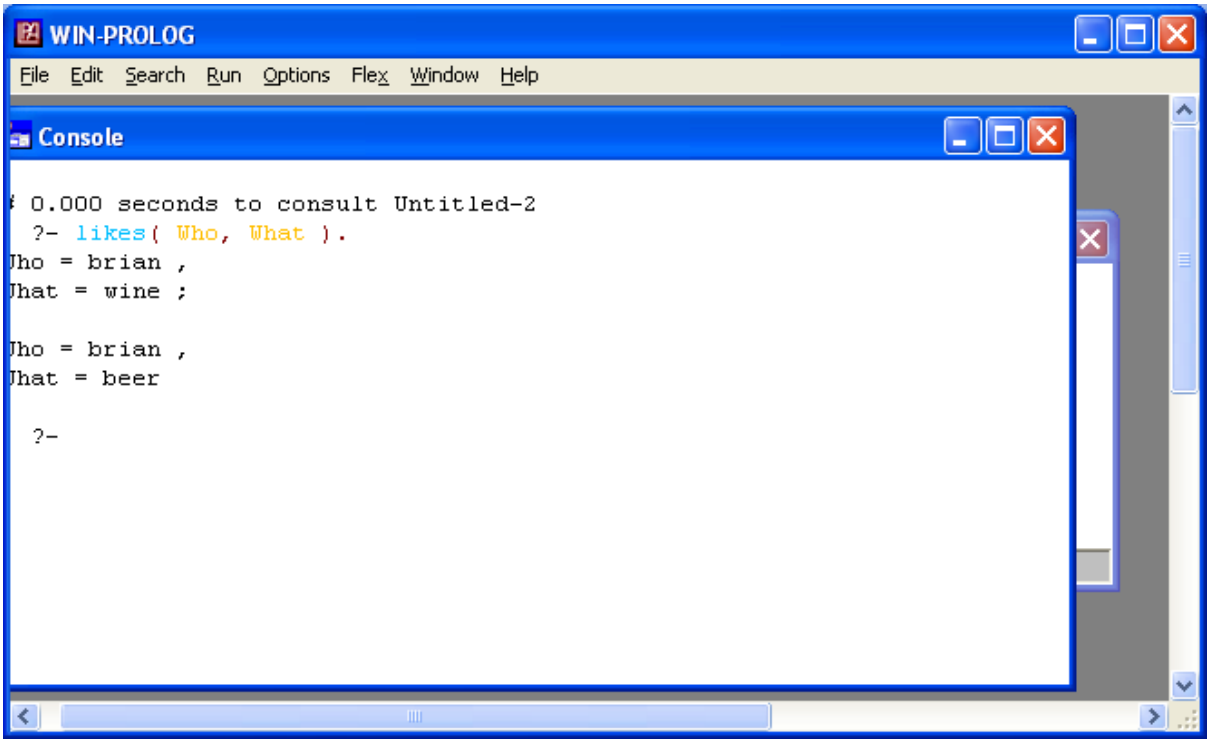


Figure 18 – Querying your simple 'likes' example program

If you got a "Predicate Not Defined" error message instead, it is probably because you forgot to compile the program.

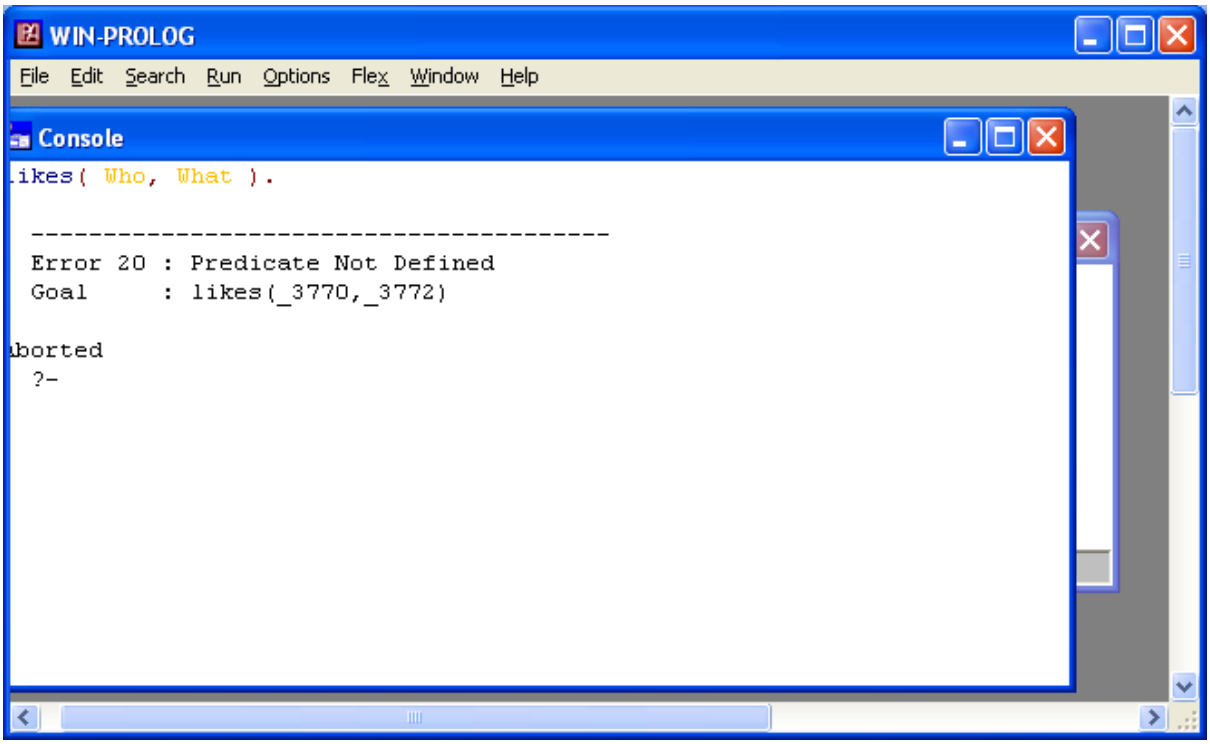


Figure 19 – What happens when you forget to compile your program

Saving your 'likes' Example Program

We are now going to save the 'likes' program to a file on disk. Click on the "File/Save" menu option. Because the program is currently in an "Untitled" window, the "Save As" dialog will be displayed, prompting us for a filename.

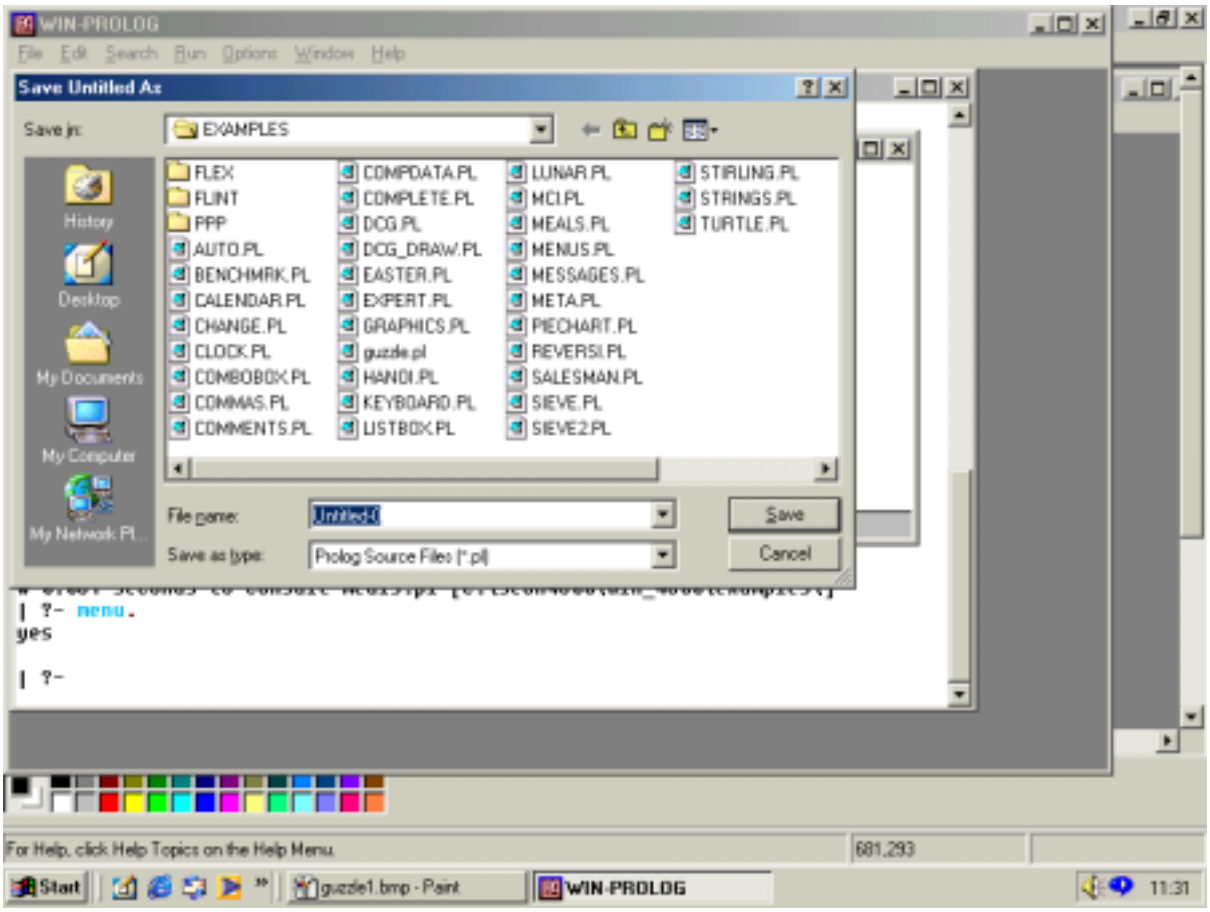


Figure 20 – The Save Untitled As dialog

Type a new name into the "File name" box. For example, type the name:

booze

and then click the "Save" button or press <*enter*>. Your "Untitled-0" window will be saved to the file *BOOZE.PL*, and the window itself will be renamed with the full pathname.

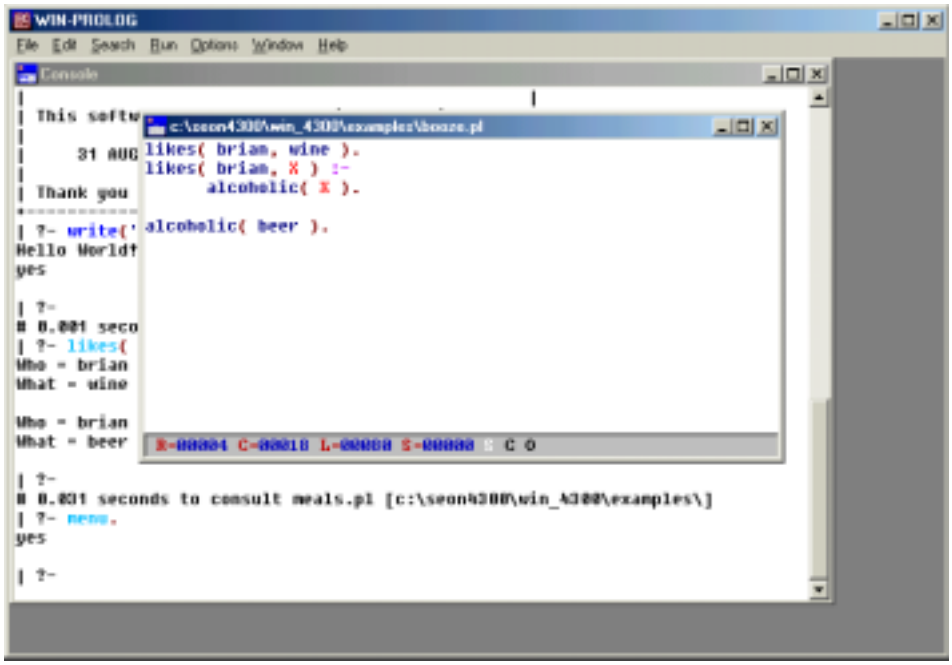


Figure 21 – Saved edit window – Note updated title bar

Opening the MEALS.PL Example Program

We are now going to turn our attention to one of the supplied Prolog examples, MEALS.PL. Select the "File/Open" menu option and when you click the mouse or press <*enter*>, you will see the "Open" dialog, one of Windows' common dialogs. Double-click on the *EXAMPLES* folder in the "File listing" box and you will then see the list of Prolog example programs within this folder. Select the one called *MEALS.PL*. When you click "Open" or press <*enter*>, a new rich edit window will be created for this file.

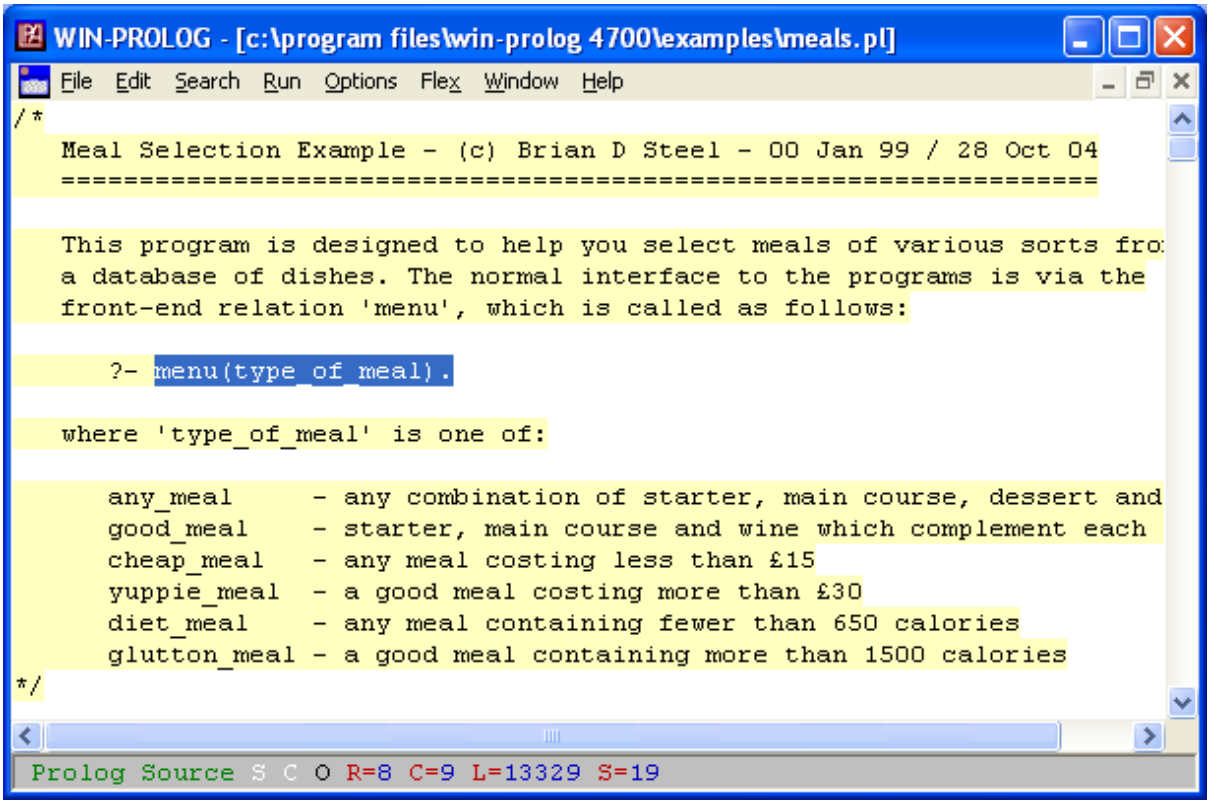


Figure 22 – The MEALS.PL rich edit window

Compiling the MEALS.PL Program

After opening the MEALS.PL example program, you need to compile it. Switch focus to the MEALS.PL edit window and then select the "Run/Compile" menu option. After you click the mouse or press <*enter*>, the Console window will display a message giving the information of how many seconds it took to consult the MEALS.PL window. Note also that the "C" in the status bar at the bottom of the MEALS.PL window changes colour.

Querying the MEALS.PL Program

We are now going to query the MEALS.PL program. Switch focus to the Console window and execute the following command at the ?- prompt:

?- **menu.** *<enter>*

Figure 23 – Execution of menu/0

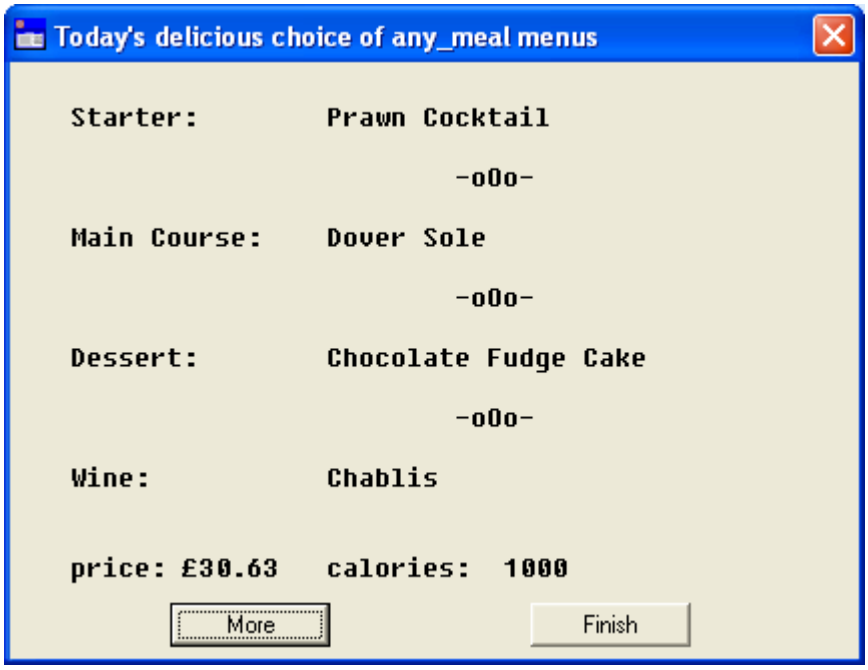


Figure 24 – Further execution of menu/0

Saving MEALS.PL under a New File Name

We are now going to save the contents of the MEALS.PL file to a new file. Change focus to the *MEALS.PL* window and then select the "File/Save As..." menu option; when you click the mouse or press *<enter>*, you will be presented with Windows' common "Save As" dialog. Type in a new file name, such as:

guzzle

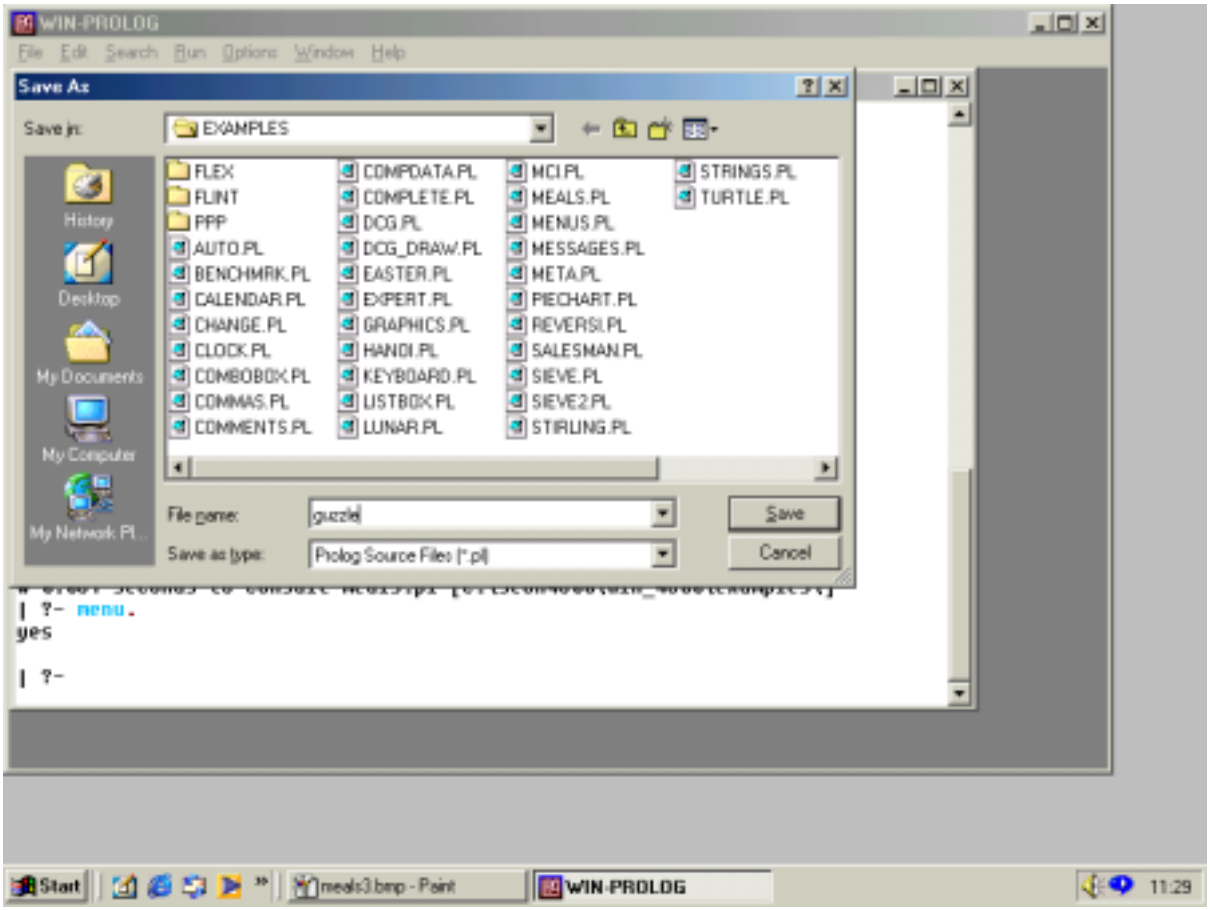


Figure 25 – The Save As dialog

When you click "Save" or press <*enter*>, the *MEALS.PL* window will be saved to the file *GUZZLE.PL*, and the window itself will be renamed with the new full pathname.

Closing the Program Windows

Switch focus to the *BOOZE.PL* window and then select the "File/Close" option. The source window will close immediately and the programs for *likes/2* and *alcoholic/1* associated with the window will be abolished from Prolog's memory at the same time. An alternative way to close a window is to click on its close icon. Switch to the *GUZZLE.PL* and try this out.

Chapter 5 - the File Menu

This chapter describes the uses of the "File" menu of *WIN-PROLOG*, covering the creating, opening, editing and saving of program files, and their relationship to edit windows.

The File Menu

The "File" menu provides a number of options related to files themselves, as well as to the edit windows used to contain their text during the writing of programs and their subsequent maintenance. None of the options in this menu are particularly related to Prolog as such: this menu is broadly the same as that in any normal Windows application. The remainder of this chapter introduces you to this menu, and shows you a few of the ways in which you can edit and maintain program files.

New... Option

The "New..." option in the "File" menu (for brevity, the term "File/New..." will be used from now on) allows you to create a new, untitled, editable window within the *WIN-PROLOG* development environment.

The "File/New" menu option results in the creation of a new "program edit" window, such as for Prolog or flex or VisiRule, depending on what toolkits you have loaded. And because *WIN-PROLOG*'s environment is extendable, other window types may exist in the future. Mostly these are text windows with syntax colouring, but in the case of VisiRule, you will get a graphical window.

Start up *WIN-PROLOG*, and when you see the Main window, select the "File/New..." option. When you click the mouse, or press *<enter>*, a dialog will appear asking you to select the class of window required, such as "Prolog Source Files (*.pl)".

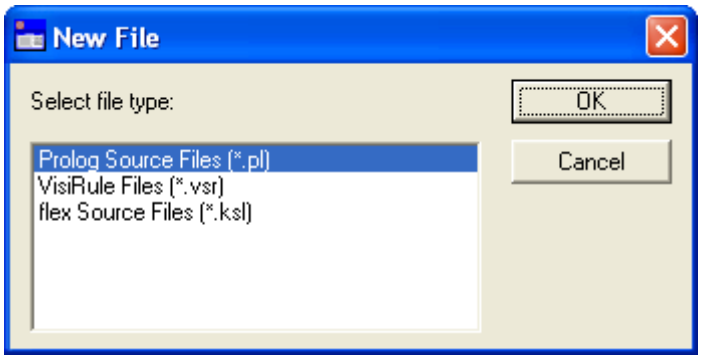


Figure 26 – Selecting the class of a new edit window

Once you select one of the entries and click "OK", an edit window will be created with the name "Untitled-0" (if it's the first "Untitled" window); you can have more than one "Untitled" window.

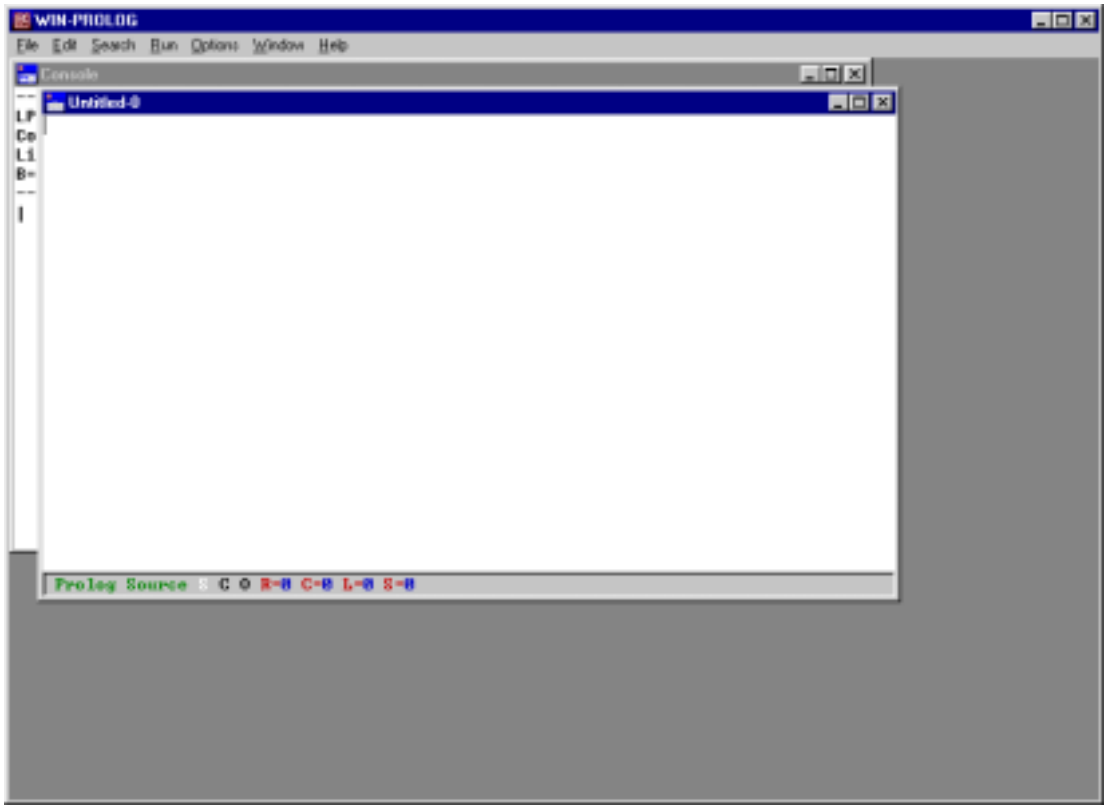


Figure 27 – An Untitled-<n> edit window

At the bottom of each window is a status bar, the left-hand side of which will inform you as to what class (e.g. Prolog Source) the window is.

The meaning of some of the flags, which are class-dependant, is as follows:

S will be shown in white* when program has been saved to disk, black* when edited but unsaved.

C will be shown in white* when program compiled, black* when uncompiled; a program will become uncompiled when edited or optimised.

O will be shown in white* when program optimised, black* when unoptimised; a program will become unoptimised when edited or compiled.

* The two colours used, which are white and black by default, are user-definable.

The meaning of some of the values, which are class-dependant, is as follows:

R=##### tells you what row (counting from 0) the caret is currently in.

C=##### tells you what column (counting from 0) the cursor is currently in.

L=##### tells you the length of (how many characters in) the window.

S=##### tells you how many characters in the current selection.

Open... Option

The File/Open...option is used to open edit windows for existing Prolog/flex source files or VisiRule charts, which are stored on disk. ***WIN-PROLOG*** uses the following file extension conventions:

.PL	Prolog source code file
.PC	Prolog object code file
.PJ	Prolog project file
.KSL	flex source code file
.PPP	Prolog++ source code file
.VRL	VisiRule source code file

A Prolog source file:

- is a file of text
- contains Prolog syntax (hopefully)
- *ends with a full stop and at least one white space character*
- has a .PL extension

The "Open" dialog allows you to open multiple source files into the ***WIN-PROLOG*** environment in a single action. Within practical limitations, you can have any number of files open for editing at any one time. Some of these may be named "Untitled-<n>"; others must be named in association with a disk file.

WIN-PROLOG will automatically detect the character encoding of the file (e.g. Unicode) and load it accordingly.

Select the "File/Open" menu option and when you click the mouse or press *<enter>*, you will see the "Open" dialog, one of Windows' common dialogs. Double-click on the *EXAMPLES* folder in the "File listing" box and you will then see the list of Prolog example programs within this folder (Flex examples will be found in the *EXAMPLES\FLEX* folder in the "File listing" box). Click on a file to select it; additional files can be selected by holding the *<CTRL>* key down. When you click "Open" or press *<enter>*, a new edit window will be created for this file (or if more than one file selected, a separate edit window for each file).

The folder which the "Open" dialog should initially point to is set in the "Win-Prolog" shortcut, although later versions of Windows may override this if files of a certain type are not found in the specified folder.

If you wish to change this default folder, click with the right mouse button over the "Win-Prolog" shortcut and select the "Properties" menu option. Click on the "Shortcut" tab in the "Win-Prolog Properties" dialog. Enter, in the "Start in" field, the full path name of the folder you wish WIN-PROLOG to initially point to. Finally, click "OK".

Alternatively, for the advanced practitioner, you can affect this programmatically by executing the following goal, all-be-it with the directory path changed to something else:

```
?- winapi( (kernel32,'SetCurrentDirectoryA'),
[`c:\temp`], 0, Handle ). <enter>
Handle = 1
```

Save Option

The File/Save option works on the topmost edit window and is used to save that window's source code (and associated object code in the case of Prolog source code). If the topmost MDI window is the Console then this option is greyed out.

If you choose to save an edit window that is already associated with a disk file, this operation will be carried out immediately saving the current contents of that edit window to its associated file.

Saving an "Untitled-<n>" edit window works in a slightly different way. Change focus to your "Untitled-0" window and then select the "File/Save" option; when you click the mouse or press <*enter*> you will be presented with the "Save Untitled As" dialog.

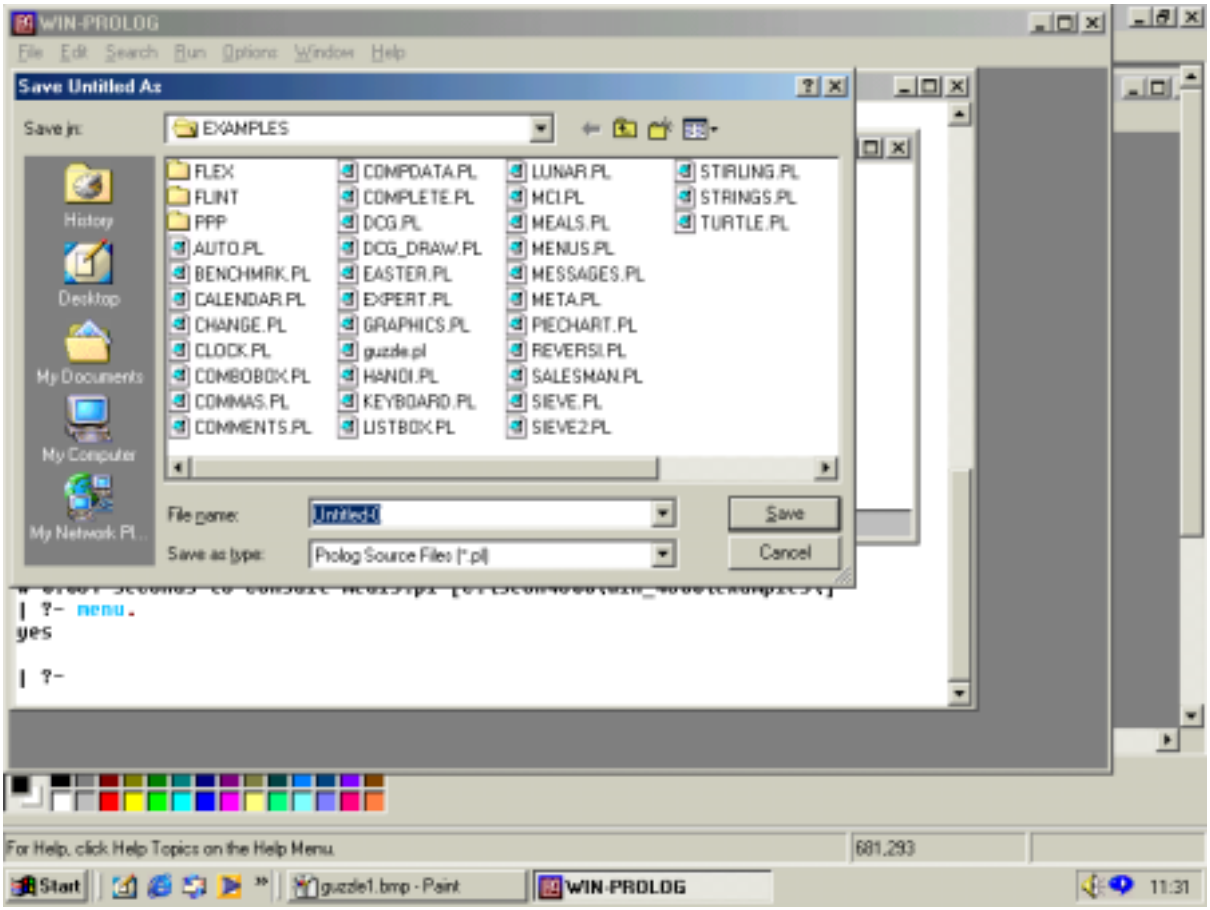


Figure 28 – The Save Untitled As dialog

Whenever you try to save an "Untitled-0" window, you will be prompted in this way for a name. Just as with the "Open" dialog you saw earlier, you can select file names or folders from the "File listing" box, or you can simply type a new name into the "File name" box and then click "Save" or press *<enter>*. The "Save as type" box will be automatically set based on the type of source code selected when the edit window was created. Your "Untitled-0" window will be saved to the new file and the window itself will be renamed with the full pathname.

The "S" in the edit window's status bar will change colour to reflect that the window has been saved.

The file will be automatically saved in the appropriate character encoding (e.g. Unicode) depending upon the characters present in the edit window. If no Unicode characters are present in the edit window, the file will be automatically saved in 'ISO' format.

If your application uses both Prolog source code and KSL source code files, which both need to be loaded at the same time, try to avoid having a .PL and a .KSL file with the same leaf name (i.e. <same_name>.PL and <same_name>.KSL) as recompiling the .KSL file will cause the code in memory for the .PL file to be abolished, and vice versa.

Save As... Option

Change focus to a rich edit window, the contents of which you wish to save under another filename, and then select the "File/Save As..." option; when you click the mouse or press <*enter*>, you will be presented with Windows' common "Save As" dialog. The "File/Save As" option is similar to "File/Save", except that you are always prompted for a file name. If the topmost MDI window is the Console then this option is greyed out.

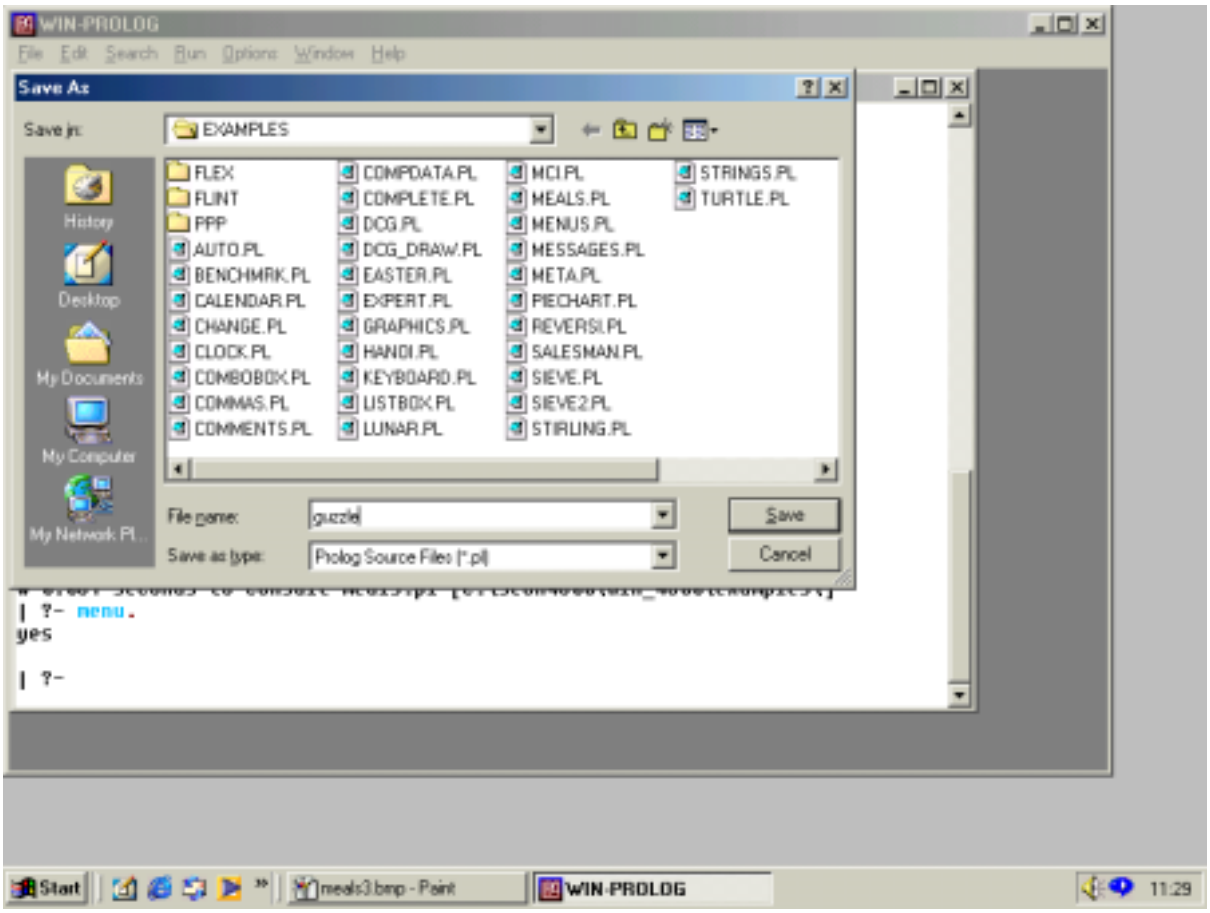


Figure 29 – The Save As dialog

Type in the new file- and/or path-name. When you click "Save" or press <*enter*>, the contents of the rich edit window currently in focus will be saved to disk under the new file- and/or path-name, and the window itself will be renamed with the new full pathname.

The "S" in the edit window's status bar will change colour to reflect that the window has been saved.

The file will be automatically saved in the appropriate character encoding (e.g. Unicode) depending upon the characters present in the edit window. If no Unicode characters are present in the edit window, the file will be automatically saved in 'ISO' format.

Save All Option

If you have edited any of the currently loaded Prolog/flex source code files then the "File/Save All" option will be enabled; this option is only available when at least one of the currently open edit windows is unsaved. Selecting this option will display the "Save All" dialog which contains a multiple-choice listbox showing all of the currently loaded files, with any edited files/windows pre-selected; you can select or deselect any of the files/windows in this list. When you click "OK" or press *<enter>*, a "save" operation will be performed on each selected item. If an "Untitled-<n>" window was selected you will be prompted via the "Save As" dialog to provide a name and folder for the source file.

The "S" in each edit window's status bar will change colour to reflect that the window has been saved.

The files will each be automatically saved in the appropriate character encoding (e.g. Unicode) depending upon the characters present in the edit window. If no Unicode characters are present in the edit window, the file will be automatically saved in 'ISO' format.

Close Option

This option is used to close an edit window, and to abolish any code associated with it from the Prolog workspace. If the Console window is currently in focus then this option is greyed out. If you attempt to close an edit window which has not been saved since you last modified it, "File/Close" will offer to save the window before closing it.

Close All Option

This option is used to close all the edit windows, and abolish any code associated with them from the Prolog workspace. If you attempt to close an edit window which has not been saved since you last modified it, "File/Close All" will offer to save the window before closing it.

Import Option

This option allows you to insert text imported from a .RTF or .TXT file.

Export Option

This option allows you to extract the selected or the complete text from an edit window and export it as a .RTF or a .TXT file.

Load Option

This option is used to load and, if necessary, compile the program in the exisitng Prolog or flex files (including .PL, .PC and .KSL extensions) into memory without creating any edit window.

This is particularly useful if you use an external text editor to edit your Prolog/flex source files, or if you want to work with object code files.

However, to get maximum value of the **WIN-PROLOG** integrated development environment (such as syntax colouring), we suggest you use the internal editor supplied. For consistency, it is recommended that any Prolog source code file created externally be given a .PL file extension.

Select the "File/Load..." option and when you click the mouse or press *<enter>*, you will see the file "Load" dialog. This dialog is almost identical to the "Open" dialog. Select the file called *MEALS.PL* in the *EXAMPLES* folder and when you click "Open" or press *<enter>* the source code for the *MEALS.PL* file will be loaded into the **WIN-PROLOG** environment. Although the load will be confirmed by a message in the Console window, you could test this by executing *menu/0* at the ?- prompt.

To load a Prolog source code file from the **WIN-PROLOG** command line, for example, *C:\PROGRAM FILES\WIN-PROLOG 4900\EXAMPLES\MEALS.PL*, into **WIN-PROLOG**, you would need to execute the following from the ?- prompt:

```
?- consult('c:\program files\win-prolog 4900\examples\meals.pl'). <enter>
```

The *source_file/1* predicate will tell you what source files are currently loaded; the *abolish_files/1* predicate will allow you to abolish one or more source files:

```
?- source_file( X ). <enter>

X = 'c:\program files\win-prolog 4900\examples\lunar.pl'

?- abolish_files('c:\program files\win-prolog 4900\examples\lunar.pl'). <enter>

yes
```

The Project Option

The "File/Project..." menu option is discussed in detail in a later chapter.

Print Option

As its name suggests the "File/Print..." option is used to print files; it uses the current default printer and port.

Select the "File/Open" option and open the *GUZZLE.PL* file. If you tried out the "File/Load..." option described above, a dialog box will appear warning you that the file has previously been loaded and asking if you wish to continue and open the file for editing. Select "Yes" and the *GUZZLE.PL* file will appear in an edit window.

Now select the "File/Print..." option. A dialog will appear saying that ***WIN-PROLOG*** is ready to print *GUZZLE.PL* to whichever printer and port you have currently set as your default. When you click "OK" or press <*enter*> a printer status dialog will appear telling you that ***WIN-PROLOG*** is currently printing page 1 of the *GUZZLE.PL* file. This dialog also gives the printer and port and an estimation of the percentage of the file that has been sent to the printer.

When the printer has finished printing you should see the contents of the *GUZZLE.PL* file printed out. The text is printed with a footer that gives the title of the file, the time and date of printing and the page number.

The "File/Print..." menu option can also be used to print call graphs.

Print Setup Option

This option is used to display one of Windows' common dialogs, the print setup dialog. This dialog allows you to perform setup operations on the printer, such as paper size, orientation, number of copies etc.

Exit Option

The remaining option is provided to allow you to exit from ***WIN-PROLOG*** at the end of a session. Its use is exactly equivalent to calling the *halt/0* predicate from the Prolog command line, and if there are any unsaved edit windows, you will be given a chance to save them before the system terminates.

Select the "File/Exit" option and when you click the mouse or press <*enter*>, your Prolog session will finish.

Chapter 6 - the Edit and Search Menus

This chapter describes the use of *WIN-PROLOG*'s "Edit" and "Search" menus, covering the writing and maintenance of program files.

the Edit Menu

The options in the "Edit" menu are all simple, text-related operations, common to virtually all Windows applications, and as such do not require detailed descriptions. Each operation (i.e. "Undo", "Redo", "Cut", "Copy", "Paste", "Clear" and "Select All") works with respect to the topmost edit window except "Select Query" which works on the Console window only. The Undo/Redo facility is multi-level and will take you all the way back to the last time a rich edit window's contents were saved.

Because the options in the "Edit" menu are ordinary text operations, and use the standard Windows clipboard, it is easy to cut, copy and paste text between any of *WIN-PROLOG*'s edit windows and its Console window, and indeed between all these and any other text-based Windows application. If you are unsure about how to use these Windows edit features, you should consult your Windows documentation.

Select Query Option

The "Select Query" option highlights (i.e. selects) the current query or command in the Console window's input zone (i.e. the query or command after the final ?- prompt). The keyboard shortcut, <CTRL-Q>, performs the same function.

Empty Console Option

The "Empty Console" option clears, after a warning, both the output and the input zones of the Console window; do not worry that the '| ?-' prompt also disappears. The Console history is unaffected.

Search Menu

The options in the "Search" menu are more closely related to Prolog than are those in the "Edit" menu. One option, "Goto Definition", is specifically linked to program maintenance, allowing you to locate instantly the definition for any compiled predicate. These options can be performed locally (within the topmost edit window) or globally (throughout all *WIN-PROLOG* windows), but cannot operate on windows belonging to other Windows applications. The remainder of this chapter examines the behaviour and use of these options.

In order to demonstrate the "Search" options, you should now load a file into an edit window. Assuming you worked through the examples earlier, you will have created two files in WIN-PROLOG's *EXAMPLES* folder, *BOOZE.PL* and *GUZZLE.PL*. Start up WIN-PROLOG, and when you see the Main window, open *GUZZLE.PL* into an edit window via the "File/Open..." option. From the "File listing" box of the "Open" dialog, select *EXAMPLES* and then *GUZZLE.PL* by clicking "Open" or pressing *<enter>*. An edit window will appear containing a copy of your example program.

Find Option

This option is used to search for a text string within a chosen amount of text. By dragging the mouse over some text within a window, you can choose to limit the search to this area; whether or not you have selected some text in this way, you can choose to perform searches within a single window, or throughout all edit windows.

Ensure the *GUZZLE.PL* edit window is the topmost window and then select the "Search/Find..." option; when you click the mouse or press *<enter>*, the "Find" dialog will appear.

The "Find" dialog is modeless, so you do not have to fill it in. In fact, once it is on display you can simply push it around with the mouse, and click on it to select it whenever you want to perform a find operation. Type the following word into the dialog's "Find" edit box:

Chablis

When you have entered this word click the "Find" button or press *<enter>*; immediately you do so, the *GUZZLE.PL* window will scroll the word into view.

Notice that the "Find" dialog is still in place: you can perform further finds of the same word by simply clicking "Find" or pressing *<enter>*, or you can edit the find string to start a new search.

The "In" radio buttons at the bottom of the "Find" dialog determine the scope of the search. Because in this case we had not selected a text area with the mouse before starting the find operation, the "Selected Text" option is greyed out.

Because it is modeless, you can leave the "Find" dialog on the screen as long as you like. When you do want to clear it, just click on its close icon. For now, leave it on the screen but move it out of the way so that you can see the *GUZZLE.PL* edit window.

Change Option

This option allows you to search for and change text strings automatically. Just like the "Search/Find..." option, you can choose to limit the search to this area by dragging the mouse over some text within a window; again, whether or not you have selected some text in this way, you can choose to perform changes within a single window, or throughout all edit windows.

Select the "Search/Change..." option; when you click the mouse, or press *<enter>*, the "Change" dialog will appear. Again, just like before, the "Change" dialog is modeless, so you do not have to fill it in until you need to. For now, type the following word into the dialog's "From" edit boxes:

main(

and then click on the "To" edit window or press *<tab>*, and type:

main_course(

You can perform find and change operations one at a time, or you can find and change all instances in one operation. Try the latter for now, by clicking on "Change All" or typing *<alt-L>*.

Just like the "Find" dialog, the "Change" dialog is modeless, and you can leave it on the screen as long as you like. When you do want to clear it, just click on its close icon. For now, leave it on the screen but move it out of the way, and you will see some of your changes.

Goto Definition Option

The third option in the "Search" menu is somewhat different from the other two: rather than perform a search for text that you enter, it automatically locates the beginning of a program definition. Because the beginning of programs is only known once they have been compiled, you will need to perform this step now.

Select the "Run/Compile" option. When you click the mouse, a message will appear in the Console window showing that the *GUZZLE.PL* file has been consulted. Now use the mouse or cursor keys to select a predicate call, such as "cost(Starter,Main,Dessert,Wine,Cost)," (to be found in the definition of the predicate *menu/1*).

Select the "Search/Goto Definition..." option. When you click the mouse, the window will select the definition of *cost/5* and scroll to bring it into view. Note that this selection also includes the comments at the beginning of the predicate.

You can also operate "Search/Goto Definition..." without first selecting a goal. Whenever the selection area includes more or less than a single predicate call the "Search/Goto Definition..." option prompts you with a list of known predicates. Try it now: click the mouse button anywhere in the *GUZZLE.PL* edit window, then select the "Search/Goto Definition..." option. Because the selection was zero, a list of all known programs is displayed.

Select a predicate, such as *diet_meal/4*, and when you click "OK" or press *<enter>*, the corresponding definition will scroll into view. Note that because this option relies on information stored by the compiler, you can only perform this operation on windows which have not been edited since their last compilation.

Unlike the "Search/Find..." and "Search/Change..." options, "Search/Goto Definition..." always works across all compiled edit windows.

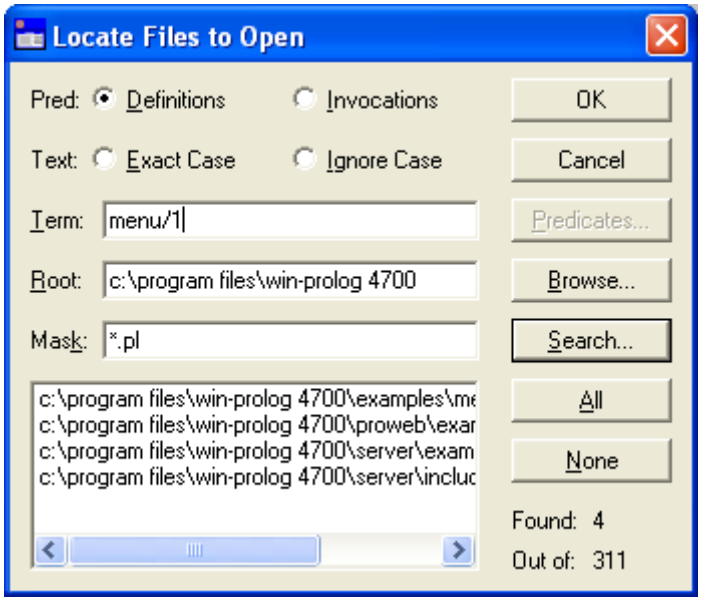
Goto Next Clause Option

The fourth option in the "Search" menu, "Goto Next Clause", works in conjunction with the "Goto Definition" option. Having highlighted the first clause for a particular predicate, clicking on the "Goto Next Clause" option clears the highlighting from the first and highlights the second. Clicking on the "Goto Next Clause" option again will highlight the third and so on. This facility is very useful if a predicate is declared as multifile (i.e. defined in more than one file). The "Goto Next Clause" option will be greyed out if no next clause exists.

Locate Files... Option

The 'Locate Files...' option leads to the "Locate Files to Open" dialog. The "Locate Files to Open" dialog allows you to search for files on disc containing a specified search string or term. Once found, you can open one or more of the files as required.

This ability to locate and find things on disc is very powerful and very helpful. For instance, if you want to see all the examples which use a certain predicate, then simply type it in and search away.



Selecting the "Definitions" radio button allows you to search for files containing a definition of the specified term of the form predicate/arity. For example, if you wanted to search for .PL files containing a definition for *menu/2*, you would select the "Definitions" radio button and type "menu/2" into the 'Term:' field and '*.pl' into the 'Mask:' field. Notice that the 'Search' button is greyed out until you have entered a valid term of the form, predicate/arity.

Selecting the "Invocations" radio button allows you to search for files containing a call to the specified Prolog term.

The "Predicates..." button leads to the "Select Search Predicate" dialog from where you can select the predicate to be searched for. This button becomes enabled when some code has been compiled.

The "Definitions" and "Invocations" radio buttons are specific to Prolog and understand the difference between definitions and calls. This is unique to LPA, and cannot be performed by, say, Windows' built-in 'Find Files' facility.

You can also search for text within any file of disc. The "Exact Case" and "Ignore Case" radio buttons allow you to search for any text anywhere in the file either matching or not matching the given case. And using the mask, you can search any kind of files.

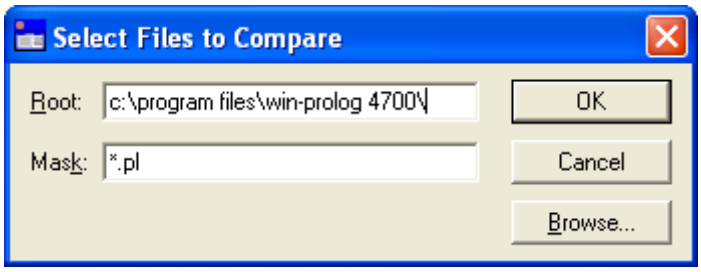
The "Root:" field holds the pathname where the search will commence and the "Level:" field holds an integer which states as to how many subfolders the search is allowed to go to. The "Browse..." button leads to the "Browse for folders" dialog from where you can select the root folder to be searched.

The "Search..." button commences a search. If the search criteria is incorrect, the "Search..." button will be disabled.

Once a search is complete, any found files will be displayed in the multibox at the bottom of the dialog. You can click on any filename to select it. The "All" button will select all filenames; the "None" button will cause all files to be unselected. Clicking on the "OK" button will open any selected file into its own rich edit window within the *WIN-PROLOG* environment.

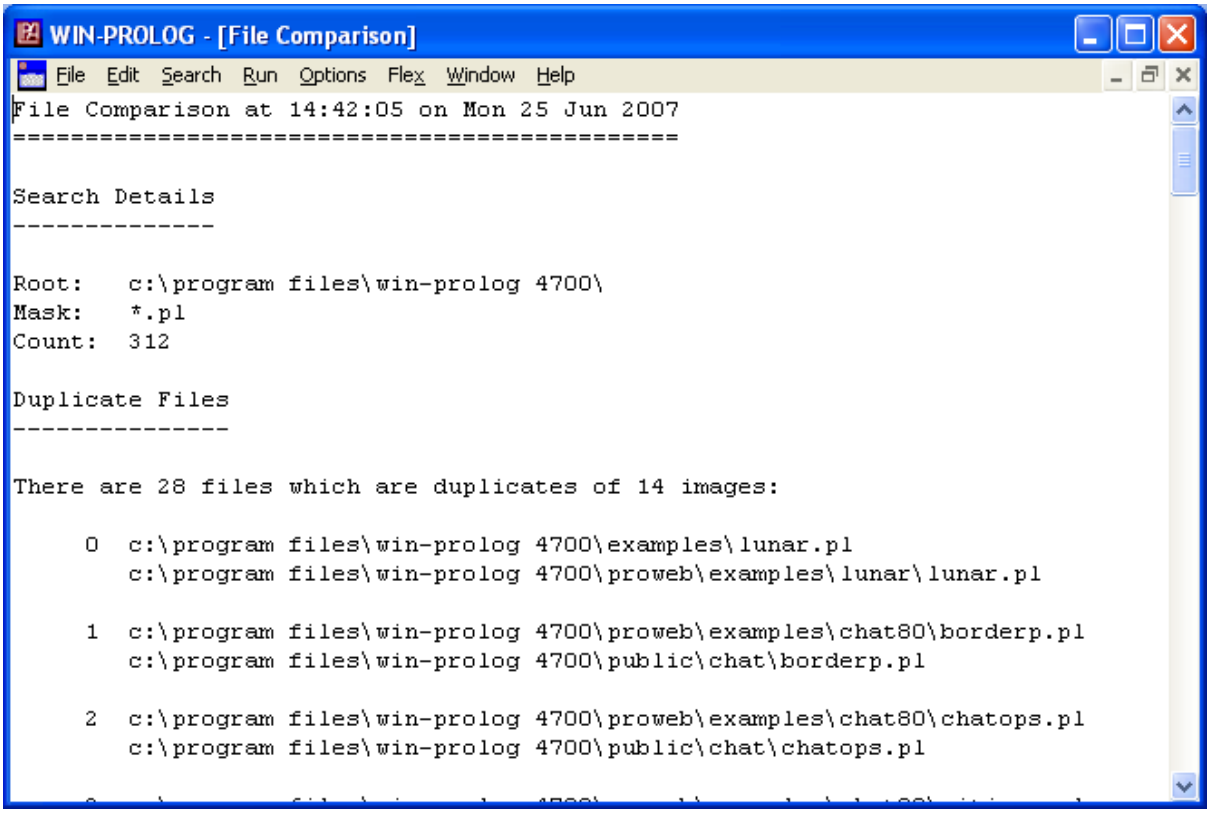
Compare Files... Option

The 'Compare Files' tool scans a disk, using a similar starting point (i.e. root, mask and level depth) as the 'Locate Files' option, and reports back lists of duplicate files, and same-named files which are different.



Unlike the features in Windows, the "duplicates" algorithm is based on file contents of files, not their names or timestamps, so will find renamed and copied files alongside originals.

This scanning function can be useful when trying to determine which photos and/or video clips which may or may not be the same!



If you click the right mouse button whilst the pointer is over a valid filename in the File Comparison window, a menu will pop up. This menu features two entries – Info and View. Clicking the Info option will cause a small dialog to appear giving statistical information about the file. Clicking the View option will cause the file itself to appear in its own rich edit window within the *WIN-PROLOG* environment.

For example, if you did:

```
C:> COPY C:\PRO386W\FOO.PL C:\TEMP\BAR.TXT
```

the 'Compare Files' tool would discover and report these as being instances of the same file:

```
There are 2 files which are duplicates of 1 images:

0   c:\pro386w\foo.pl
    c:\temp\bar.txt
```

The "differents" algorithm is sort of the same in reverse: it looks for sets of files with the same name, but of which there is at least one instance that differs from the others. So if, for example, you had three copies of "SUX.PL" in different folders, and two were identical (duplicates - see above), but the third was different, you'd get a report saying something like:

```
File "sux.pl" occurs 3 times in 2 divergent images:

    0       c:\program files\win-prolog 4200\examples\sux.pl
            c:\program files\win-prolog 4900\examples\sux.pl
```

```
1       c:\program files\win-prolog 4320\examples\sux.pl
```

The two algorithms help manage the multiple version problem, where you install, say, several different copies of **WIN-PROLOG**, which may or may not contain some of the identical examples, while others have been modified. Irrespective of timestamps or even file names, this tool helps work out which files are identical, and which are named the same, but different.

Chapter 7 - the Run Menu

This chapter describes the uses of the "Run" menu of ***WIN-PROLOG***, covering the compiling, testing and execution of programs, as well as the saving of applications. If you have any additional ***WIN-PROLOG*** plug-ins, such as the Dialog Editor, the menu items for these may also appear at the bottom of this menu.

the Run Menu

Unlike most of the other menus, the "Run" menu provides options which are closely associated with the needs of Prolog, and are unlike those found in the majority of Windows applications. These include the running of Prolog queries, and compiling or optimising Prolog programs.

Check Syntax Option

This option provides a quick way to check your Prolog source code without compiling it. It works simply by attempting to read all terms in the topmost edit window, and stops if it encounters a syntax error. If the topmost edit window contains flex source code or is the Console window this option will be greyed out.

In order to demonstrate this and the other "Run" options, you should now load a file into an edit window. Assuming you worked through the examples earlier, you will have created two files in ***WIN-PROLOG***'s *EXAMPLES* folder, *BOOZE.PL* and *GUZZLE.PL*. To open *GUZZLE.PL* into an edit window, click on "File/Open" to display the "Open" dialog, select the *EXAMPLES* folder and then *GUZZLE.PL* from the "File listing" box and then click "Open" or press *<enter>*.

Select the "Run/Check Syntax" option and when you click the mouse, or press *<enter>*, a status box will appear showing the contents of the window being analysed. Assuming there are no syntax or other errors, a message box will appear stating that the syntax of the file is ok.

Cross Reference Option

The Cross Reference is documented later in this user guide.

Compile Option

This option works on the topmost edit window only and provides the main way in which to compile your programs incrementally, so that they can be tested and debugged. If the topmost edit window is the Console window or a VisiRulke chart or graphic window, this option will be greyed out.

Select "Run/Compile" and when you click the mouse, or press *<enter>*, after the file has finished compiling a moment later, the compilation will be confirmed by a message in the Console window. Each time you compile a window, your existing code is replaced with the new definition.

The "C" in the edit window's status bar will change colour to reflect that the window has been compiled.

You are now ready to run queries on your program. Switch focus to the Console window and enter the following command:

?- **yuppie_meal(Starter,Main,Dessert,Wine).** *<enter>*

As soon as you press *<enter>*, the first solution will be displayed in the Console window. You can get further solutions by pressing the *<space bar>* or clicking the left mouse button. If you press *<enter>* or *<esc>* or click the right mouse button, the query will stop.

Compile All Option

This option compiles all the currently "dirty" text edit windows, so that they can be tested and debugged. This menu option is only available when there are source windows present that need compiling.

The "C" in each edit window's status bar will change colour to reflect that the window has been compiled.

Optimize Option

Like "Run/Compile", this option works on the topmost edit window only and provides the main way in which to optimise your programs, so that they can be executed at maximum speed. If the topmost edit window is the Console window this option will be greyed out.

Bring the *GUZZLE.PL* window back into focus by pressing *<ctrl-tab>* a few times, and select "Run/Optimize". When you click the mouse, or press *<enter>*, the file will be optimised. The "O" in the edit window's status bar will change colour to reflect that the window has been optimised. You are now ready to run queries on your optimised program.

The first time in any one *WIN-PROLOG* session that you optimise a program, there will be an initial delay while the optimising compiler code is loaded from the disk. Thereafter, the optimiser will remain in memory ready for use next time you need it.

Appendix C documents the types of compilation available.

Optimize All Option

Like "Run/Optimize", this option provides a way in which to optimise your programs, so that they can be executed at maximum speed. But whereas "Run/Optimize" optimises just the one window, and then only if the window is "dirty"; this option optimises each "dirty" edit window in turn. This option will be greyed out if there are no program rich edit windows which need optimising.

The "O" in each edit window's status bar will change colour to reflect that the window has been optimised.

Abolish Option

The "Run/Abolish" option acts on the edit window currently in focus and removes its compiled/optimised program from Prolog's memory.

Abolish All Option

The "Run/Abolish All" option acts on all currently open edit windows and removes all compiled/optimised programs from Prolog's memory.

Application Option

This option is only available if you have a Developer Edition of ***WIN-PROLOG***. It allows you to test and save stand-alone applications. The process of creating a stand-alone application appears in ***Chapter 14 - Stand-Alone Applications*** on page *126*.

Additional Plug-ins

At the bottom of the "Run" menu you might find some additional menu items. These correspond to various plug-ins you might have added to your ***WIN-PROLOG*** system. These options include the Dialog Editor and the Call Graph and are documented later in this user guide.

Chapter 8 - The Options Menu

This chapter describes the uses of the "Options" menu of ***WIN-PROLOG***, covering the settings of the ***WIN-PROLOG*** environment.

In order to demonstrate the "Options" menu options, you should now load a file into an edit window. Assuming you worked through the examples earlier, you will have created a file called *GUZZLE.PL* in ***WIN-PROLOG***'s *EXAMPLES* folder. To load this file, select the "File/Open" option. After the file has finished loading, select the "Run/Compile" option to compile it.

Trace Option

This option is used to set the debug mode to "trace". When "trace" mode is on, any queries, from the command line or the "Query" dialog, will invoke the debugger. When the debug mode is already set to "trace" this menu item will be ticked.

Debug Option

This option is used to set the debug mode to "debug". When "debug" mode is on, any spied predicates encountered during the evaluation of queries, from the command line or the "Query" dialog, will invoke the debugger. When the debug mode is already set to "debug" this menu item will be ticked.

Spypoints Option

This option is used to set or clear spypoints on the source predicates contained in the topmost edit window. When a spied predicate is executed in debug mode, it fires up whichever debugger you have chosen in the "Prolog Flags" dialog (see the "Options/Prolog Flags" option described below).

Switch focus to the *GUZZLE.PL* edit window and then choose the "Options/Spypoints..." option. The "Spypoints" dialog subsequently displayed contains a multi-choice listbox labelled "Predicates:" filled with the predicate specifications of the file *GUZZLE.PL*. Selecting a predicate specification from this listbox will place a spypoint on that predicate. Any predicates which already have spypoints will be pre-selected and you can clear existing spypoints by deselecting them.

Select the "any_meal/4" entry and then click "OK" or press *<enter>*. If you choose "Options/Spypoints..." again, you will notice that this entry is already highlighted, indicating that it is already "spied". Details of how to use the debugger are given in ***Chapter 12 - The Debuggers*** on page 95.

If you want to choose from all the predicates currently loaded, regardless of which source files the predicates are defined in, you should first select the Console window, to bring it to the top, and then select the "Options/Spypoints..." option. In the "Predicates:" listbox you will then see all of the currently non-optimised (i.e. interpreted) predicates.

Prolog Flags Option

WIN-PROLOG has a number of default settings that govern the way that its environment behaves. Selecting the Prolog Flags option displays a dialog that allows you to change these settings. These Prolog Flag settings are dealt with in greater detail in a later chapter.

Font... Option

This option is used to change the fonts used by *WIN-PROLOG* in its Console and edit windows.

The Console window can only use one font at a time. Each rich edit window can use multiple fonts. Just select the text for which you wish to change the font and go to the "Font" dialog. If there is no current selection, the new font settings will be applied to any text typed at the current caret position.

Select the "Options/Font..." option and the "Font" dialog will be displayed. The three parameters of a font can be selected in the dialog: the "Font" menu gives a list of the currently defined system typefaces, the "Style" menu shows the available font styles and the "Size" menu gives a list of point sizes.

If the "Courier" font is on your system, select it from the "Font" menu. Notice how each time you select a parameter the "Sample:" field is updated to illustrate your selection. If the character you wish to see in a particular font is not amongst the sample text, the sample text may be edited to include it.

Now set a size of 24 points for the font by selecting from the "Size" menu. Finally select the "Bold" option from the "Style" menu, then click "OK" or press *<enter>* to accept the new font settings; the Console window and edit windows will now have the font that you have selected.

Console History... Option

This option allows you to change the depth of the Console history. Select the "Options/Console History..." option and the "Console History" dialog will be displayed.

Status Bars... Option

This option allows you to enable (show) or disable (hide) the status bars which appear at the bottom of edit windows. Select the "Options/Status Bars ..." option and the "Status Bars" dialog will be displayed.

Syntax Colouring... Option

This option allows you to disable the syntax colouring or enable it and set the millisecond delay value. Select the "Options/Syntax Colouring..." option and the "Syntax Colouring" dialog will be displayed.

Colour Settings... Option

This option leads to the "Colour Settings" dialog which allows you to change the foreground and/or background colour for one or more "items". Selecting a particular item in the "Item:" listbox will cause its corresponding "Colour:" square to be highlighted. Clicking with the right button over a "Colour:" square will display the "Change Colours" dialog which allows you to change the foreground and/or background colour for that "Colour:" square and hence any "item" associated with it.

The sample code in the text box at the bottom of the dialog always reflects the "current" syntax colouring scheme as it is modified. If you dislike the sample code, please feel free to edit it.

The "Defaults..." button will restore the default colour settings.

Save Settings on Exit Option

This option is used to save the settings, made using the options menu, at the end of a ***WIN-PROLOG*** session. You would use this option if you want the settings you have made to persist to your next ***WIN-PROLOG*** session.

Select the "Options/Save Settings on Exit" option and when you have clicked the mouse or pressed *<enter>*, select the "File/Exit" option to quit from ***WIN-PROLOG***. Having done that, start ***WIN-PROLOG*** again and all the settings that you made during the course of this chapter will still be set in this new ***WIN-PROLOG*** session.

The settings are saved in a file called *PRO386W.INI* in your ***WIN-PROLOG*** "home" folder. If you want to reset the environment settings to their default values, simply delete or rename this file.

Saving the Command History between Sessions

The command history can be saved in an encrypted file upon exiting ***WIN-PROLOG*** by setting the Prolog flag, 'save_history', to 'on'.

?- **prolog_flag(save_history, X, on).** *<enter>*
X = off

Please note: The saving of the command history relies on there being a folder named 'cache' within *WIN-PROLOG*'s 'system' folder; make sure that such a folder exists.

Such a 'cache' folder can be moved to another location, so long as you tell *WIN-PROLOG* where it is via the 'cache_directory' Prolog flag:

?- **prolog_flag(cache_directory, X).** *<enter>*
X = ''
?- **prolog_flag(cache_directory, X, 'C:\Temp\Cache').** *<enter>*
X = ''

Saving the Text Formatting

By default, the saving of text formatting only occurs for files saved in *WIN-PROLOG*'s 'files' directory, and then only if you have a cache directory. However, such saving of text formatting can be applied to another folder via the 'files_directory' Prolog flag:

?- **prolog_flag(files_directory, X).** *<enter>*
X = ''
?- **prolog_flag(files_directory, X, 'C:\MyPrologFiles').** *<enter>*
X = ''

By default, the saving of text formatting relies on there being a folder named 'cache' within the *WIN-PROLOG* folder; such a 'cache' folder can be moved to another location, so long as you tell *WIN-PROLOG* where it is via the 'cache_directory' Prolog flag:

?- **prolog_flag(cache_directory, X).** *<enter>*
X = ''
?- **prolog_flag(cache_directory, X, 'C:\Temp\Cache').** *<enter>*
X = ''

Chapter 9 - The Window Menu

This chapter describes the uses of the "Window" menu of *WIN-PROLOG*, covering the handling of *WIN-PROLOG* system windows. This menu provides the standard Windows functions for arranging and finding windows on the screen.

Cascade Option

This option cascades the windows that are not currently iconised. Cascading arranges the windows so that they cascade down the screen from the top left corner towards the bottom right corner. The first window is placed at pixel co-ordinates (0,0) in the Main window's client area and each successive window is placed on the screen shifted down and to the right by 22 pixels.

Tile Option

This option tiles the windows that are not currently iconised. Tiling arranges the windows vertically so that they fill the Main window's client area and no window overlaps another.

Stack Option

This option stacks the windows that are not currently iconised. Stacking arranges the windows horizontally so that they fill the Main window's client area and no window overlaps another.

Arrange Icons Option

This option arranges any icons present in the Main window's client area. Arranging icons places them in order at the bottom left hand corner of the Main window's client area.

The Lower Section of the Window Menu

Each time a new "edit" or "user" type window is created the title of the window is appended to the lower section of the "Window" menu; this includes the Cross Reference window. If you make a selection from the lower part of the "Window" menu the window whose name you selected will be brought into focus.

Chapter 10 - The Help Menu

This chapter describes the uses of the "Help" menu of ***WIN-PROLOG***. This menu provides the standard Windows functions for accessing the ***WIN-PROLOG*** help file itself (and those of other installed toolkits) and information on how to use it.

The WIN-PROLOG Help Option

This option opens the ***WIN-PROLOG*** help file at the contents topic. The contents topic of the ***WIN-PROLOG*** help file contains links to:

An alphabetically sorted list of predicate descriptions.

A logically sorted list of predicates. Where you can find the predicate description you want according to its function.

Some reference information on error messages and command line switches.

Information on technical support.

If the help file fails to appear when ***WIN-PROLOG*** is being run over a network, ensure that you have write access as the Help (PRO386W.HLP) file needs to generate a PRO386W.GID file.

The How to Use Help Option

This option loads the "How to Use Help" topic into the Microsoft help application. This topic should be fairly self-explanatory; simply follow the instruction for guidelines on how to use the Microsoft help application.

The About *WIN-*PROLOG... Option

The "Help/About WIN-PROLOG..." option is used to display version and licence information about your copy of ***WIN-PROLOG***, combined with the ***WIN-PROLOG*** bitmap.

Additional Menu Entries

Additional menu entries for the help files of installed toolkits may also appear on the Help menu. For example, if you have flex installed, the Help menu will gain two additional menu options, namely "Flex Help" and "About flex..."; the "Flex Help" option launching the flex online help file, FLEXHELP.HLP.

Chapter 11 – Prolog Flags

This chapter describes the Prolog Flags dialog.

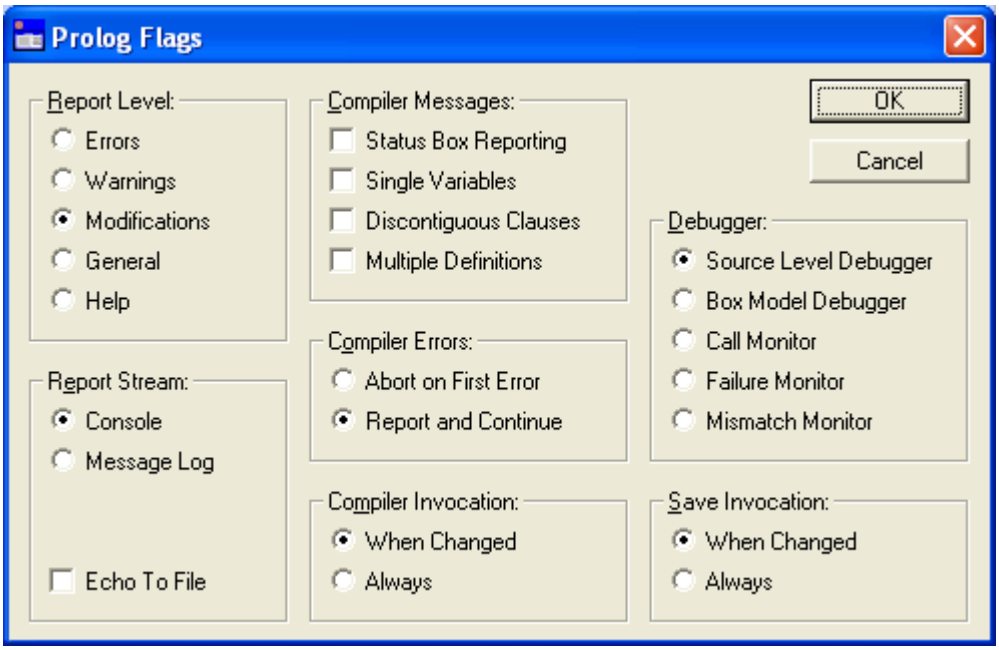


Figure 30- Prolog Flags dialog

In order to demonstrate the "Options/Prolog Flags..." menu option, you should now load a file into an edit window. Assuming you worked through the examples earlier, you will have created a file called *GUZZLE.PL* in ***WIN-PROLOG***'s *EXAMPLES* folder. To load this file, select the "File/Open" option. After the file has finished loading, select the "Run/Compile" option to compile it.

The Prolog Flags Dialog

Select the "Options/Prolog Flags..." menu option to display the "Prolog Flags" dialog.

The Compiler Invocation Group

The first setting we are going to change governs when the compiler is allowed to compile. Switch the radio button in the "Compiler Invocation:" group from "When Changed" to "Always". Setting this option means that whenever the "Run/Compile" option is selected the topmost edit window will be compiled regardless of whether it has changed since the last compilation.

The Compiler Messages Group

Before clicking "OK" or pressing *<enter>* to accept the dialog, we are also going to set two other options, this time in the "Compiler Messages:" group. Click the "Single Variables" check box. When this option is set, every time a named variable is encountered in a clause that is being compiled, and that variable is used only once in the clause, it will be reported in a warning. Now click on the "Status Box Reporting" check box. When this option is set the status of the compilation of files, as they are compiling, is displayed in a status box. Having set the "Compiler Messages:Single Variables", the "Compiler Messages:Status Box Reporting" and the "Compiler Invocation:Always" options click "OK" or press *<enter>* to accept the dialog.

Now select the "Run/Compile" option again to compile the file; this time, when you click the mouse, or press *<enter>*, the status box will be displayed giving constantly updated information on the status of the file GUZZLE.PL. Notice the "Warnings :" field at the bottom of the status box, showing a running count of the warnings as they are encountered in the file. When compilation has finished a message box will be displayed stating that several warnings were encountered during compilation; click "OK" or press *<enter>* to close the message box. Compilation of the file will take longer than normal; this is because each clause that is being compiled is reported to the status box.

Note: warnings are not as serious as errors because, as in the case of GUZZLE.PL, it may be that there is nothing actually wrong with the programs that exhibit them.

The warnings, by default, are sent to the Console window. If you now bring that window to the front and look at the second warning it should look something like:

```
* Single use variable(s) in clause 2 of predicate menu /1
* Meal
```

Notice that the warnings are denoted by the "*" symbol. Now bring the GUZZLE.PL window to the front and scroll the window until you can see the second clause of the *menu/1* predicate; it should look like:

```
menu(Meal):-
        wclose(display_window).
```

as you can see the variable "Meal" is not used anywhere else in the clause and good style would indicate that this named variable should be replaced by an underscore.

The usual reason for single use variable warnings is a variable name has been mistyped; if you stick to the style convention that truly unused variables are anonymous (represented by underscores), turning on the single variable option will lead you straight to these possible typing errors.

The other warnings that can be set from the "Compiler Messages:" group are "Discontiguous Clauses" - warnings about clause definitions that are interspersed with the definitions of other clauses, and "Multiple Definitions" - warnings about clauses not declared as multifile that exist in more than one file. The warnings come in the same way as for the single variables warnings described above.

The Compiler Errors Group

The other setting in the "Prolog Flags" dialog that deals with the compiler is the "Compiler Errors" group. This allows you to govern whether the compiler aborts compilation upon the first error encountered or whether it carries on to see if the rest of the file will compile. By default this option is set to "Report and Continue"

Reports

In the "Prolog Flags" dialog, in addition to controlling the compiler options, you can also control the type of reports made by ***WIN-PROLOG*** and where these reports are displayed. Select the "Options/Prolog Flags..." option; this will display the "Prolog Flags" dialog. In this dialog you will see two radio button groups that deal with the reporting of information in ***WIN-PROLOG***. These groups are: the "Report Level:" group, where you can set the range of reports made, and the "Report Stream:" group, where you can set the destination of the reports.

The Report Level Group

Runtime reports are divided into separate levels. The lowest level is the "Errors" level; if this is selected, only error reports are made. The highest level is the "Help" level; if this is selected, all errors, warnings, modifications, general and help reports are made.

Select the "Warnings" radio button in the "Report Level:" group. Selecting this button means that only errors and warnings will be reported.

The Report Stream Group

Before selecting the "OK" button and accepting the dialog, we are going to set one other report preference. This time to specify the destination of the reports.

The "Report Stream:" group gives you three choices as to where to send messages, with the additional capability of echoing the reports to a disk-based file. If the "Console" button is selected then all reports are sent to the Console, regardless of their origin. If the "Message Log" button is selected then all reports are sent to a special "Messages" log window. Selecting the "Echo To File" check box with any of the above radio buttons will duplicate any reports made to a file called *PRO386W.MSG* saved in the Windows temporary directory; this default file extension, *.MSG*, can not be altered.

Set the report stream by clicking on the "Message Log" radio button, then click "OK" or press *<enter>* to accept the dialog. If you then select the "Run/Compile All" option, the file GUZZLE.PL will be compiled and will generate the same warnings as before, but this time instead of going to the Console window they will be sent to the "Messages" log window. Notice how all of the reports begin with the "*" symbol; this shows that they are all warnings. None of the modification reports - ones that begin with the "#" symbol - have been displayed, because of the selection we made in the "Report Level:" group.

The Debugger Group

In the "Prolog Flags" dialog there is a group of radio buttons that allow you to set the *WIN-PROLOG* debugger. The debugger you set in this dialog will be invoked when the trace or debug modes are on. The operation of the debuggers themselves is described in ***Chapter 12 - The Debuggers*** on page 95.

The Save Invocation Group

While the "Prolog Flags" dialog is still open, there is just one more group to be discussed. The "Save Invocation:" group allows you to set when a source file can be saved from the "File" menu. The first choice is "When Changed"; when this radio button is selected the "File/Save" option will be greyed out after a file has been saved and until the file is changed again. The second choice is, "Always"; when you select this radio button the "File/Save" option will always be available for a source file. Click on the "Cancel" button or press the <esc> key to cancel the dialog.

Chapter 12 - The Debuggers and Monitors

This chapter describes the debuggers and monitors of ***WIN-PROLOG***, outlining the different models available, and the source level debugger in detail.

The Debugger Models

There are five different debugger "models" supplied with ***WIN-PROLOG***: although each is a completely independent program, only one may be in use at a time. Two of the models, the "source level" and "box model" debuggers, are fully interactive; the remaining three are simply "monitors" which echo information about predicate calls, argument mismatches and predicate failures.

In addition, you can define and integrate your own debugger (using the '?DEBUG?'/1 hook).

Also, a library file, TRACE.PL, is supplied as a configurable debugging utility.

Selecting a Debugger

By selecting the "Options/Prolog Flags..." menu option, the "Prolog Flags" dialog will be displayed. In this dialog, you will see a group labelled "Debugger:". The radio buttons in this group list the five ***WIN-PROLOG*** debuggers, with the currently selected debugger indicated by the active radio item. This debugger will be invoked automatically whenever a "spied" predicate (one upon which a spypoint has been set) is executed while in "debug" mode, or, whenever you run a query, if you are in "trace" mode.

You can programmatically change the debugging model, without going via the Prolog Flags dialog, simply by using the *prolog_flag/3* predicate with the atom, debug_file, as the first argument.

The Source Level Debugger

Of the five debugger models, the source level debugger is by far the most sophisticated, and is the main subject of this chapter. It is a highly interactive debugger with programs displayed in a scrollable, resizeable dialog with two panes, an adjustable horizontal splitter bar, and ten buttons. Programs are displayed clause by clause in the top pane, with the current call indicated through highlighting. Additional information, such as the name and arity of the predicate being debugged, as well as clause number, next clause number and recursion depth, is displayed in a number of status boxes. Variable bindings are displayed in the bottom pane, either as a list or 'in-situ'.

Compiling for Debugging

Before you can debug a program, it is necessary to compile it. The debugging mechanism only works on code which has been incrementally compiled; optimised code is not traceable. You should use the "Run/Compile" option, and not "Run/Optimize", to compile your program.

In order to demonstrate the debuggers, you should now load a file into an edit window. Assuming you worked through the examples in ***Chapter 5 - the File Menu*** on page *66*, you will have created two files in ***WIN-PROLOG***'s *EXAMPLES* folder, *BOOZE.PL* and *GUZZLE.PL*. Open *GUZZLE.PL* into an edit window via the "File/Open..." option.

Once the *GUZZLE.PL* program has loaded, select "Run/Compile" to consult the program, as described in ***Chapter 7 - the Run Menu*** on page *82*. If everything is OK, you should now see a simple confirmation message in the Console window, as shown in *Figure 31*.

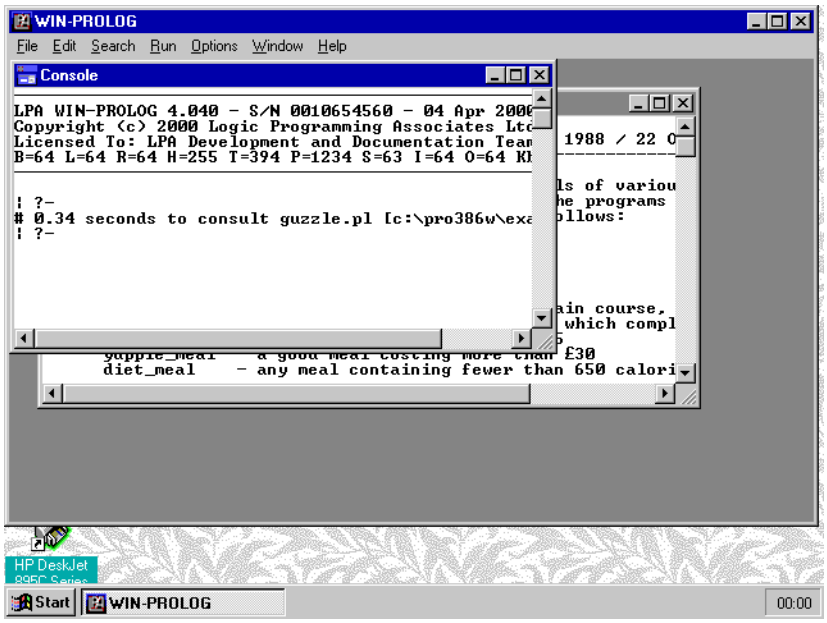


Figure 31 - Confirmation that GUZZLE.PL has been consulted

Selecting the Debugger Model and Setting the Debug Mode

By default, ***WIN-PROLOG*** assumes you will be using the source level debugger in debug mode. To be certain you are using the source level debugger in debug mode, check these options now. Select "Options/Prolog Flags..." from the Main window, as shown in

Figure 32, and when you click the mouse or press *<enter>*, make sure the "Source Level Debugger" radio button is selected in the "Prolog Flags" dialog, as shown in *Figure 33*, and then click "OK" or press *<enter>*.

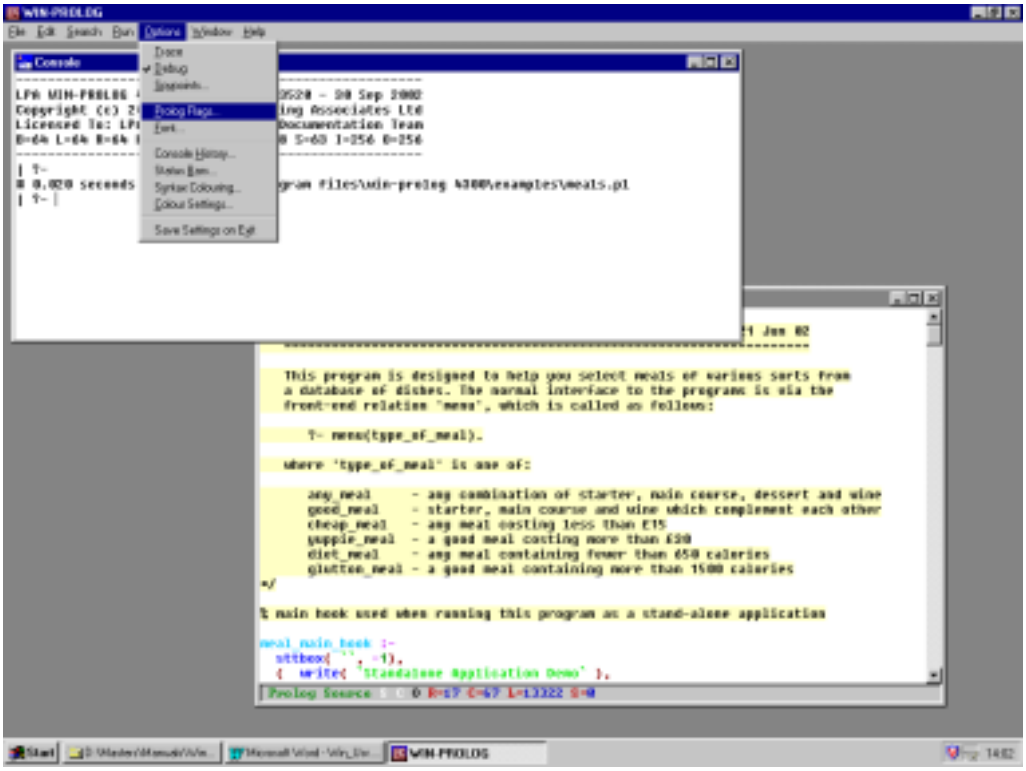


Figure 32 - The "Options/Prolog Flags..." option

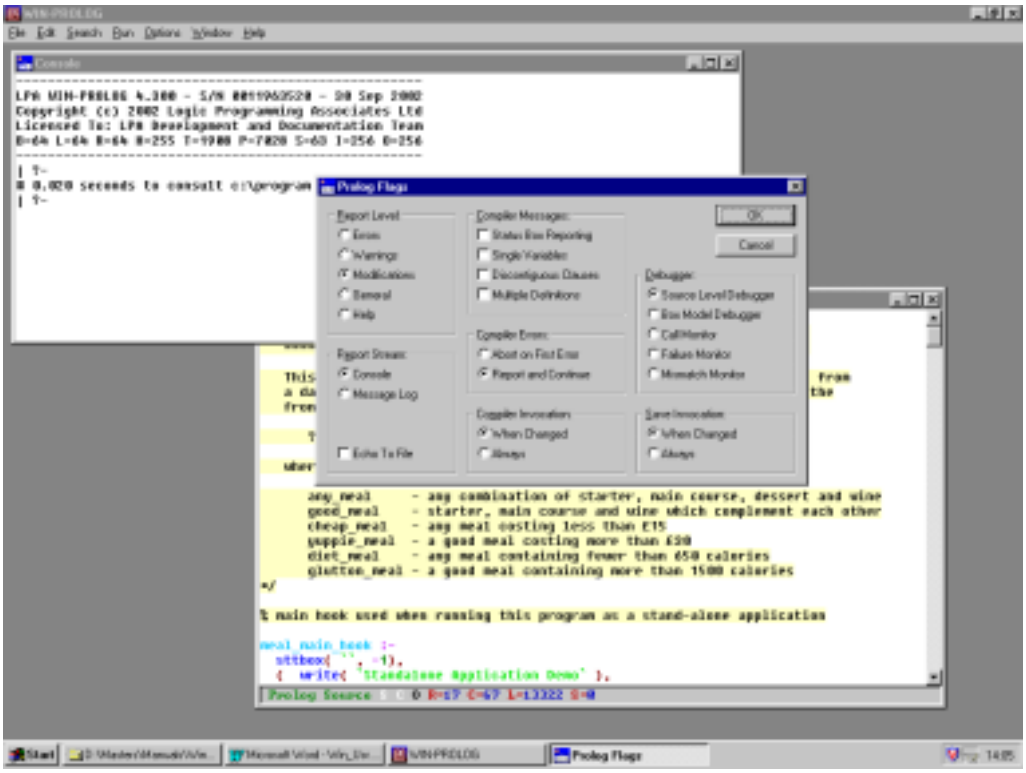


Figure 33 - Selecting the source level debugger

Setting Spypoints

The next stage to debugging is to set one or more "spypoints" on programs. A spypoint is a hidden flag which causes the debugger to be switched on whenever a "spied" predicate is called. Select the "Options/Spypoints..." option, as shown in *Figure 34*, and when you click the mouse or press *<enter>*, you will see a multi-choice list of all the currently loaded source predicates. Click on the "cheap_meal/4" entry, as shown in *Figure 35*, and then click "OK" or press *<enter>*. You should now have set a spypoint on the predicate *cheap_meal/4*.

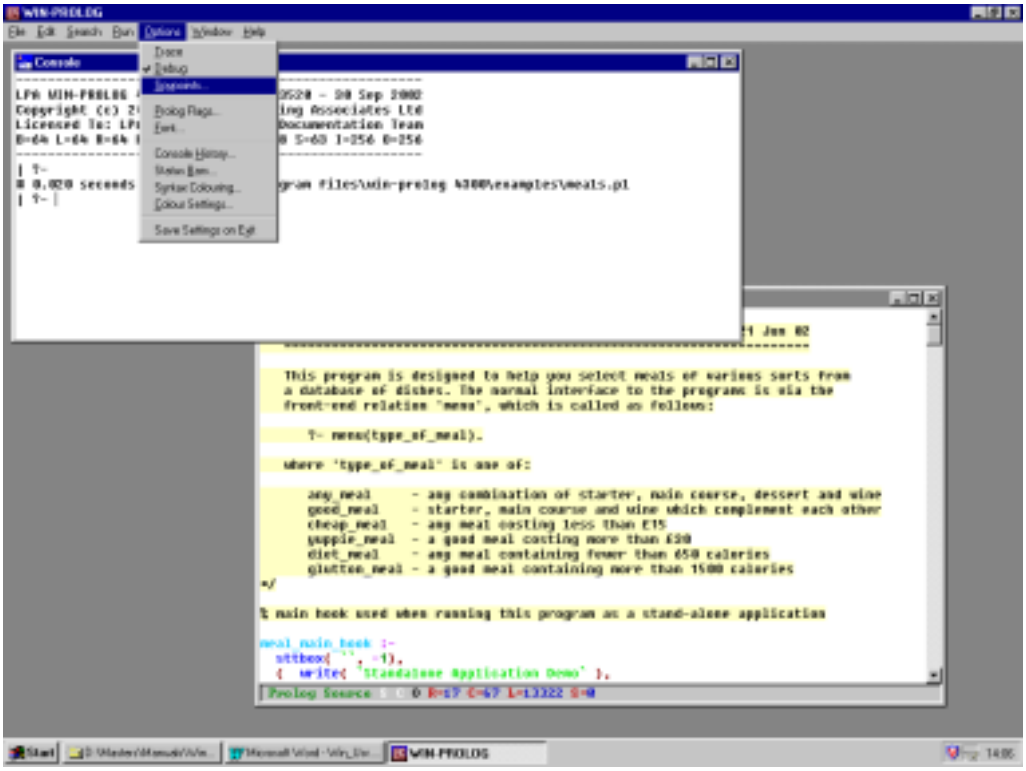


Figure 34 - The "Options/Spypoints..." option

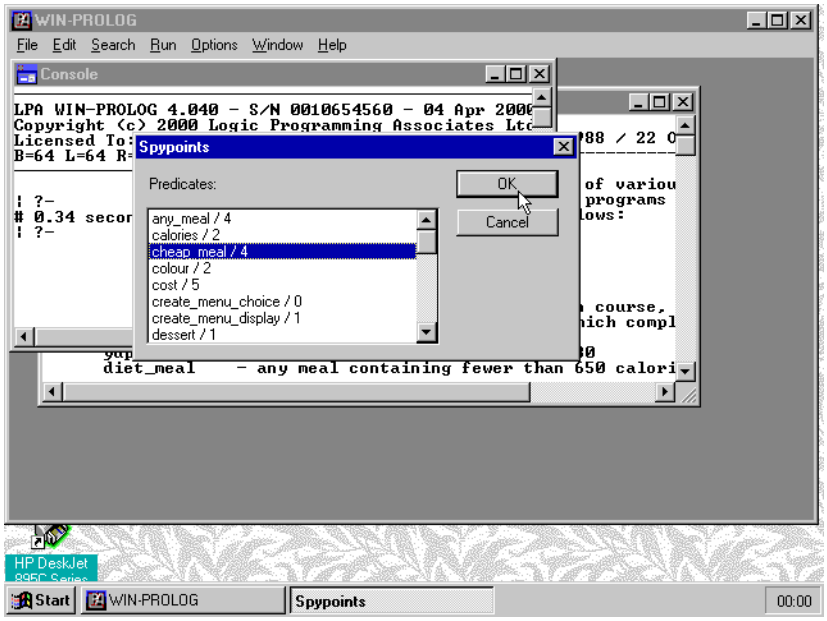


Figure 35 - Setting a spypoint on a predicate

Spypoint Information

When you set a spypoint on a predicate, and the report level is set to either "Modifications", "General" or "Help", information about the spypoint just set is written to the Console. You should see the report as shown in *Figure 36*.

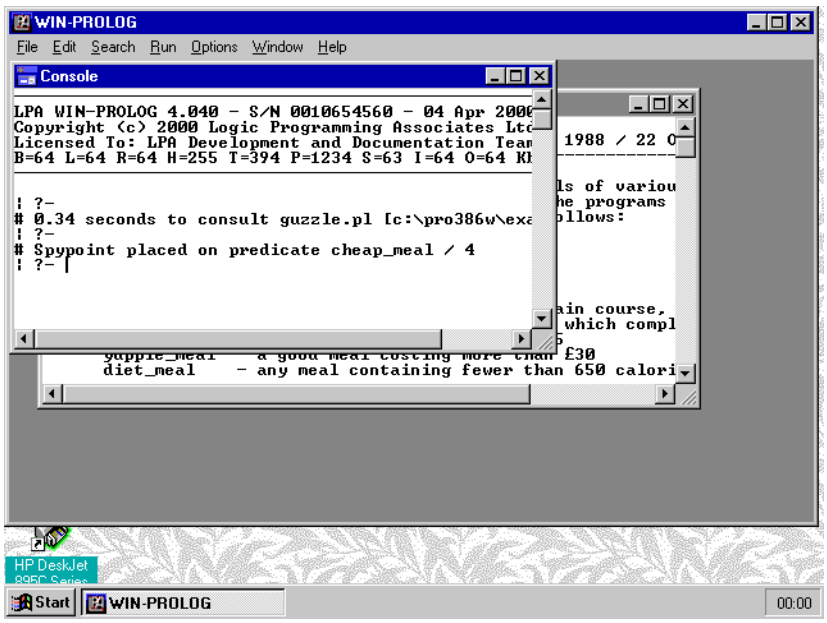


Figure 36 - Information on the set spypoint

Running a Query

Once you have set up the debugger as just described, all you need to do is run a query. When execution reaches a spied relation, the debugger will switch in automatically. Switch focus to the Console window and type the following query into its input zone:

?- **cheap_meal(S,M,D,W).** *<enter>*

Once you have typed the query and pressed *<enter>*, there will be a short delay while the source level debugger is loaded.

The Source Level Debugger Dialog

When the source level debugger is activated, it creates a large multi-window dialog, as shown in *Figure 37*. This serves two purposes: firstly, it neatly displays your program and other debugging information, and secondly, it accepts debugging commands in reaction to clicks on its row of ten buttons.

Directly underneath the command buttons is a large, scrollable window. This is the "source" window, and is used to display the clause being debugged. The goal which is being tested at any one point is highlighted, and one of a number of symbols is displayed alongside the goal to indicate the direction of flow (i.e. "C" for call, "F" for fail, "D" for done, "R" for redo and "E" for exit).

Beneath the source window is a collection of status boxes, listing the predicate name, current clause number, next candidate clause number (if any), and recursion depth.

At the bottom of the "Source Level Debugger" dialog is a second scrollable window, this time used to display variable bindings. You can display these in one of two ways: by default, variables are simply listed with their bindings; alternatively, you can display them in context, in an instantiated version of the clause.

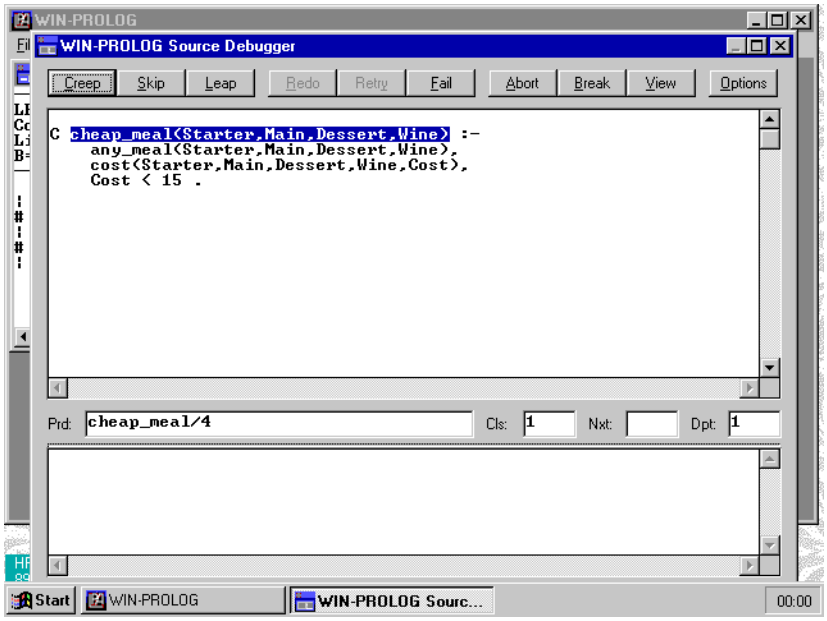


Figure 37 - The "Source Level Debugger" dialog

The "Option" command lets you change this and other aspects of the debugger (see above).

The Creep Command

This is the basic debugger command, allowing you to single-step through your program. If you click on "Creep", or press <C>, the highlight box will advance, as shown in *Figure 38*.

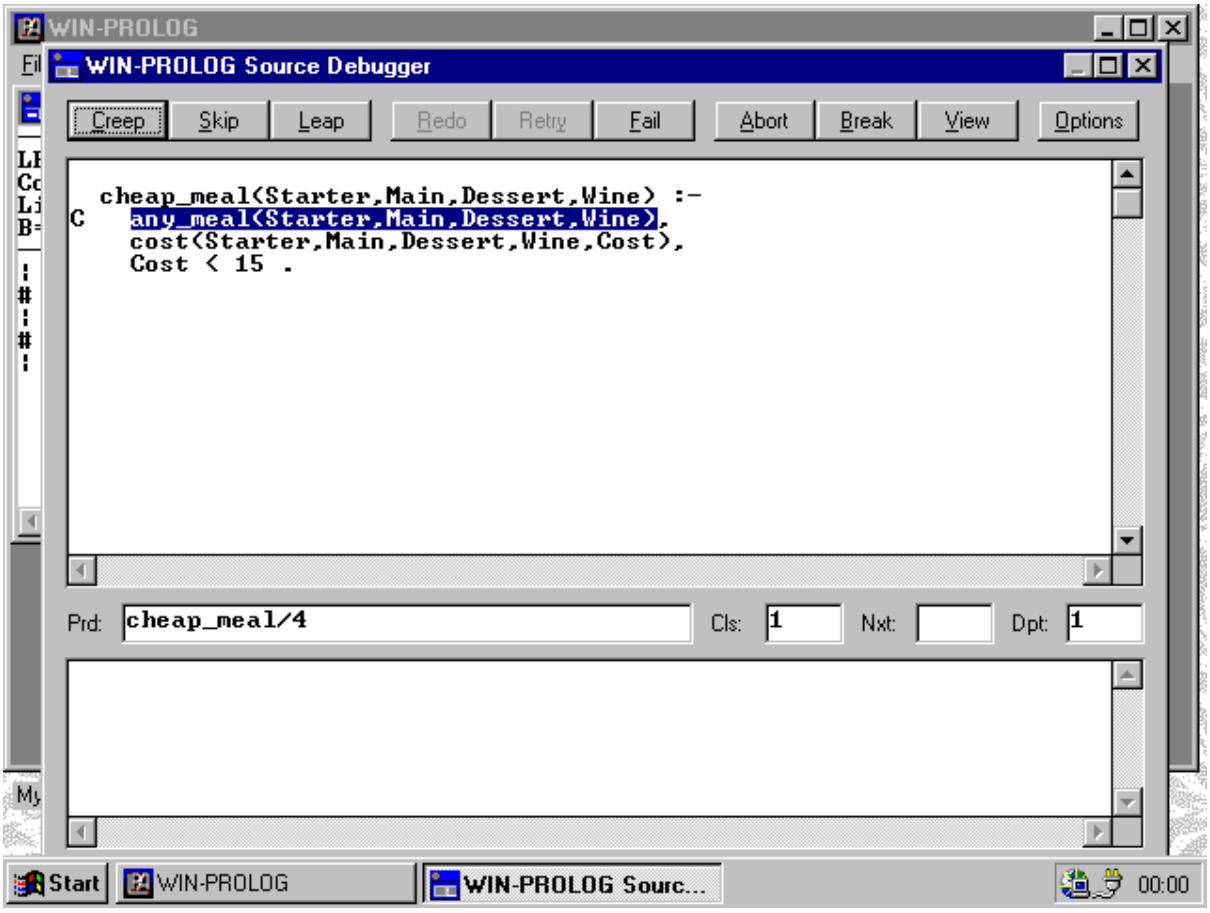


Figure 38 - Advancing into the body by the creep command

The "C" symbol indicates that you are now about to call the goal *any_meal(...)*: once again, click on "Creep" or press <C>. As you creep into a goal, the program being called is brought into view, as shown in *Figure 39*.

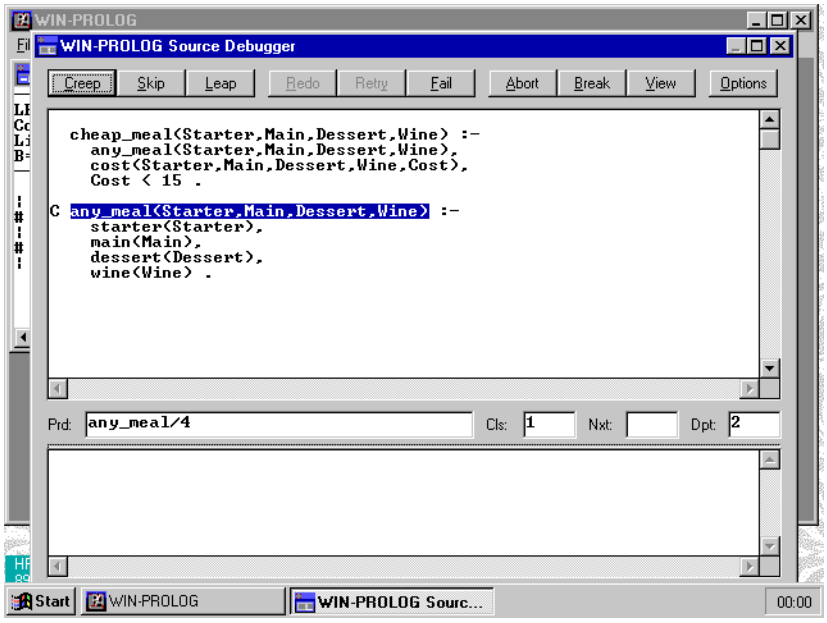


Figure 39 - Further source code displayed by creep

Perform two more creeps, to bring the *starter/1* relation into view, as shown in *Figure 40*.

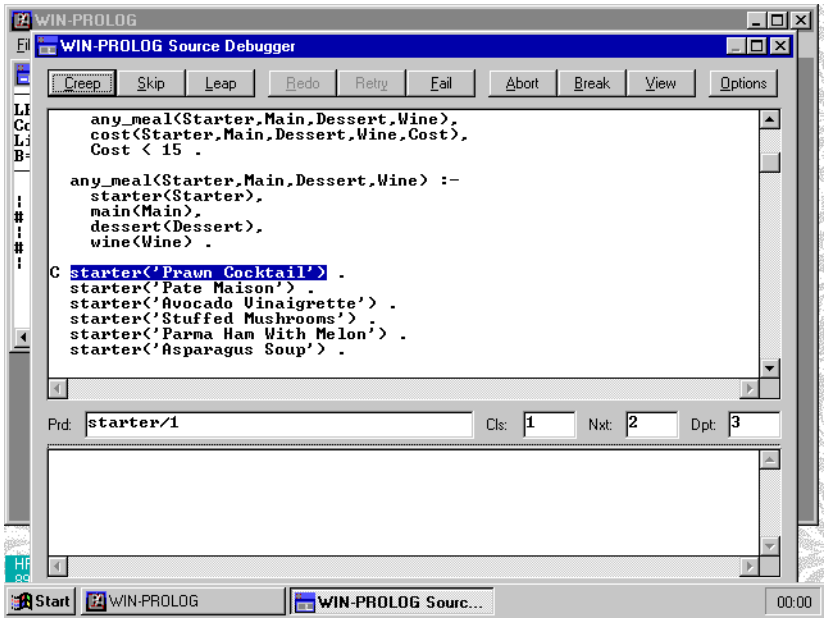


Figure 40 - Advancing still further with creep

Press creep once again and notice that the "C" symbol has been replaced with an "E"; this symbol means that the highlighted goal item has completed execution, and is ready to continue. Select creep once more, and the highlight will return to the *any_meal/4* clause, again with the "E" symbol to indicate completion of a goal, as shown in *Figure 41*. Notice also that the variables window has been updated to show the variable "Starter = 'Prawn Cocktail'".

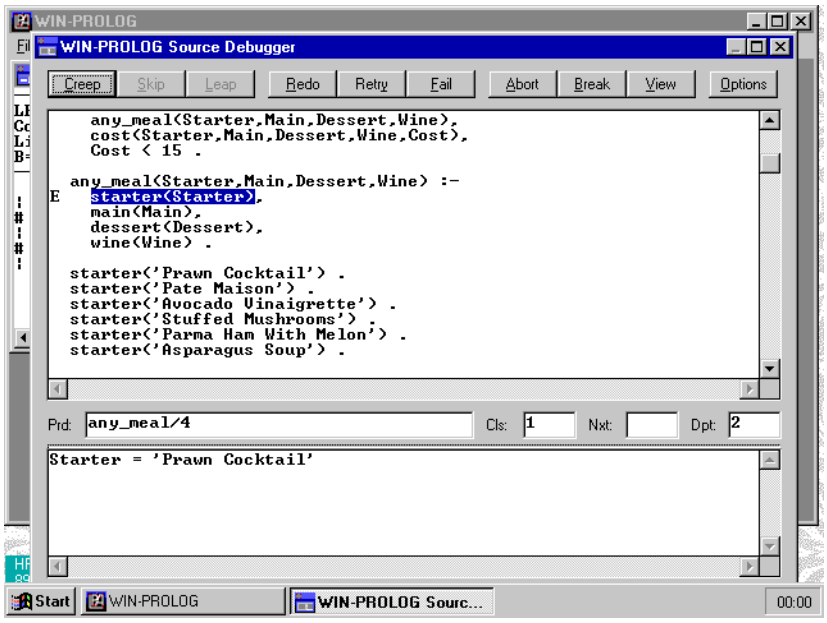


Figure 41 - Exiting a successful call with creep

Creep one more time, to land on the call to *main(...)*, as shown in *Figure 42*. Once again, the "C" symbol is shown to indicate that you are about to make a call.

Note that if a goal exits and there are no other solutions possible, the "E" symbol is replaced by a "D", meaning that the query has been done for the last time.

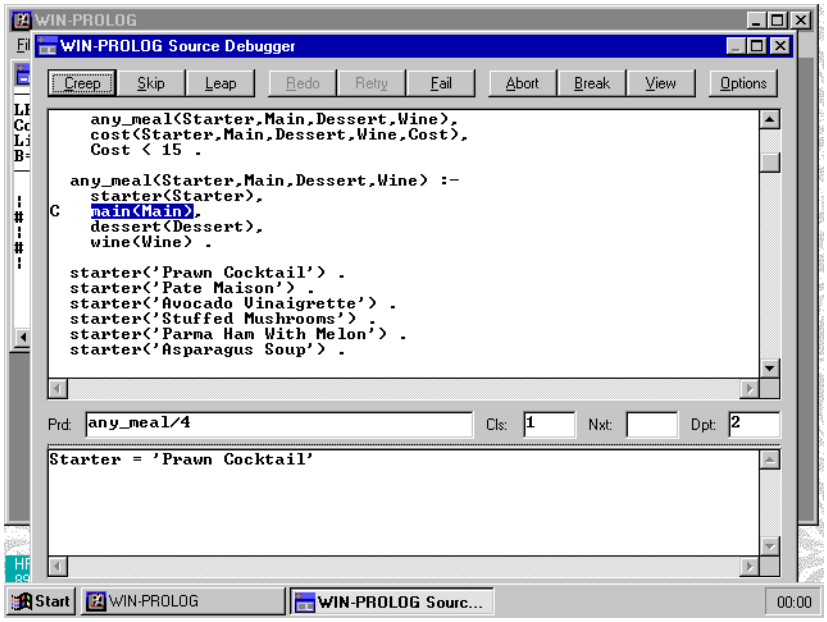


Figure 42 - Moving to the next goal with creep

The Skip Command

While creep allows you to single-step through your program, following every predicate call to its completion, there will be times when you want to skip over a given call without tracing it. This is exactly what the skip command lets you do. If you click on "Skip", or press <S>, the call to *main(...)* is made without the debugger interacting until the call has completed, and you are ready for the next call, as shown in *Figure 43*.

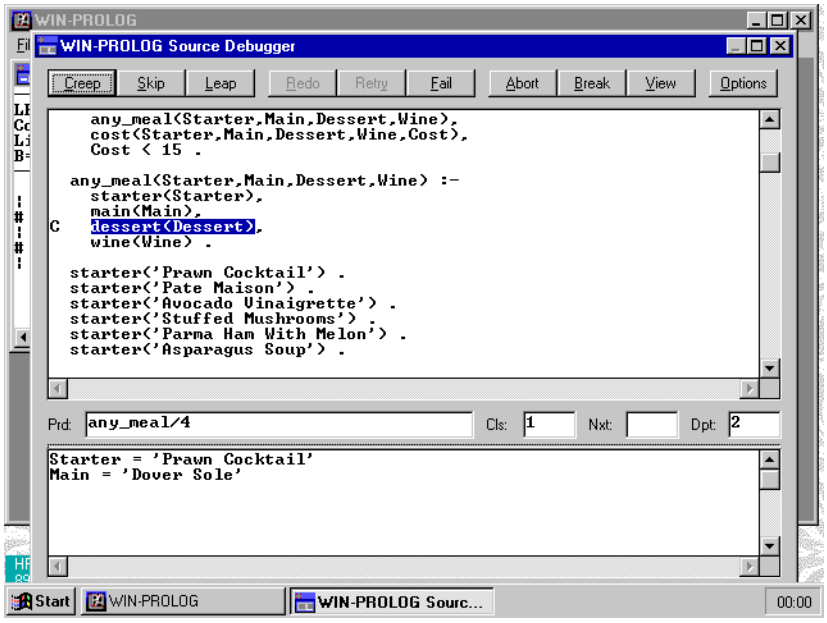


Figure 43 - Advancing without tracing with the skip command

Notice that no source code was brought in for *main/1*, since you skipped over that call. Notice also that another variable has been bound, this time "Main = 'Dover Sole'". Perform two more skips, to return the highlight to the head of the *any_meal/4* clause, as shown in *Figure 44*.

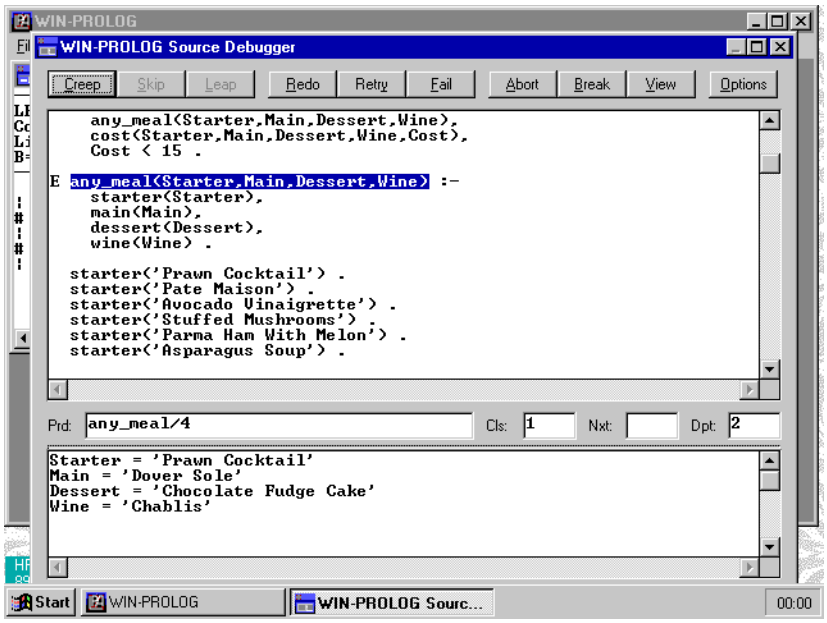


Figure 44 - Back at the clause head after two further skips

At this point, you cannot skip again, because the clause is completed. You must use creep to return to the *cheap_meal/4* clause, as shown in *Figure 45*.

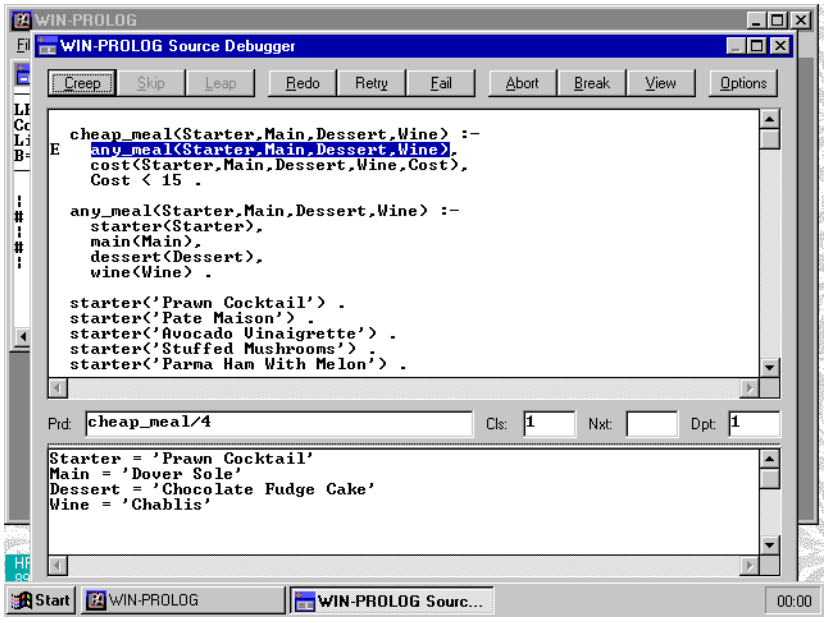


Figure 45 - Returning to the original clause with creep

Perform a creep to highlight the *cost(...)* goal, as shown in *Figure 46*, and then skip to get to the "Cost < 15" line, as shown in *Figure 47*.

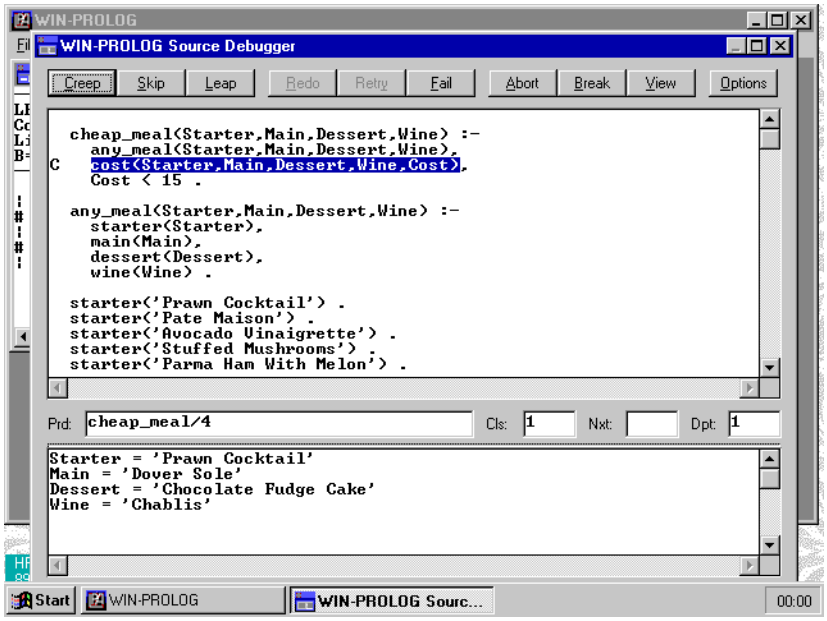


Figure 46 - Advancing to the next goal with creep

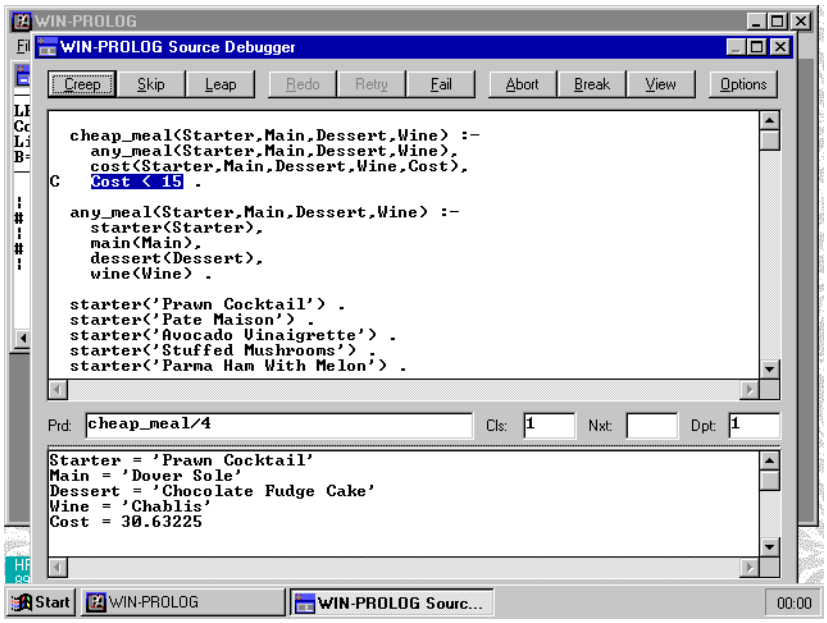


Figure 47 - Advancing to the next goal with skip

Attempt one more skip: if the comparison had worked, the highlight would have returned to the head of the *cheap_meal/4* clause as before, but in this case the call has failed. Failure is indicated by an "F" symbol, as shown in *Figure 48*. Perform a creep, and the highlight will return to the last call that which you entered with a creep, as shown in *Figure 49*. Note that this time the symbol is an "R"; this is used to indicate that there are further choice points in the call. If you select creep again, then you will return to the *any_meal/4* clause, just as before. If you prefer, you can skip the attempt to redo the goal: this is useful if you don't want to step through the program over and over each time it backtracks for further solutions.

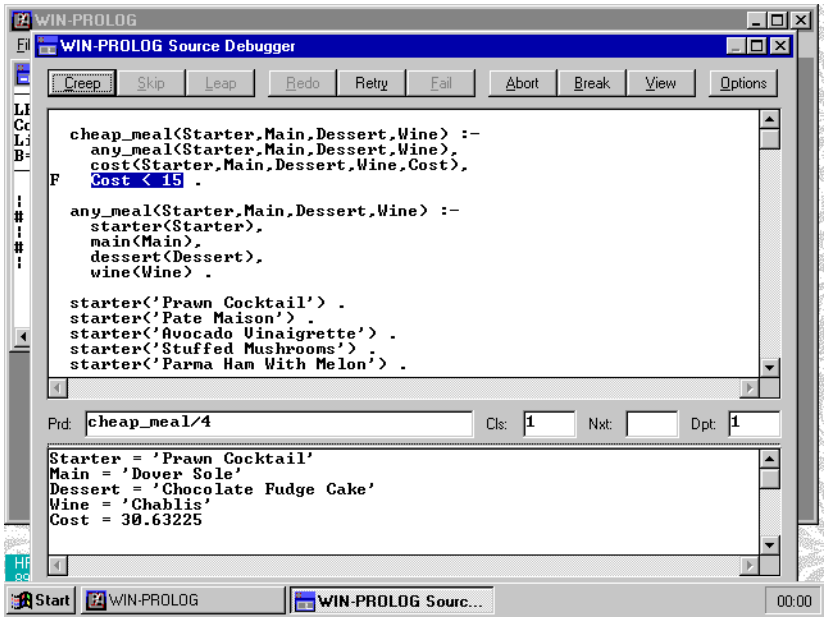


Figure 48 - Indicating failure after a skip

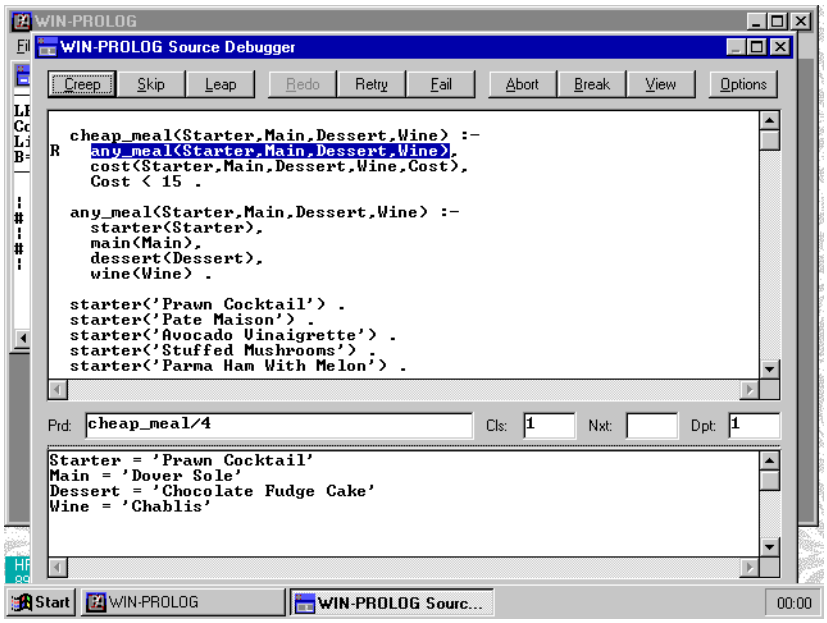


Figure 49 - Returning to a choice point with creep

The Leap Command

Just as skip allows you to skip over the debugging of a single goal, the leap command lets you turn off all debugger interaction until the next time a spypoint is reached. Because we only set one spypoint, on *cheap_meal/4*, we cannot use leap just yet. Instead, you should now abort the current query by clicking "Abort" or pressing <*A*>, to return to the Main window. Now select "Options/Spypoints..." as before, and click on the "cost/5" entry to select it in addition to "cheap_meal/4", as shown in *Figure 50*. When you click "OK" or press <*enter*>, the additional spypoint will be set.

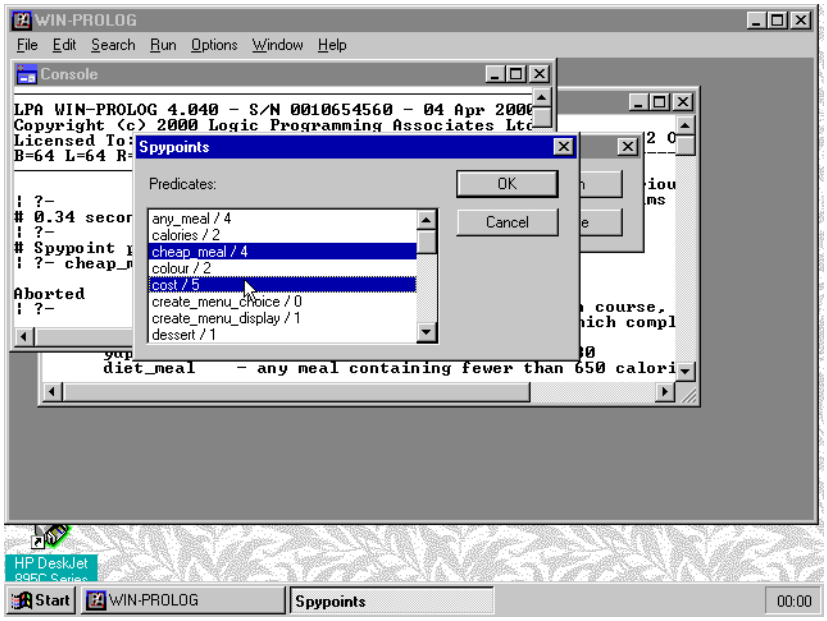


Figure 50 - Setting a spypoint on an additional relation

Now return to the Console window and use *<ctrl>+<page_up>* and *<ctrl>+<page_down>* to bring back your previous *cheap_meal(S,M,D,W)* command into the input zone; press *<enter>* to re-enter it. At once, the "Source Level Debugger" dialog will reappear on the screen. This time, click "Leap" or press *<L>*. The debugger will run, instantiating variables as it goes, until it encounters a call to a program which has been spied, in this case *cost(...)*, as shown in *Figure 51*.

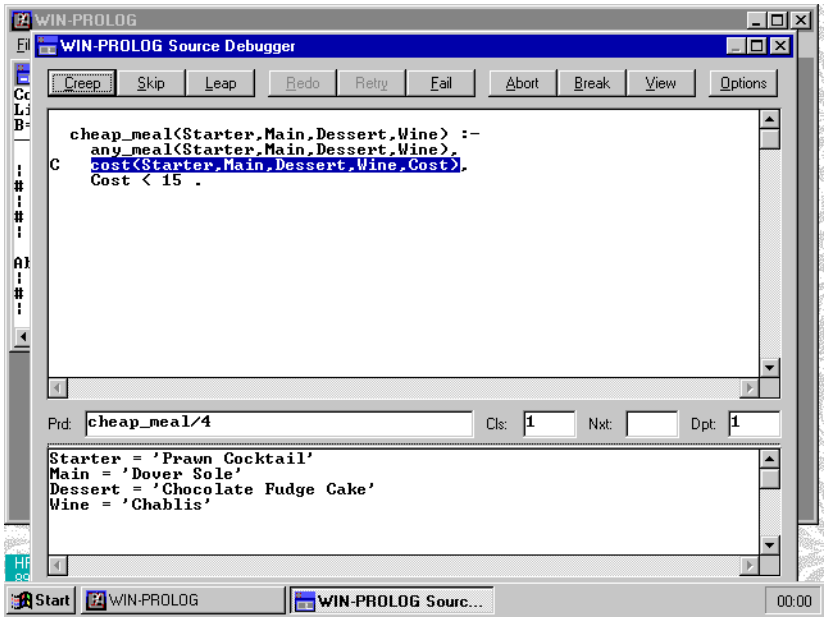


Figure 51 - Using leap to execute until the next spied relation

As soon as the leap command lands on a spied relation, you can continue debugging as if you had performed creeps all the way there. You can now proceed with creep, skip and further leap commands as you like.

The Redo Command

This is the first of three commands which can be used to modify the execution of a program being debugged, rather than simply view the program working as written. The redo command is available whenever an exit port is displayed (the "E" symbol), and allows you to force the call that has just exited to backtrack and find an alternative solution. To see this, click on "Abort" or press <*A*>, and then use the Console's <*ctrl*>+<*page_up*> and <*ctrl*>+<*page_down*> facility to run your query once again. Perform a creep six times, until you exit the call to *starter(...)*, as shown in *Figure 52*.

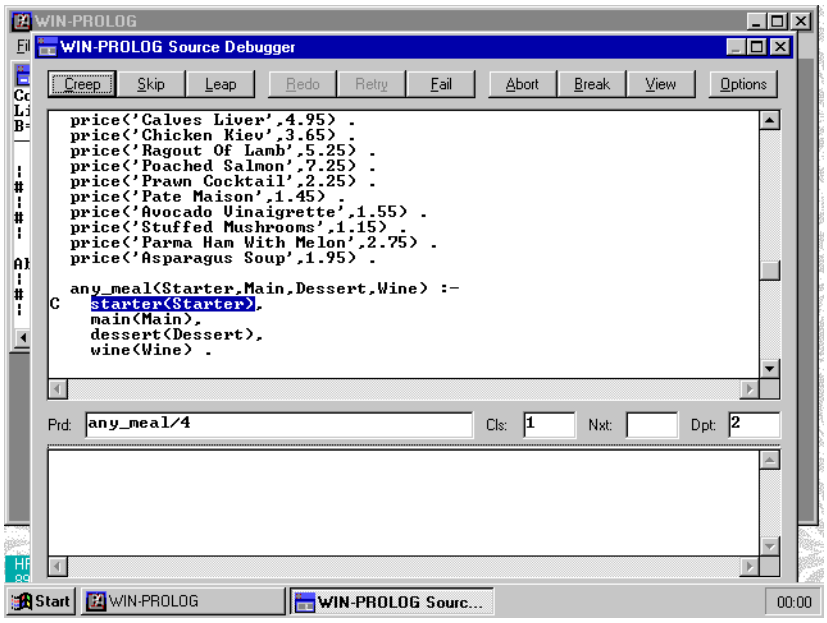


Figure 52 - An exit port reached after six creeps

The "E" symbol shows that the call has just exited, with the solution "Starter = 'Prawn Cocktail'". You can now force this call to redo its evaluation, to come up with the next solution available by backtracking. Click on "Redo", or press <*R*>, and the highlight will move to the second clause of *starter/1*, as shown in *Figure 53*. Now perform two creeps, and you will be returned to the *starter(...)* call as before, but this time with the solution "Starter = 'Pate Maison'", as shown in *Figure 54*.

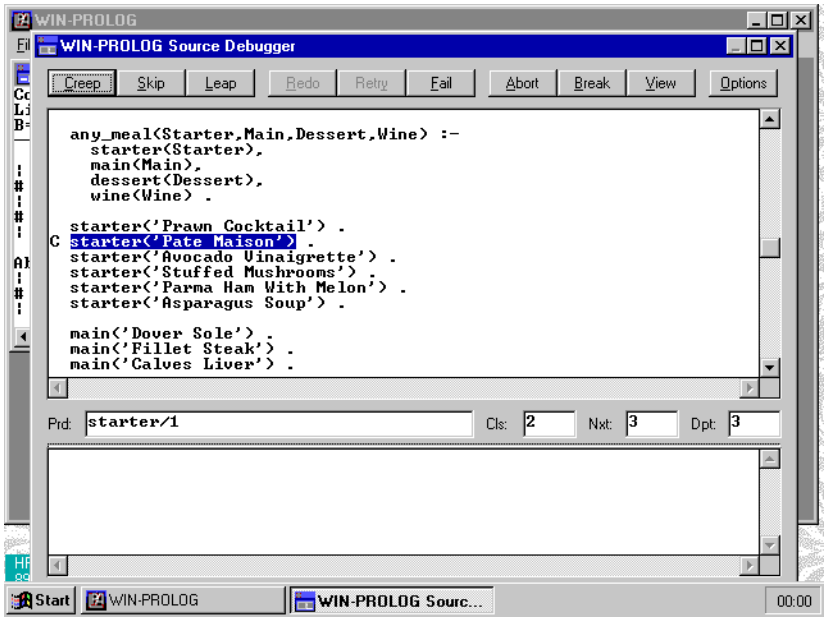


Figure 53 - Backtracking into a relation using redo

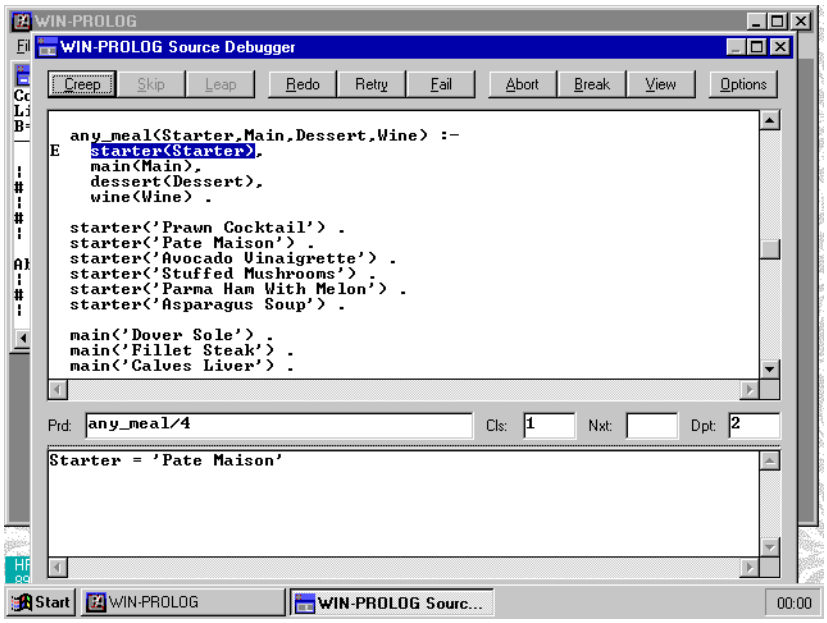


Figure 54 - The exit port with the new solution after two creeps

The redo command allows you to test backtracking over a single call, until you get back a solution that you want to test further by creeps and skips.

The Retry Command

This command is similar to redo, except that it actually runs a goal a second time without backtracking. It enables you to start over with the debugging of a given call. Click on "retrY" or press <*Y*>, and the call *starter(...)* will revert to the state it was in before its first call, as shown in *Figure* 55. Now if you creep, you will once again land on the first clause of *starter/1*, returning after another two creeps with the "Starter = 'Prawn Cocktail'" solution.

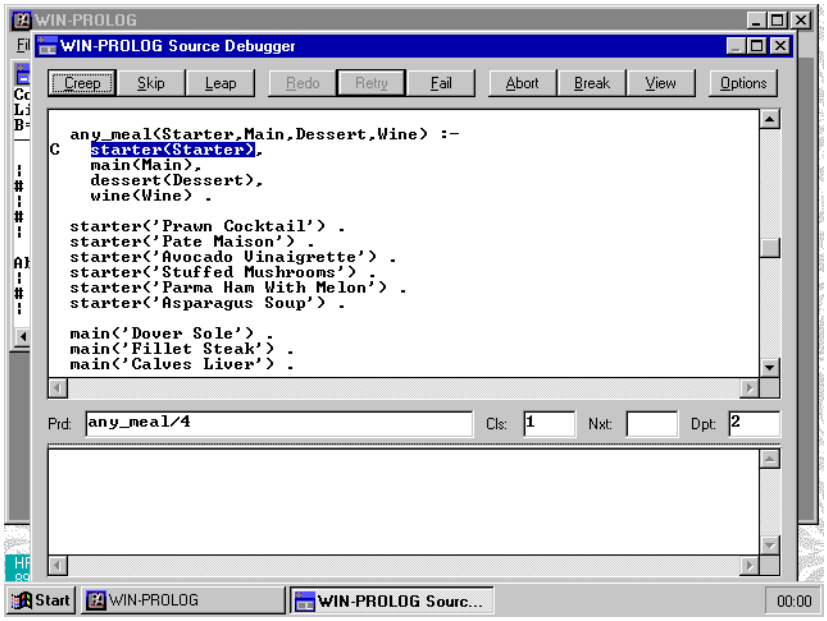


Figure 55 - Re-evaluating a goal with the retry command

Note that if you retry a goal which performs side effects, such as output or database updates, then these will eventually be performed more than once. The retry option can only reset variable bindings and the execution stack, and not side effects.

The Fail Command

This is the third command which modifies the behaviour of your program, and is used to force a goal to fail altogether, forcing backtracking in any earlier choice points. Click "Fail", or press *<F>*, and the call to *starter(...)* will fail, returning the highlight to the head of the *any_meal/4* clause, as shown in *Figure 56*.

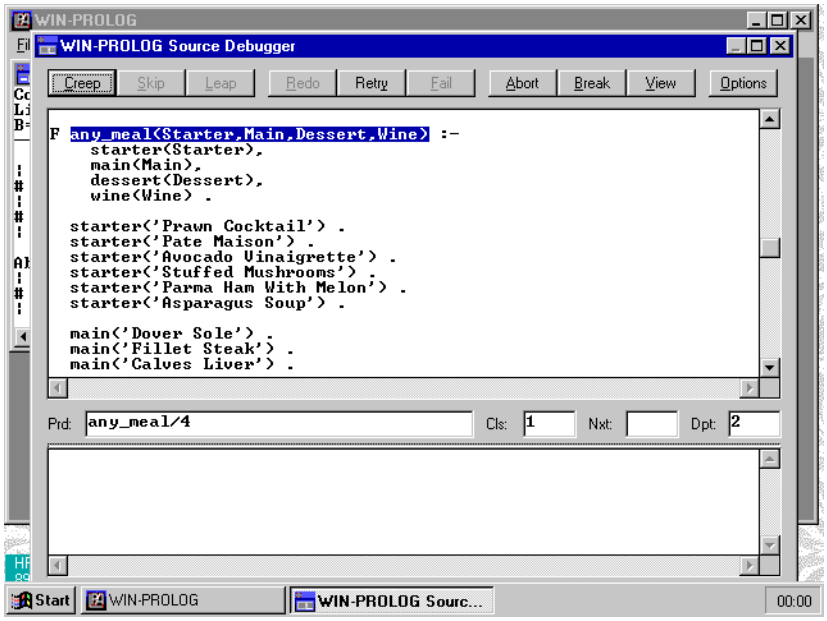


Figure 56 - Failure indicated after the fail command

Notice the "F" symbol by the head: this indicates that the fail command has caused the whole clause to fail, because there were no other choice points in the clause prior to the call to *starter(...)*. At this point, you can either creep back out into the clause for *cheap_meal/4*, or retry the *any_meal/4* clause.

The Abort Command

The final group of four commands affect the debugger itself, rather than the program being debugged. You have already used the abort command, which terminates the debugger and returns you to the Main window command line. When you abort, which you can do by clicking on "Abort" or pressing <*A*>, the execution is simply abandoned, and you are free to start again, edit your programs, recompile, and so forth. The only thing that you cannot do is resume debugging at the point you aborted.

The Break Command

This command is like abort, except that rather than abandon the query that you are debugging, it suspends it while returning you to the Main window command line. Click on "Break" or press <*B*>, and the "Source Level Debugger" dialog will disappear, leaving you at the Main window command line, as shown in *Figure 57*. You can perform most normal operations while in "break" mode, although you cannot change debugger model (remember, the debugger is still running!), and you should not edit, compile or otherwise modify the program you are debugging. When ready to resume debugging, press <*Esc*>, and the debugger will appear where you left off.

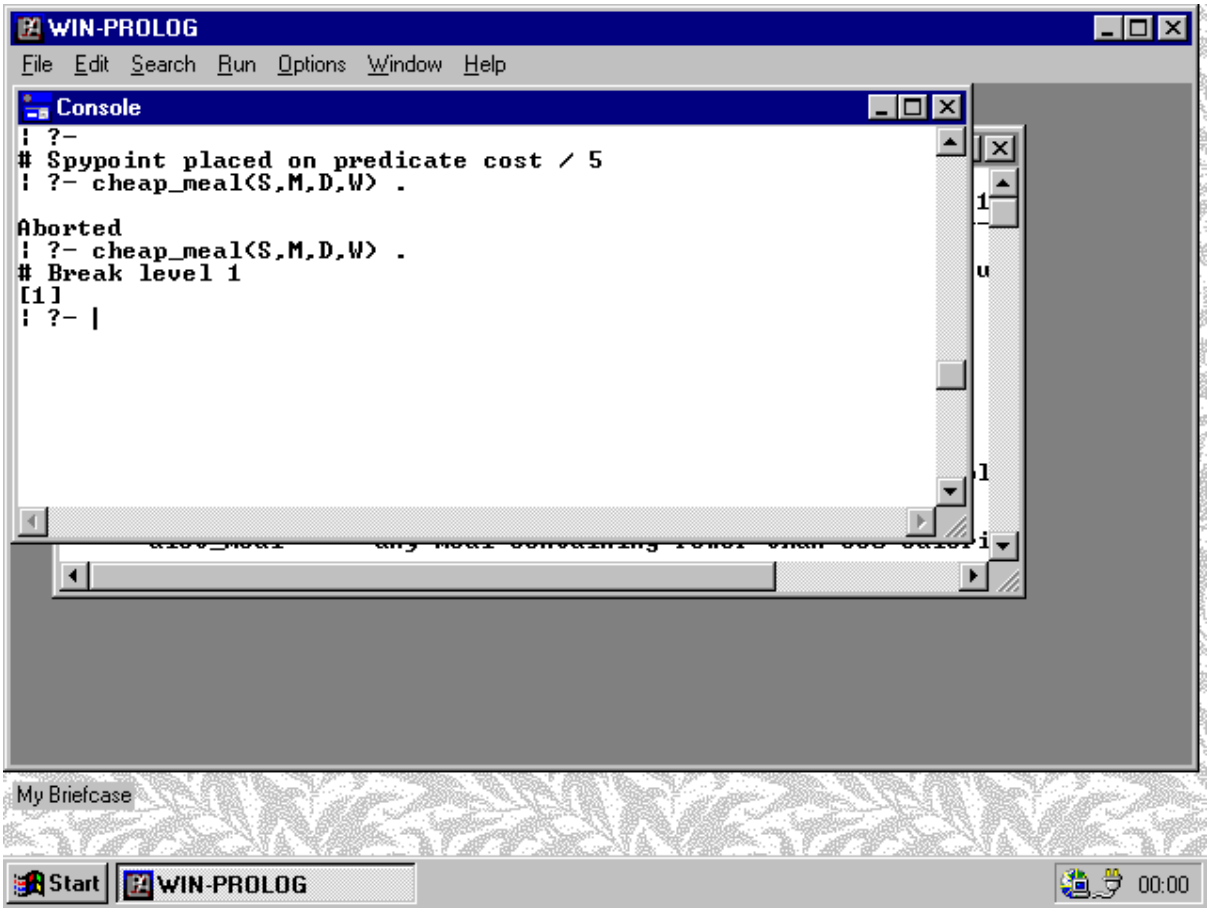


Figure 57 - Shelling to the Main window using break

The View Command

While the break command allows you to suspend debugging temporarily, so that you can perform some other commands or queries, there are times when you may merely wish to view the output, windows and graphics produced by your program. This is the function of the view command: if you click on "View" or press <*V*>, the "Source Level Debugger" dialog is hidden, revealing the other windows. When you want to continue, simply click the left mouse button.

You cannot actually perform any operations in view mode; its sole intention is to allow you to inspect your program's output. Note that the keyboard is locked while in this mode; you must use the left mouse button to exit back to the debugger.

The Options Command

The debugger final command allows you to set a number of options to customise the way the source level debugger works. If you click on "Options", or press <*O*>, you will be presented with the "Debugger Options" dialog as shown in *Figure 58*. The "Skip" options allow you to decide how much debugging information is associated with the skip command. By default, spypoints are ignored during a skip; you can enable them by checking the first box. Normally, skip does not display the exit ("E") port upon completion, but you can force it to do so by unchecking the second box.

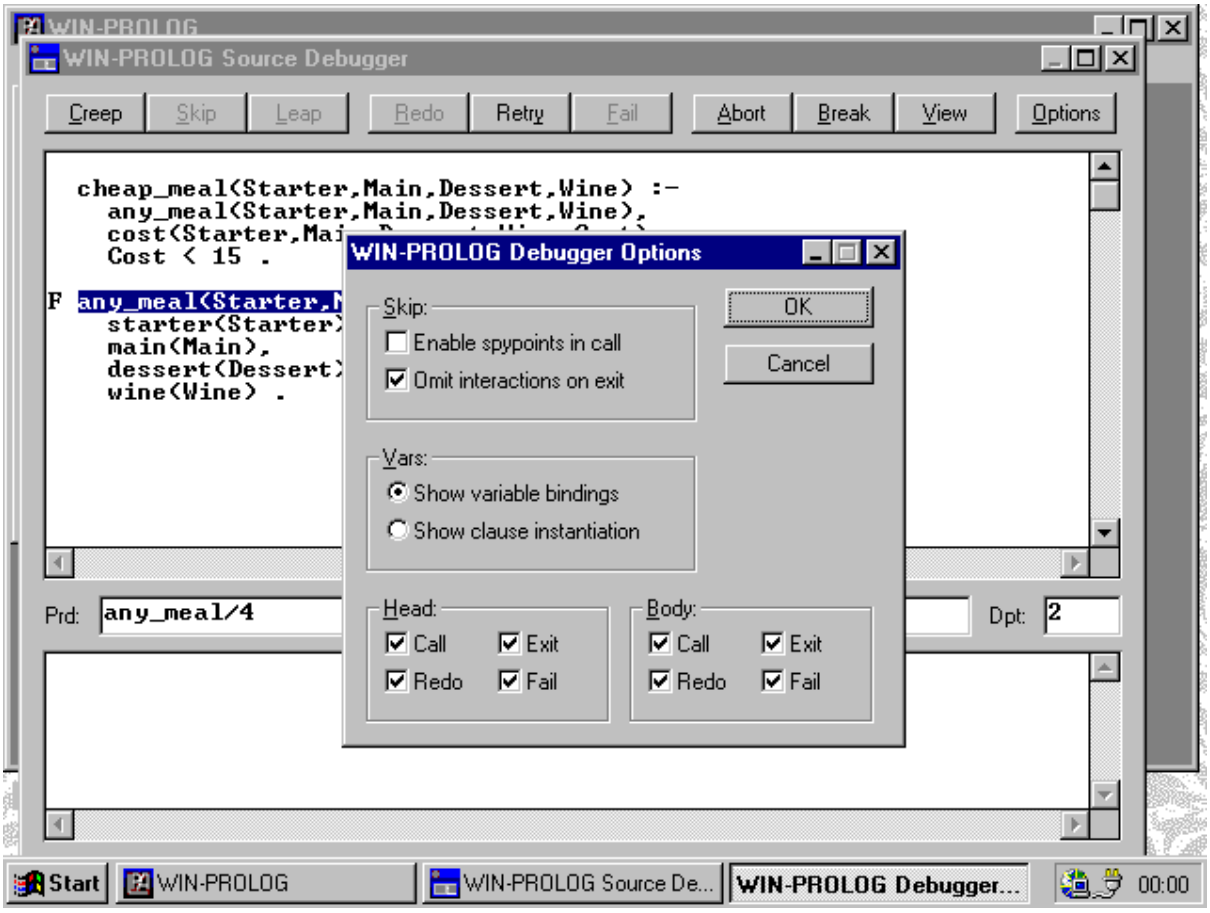


Figure 58 - The "Debugger Options" dialog

The "Vars" options allow you to decide in which of two ways to display variables. By default, they are displayed as an enumerated list of bindings, but by checking the second button, you can display them in situ in their clause if you prefer.

The "Head" and "Body" options allow you to say which of the "Call", "Exit", "Redo" and "Fail" ports you want to be able to interact with in the heads and bodies of clauses you are debugging. By default, all these ports are enabled; you can disable any that you don't want to see by unchecking the relevant boxes.

Click on the "Show clause instantiation" radio button, as shown in *Figure 59*, and then click on "OK" or press *<enter>*. When you return to the "Source Level Debugger" dialog, click on "retrY" or press *<Y>* to perform a retry. Notice how the variable listing in the bindings window has been replaced by a copy of the *any_meal/4* clause, as shown in *Figure 60*.

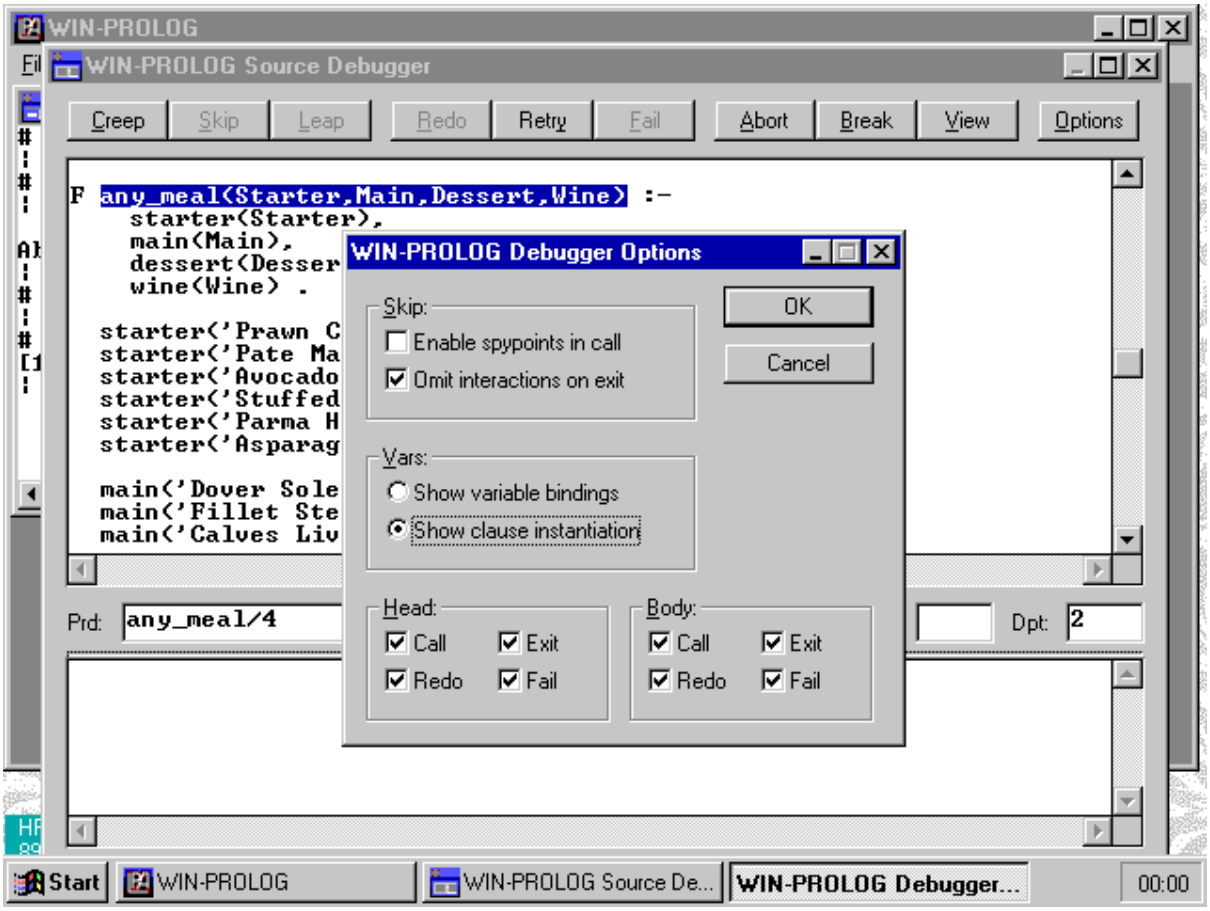


Figure 59 - Setting the "Show Clause Instantiation" option

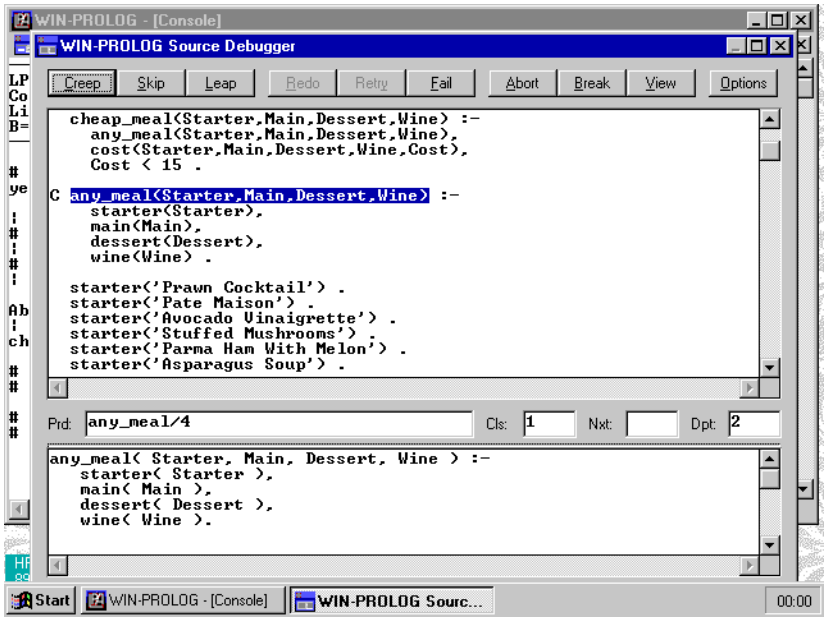


Figure 60 - Clause instantiation displayed after a retry

Now creep to the call for *starter(...)* as before, and then skip the call. See how the clause shown in the bindings window is updated to include "prawn_cocktail" in place of the variable "Starter", as shown in *Figure 61*. If you now skip three more times, so that the highlight moves back to the head of *any_meal/4*, the bindings window will contain the clause fully instantiated, as shown in *Figure 62*.

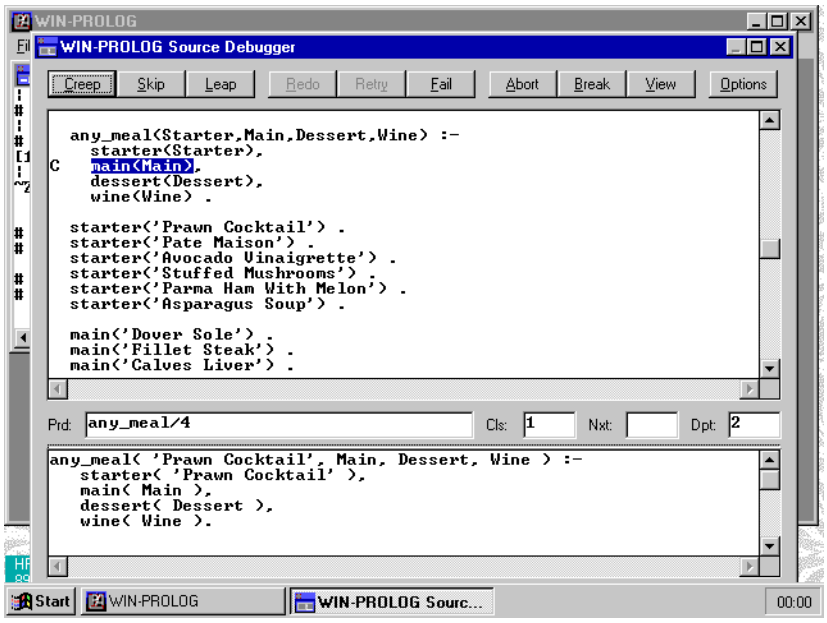


Figure 61 - One of the variables instantiated in the clause

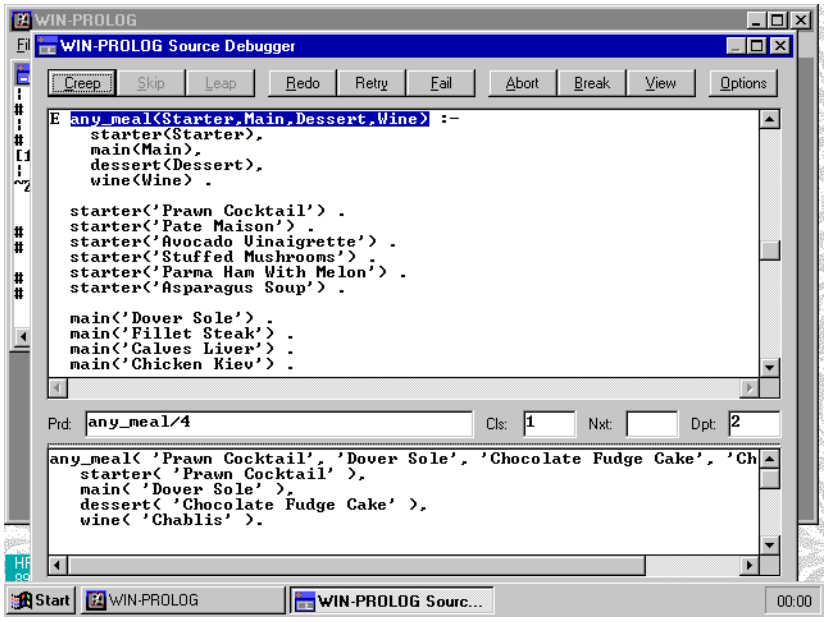


Figure 62 - All variables instantiated in the clause

The Box Model Debugger

This debugger works by tracing execution a step at a time, displaying goals and their results as they happen. Source code is not displayed, but instead the run-time instantiations of goals are shown in a scrolling window. The box model debugger is controlled in a similar way to the source level debugger just described, and has its own "Box Debugger" dialog. Click on "abort" or press <*A*> to abort the query, and then from the Main window select the "Options/Prolog Flags..." option. Click on the "Box Model" radio button, as shown in *Figure 63*, and then click on "OK" or press <*enter*>. Once again, bring your query back into the Console's input zone via <*ctrl*>+<*page_up*> and <*ctrl*>+<*page_down*> and then press <*enter*> to re-enter your query. After a few moments, the "Box Debugger" dialog will appear, as shown in *Figure 64*.

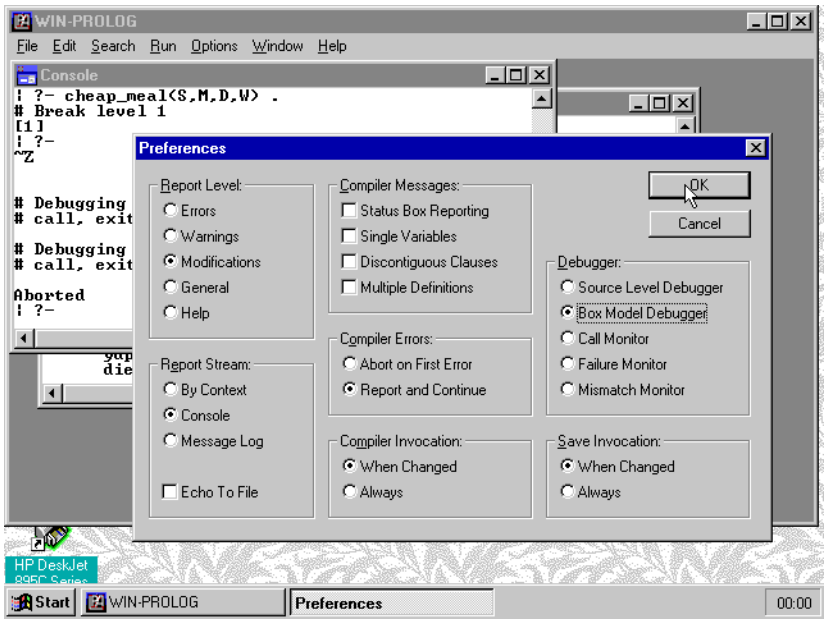


Figure 63 - Selecting the box model debugger

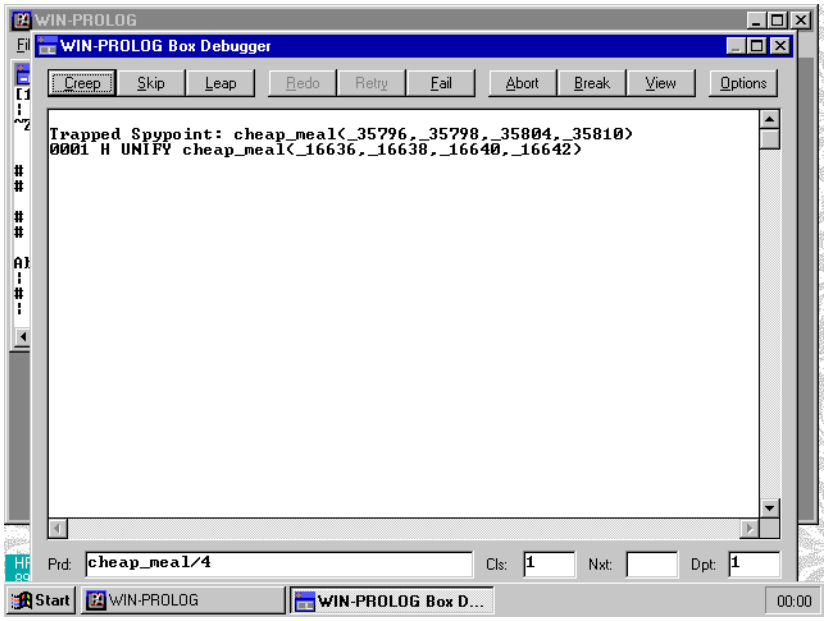


Figure 64 - The "Box Debugger" dialog

The box model debugger works with the same commands as the source level debugger, but displays calls and their results in line. After a number of creep and skip commands, the output will look something like *Figure 65*.

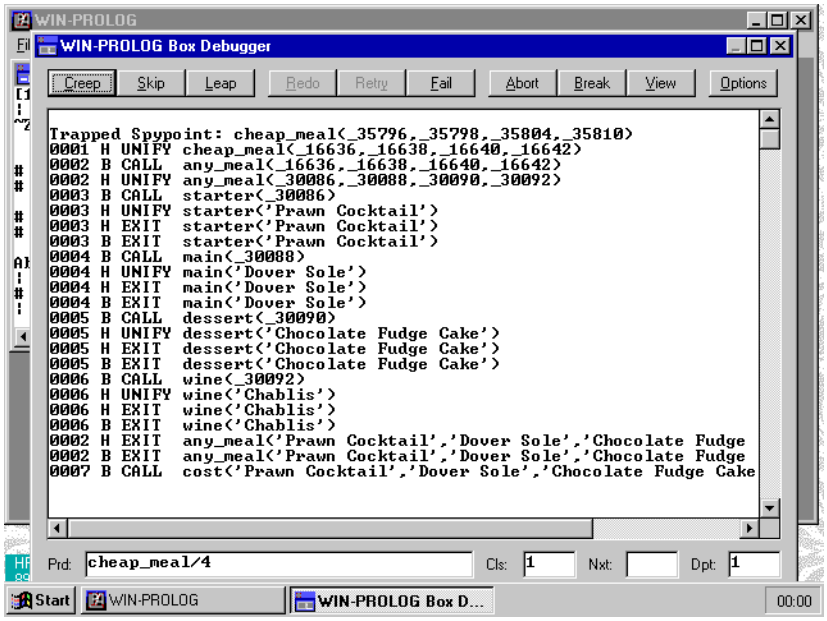


Figure 65 - Typical box model debugger output

Notice that each call is assigned a unique 4-digit number: this can be used to match any given exit or fail to its original call.

Resizing the Debugger Window

You can resize the debugger window by clicking on the maximize button or by clicking and dragging an edge of the window, as shown in Figure 66.

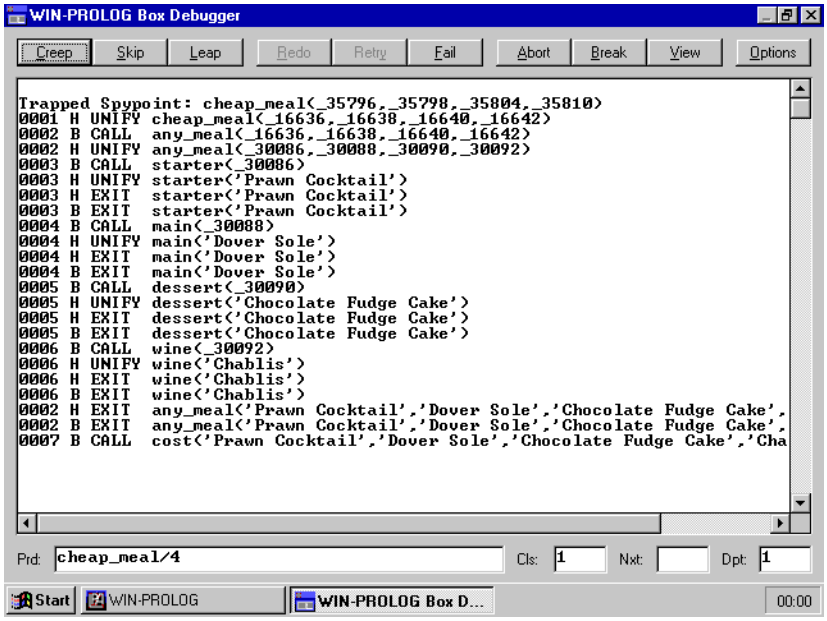


Figure 66 – Resizing the debugger window

You can also, in the source debugger window, resize the variable binding window by grabbing and dragging the "splitter" resize bar above the variable binding window, as shown in Figure 67.

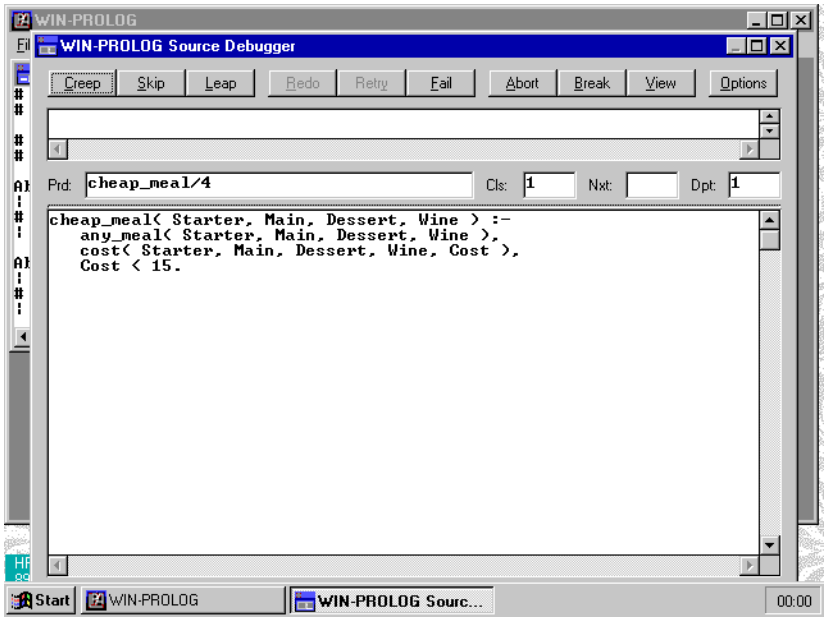


Figure 67 – Resizing the variable window

The Call Monitor

This is a simple, non-interactive version of the box model debugger; its output looks just like that of the box model debugger, except that it does not pause for commands. Effectively, it automatically performs a creep operation at each displayed step.

The Mismatch and Fail Monitors

These two debuggers are special purpose tools to help you find points at which your program is failing. While you can use either of the interactive debuggers to step through a program until you encounter a failure to unify, or a program which fails for other reasons, it is often quick to allow one of these monitors to search for the failure.

In both cases, these monitors simply run your program either until an attempt is made to call a program for which there is no matching clause (the mismatch monitor) or which ultimately fails for other reasons (the fail monitor). When the mismatch or fail occurs, you are prompted with information about it, and asked whether you want to continue the execution, or abort.

Programmatic Access to the Debugger

There are a number of predicates that allow you to programmatically access and control the debugger; *debugging/0* writes the current status of the debugger to the standard output stream and *nodebug/0* switches the debug mode to "off".

For instance, if you turned off interaction within the Options dialog for the exit ports, you might then get:

```
?- debugging.
[State : debug]
[Head  : call redo fail ]
[Body  : call redo fail ]
[Spied : menu/1 ]
yes
```

Debugger-Related Prolog Flags

WIN-PROLOG possesses a number of Prolog flags; two of these, debugging and debug_file, state whether debugging is currently on or off and hold the value of the current debug mode respectively. When you toggle the Options/Debug menu option or select a different debugger in the Prolog Flags dialog, it is *WIN-PROLOG*'s debugging and debug_file flag's value that gets changed.

The values of such flags can be obtained via the *prolog_flag/2* predicate. Let's assume debugging is currently off. Execute the following from the Console window:

```
?- prolog_flag(debugging,X). <enter>
X = off
```

Now toggle the Options/Debug menu option to turn debugging on. The following will be written to the Console window:

```
# Debugging mode switched to debug
```

From the Console window, execute *prolog_flag/2* again:

```
?- prolog_flag(debugging,X). <enter>
X = debug
```

You can change to or from debugging mode without going via the Options/Debug menu option, simply by using the *prolog_flag/3* predicate. The following will change the debugging mode to off:

```
?- prolog_flag(debugging,debug,off). <enter>
# Debugging mode switched to off
yes
```

Trace Utility

The library file, TRACE.PL, is supplied as a configurable debugging utility. Amongst other things, it can write its output to a text file for external inspection. For example, you can run the goal:

```
?- ?(myprogram,tracefile).
```

Notice, that the utility uses '?'/[1,2] as the way of invoking the utility.

And now, if you press Scroll Lock, your entire program will run, a bit slower, while a complete trace of its calls, exits and fails are written out to whatever file you have elected to use. If you get bored, turn off Scroll Lock, and press <Esc>. Remember to close the file before trying to read it.

Check in the comments of TRACE.PL for more info on this handy debugging utility (it's the one LPA use all the time for system-level programming!).

It can be used in almost any situation, including from within a VB session under the Intelligence Server or on the Web in conjunction with ProWeb Server.

Intercepting the Link between a Spied Predicate and the Debugger

When a spied predicate is executed in debug mode, the debugger is invoked. You can intercept the 'link' between a spied predicate and the debugger by defining your own '?DEBUG?'/1 clause:

```
'?DEBUG?'( Goal ) :-
  write( Goal ),
  force( Goal ).
```

or to go on and call the debugger itself:

```
'?DEBUG?'( Goal ) :-
  write( Goal ),
  force( Goal ),
  debug_hook( Goal ).
```

You could, for example, use TRACE.PL as your debugger by defining the following clause:

```
'?DEBUG?'( Goal ) :-
  ?( Goal ).
```

Chapter 13 – Project Files

This chapter describes the creation, loading and amendment of a project file.

Introduction

Projects in ***WIN-PROLOG*** provide a means for conveniently managing the development of programs. By loading a single project file you can load several source code windows into given positions on the screen and have them automatically compiled. In fact you can perform any valid ***WIN-PROLOG*** environment function automatically.

Creating a Project File

The "File/Project..." option lets you create a new project file. It is enabled whenever one or more program edit windows associated with a disk file exist. You are advised to have no "Untitled-<n>" windows open when creating a project file.

This option automatically generates the project code that describes the current state of the ***WIN-PROLOG*** windows. It then saves that code in a project file (.PJ) under the filename you supply. A project file uses *system_menu/3* commands to run the ***WIN-PROLOG*** menu options described in this user guide.

Use the "File/Open..." option and make sure that both the files *BOOZE.PL* and *GUZZLE.PL* are open. Now select the "File/Project..." option; this will generate some project code that describes the current state of the ***WIN-PROLOG*** editing environment. The "Create Project" dialog will then be displayed inviting you to save the project code as a project file (.PJ) under the filename of your choice. Fill the "File name" edit box with the name:

victuals

When you click "Save" or press *<enter>*, a *VICTUALS.PJ* file will be saved to disk.

Loading a Project File

Normally when handling a project, you would not want the source code for the project file to appear as if it were part of the project. The best way to load a project is by loading its project file using the "File/Load..." option.

Select the "File/Load..." menu option; the "Load" dialog (a dialog almost identical to the "Open" dialog) will then appear. You should now see a list of example programs in the "File listing" box. To display the project files select the "Prolog Project Files (*.pj)" option from the "Files of type" menu. Select the file called *VICTUALS.PJ* and when you click "Open" or press <*enter*> the source code text for the *GUZZLE.PL* and *BOOZE.PL* files will be loaded into the ***WIN-PROLOG*** environment. The reason for this is that the *VICTUALS.PJ* project file contains a number of commands, generated using the "File/Project..." option, that get run automatically when the project file is consulted.

Amending a Project File

Under normal circumstances you should never need to view the source code of a project file. However, you may like to know what *VICTUALS.PJ* looks like should you ever need to manually amend such a project file.

Most of the menu options in the ***WIN-PROLOG*** development environment are callable programmatically via the *system_menu/3* predicate; the first command in *VICTUALS.PJ* programmatically executes the "File/Close All" menu option to close any source files currently loaded into the ***WIN-PROLOG*** environment:

```
% Close existing program windows

:- system_menu(_,file,close_all).
```

The file then contains commands to run the "File/Open..." option to open the source files *BOOZE.PL* and *GUZZLE.PL* into the ***WIN-PROLOG*** environment.

```
% Open project program windows

:- system_menu( _, file, open( 'C:\PROLOG\EXAMPLES\BOOZE.PL'( 22, 22, 518,
252 ) ) ).

:- system_menu( _, file, open( 'C:\PROLOG\EXAMPLES\GUZZLE.PL'( 44, 44, 518,
252 ) ) ).
```

The file *BOOZE.PL* will be opened into a window with a width of 518 and a height of 252 with its top left corner at position 22, 22.

The last command resets the size of the Console window and sets this as the focus:

```
% Restore the Console window

:- wshow(1,1), wsize(1,0,0,518,252), wfocus(1).
```

To display *VICTUALS.PJ* in a window within the ***WIN-PROLOG*** environment, click on the "File/Open..." menu option. When the "Open" dialog appears, type in the "File name" field:

```
*.pj
```

and press <*return*>.

Select *VICTUALS.PJ* from the .PJ files displayed and click "Open" or press <*return*>. This project file will then be displayed in a window where it may be amended as required.

To save the modified project file simply select the "File/Save" option, as you would for any other window.

To test your amended project file, select the "File/Close All" option to clear the ***WIN-PROLOG*** workspace and then load the project file in the way described above.

Chapter 14 - Stand-Alone Applications

This chapter describes the process of creating stand-alone applications with *WIN-PROLOG*. The topics covered include some background on the basic anatomy of an application, and an example of how to create one. **Note that this chapter only applies if you have purchased the Developer Edition of *WIN-PROLOG*.**

The Anatomy of an Application

Any application created with *WIN-PROLOG* will include two basic components: the Prolog system files, and your compiled Prolog programs. With reference to *WIN-PROLOG* specifically, there are two system files that must be distributed as part of your stand-alone application:

```
PRO386W.EXE     - The WIN-PROLOG kernel

MYAPP.OVL       - Your Prolog code overlay file
```

The first file, PRO386W.EXE, must be renamed to whatever your application is to be called: this file and the overlay file (.OVL) *must* have the same root name, and must retain their existing extensions. For example, you could name these files:

```
FOO.EXE         )
                )       OK: One root, original extensions
FOO.OVL         )
```

But you could not rename them:

```
FOO.EXE         )
                )       NOT OK: different roots
SUX.OVL         )
```

or:

```
FOO.COM         )
                )       NOT OK: incorrect extensions
FOO.SYS         )
```

WIN-PROLOG's own overlay file, *PRO386W.OVL*, is not included directly in any application you develop: rather, its code is combined with your own to produce a new, application-specific overlay file.

Creating an Application

The process of creating a stand-alone application is very simple. First you write, debug and compile your application code, then you combine this with parts of the existing *PRO386W.OVL* file to create a new overlay file, and finally you ship this new overlay file (say, "*FOO.OVL*") with a copy of *PRO386W.EXE* renamed, say, "*FOO.EXE*". Among the example programs in your *EXAMPLES* folder, you should find the file, *MEALS.PL*; we will use this file to illustrate the creation of a simple stand-alone application.

The first stage is to load all the files for your application into memory. Only after this has been done can you test or create your application using the "Run/Application..." menu option.

main/0

At startup, ***WIN-PROLOG*** performs various internal initialisations, like creating the Main and Console windows, loading icons and bitmaps, and preparing the heaps and stacks. Next, it loads an overlay file – either PRO386W.OVL if you are running Prolog itself, or a user application .OVL file. Once this is in memory, ***WIN-PROLOG*** transfers control to an LPA-defined Prolog routine called "main/0", which performs additional Prolog-code setting up before ultimately transferring control to the user-defined main hook.

The Application Loop

For each stand-alone application, two primary hooks need to be specified - the main hook (i.e. the main goal) and the abort hook (i.e. the abort goal).

When you run your application, the main hook, which you specify in the "Application" dialog (see below), is executed automatically after the overlay has been loaded. Once a user-defined main hook is called, this program is simply run once. The main hook can be thought of as a start-up procedure that performs any initialisations you want (such as the declaration of operators) and then transfers control to your application, probably via an "abort". If the main hook succeeds or fails, the application terminates. If an error occurs in the program, or the program itself calls the "abort/0" predicate, then execution of the main hook is stopped, and the abort hook (i.e. the central application loop) is invoked in its place (i.e. performing an "abort" causes execution to restart at the abort hook).

```
my_main_hook :-
  % <perform any initialisations you might want here>,
  abort.
```

Your main hook must NOT be called *main/0,* otherwise when you save the application overlay, your "main/0" will get muddled up with ***WIN-PROLOG***'s built-in (and normally hidden) main startup point which is called, you guessed it, "main/0". By naming your main hook "main", you would inadvertently over-write the LPA version of "main", so the Prolog initialisations would fail causing the application to abort!

The abort hook, also specified in the "Application" dialog, is executed if an error occurs or an explicit call is made to the *abort/0* predicate.

Your application itself should be built into the abort hook; the abort hook can be viewed as a restart procedure where the central application loop is found. If you want your application to run indefinitely, you need some kind of repeat/<do something>/fail loop. If you want it simply to do one thing and then quit, you don't even need that. If the abort hook succeeds or fails, the application terminates automatically. By putting your "loop" code in the abort hook, rather than the main hook, your application will automatically recover after an error.

```
my_abort_hook :-
  % <run the application itself here>,
  halt.
```

Note: when you create a stand-alone application, your main and abort hooks completely replace the standard read/evaluate/write loop. Because neither the main hook nor the abort hook should ever succeed or fail, actions which would instantly terminate the application, they must be given something to do throughout the duration of the application. If either of these two hooks terminates prematurely, either with success or failure, ***WIN-PROLOG*** will itself terminate with a DOS return code of 0 or 1 respectively. If you want your application to run indefinitely, your definitions of the main and abort hooks must continue processing forever. Because most Windows applications involve displaying one or more dialogs and windows, and then waiting for messages from the mouse and the keyboard, the recommended "do nothing while waiting" loop is:

```
repeat,
flag(1),
wait(0),
fail.
```

As you know, "repeat,fail" simply runs forever. The trouble is it also hogs CPU time and, when optimised, does not even allow message interrupts. Under Windows, to ensure your application behaves respectfully to other Windows applications, your central application loop should regularly yield control to Windows. To do this, call *flag(1)* followed by *wait/1* with the argument 0 (yield for one message cycle). The call to flag(1) just ensures that ***WIN-PROLOG*** permits message interrupts; the call to *wait(0)* causes ***WIN-PROLOG*** to yield control for up to 55ms, unless a message is waiting to be processed. This latter call both releases the CPU, and also checks for messages. If a message occurs, it will be passed to the appropriate window whose handler can then treat it as normal. The definition of a typical "abort" hook would be:

```
my_abort_hook :-
        repeat,
        flag(1),
        wait(0),
        fail.
```

The *flag/1* predicate is now, however, completely unnecessary in a null loop such as this; the following is sufficient in ***WIN-PROLOG*** 4.0 and later:

```
my_abort_hook :-
        repeat,
        wait(0),
        fail.
```

When *my_abort_hook/0* is run, execution is trapped forever between *repeat/0* and *fail/0*; however, the *wait/1* predicate is a special one in ***WIN-PROLOG*** which actually yields to Windows, allowing messages (like menu, dialog and other messages) to interrupt this null loop.

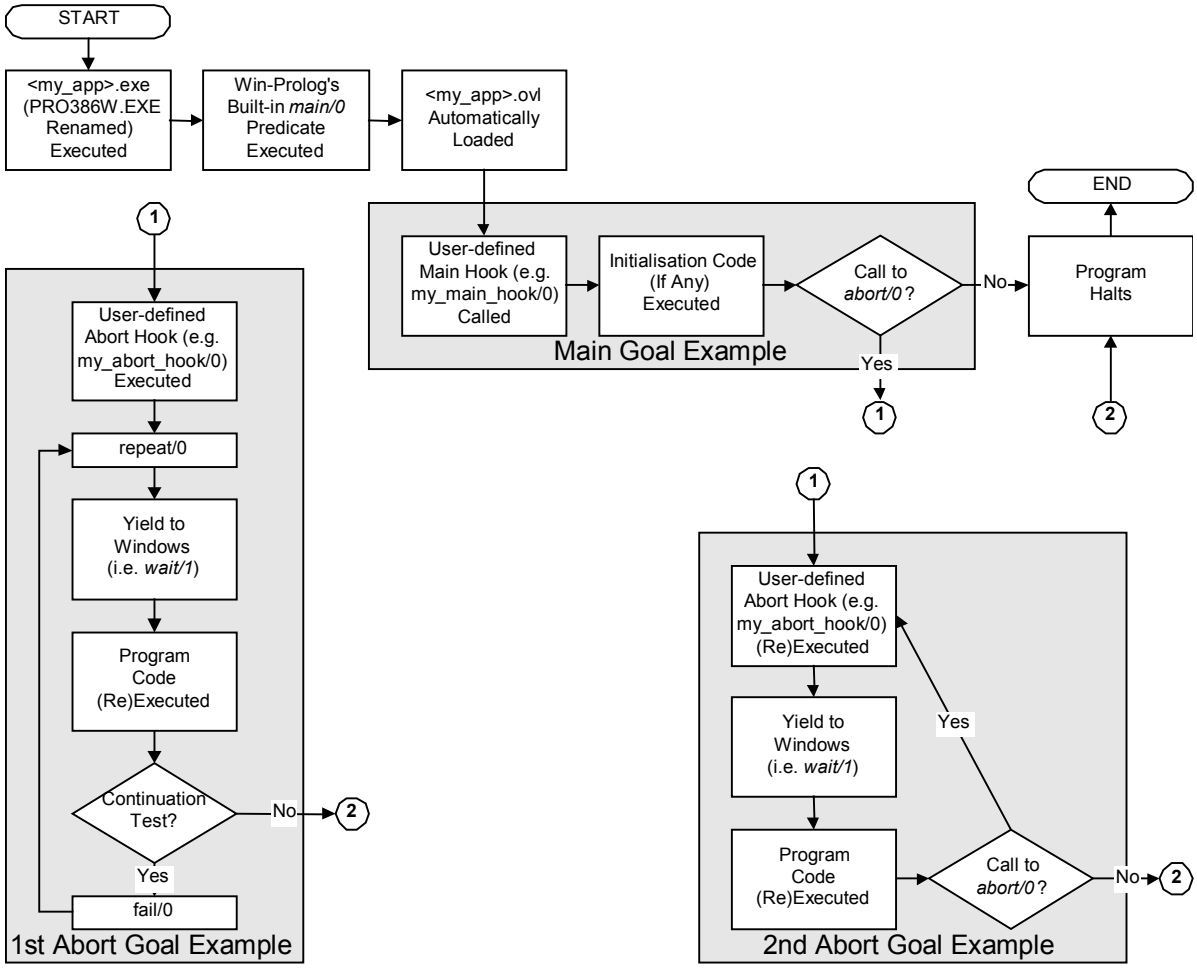


Figure 68 - Stand-alone application flowchart

Testing a Stand-Alone Application

Prior to going through the process of saving your application overlay file it is recommended that the source code for the application be tested thoroughly. The ***WIN-PROLOG*** environment contains the "Run/Application..." menu option that leads to the "Application" dialog, from where stand-alone applications can be both tested and generated.

Testing a Stand-Alone Application: The Meals Example

Run *WIN-PROLOG* and load *MEALS.PL* into a new edit window via the "File/Open..." option. Now select the "Run/Compile" option to compile the file. The program code is now in memory.

The Application Loop

In the meals example, the main hook is a call to the *meal_main_hook/0* predicate:

```
meal_main_hook :-
 sttbox( ``, -1),
 ( write( `Standalone Application Demo` ),
   nl,
   write( `--------------------` ),
   nl,
   nl,
   write( `written by ...` ),
   nl,
   nl,
   write( `Brian D Steel and Dave Westwood` )
 ) ~> AboutString,
 abtbox( `Meal Selector V1.1`, AboutString, 0 ),
 putb( 12 ),
 abort.
```

and the abort hook is a call to the *meal_abort_hook/0* predicate:

```
meal_abort_hook :-
 ( menu
 -> repeat,
   wait( 0 ),
   \+ wndhdl( menu_display, _ ),
   msgbox( `Meal Selector`, `Do you want another go?`, 36, Yes ),
   ( Yes = 6
   -> abort
   ; halt
   )
 ; halt(1)
 ).
```

both of which are defined in the *MEALS.PL* file.

Testing the Application

To test the meals example application, select the "Run/Application..." menu option and fill the "Application" dialog with the information shown in *Figure 69*.

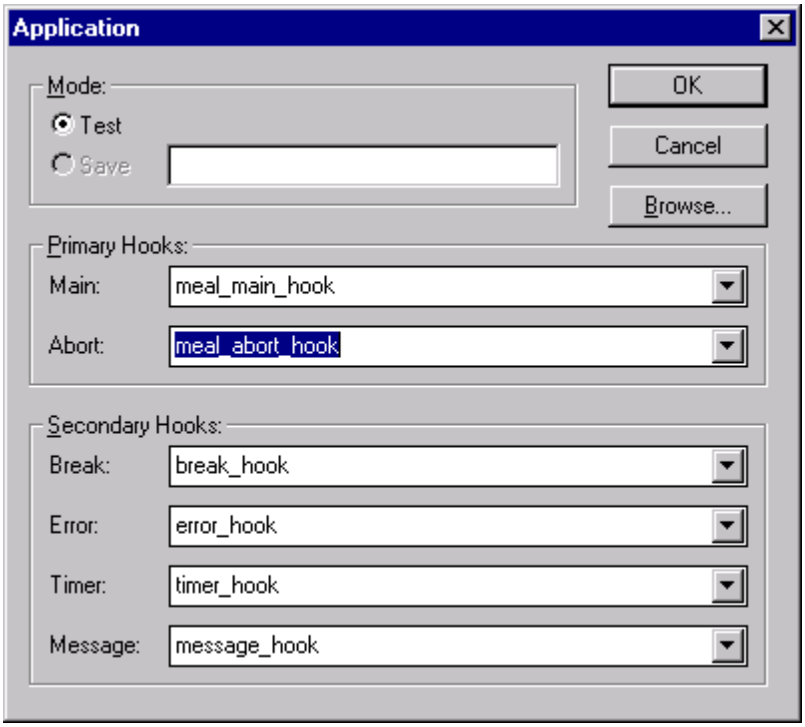


Figure 69 - Testing the meals example with the applications dialog

When you click on the "Ok" button, the ***WIN-PROLOG*** development environment will disappear and you will see the ***WIN-PROLOG*** banner appear, followed by an about box showing the authors of the stand-alone meals application.

Figure 70 - The Win-Prolog banner

Click on the "OK" button and the "Meal Selector" dialog, containing a meal choice listbox, will appear on the screen.

Figure 71 - The Meal Selector dialog

If you now click on the "Cancel" button the test application will terminate and a dialog will appear that shows the exit code returned, as shown in *Figure 72*. Note: this dialog will only appear during the testing of the application.

Figure 72 - The termination dialog showing an exit code of 1

The exit code in this case is 1; this has been set explicitly by the application using a call to *halt/1* when handling the cancellation of the dialog.

Creating the Application Overlay File

After you have tested your code the next thing you will need to do is to create the overlay file that will become part of the application. The "Run/Application" menu option also allows you to save the ***WIN-PROLOG*** workspace into a named file, together with the specified "main" and "abort" hooks, but first there is one more thing you should do.

Starting a New Session

When you create a stand-alone application, a new system overlay file is created containing all the code which is currently in the ***WIN-PROLOG*** workspace. This means that any utilities - like debuggers, optimising compilers, or experimental programs - will be included in the overlay, adding considerably to its size and load time. The safest way to ensure that your application contains only as much code as it needs is to start a new session.

So exit from ***WIN-PROLOG***, if you haven't already done so, and launch a fresh ***WIN-PROLOG*** session, and when you see the "?-" prompt, type the following command:

```
?- ensure_loaded(examples(meals)).          <enter>
```

In this and all other examples in this document, type only those letters shown in bold, and press the named key for anything bracketed in <*italics*>.

To create an overlay file for the meals application, select the "Run/Application..." menu option; this will display the "Application" dialog. Enter the overlay's filename, *meals*, in the edit box to the right of the "Save" radio button (as shown in *Figure 73*). This action will cause the "Save" radio button to be selected automatically. A ***WIN-PROLOG*** application's overlay file must have the .OVL extension; if this extension is not specified, it will be added automatically.

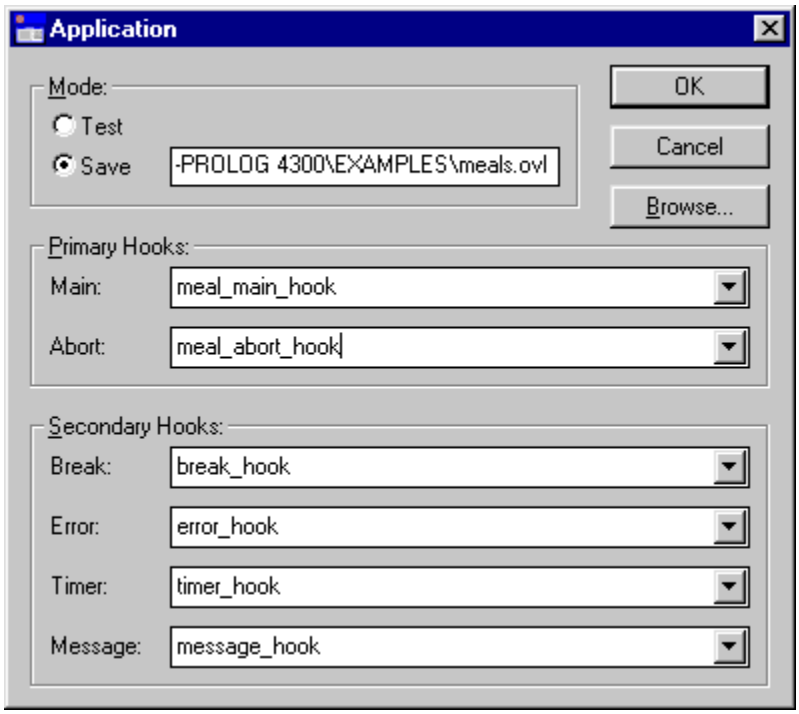


Figure 73 - The applications dialog with the MEALS information

The "Browse..." button can be used to specify the path and name for the .OVL file. When you click on the "OK" button the application saving status box will appear, as shown in *Figure 74*, showing that the overlay file (i.e. "*MEALS.OVL*") is being created and the folder it is being created in.

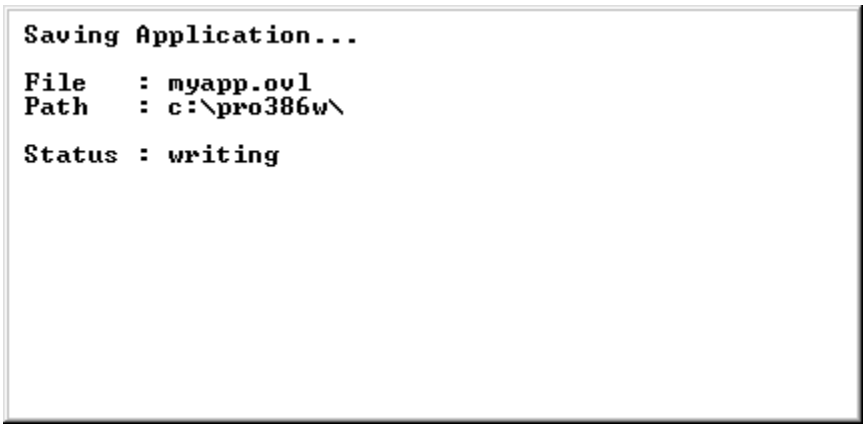


Figure 74 - The application saving status box

Copying the Application

Once the overlay file has been created you can continue with your WIN-PROLOG session as normal. For now though, exit from the WIN-PROLOG environment using the "File/Exit" option, so that the files for the application can be prepared.

All that remains to do is to copy the application and system files to another folder; this is for tidiness and to ensure you know exactly what files make up the standalone application. You should create a folder called *MEALS* and then copy the relevant files from the WIN-PROLOG home directory to the *MEALS* folder:

Original File	Name in MEALS Folder
c:\program files\win-prolog 4900\pro386w.exe	c:\meals\meals.exe
c:\ program files\win-prolog 4900\examples\meals.ovl	c:\meals\meals.ovl

Table 1 - The MEALS application files

The Secondary Hooks

The secondary hooks, namely break, error, timer and message, are predefined in your WIN-PROLOG system and should not need to be altered. For further information, you are referred to *Appendix G - Programmable Hooks and Handlers* in the WIN-PROLOG *Technical Reference* manual. Note that when defining these hook yourself, be sure to get the arity right, otherwise they will not appear in the relevant pull-down menu.

Changing the Banner

In addition to containing the WIN-PROLOG welcome banner, the file *PRO386W.DLL* also contains three additional predicates, *sttbox/2*, *abtbox/3* and *fonts/1*; if your stand-alone application utilises any of these predicates, you will also need a copy of PRO386W.DLL, renamed to *<leafname of standalone application>.DLL*, and placed in the same directory as *<leafname of standalone application>.EXE*.

The "/V1" switch can be included on the command line to suppress the WIN-PROLOG welcome banner when your application runs. When you create a Shortcut for your new application, you can also include this switch in the item's command line so that the banner is automatically suppressed. If you want WIN-PROLOG to display your own banner, you will need to exchange the bitmaps within *<leafname of standalone application>.DLL* (a copy of *PRO386W.DLL* renamed); if you edit this with a 32-bit resource editor, you should be able to see and replace the bitmaps.

Reading the Command Line

Your stand-alone application can read the command line just the same as any other input. For example, if *my_main_hook/0* includes the calls:

```
my_main_hook :-
  (  at_end_of_file
  -> msgbox( `My Application`, `No command line argument was given!`, 48, _ )
  ;  fread( s, 0, 0, String ),
     msgbox( `My Application`, String, 48, _ )
  ),
  abort.

my_abort_hook :-
 halt.
```

then "String" will contain a string with whatever was appended to the command line, such as `hello(world).`

Testing Your Stand-Alone Application Properly

To test your application properly, you should really attempt to run it on a computer which has Windows, but not ***WIN-PROLOG***, installed on it: this is the only way you can really be certain that some component of your application has not been "left behind" in the ***WIN-PROLOG*** home directory! If this is not possible, then at the very least you should remove the ***WIN-PROLOG*** home directory from the DOS path during testing. To do this, click on the Start\Settings\Control Panel menu option, click on System in the Control Panel dialog, select the Environment tab in the System Properties dialog, amend the Path system variable as required and click OK.

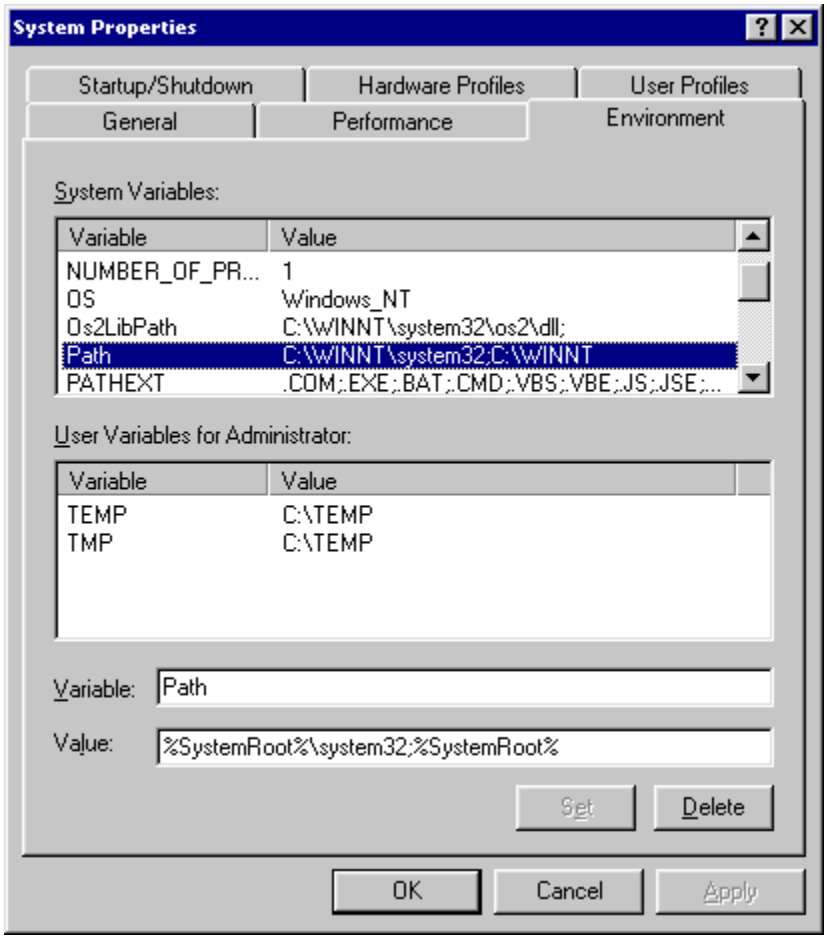


Figure 75 – Amending the PATH system variable

To run the stand-alone meals application, double-click on the *MEALS.EXE* file in the *MEALS* folder.

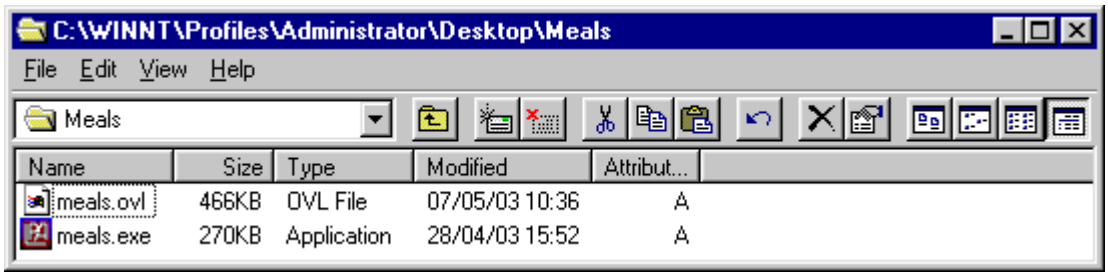


Figure 76 - The MEALS folder

Generating a .INI File

A stand-alone application does not generate a .INI file automatically. Your application will require code to generate this itself. See the predicate, *profile/4*, in the *Technical Reference* manual.

Loading Overlay Files

WIN-PROLOG has the ability to load a particular overlay file. To achieve this, you simply put the name of the overlay file on the command line after the PRO386W application name, preceded by a space and the '@' symbol. For example, the following command line will cause PRO386W.EXE to load the *MYOVL.OVL* overlay file instead of the default one, PRO386W.OVL:

```
PRO386W.EXE @MYOVL
```

or

```
PRO386W @MYOVL
```

This example assumes that the *MYOVL.OVL* file is in the same folder as PRO386W.EXE. Note that there is no space between the @ symbol and the overlay filename and that the overlay filename does not include its extension.

Displaying Prolog's Main Window

Prolog's Main window, including the Console window, is normally hidden in a standalone application unless required for user input; it is programmed to (re)appear whenever keyboard input is requested from the "user" stream (e.g. *getb/1*,*read/1*, etc.). If you want Prolog's Main window (including the Console window) to be permanently visible in your stand-alone application, you will need to execute *wshow(0,1)*:

```
my_main_hook :-
  wshow( 0, 1 ),
  abort.

my_abort_hook :-
 repeat,
 write(`Here is some text`),
 nl,
 ttyflush,
 msgbox( `My Application`, `Click 'Yes' to Continue`, 36, Button ),
 (  Button = 6
 -> fail
 ;  halt
 ).
```

Seeing the Console window may reveal error messages generated by your application that you were unaware of.

Initialization Directives

A *:- initialization* directive is NOT recommended in a stand-alone application: this directive is only guaranteed to work when loading a .PL or .PC file at runtime, and does not mean it will also happen at overlay-initialisation time. The recommended (and only guaranteed) approach is to use the main hook; indeed, this is the entire and sole purpose of the main hook.

Copying Toolkit Files

If your standalone application uses one or more LPA toolkits, there may be additional files which need to be copied to the application's own directory. In the case of Prodata, the files DBLINK.PC and LPADBW.DLL need to be copied.

Troubleshooting

In the event of a problem with your standalone application, you could try using the *sttbox/2* predicate to display a status box on the screen. This will show how far your standalone application progressed through its code before a problem occurred or indeed which branch was taken through the code:

```
my_main_hook :-
  sttbox( `Start of Initialisation`, 0 ),
  ms( repeat, T1 ),
  T1 > 2999,
  % <Initialisation Code>
  sttbox( `End of Initialisation`, 0 ),
  ms( repeat, T2 ),
  T2 > 2999,
  abort.

my_abort_hook :-
  halt.
```

This program will display a status box containing the phrase 'Start of Initialisation' for three seconds and then displays another status box containing the phrase 'End of Initialisation' also for three seconds before exiting.

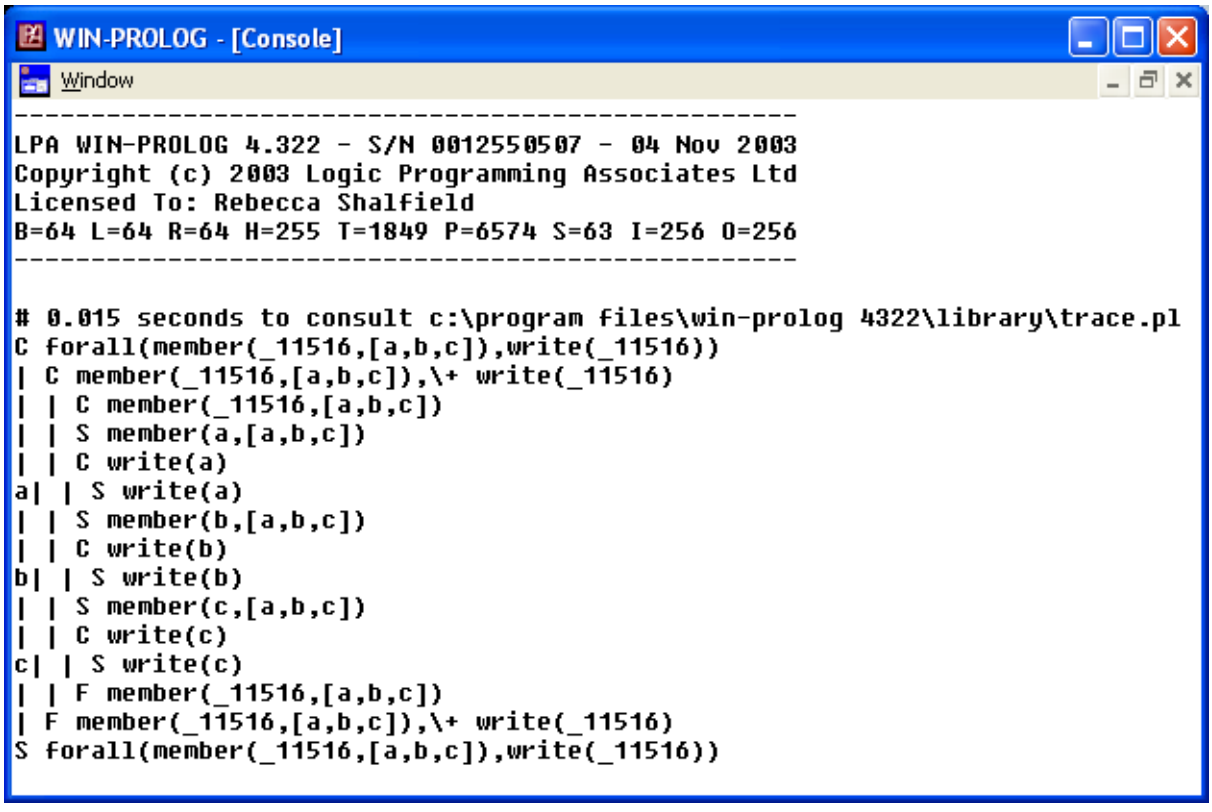
Tracing Your Stand-Alone Application

You can trace your stand-alone application using the Library file, TRACE.PL:

```
my_main_hook :-
  ensure_loaded( library(trace) ),
  ?((
      forall( member( X, [a,b,c] ),
              write( X )
            )
   )),
  abort.

my_abort_hook :-
  repeat,
  wait( 0 ),
  fail.
```

When run as a stand-alone application, the *WIN-PROLOG* console window will appear and you will be able to step through your program's execution using the space bar or the scroll lock:



Using the Flex Toolkit in a Stand-Alone Application

To use flex in a stand-alone application, you need to include the two files, FLEX.PC and FLEXDLG.PC; do not include the file FLEXENV.PC.

Chapter 15 - Dynamic Data Exchange Extension

The Dynamic Data Exchange (DDE) extension allows easy communication between ***WIN-PROLOG*** and other 32-bit applications supporting DDE in the Windows Win32 environment. This section tells you how to load the extension, introduces the concepts associated with DDE and explains the terminology related to ***WIN-PROLOG***'s DDE interface.

What is DDE?

DDE is a means of communication between applications in Microsoft Windows. As its name suggests DDE is a means of passing data between applications, but this is not all; using DDE, applications can control, and be controlled by, other applications.

Because DDE is used to communicate between applications it is of prime importance in any DDE interface that the standard DDE functions be supported. In order to ensure that the standard functions are available, the ***WIN-PROLOG*** DDE extension has been implemented using the 32-bit *Dynamic Data Exchange Management Library* (DDEML).

Loading the DDE Extension

Before describing any of the details and terminology of the DDE interface the best place to start is the process of loading the extension. The DDE extension consists of a Prolog object code file and a Dynamic Link Library (DLL) file, called *DDE32.PC* and *DDE32.DLL* respectively. These files reside in the ***WIN-PROLOG*** *SYSTEM* folder and the *DDE32.PC* file may be loaded using the normal Prolog load predicate *ensure_loaded/1*.

```
?- ensure_loaded(system(dde32)).                    <enter>
```

In this, and all other command line examples, only type the characters shown in **bold** letters, and press the named key for anything bracketed in *<italics>*. To load the *DDE32.DLL* file and set up the interface you should run the following goal after the *DDE32.PC* file has been loaded.

```
?- dde_load.                                        <enter>
```

Terminology

A lot of existing terminology surrounds the DDE interface; the following sections are intended to make clear the relationship between the terminology and the way that the ***WIN-PROLOG*** DDE interface is implemented.

Sources, Destinations and Conversations

Two programs can exchange data by conversing with one another through DDE. This is similar to a phone call between two people, where DDE acts as the telephone connection. The application that starts the conversation is called the *destination*; the application answering the destination is the *source*. To continue with our telephone analogy, the first application would be a person phoning another to ask a question, even though the first person initiates the conversation, that person will eventually be the *destination* for the answer to their question. The second application would be the person on the other end of the line who answers the question, that person is the *source* of the answer to the question.

A program can handle many conversations simultaneously, as the destination in some and as the source in others. In the ***WIN-PROLOG*** DDE interface, Prolog can be programmed either as a source, a destination or both, although it can not have a conversation with the same instance of itself.

Applications, Topics, and Items

A conversation with a particular source on a particular topic is initiated by a request from a destination application. When the source application responds, the conversation is established (the source application and topic are then fixed for the life-time of the conversation).

In the ***WIN-PROLOG*** DDE interface, each conversation, with a specified source and topic, also has a logical name, which is given by the programmer when the conversation is initiated. This is done using the predicate *dde_open/3* (see the heading ***WIN-PROLOG*** *as a Destination Application"* below).

Once a conversation has been established, data can be exchanged on one or more *items*. An item is a reference to data that is meaningful to both the source and the destination applications. For example, let's suppose that Microsoft Excel is the source application and the topic is the name of an excel file. The data in each cell of that file could be referenced by an item that refers to its row and column (i.e. the item `R1C1` refers to the data in row 1 column 1).

The data passed between the source and destination is uniquely defined by the combination of source application, topic and item. In the ***WIN-PROLOG*** DDE interface this is identified by the logical conversation name (which refers to a particular source and topic) and the item.

Application Names

Each application used as a DDE source has a unique *application name*. Usually this is the executable file name without any extension. The default application name when ***WIN-PROLOG*** is the source in a conversation is "PRO386W", you can set your own using *dde_load/1* (see the heading *"Registering* ***WIN-PROLOG***'s *Application Name"* below). The application name is not case-sensitive.

Topics

The *topic* defines the subject of a DDE conversation and is usually taken from a unit of data that is relevant to the source program. As a rule, a document name is recognised by an application as a topic for a DDE conversation. For example, Word for Windows recognises a file name ending in .DOC. In the ***WIN-PROLOG*** DDE interface, topics can be defined programmatically using the *dde_open_topic/2* predicate.

Many programs that perform DDE define a topic called "System". This topic can be used to request information from the program (for example, what other topics and data formats it supports). In the ***WIN-PROLOG*** DDE interface a system topic can be defined programmatically using *dde_open_topic/2* that responds to the various system items appropriately. As with application names, the topic name is not case-sensitive.

Items

The item names a piece of data being passed in a DDE transaction. For example, Word for Windows recognises any bookmark in a document as an item in a conversation. When the ***WIN-PROLOG*** DDE interface is acting as a source application, items are defined programmatically. Like application names and topics, item names are not case sensitive.

Poke, Request and Execute Commands

In a conversation between a destination and source application, each topic and item normally supports three types of command: the poke command, for poking data into the source item; the request command, for requesting data from the source item and the execute command, for sending instructions to be executed by the source.

A ***WIN-PROLOG*** destination application can send these commands using *dde_poke/3*, *dde_request/3* and *dde_execute/2*. When a poke, request or execute command is sent to a ***WIN-PROLOG*** source application these are handled by attaching a Prolog program to the topic using *dde_open_topic/2*.

Advise Loops

If you want Prolog to be notified of changed data in a specific item of a topic, you can do so by requesting an *advise loop* with the source using the predicate *dde_open_advise/4*. Then when that item's data is changed Prolog will be notified. For example, in Excel, given the open conversation "excel_channel", you can request an advise loop on the item "r1c1:r3c3" using the following goal:

```
dde_open_advise(excel_advise, exel_channel, 'r1c1:r3c3', excel_advise_handler)
```

Then whenever data in the cells r1c1 to r3c3 changes the user-defined handler, *excel_advise_handler/2* will be notified with the changed data.

You can open many advise loops for a single conversation.

*WIN-*PROLOG as a Destination Application

The following example illustrates the use of ***WIN-PROLOG*** as a DDE destination by creating an additional sub-menu on the Windows start menu and adding a shortcut to the application. To start with, you need to load the DDE interface extension. Start up ***WIN-PROLOG***, and when you see the Main window, type the following at the command line:

```
?- ensure_loaded( system(dde32) ), dde_load.          <enter>
```

The call to *ensure_loaded/1* loads the DDE32.PC file from ***WIN-PROLOG***'s SYSTEM folder and the call to *dde_load/0* loads the DDE32.DLL.

The next thing you need to do is to establish a conversation with the "Program Manager". The Program Manager can be referenced by the application name "progman" and a topic also called "progman". By sending commands to this topic you can do things like create and delete "Start Menu Folders" and "Shortcuts".

Entering the following goal opens a DDE conversation called "example1" with the Program Manager (progman) on the topic "progman".

```
?- dde_open( example1, progman, progman ).          <enter>
```

The DDE Execute Command

Having established the "example1" conversation you are now ready to send a command to the Program Manager to create a new start menu folder. In a DDE conversation commands are usually sent using the DDE execute command. In ***WIN-PROLOG*** this is done using the *dde_execute/2* predicate.

The following goal creates a new folder called "Example1" using the "CreateGroup" command (The available Program Manager DDE commands are documented in the Microsoft Windows Software Development Kit). Note that after entering the command focus will switch to the new folder.

```
?- dde_execute( example1, '[ CreateGroup( "Example1" ) ]' ).          <enter>
```

Now that you have created the "Example1" folder you can now create a shortcut for the ***WIN-PROLOG*** application. To do this you need to get the path of the *PRO386W.EXE* file and send this to the Program Manager as part of the "AddItem" command. Switch focus back to the ***WIN-PROLOG*** application (using *<alt-tab>*, or some other means) and enter the following goals.

```
?- absolute_file_name(prolog('pro386w.exe'),Prolog),   <enter>
  ( write( '[AddItem("' ),                             <enter>
    write( Prolog ),                                   <enter>
    write( '")]' )                                     <enter>
  )                                                    <enter>
  ~> ExecString,                                       <enter>
  dde_execute( example1,   ExecString ).               <enter>
```

The above call to *absolute_file_name/2* retrieves the path and name of the *PRO386W.EXE* file. This is then combined with the "AddItem" command, using the three write statements whose output is redirected into a string, and returned in the variable *ExecString*. The resultant *ExecString* is then sent to the Program Manager using *dde_execute/2*. If you now click on the Start Menu and select the "Programs" sub-menu, you should see a further sub-menu called "Example1" containing a shortcut called "Pro386w".

To tidy things up, it is advisable to delete the folder you have just created. Switch focus back to the ***WIN-PROLOG*** application and type the following "DeleteGroup" command at the command line.

```
?- dde_execute( example1, '[ DeleteGroup( "Example1" ) ]' ).   <enter>
```

Finally, enter the following call to *dde_close/1* at the ***WIN-PROLOG*** command line to terminate the no longer needed "example1" conversation.

```
?- dde_close( example1 ).   <enter>
```

*WIN-*PROLOG as a Source Application

The following example illustrates ***WIN-PROLOG*** acting as a source application. The topic that is set up provides a simple connection to ***WIN-PROLOG*** that allows information to be requested and poked and for Prolog queries to be executed and their results stored for later request.

To start with, you need to load the DDE interface extension. Start up ***WIN-PROLOG***, and when you see the Main window, type the following at the command line:

```
?- ensure_loaded( system(dde32) ).   <enter>
```

Registering *WIN-*PROLOG's Application Name

To allow the ***WIN-PROLOG*** source application to be accessed via DDE, an application name must first be registered with the DDE32.DLL. From then on, this registered name is used in any DDE transaction with the ***WIN-PROLOG*** source. Multiple copies of ***WIN-PROLOG*** may be registered with different application names and accessed separately. We could use *dde_load/0* and have the default application name "PRO386W", but in this example we will register the name `MySource` by entering the following *dde_load/1* query at the command line:

```
?- dde_load( `MySource` ).   <enter>
```

Registering a *WIN-*PROLOG Handler for the Topic

Now that the source name has been registered, the next step is to set up the topics. This is done by registering a topic name and corresponding ***WIN-PROLOG*** handler. The handler is a Prolog program of the form:

```
functor( Transaction, Item, Data ).
```

where the *Transaction* parameter is bound to the type of transaction (one of execute, poke or request). *Item* is the item for the topic and *Data* is the data sent with a "poke" transaction.

The following goal opens the "eg" topic and sets the *eg_handler/3* program ready to respond to commands.

```
?- dde_open_topic( eg, eg_handler ).                    <enter>
```

Reacting to a DDE Request Command

The following clauses for the *eg_handler/3* program show how *WIN-PROLOG* could react to a request command. When the destination application sends a request on some item in the "eg" topic the following handler tries to find a data value (stored in a local data clause *data_value/2*) corresponding to that item. If there is a matching *data_value/1* clause the handler returns that data to the destination:

```
eg_handler( request, Item, _ ) :-
  data_value( Item, Data ),
  dde_put_data( Data ),
  !.
```

If no matching *data_value/1* clause is found then the following handler clause returns the information "No Data Found" to the client:

```
eg_handler( request, _, _ ) :-
        dde_put_data( `No Data Found` ).
```

Reacting to a DDE Poke Command

The following clause for the *eg_handler/3* shows how *WIN-PROLOG* could react to a "poke" command. When the destination application sends a "poke" transaction with some data relating to a given item in the "eg" topic, the following handler stores the item and data in a local data clause *data_value/2*:

```
eg_handler( poke, Item, Data ):-
  assert( data_value( Item, Data ) ).
```

Reacting to a DDE Execute Command

The final clause for the *eg_handler/3* program is an example of how to react to an "execute" command. When the destination application sends an "execute" transaction on some item, the following handler reads a goal from the item string, executes the goal and redirects the output from running the goal to a string (which is stored for later retrieval).

NOTE: In this example it is *assumed* that the destination has sent a valid Prolog goal term as the item string. To keep the example simple, no guard is made to check that this string *is* a valid *WIN-PROLOG* goal term. For safety, the reading and execution of the goal should be wrapped up in calls to *catch/2*.

```
eg_handler( execute, Item, _ ) :-
  read( Goal ) <~ Item,
  Goal ~> OutputString,
  assert( data_value( Item, OutputString ) ).
```

Once a goal has been "executed", the output from running the goal can be retrieved by the destination application by sending a "request" transaction using the goal term as the "item".

An Outline of the Prolog Interface Predicates

The *WIN-PROLOG* predicates in the DDE interface can be divided logically into three groups: general predicates, which are of use to both destination and source programs: destination predicates, where *WIN-PROLOG* requests services from a source: and source predicates where *WIN-PROLOG* provides services to destination applications.

General Predicates

dde_advise_dict/1	Get a dictionary of open advise loops
dde_channel_dict/1	Get a dictionary of open source channels
dde_dll_name/1	Get the DDE DLL's absolute file name
dde_load/0	Load the DDE DLL
dde_load/1	Load the DDE DLL and register an application name
dde_timeout/1	Set the global timeout value
dde_topic_dict/1	Get a dictionary of registered topics
dde_unload/0	Unload the DDE DLL

Table 2 - DDE general predicates

Destination Predicates

dde_close/1	Close a source channel
dde_close_advise/2	Close an advise loop
dde_close_all/0	Close all source channels and associated advise loops
dde_enable_state/2	Set/Get the enable state of a source channel
dde_execute/2	An execute transaction
dde_fetch_data/1	Fetch data for a transaction
dde_open/3	Open a source channel
dde_open_advise/4	Open an advise loop on an open source channel
dde_poke/3	A poke transaction
dde_put_data/1	Put data for a transaction
dde_request/3	A request transaction

Table 3 - DDE destination predicates

Source Predicates

dde_close_all_topics/0	Close all registered topics
dde_open_topic/2	Open a topic
dde_close_topic/1	Close a registered topic

Table 4 - DDE source predicates

Error Handling

You can define your own error handler for DDE by defining the relation *'?DDE_ERROR?'/2*. For example the following clause for the *'?DDE_ERROR?'/2* program converts the error number generated into an error message and then writes that message to the Console:

```
'?DDE_ERROR?'( ErrorNumber, Goal ) :-
        error_message( ErrorNumber, Message ),
        write( Message ), nl,
        Goal.
```

where *ErrorNumber* will be bound to the reported error and *Goal* will be bound to the goal that the error interrupted.

Description of Errors

100	An advise transaction has timed out.
101	The response to a transaction is that the source is busy.
102	A request for data has timed out.
105	A request for an execute transaction has timed out.
108	A memory allocation by the interface has failed
109	A transaction failed.
110	A destination's attempt to establish a conversation has failed.
111	A request for a poke transaction has timed out.
114	The source terminated before a transaction was completed.
115	An internal error in the DDEML has occurred.
116	A request to end an advise loop has timed out.
118	There are no slots available to create a new topic.
119	A conversation was not established.
120	The data was unavailable from the source.
121	There are no slots available for a new conversation.
122	There are no slots available for a new item.
123	An advise loop was not opened.

Table 5 - DDE error numbers

Chapter 16 - OLE Automation Extension

The OLE Automation library extension allows you to control "OLE automation objects" exposed by other Windows applications. This chapter tells you how to load the OLE Automation library extension, introduces the concepts associated with OLE Automation and explains the terminology related to ***WIN-PROLOG***'s OLE Automation interface.

What is Automation?

OLE Automation, Microsoft's replacement for DDE, is a means of communication and control between applications in Microsoft Windows. The Object Linking and Embedding (OLE) Automation library extension allows ***WIN-PROLOG*** to pass data between and control OLE Automation objects exposed by other Windows applications such as Microsoft Word, Excel or Access. ***WIN-PROLOG*** itself does not expose any objects to other Windows applications, and so, can not be controlled by them. Examples of using Automation from ***WIN-PROLOG*** might be to create and format a document in Microsoft Word or control Microsoft Internet Explorer.

Each Automation object has a "dispatch" interface; this defines the methods and properties of the object and though which the object is controlled.

Using the Automation Library

Six steps are involved in the use of Automation objects:

1. Load the OLE Automation extension.
2. Initialise OLE Automation.
3. Create an instance of an object.
4. Invoke the methods and properties of an object.
5. Close an instance of an object.
6. Uninitialise OLE Automation.

We will now go through each of these steps in turn.

Step 1 - Load the OLE Automation Extension

The OLE Automation extension consists of a Prolog object code file called OLE.PC and a Dynamic Link Library (DLL) file named LPAOLE.DLL; these files reside in the ***WIN-PROLOG*** SYSTEM folder. The OLE.PC file may be loaded using the normal Prolog load predicate *ensure_loaded/1*:

?- **ensure_loaded(system(ole)).** *<enter>*

In this, and all other command line examples, only type the characters shown in **bold** letters, and press the named key for anything bracketed in *<italics>*.

Step 2 - Initialise OLE Automation

The next step is to load *WIN-PROLOG*'s OLE Automation DLL and then initialise the OLE system library; this is done by executing the *ole_initialize/0* predicate:

?- **ole_initialize.** *<enter>*

Step 3 - Create an Instance of an Object

You now need to create your "root" object using the predicate *ole_create/2*. This predicate names an object and attempts to load its associated application.

If you wanted to create a Microsoft Word application object you would call:

?- **ole_create(word, 'word.application').** *<enter>*

If you wanted to create a Microsoft Excel application object you would call:

?- **ole_create(excel, 'Excel.Application').** *<enter>*

The first argument is a user-defined reference to the object that is being created; this can be called anything you like. The second argument is the 'programmatic identifier' or ProgID; this is of the form App.Object where App is the application name and Object is the type of object that you wish to create.

To clarify the above, here is another example; if you wanted to create a Word Basic object, the following code would be called:

?- **ole_create(wordbasic, 'word.basic').** *<enter>*

OLE will then open the application with the word.basic object, this object becoming another root object.

We can see what objects we have created by displaying the object dictionary with the *ole_dict/1* predicate:

?- **ole_dict(X).** *<enter>*
X = [wordbasic, excel, word]

If you wanted *WIN-PROLOG* to control, say, a second Word object, you would simply call *ole_create/2* again with the first argument set to a different user-defined reference from that used before, such as *word2*.

Objects can also be created by and returned from function calls or properties; these objects being derived from the root object.

WIN-PROLOG can control many OLE Automation objects simultaneously.

Step 4 - Invoke the Methods and Properties of an Object

Once an object has been created, you can then invoke its methods and properties; four predicates are provided for this purpose:

ole_function/4	calls an object's function with a list of arguments
ole_get_property/4	gets the value for an object's property
ole_procedure/3	calls an object's procedure with a list of arguments
ole_put_property/3	sets the value of an object's property

Using the word object we created earlier as an example, we can make it visible by putting the value –1, meaning true, to its "visible" property:

```
?- ole_put_property( word, visible, -1 ). <enter>
```

To discover the available values for a particular property, you will need to refer to such sources as on-line help; in this case, the *Microsoft Word Visual Basic Reference*.

We can then get back the current value for this property with:

```
?- ole_get_property( word, visible, [], Value ). <enter>
Value = true
```

The first two arguments are the object and the property respectively. The third argument is a list of parameters that may be required by the call. The fourth argument is a variable which gets instantiated to the returned value.

To maximise the word object's now open window, you need to execute:

```
?- ole_put_property( word, windowstate, 1 ). <enter>
```

If you wanted to hide the word object's window again, you would use:

```
?- ole_put_property( word, visible, 0 ). <enter>
```

Getting the value for a property can also return an object. For example, the call:

```
?- ole_get_property( word, documents, [], Documents ). <enter>
```

returns the word object's documents collection object:

```
Documents = object( word : documents )
```

The first argument is the object path; this can take two forms: Root:Obj1:Obj2 or object(Root:Obj1:Obj2).

If you execute *ole_dict/1* again, you will find that the word:documents object has been added to the object dictionary and can be used just like its root object:

```
?- ole_dict(X). <enter>
X = [ word:documents, wordbasic, word ]
```

We now have three uniquely named objects. If we were, for example, to ask for the word:documents object again, the previous instance of this object would need to be closed prior to a new instance being created.

We can add a new document to the word:documents object by executing the following:

```
?- ole_function( word:documents, add, [], Add ). <enter>
Add = object( word : documents : add )
```

This newly created document is itself an object, accessible as word:documents:add. You will notice that an object's name is built up by appending the function or property name to the referenced object; you can liken this to a directory path from the root object to the named object.

Parameters can be passed to an object's function or procedure through the third argument of *ole_function/4* and *ole_procedure/4* respectively. When creating a range object, for example, the start and end points for the range need to be specified:

```
?- ole_function( word:documents:add, range, [0,0], Range ). <enter>
Range = object( word : documents : add : range )
```

Having created a range object, you could then set its text property with the following call:

```
?- ole_put_property(word:documents:add:range,text,'my text'). <enter>
yes
```

or more simply:

```
?- ole_function( word:documents:add, range, [0,0], Range ), ole_put_property(
Range, text, 'my text' ). <enter>
yes
```

From the above example, you can see how an object returned from one call can be passed straight into another.

Execute *ole_dict/1* again to see the current list of objects:

```
?- ole_dict(X). <enter>
X = [ word:documents:add:range, word:documents:add, word:documents, wordbasic,
word ]
```

Step 5 - Close an Object

When you have finished with an object, it is always a good idea to close it down; in some cases though, you may need to tell the object to quit first. To close the word object, you would call:

```
?- ole_function( word, quit, [0], X ). <enter>
X = ok
```

followed by:

```
?- ole_close( word ). <enter>
yes
```

This will close not only the word object but all its sub-objects, releasing its interfaces in the process.

Executing *ole_dict/1* again will show that the only objects remaining will be those not derived from the word object, namely the wordbasic object:

```
?- ole_dict(X). <enter>
X = [wordbasic]
```

Step 6 - Uninitialise OLE Automation

Finally, we need to uninitialise the OLE Automation system library:

```
?- ole_uninitialize. <enter>
yes
```

This will uninitialise the OLE Automation system library, release any open objects and then unload/close the OLE Automation DLL.

Complete Microsoft Excel Example

Here is a complete example which read, via OLE, a given Microsoft Excel spreadsheet file and writes each record out to the console window. It is designed to read an Excel spreadsheet with two columns - the first containing the name of a person and the second column their email address:

```
:- ensure_loaded( system(ole) ).
```

```
read_excel_file( ExcelFile ) :-
  ole_initialize,
  ole_create( excel, 'Excel.Application' ),
  ole_get_property( excel, workbooks, [], WorkBooks ),
  ole_get_property( WorkBooks, open, [ExcelFile], _ ),
  ole_get_property( excel, cells, [], CellsObj ),
  integer_bound( 1, RecordNo, 65535 ),
  ole_get_property( CellsObj, item, [RecordNo,1], FirstColCellObj ),
  ole_get_property( FirstColCellObj, value, [], Name ),
  ole_get_property( CellsObj, item, [RecordNo,2], SecondColCellObj ),
  ole_get_property( SecondColCellObj, value, [], EmailAddress ),
  (  Name = ok,
     EmailAddress = ok
  -> !,
     true
  ;  write( Name ),
     nl,
     write( EmailAddress ),
     nl,
     nl,
     fail
  ),
  ole_function( excel, quit, [], _ ),
  ole_close(excel),
  ole_uninitialize.
```

Execute a goal similar to the following:

```
?- read_excel_file( `c:\my_data.xls` ). <enter>
```

It will read each record (i.e. row of two columns) until a blank row is encountered, in which case, 'ok' and 'ok' are returned.

Version Details

The *ole_version/4* predicate provides information on the OLE Automation library itself:

```
?- ole_version( Major, Minor, Date, Author ). <enter>
Major = '1' ,
Minor = '000' ,
Date = '25 April 1997' ,
Author = 'Alan Westwood'
```

The OLE Automation System Library Predicates

The *WIN-PROLOG* predicates in the OLE Automation interface are as follows:

ole_close/1	Closes object and all its derivatives
ole_create/2	Creates a user-defined object named by program-id
ole_dict/1	Returns a list of open objects
ole_function/4	Calls an object's function with a list of arguments
ole_get/2	Links to a running user-defined object named by program-id
ole_get_property/4	Gets the value of an object's property
ole_initialize/0	Loads the Automation DLL and initialises the OLE libraries
ole_last_error/4	Returns details of the last error to occur
ole_procedure/3	Calls an object's procedure with a list of arguments
ole_put_property/3	Puts a value to an object's property
ole_uninitialize/0	Uninitialises OLE libraries, releases open objects, closes DLL
ole_version/4	Provides version information on the OLE Automation library

Error Handling

In the event of an error being returned, you may need to use the *ole_last_error/4* predicate to obtain further information as to why the error occurred; for example:

```
?- ole_get_property(word,documents,[1],Documents). <enter>
! ----------------------------------
! Error 212 : OLE Error: Exception
! Goal    : ole_get_property(word,documents,[1],_1960)

Aborted

?- ole_last_error(Source,Desc,HelpFile,HelpContext). <enter>
Source = `Microsoft Word` ,
Desc = `'Item' is not a property.` ,
HelpFile = `wdmain8.hlp` ,
HelpContext = 25342
```

where Source defines the source (ie Windows application name) of the exception, Desc is a description of the exception, HelpFile identifies the relevant help file and HelpContext is an ID that can be used to access the specific help for the exception.

A call to *ole_last_error/4* must be done prior to any other OLE Automation library predicate being called, otherwise the stored information about a particular exception may be discarded or overwritten.

The *ole_last_error/4* predicate is only of use if the error number returned is either 212 (an exception error has occurred) or 246 (an unknown error has occurred); in the case of an unknown error, details of the error will be returned in the first argument of *ole_last_error/4*, the last three arguments having no meaning. For all other errors, the arguments of *ole_last_error/4* have no meaning.

To display the relevant help file for an exception, execute the following:

```
?- ole_last_error(_,_,HelpFile,HelpContext), help(HelpFile,1,HelpContext).
<enter>
```

The OLE Automation errors are as follows:

200 S_FALSE returned

201 Object does not support aggregation

202 EXE launched but did not register class

203 EXE not found

204 EXE can only be launched once

205 DLL not found

206 The registered CLSID for the ProgID is invalid

207 EXE has an error in image

208 DLL has an error in image

209 Object not connected

210 Wrong number of parameters

211 One of the parameters is not a valid type

212 Exception

213 The requested member does not exist

214 Server does not support named parameters

215 One or more of the parameters could not be coerced

216 A required parameter has not been found

217 One or more parameters could not be coerced

218 Illegal use of a reserved function

219 Unknown national language

220 Unknown method or property name

221 A required parameter was omitted

222 Object application not running

223 Wrong version of COMPOBJ.DLL

224 CLSID is not properly registered

225 Object interface not properly registered

226 Error reading from the registration database

227 Error writing to the registration database

228 <UNKNOWN>

229 The type library element has not been found

230 The type library could not be found

231 The type library is not registered

232 The type library has an old format

233 Type mismatch

234 Access denied

235 One or more arguments are invalid

236 Object does not support the requested interface

237 Out of memory

238 An unexpected error occurred

239 Incompatible OLE version

240 Invalid object specified

241 Unsupported return parameter type in invocation

242 Unsupported input parameter type in invocation

243 OLE library has already been initialized

244 Error accessing object dispatch table

245 Object not found

246 Unknown error

Chapter 17 - The Dialog Editor Plug-in

The Dialog Editor plug-in allows easy creation and maintenance of *WIN-PROLOG* dialog code. This chapter tells you how to use the Dialog Editor, how to create and edit dialogs and how to transfer dialog code between the Dialog Editor and your application. **The Dialog Editor generates the Prolog code for creating a dialog; it does not write any of the window handling code required to respond to mouse clicks, menu selections, etc.** Note: this chapter only applies if you have installed the Dialog Editor toolkit.

Starting the Dialog Editor

Once you have installed the Dialog Editor from the *WIN-PROLOG* CD-ROM, the next time you start up *WIN-PROLOG* you will find that a new "Dialog Editor..." menu option has been added to the "Run" menu.

Selecting this option starts the Dialog Editor. When the Dialog Editor starts, you see three windows, as shown below.

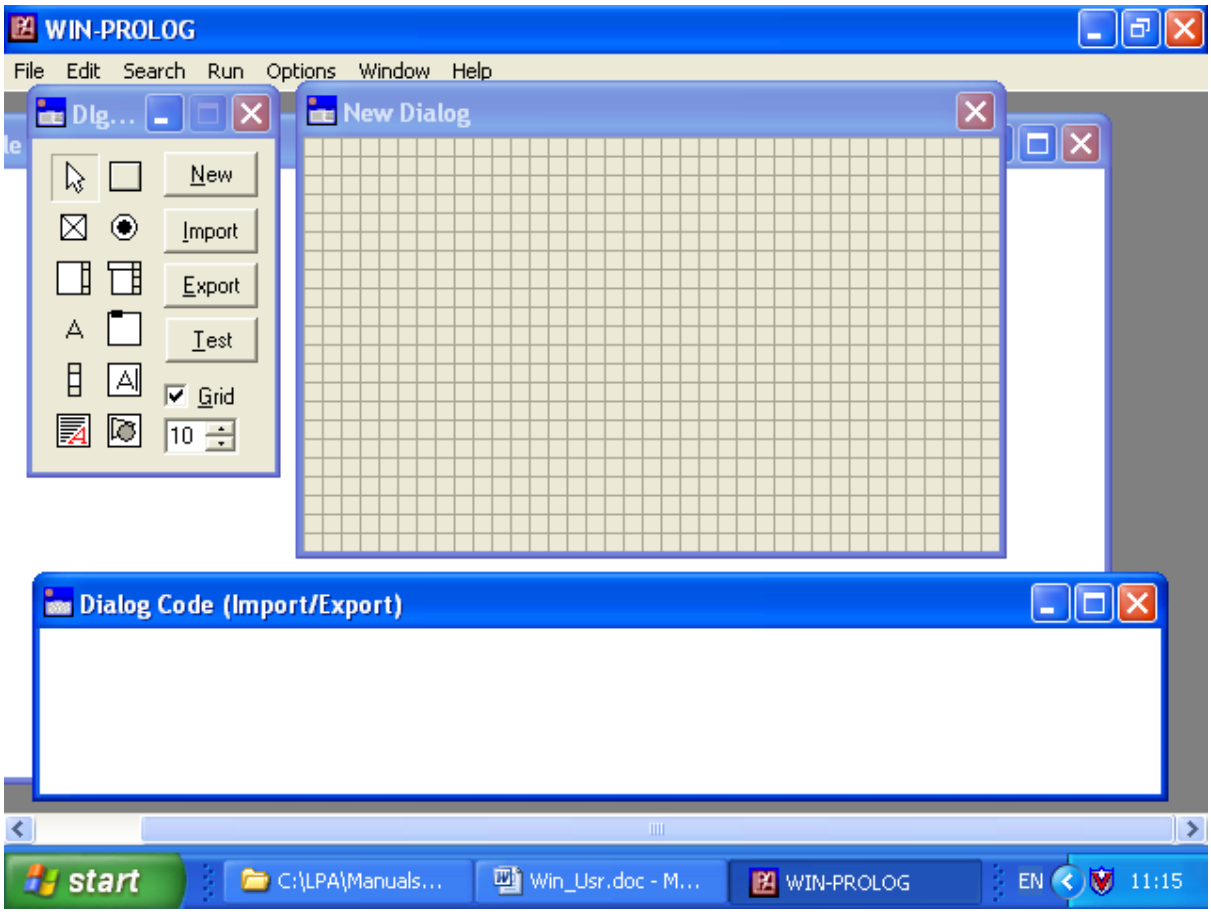


Figure 77 - The Dialog Editor windows

The Scratch Window

Dialogs are drawn and edited in the window that appears on the right hand side of the screen, which initially has the caption "New Dialog". For convenience we refer to this window as the scratch window. The scratch window is a graphical representation of the dialog being edited. Placing, moving and re-sizing controls on this window affect the controls in the dialog being created.

The Toolbox

The window that appears on the left, titled "DlgEdit", contains tools for controlling the editor. For convenience, we refer to this window as the toolbox as shown in *Figure 78*. To create a dialog control you simply select the appropriate tool in the toolbox then click and drag the mouse in the scratch window. The toolbox also lets you control the grid that appears in the scratch window, used to set standard sizes and positions for new controls.

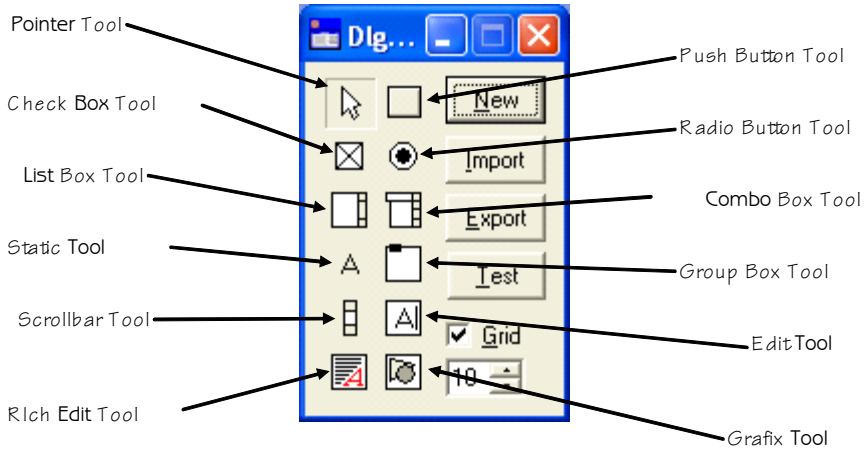


Figure 78 - The toolbox

Drawing a Dialog

Let's start by creating a very simple dialog with just two controls, a button and a static text control. Select the button tool (the second one in from the top-left hand corner, as shown in *Figure 79*) and create a button in the scratch window by clicking and then dragging with the mouse, as shown in *Figure 80*.

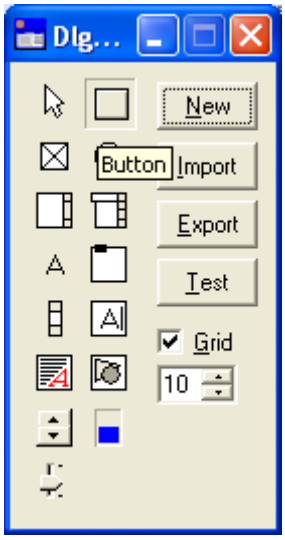


Figure 79 - Selecting the button tool

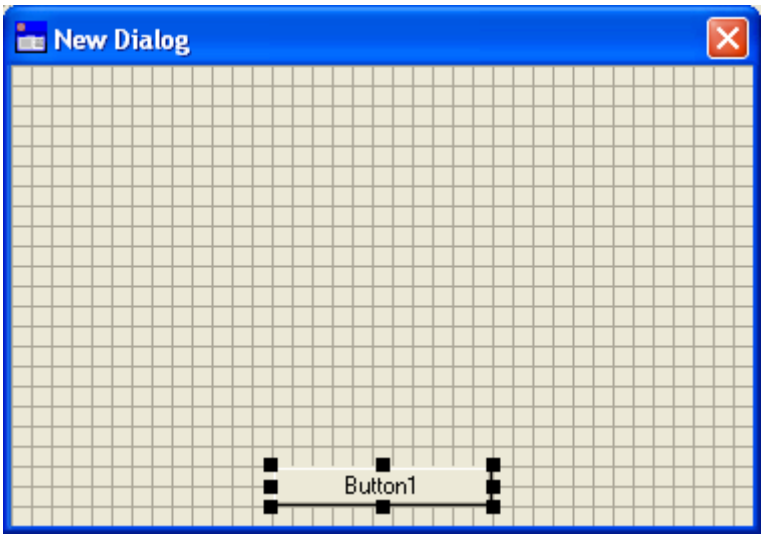


Figure 80 - Creating a button in the scratch window

Now select the static control tool (first column, fourth row, as shown in *Figure 81*) and create a static text control by clicking and dragging with the mouse at the required position in the scratch window, as shown in *Figure 82*.

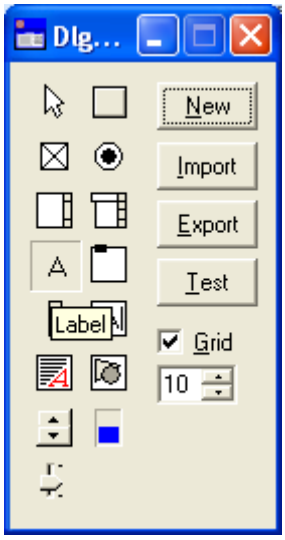


Figure 81 - Selecting the static tool

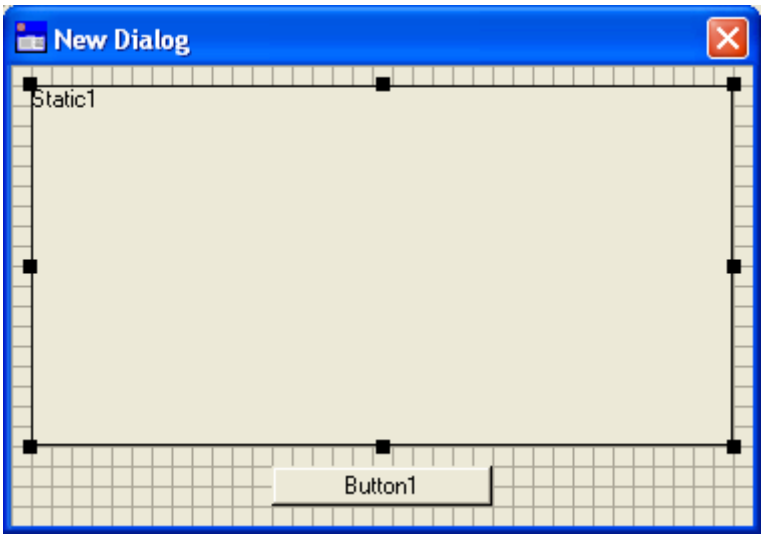


Figure 82 - Creating a static control in the scratch window

Testing the Appearance of Dialogs

Now that we have created the scratch version of our simple dialog, we can test how the actual dialog would appear. To do this, select the "Test" button on the toolbox. This will show our edited dialog as it would actually appear in an application; this can be seen in Figure 83. You will also notice that the text of the "Test" button on the toolbox has changed to "Edit"; selecting this button will return the Dialog Editor to edit mode, as will clicking the close icon on the dialog being tested.

Figure 83 – Testing/Running the edited dialog

Importing and Exporting Dialogs

Having "created" a dialog and seen how it will appear, the next step will be to generate the code for creating the dialog and place this into an application. For this purpose the Dialog Editor has a third window titled "Dialog Code (Import/Export)". For convenience, we refer to this window as the I/E window. Select the "Export" button on the toolbox and some dialog code will appear in the I/E window:

```
new_dialog :-
  _S1 = [ws_sysmenu,ws_popup,ws_caption,dlg_ownedbyprolog],
  _S2 = [ws_child,ws_visible,ws_tabstop,bs_pushbutton],
  _S3 = [ws_child,ws_visible,ss_left],
  wdcreate( new_dialog,      `New Dialog`,    164, 39, 376, 262, _S1 ),
  wccreate( (new_dialog,1000), button, `Button1`, 130, 200, 110, 20, _S2 ),
  wccreate( (new_dialog,11000), static, `Static1`, 10, 10, 350, 180, _S3 ).
```

Hiding the Dialog Editor Windows

The toolbox and scratch windows have been designed to always appear on top of the *WIN-PROLOG* application window. It is not always convenient to have these windows in the way, but fortunately, there is a simple solution to this problem. Click on the minimise button at the top-right hand of the toolbox and both the toolbox and the scratch window will be minimised.

The Generated Code

With the Dialog Editor window now minimised, you can take a look at the code in the I/E window. You will notice that a Prolog program, *new_dialog/0*, has been generated. This program consists of one call to *wdcreate/7* and two calls to *wccreate/8*, one for each control (i.e. button and static) in the dialog. The controls are listed in accordance with their tab ordering, which initially is the same as the order of creation.

You will notice that the dialog itself has been allocated the styles *ws_sysmenu*, *ws_popup*, *ws_caption* and *dlg_ownedbyprolog*, the button control *ws_child*, *ws_visible*, *ws_tabstop* and *bs_pushbutton* and the static control *ws_child*, *ws_visible* and *ss_left*. You are referred to Microsoft's own documentation which gives a detailed explanation of what each style means except for those unique to ***WIN-PROLOG*** such as *dlg_ownedbyprolog*.

Select and copy the *new_dialog/0* program in the I/E window so that it can be pasted into a ***WIN-PROLOG*** source code window; if this is the last piece of code in the destination window, ensure that you have entered at least one *<return>* character after the code.

Creating a Dialog

Change focus to the ***WIN-PROLOG*** Console window and select the "File/New..." menu option. Paste the *new_dialog/0* program into the "Untitled-0" window and then select the "Run/Compile" menu option. You should now be able to run the following goal at the ***WIN-PROLOG*** command line:

```
?- new_dialog.                                          <enter>
```

In this, and all other command line examples, only type the characters in **bold** letters, and press the named key for anything bracketed in *<italics>*.

This will create the dialog but not display it. If you execute *wdict/2* at the ***WIN-PROLOG*** command line, you will see that 'new_dialog' has been added to the list of currently defined windows owned by ***WIN-PROLOG***:

```
?- wdict( 0, X ).                                       <enter>
X = [new_dialog,'Untitled-0']
```

Methods of Invocation

Under Windows, a dialog can be invoked in one of two modes - modal or modeless. Any dialog generated using the Dialog Editor is mode independent; the method of invocation determines whether it is a modal or a modeless dialog.

Predicate to Invoke a Modal Dialog	**Predicate to Invoke a Modeless Dialog**

call_dialog/2	*show_dialog/1* *wshow/2*

Invoking as a Modal Dialog

Now, to show and handle the dialog as a modal dialog, run the following goal at the *WIN-PROLOG* command line:

```
?- call_dialog( new_dialog, Result ).                    <enter>
```

This will create a modal dialog; once displayed, all other *WIN-PROLOG* windows will be disabled until you click on either the "Button1" button or the dialog's close icon. If you click on the "Button1" button, the dialog will close and *Result* will be instantiated to 'button1'. Re-run the above goal, but this time, click on the close icon; the dialog will close and *Result* will become instantiated to 'close'. *Result* is given whatever value has been returned by the dialog's "*window handler*", which in this case, is the default built-in window handler.

If your dialog contained other controls and you wanted to return these to your program as well, *call_dialog/2* is inadequate on its own. You may like to take a look at the supplied DLGCALL.PL library file in the LIBRARY directory, which calls a modal dialog and returns, via some post processing, both the button pressed and a list of controls and their values.

Control Type	Predicate to Get Value
combobox	*wcmbsel/3*
edit	*wedttxt/2*
listbox	*wlstsel/3*
radio	*wbtnsel/2*
rich edit	*wrchsel/3 or wrchtxt/3*
scrollbar	*wthumb/3*

Invoking as a Modeless Dialog

We are now going to look at invoking our dialog as a modeless dialog. This involves showing the dialog first using *wshow/2* and then defining our own window handler to handle it. To show the dialog, you should run the following goal at the command line (the result of this is shown in *Figure 84*).

```
?- show_dialog( new_dialog ).                    <enter>
```

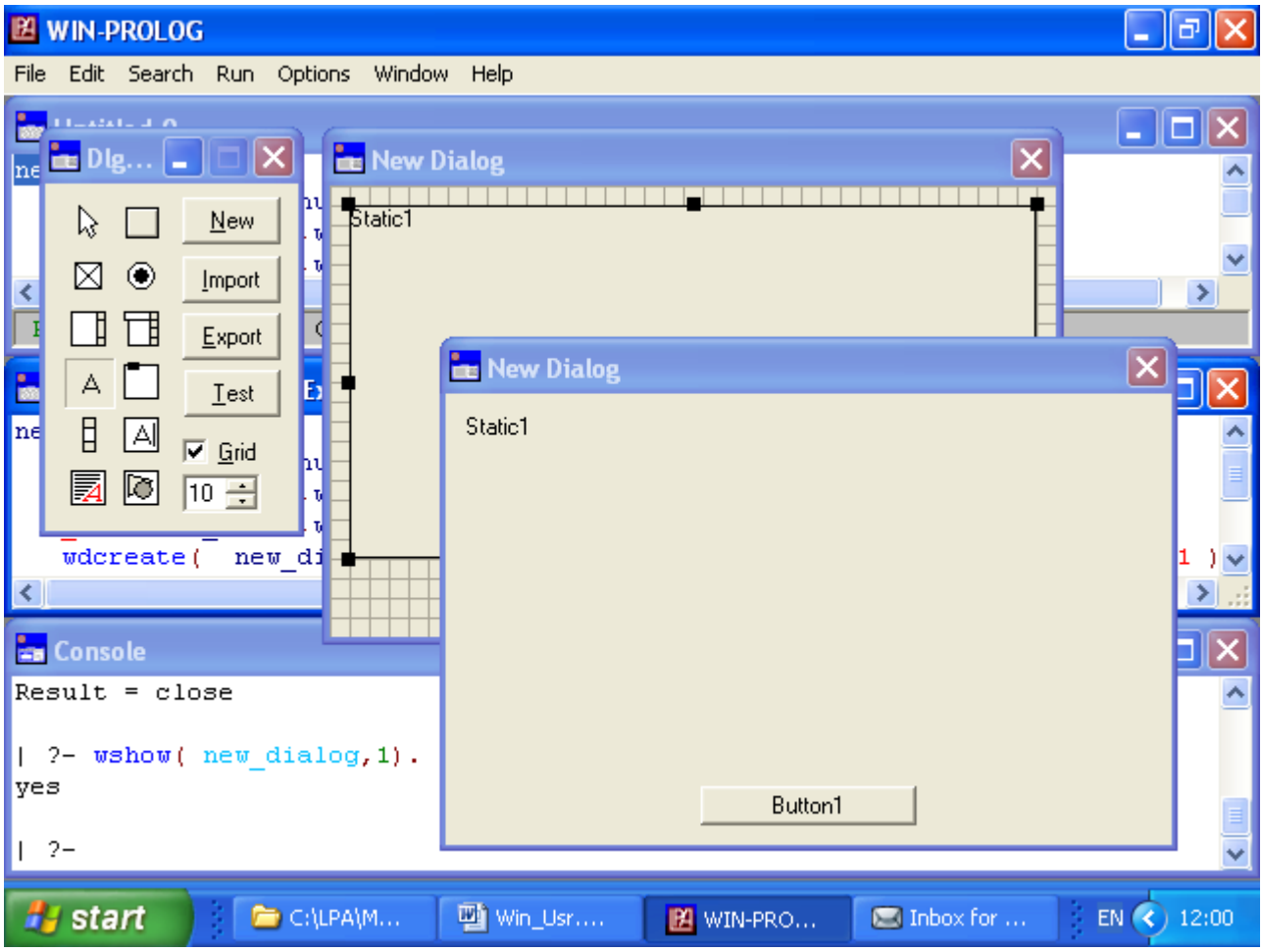


Figure 84 - Showing the new_dialog window

You can close the dialog either by clicking on its close icon or by executing *wclose/1* at the *WIN-PROLOG* command line:

```
?- wclose( new_dialog ).                         <enter>
```

A Window Handler for a Modeless Dialog

The modeless dialog that we have just displayed needs a window handler defined for it in order for it to be of any use. Let's just make the "Button" button close the dialog when clicked upon and return you to the *WIN-PROLOG* Console window. Enter the following program into the "Untitled-0" window to go with the *new_dialog/0* clause you already have:

```
test :-
  new_dialog,
  wshow(new_dialog,1),
  window_handler( new_dialog, new_dialog_handler ).
```

The call to *window_handler/2* is saying that the dialog, *new_dialog*, is to be handled by the user-defined window handling predicate, *new_dialog_handler/4*, which is defined as follows:

```
new_dialog_handler( (new_dialog,1000), WindowsMessage, _, _ ) :-
  WindowsMessage = msg_button,
  !,
  beep( 440, 250 ),
  wclose(new_dialog).

new_dialog_handler(Window,Message,Data,Result) :-
  write( Window - Message - Data - Result ),
  nl,
  ttyflush,
  window_handler(Window,Message,Data,Result).
```

A window handling predicate, be it the built-in *window_handler/4* or a user-defined one, has four arguments. The first argument is the window handle itself (e.g. (new_dialog,1000) is the 'Button1' button). The second argument is the message received from Windows (e.g. msg_button).

The first clause matches with the "Button1" button being pressed; a msg_button message will be received from Windows for this control. The body of the clause then beeps and closes the 'new_dialog' dialog.

The second clause is a 'catch all' and passes all other messages received for the 'new_dialog' dialog to the built-in *window_handler/4* predicate. The call to *write/1* outputs to the console information about every message received concerning this window.

Running the Program

Having entered the window handling code into the "Untitled-0" window, you now need to compile the code by selecting the "Run/Compile" menu option. You should now be able to run the following goal at the **WIN-PROLOG** command line:

```
?- test.                                    <enter>
```

Try moving the dialog, clicking on controls, moving the pointer in and out of the dialog, hovering over controls, etc. to see what messages, as received from Windows, are written out to the console window. A selection of possible messages, in Windows Message order, is as follows:

```
?- new_dialog - msg_close - 0 - _10208
?- new_dialog - msg_focus - (1,1) - _35138
?- (new_dialog,1000) - msg_focus - new_dialog - _15506
?- new_dialog - msg_fuzzy - (new_dialog,1000) - _35402
?- (new_dialog,1000) - msg_key - (down,29,17) - _23746
?- (new_dialog,1000) - msg_key - (repeat,29,17) - _37470
?- (new_dialog,1000) - msg_key - (up,29,17) - _51842
?- new_dialog - msg_menu - 2 - _26438
?- new_dialog - msg_mouseenter - (1,84) - _31096
?- (new_dialog,1000) - msg_mouseenter - (14,19) - _7784
?- new_dialog - msg_mousehover - (73,86) - _22740
?- (new_dialog,1000) - msg_mousehover - (20,9) - _49634
?- new_dialog - msg_mouseleave - (72,221) - _21970
?- (new_dialog,1000) - msg_mouseleave - (21,1) - _26600
?- new_dialog - msg_mousemove - (1,84) - _31288
?- (new_dialog,1000) - msg_mousemove - (18,13) - _40936
?- new_dialog - msg_move - (167,68) - _34920
?- (new_dialog,1000) - msg_paint - button_up - _15792
?- new_dialog - msg_popup - (42,39) - _18648
?- (new_dialog,1000) - msg_popup - (83,10) - _42988
?- new_dialog - msg_size - (370,230) - _34728
```

When you finally click on the "Button1" button, a beep will sound and the dialog will close. So now we've demonstrated how to create a dialog that has a button called "Button1" and a static field with the text "Static1" in it.

Returning To the Dialog Editor

Let's return to the Dialog Editor and do something more useful with our dialog. You can bring back the Dialog Editor by clicking on the DlgEdit window's "Restore Up" icon. This will re-instate both the toolbox and the scratch window. Notice that our dialog is still in the scratch window ready for editing.

Changing the Text of a Push Button

At present we have a dialog with a single text field and button. Let's convert this into a simple "notify message" dialog, one where the text of the button is normally set to "Ok". To do this, double-click on the button picture in the scratch window. This will cause the "Push Button Style" dialog to be displayed, as shown below.

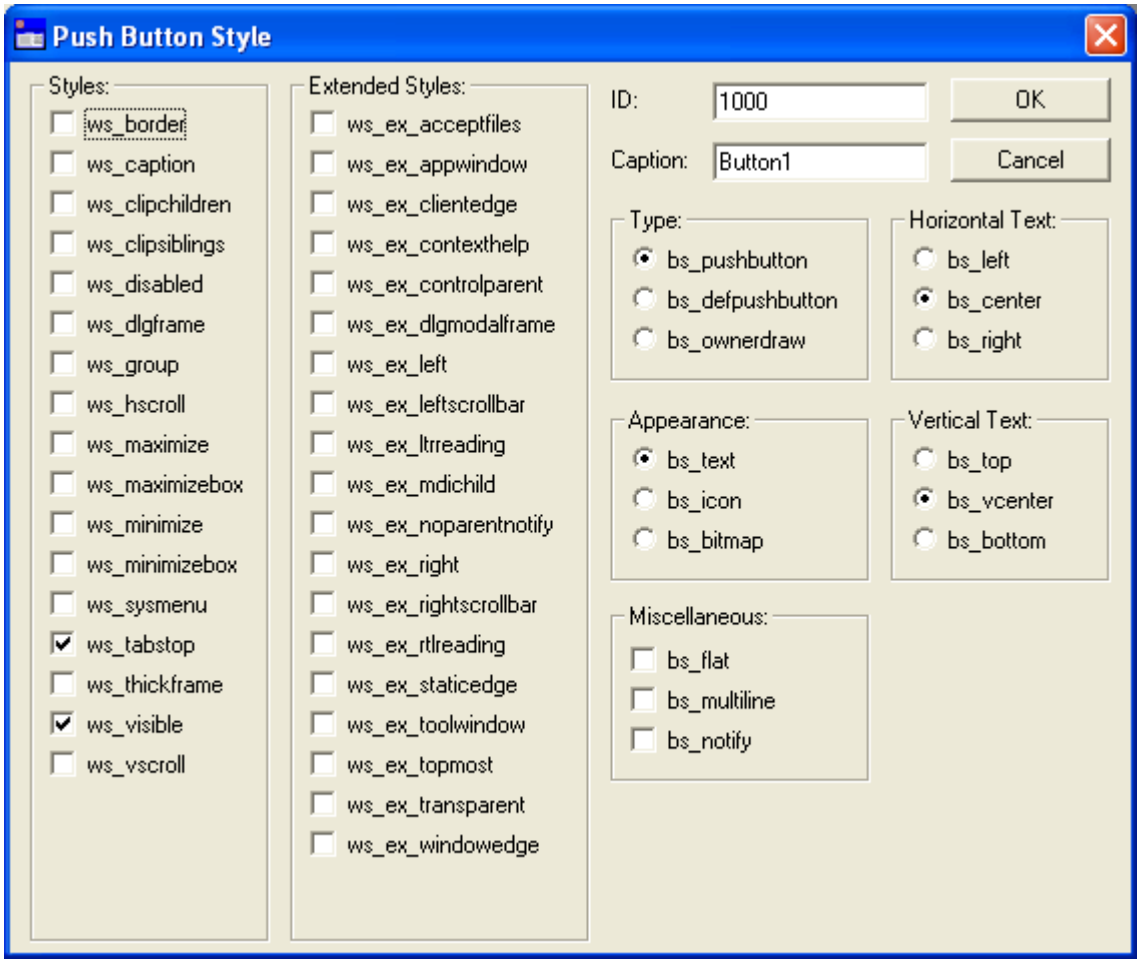


Figure 85 - The push button style dialog

To change the text of the button, replace the contents of the "Caption" edit field with the text "OK" and accept the dialog by clicking on the "OK" button. The text of the button should now be "OK", as shown in *Figure 86*.

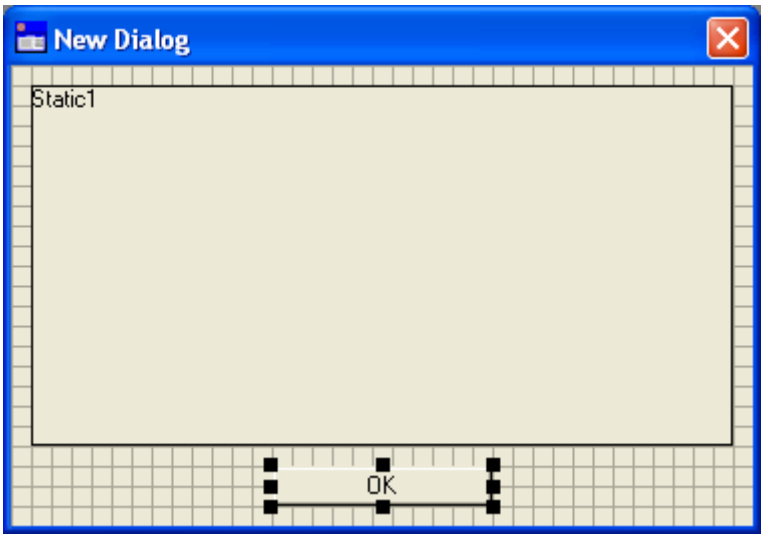


Figure 86 - The button control with the text "Ok"

Now we will change the text of the static control to contain an appropriate message. Double-click on the static control and the "Static Style" dialog will be displayed, as shown in *Figure 87*.

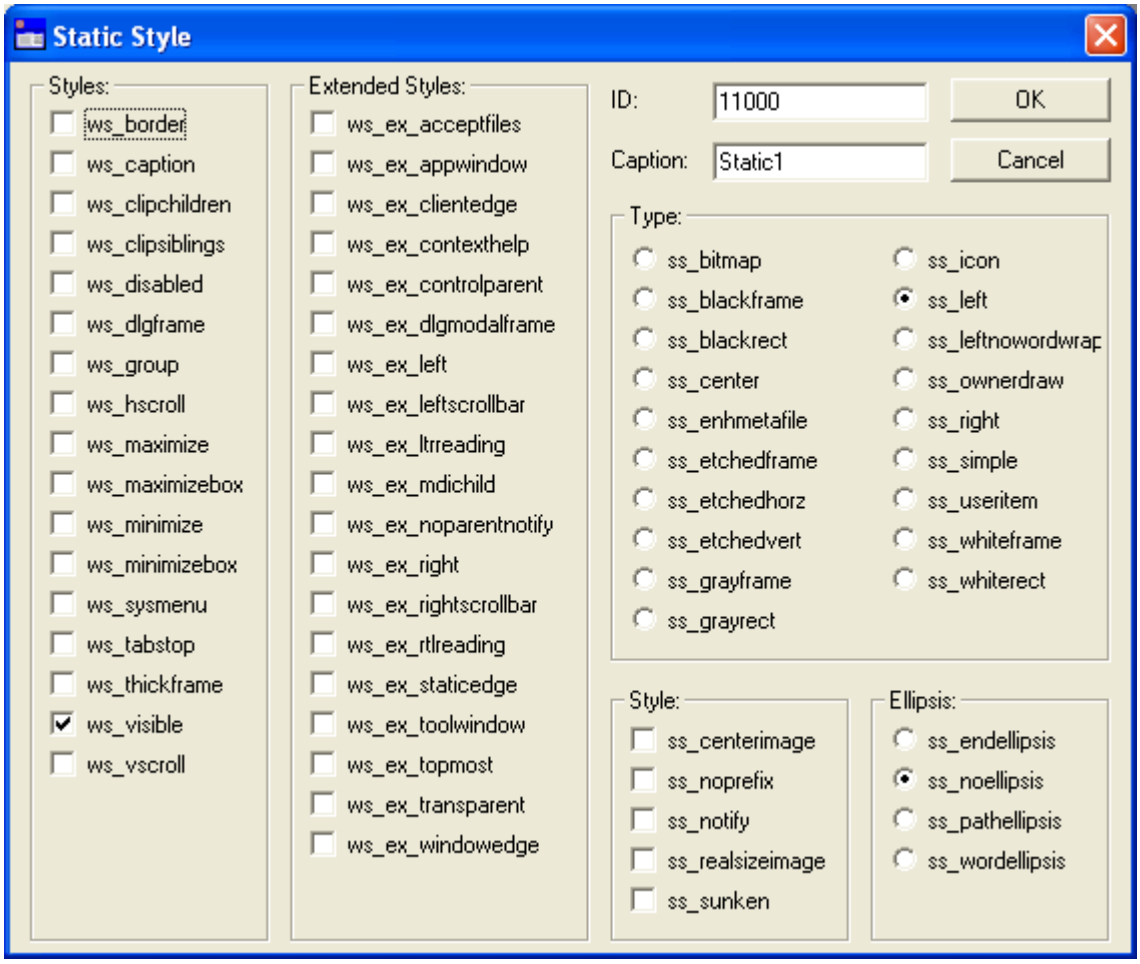


Figure 87 - The static style dialog

Now type: "The program has now finished running" in the "Caption:" field.

Changing the Style of a Control

Notice that there are many other parameters for a control other than the caption that may be changed. Before accepting the dialog, let's first select the "ss_center" radio button. Now accept the dialog by clicking on the "OK" button or pressing *<enter>*.

The text of the static field should now display its new contents, as shown in *Figure 88*.

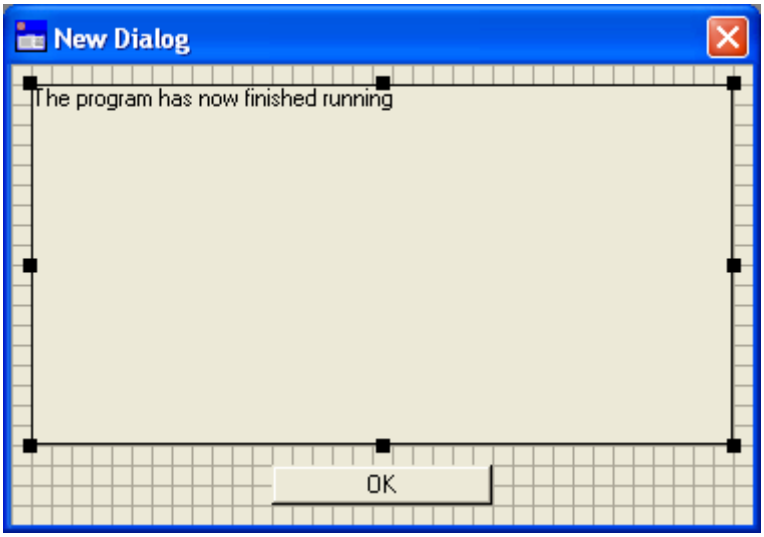


Figure 88 - The new text of the static control

Although the new text is displayed, the style change is not displayed. To see how the dialog will actually look when it is created, click on the "Test" button.

The Dialog Caption and Style

Now we want to set the title of our dialog. Double-clicking on any visible part of the gridded background area in the scratch window will display the "Dialog Style" dialog as shown in

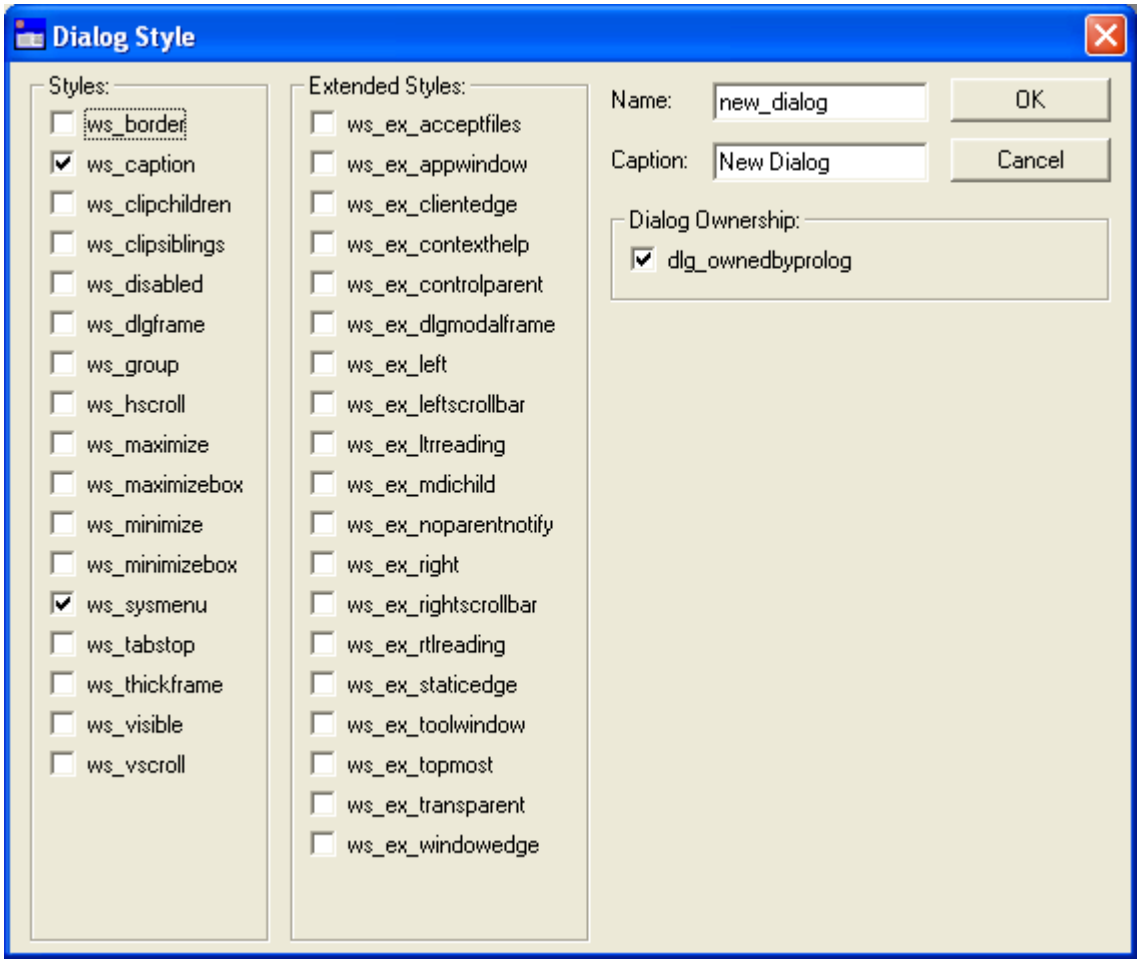


Figure 89 - The Dialog Style dialog - Before amendment

Set the text of the "Caption:" field to "Program Termination" and then we'll take a look at some of the other options in the dialog.

The "Name:" field sets the name given to the dialog window and the name used by the Dialog Editor when exporting the program for creating the dialog. Let's change this to "notify_dialog".

The "Dialog Ownership:" group allows you to say whether the dialog is owned by the desktop, in which case it exists as a separate entity to ***WIN-PROLOG*** as contrasted to being owned by Prolog, in which case it always appears on top of the ***WIN-PROLOG*** application window and gets minimised when the ***WIN-PROLOG*** application window is minimised. Given the type of dialog we are creating it probably doesn't matter whether the owner is ***WIN-PROLOG*** or the desktop.

The "Styles" and "Extended Styles" sections allow you to include or omit various features on the dialog, such as system menus, maximise and minimise boxes. None of which really apply to a simple notification dialog. Many of the attributes shown here will not affect the display in the scratch window, however, testing the dialog will show the changes. Now we have set the dialog, as shown in *Figure 90*, accept it by clicking on the "OK" button or pressing *<enter>*.

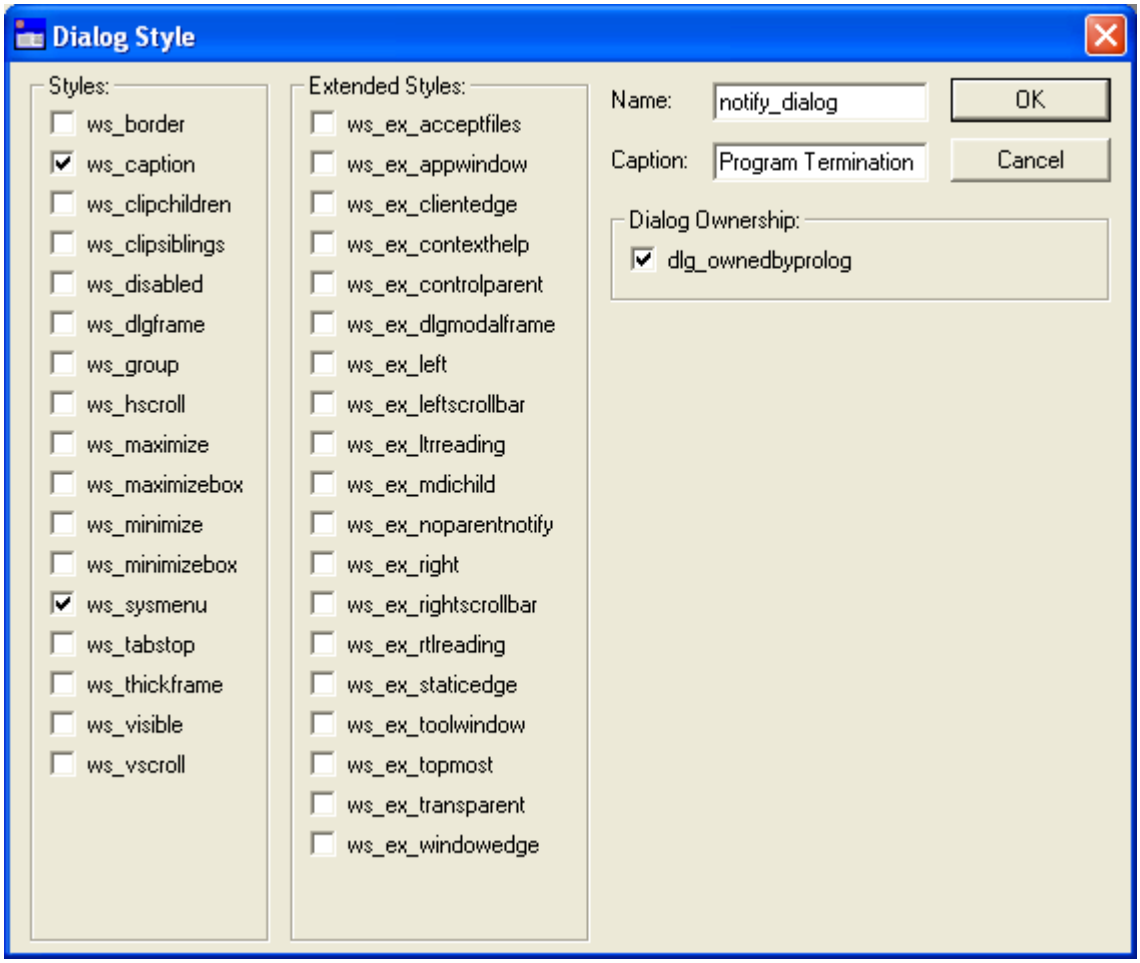


Figure 90 - The Dialog Style dialog - After amendment

You should now see that the caption of the scratch window has changed to "Program Termination".

Re-sizing Controls

As it stands, there is now only one thing wrong with our dialog. The dialog and its controls are too big for the amount of text that we've put into it. This gives us a good opportunity to demonstrate how to re-size and reposition the elements of the dialog. Select the "Static" control in the scratch window by single-clicking with the mouse on its rectangle. Some black rectangles, or "resize handles", will appear at the border of the control, as shown in *Figure 91*.

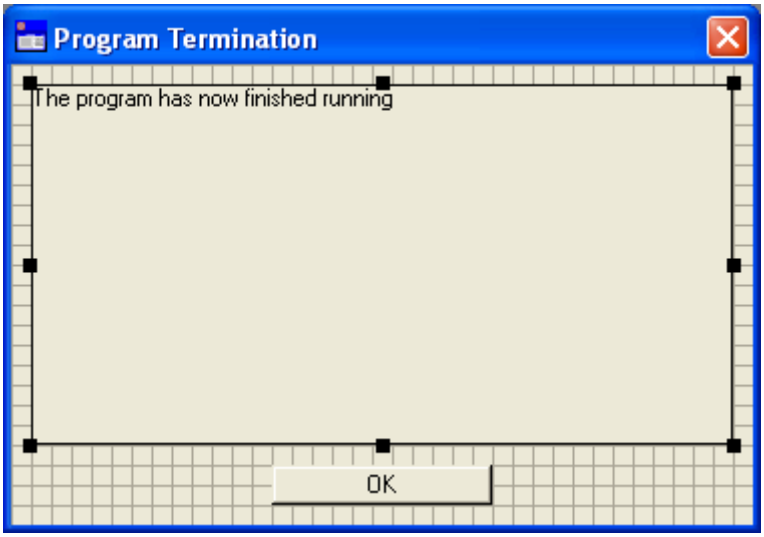


Figure 91 - The re-size handles on the static control

Control Handles

The control handles are areas on the screen that you can "grab" with the mouse by positioning the mouse cursor on top of one and then clicking and holding down the left mouse button. If you then drag an area with the mouse a resizable box will follow the mouse position. Releasing the mouse button will resize the current selection to this box. The central top, right, bottom and left handles constrain the resizing to the appropriate direction. For example, if you grab the bottom-centre handle and drag it, then the control will only be resized in the vertical direction. The handles at the corners allow you to freely resize the box in both horizontal and vertical directions.

If the grid snapping is on (i.e. the "Grid" checkbox in the DlgEdit dialog is ticked) and a control is resized then the control will be snapped to the grid. It is only snapped according to the resize handle you have used. For example, if you grabbed the bottom-right handle of a control and dragged it, then the bottom and right of the control will be snapped to the grid, but the top and left sides of the control will be left in place.

Position the mouse cursor over the bottom-right hand resize handle, you will see the cursor change to a diagonal arrow. Pick up the resize handle, by clicking and holding down the left mouse button, and move the mouse until the corner of the static control is one grid square below the top of the static control and one square beyond the right hand end of the text, as shown in *Figure 92*.

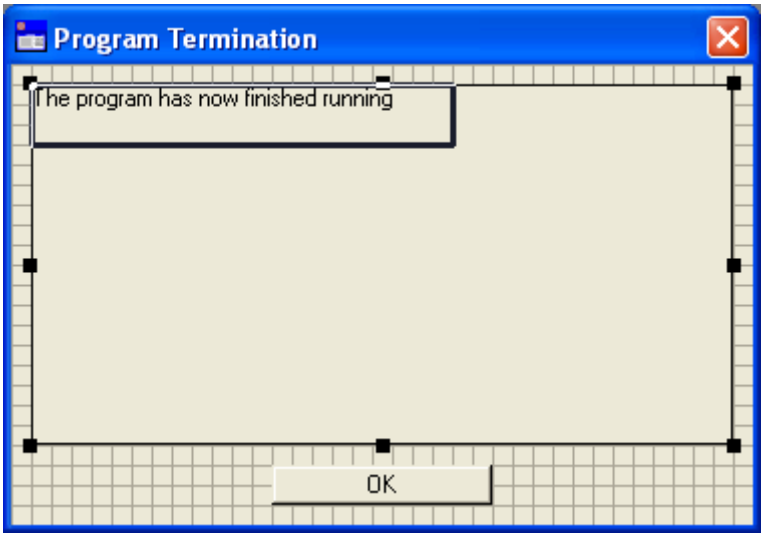


Figure 92 - Resizing the static control

As the grid-snap is on, the bottom and right borders of the control will be snapped to the nearest grid lines. Now let go of the mouse and the window will be re-sized as shown in *Figure 93*.

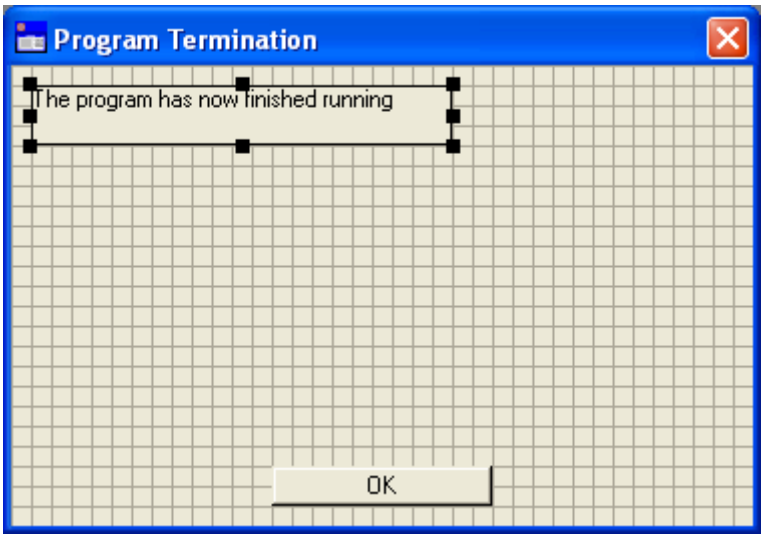


Figure 93 - The resized static control

Moving Controls

Now we are going to move a control, leaving its size the same. Select the button control, by clicking on it once with the mouse, and position the cursor at its centre. The cursor should be an arrow pointing up, down, left and right. Pick up the control, by clicking and holding down the left mouse button, and move it until it lies one row below the static control and the left hand side of the button is five squares in from the left hand side of the static control. Release the mouse button and the button control will be moved to its new location, as shown in *Figure 94*. Note, you can also move controls by using the up, down, left and right arrow keys, this is most useful for positioning controls in non-snap mode.

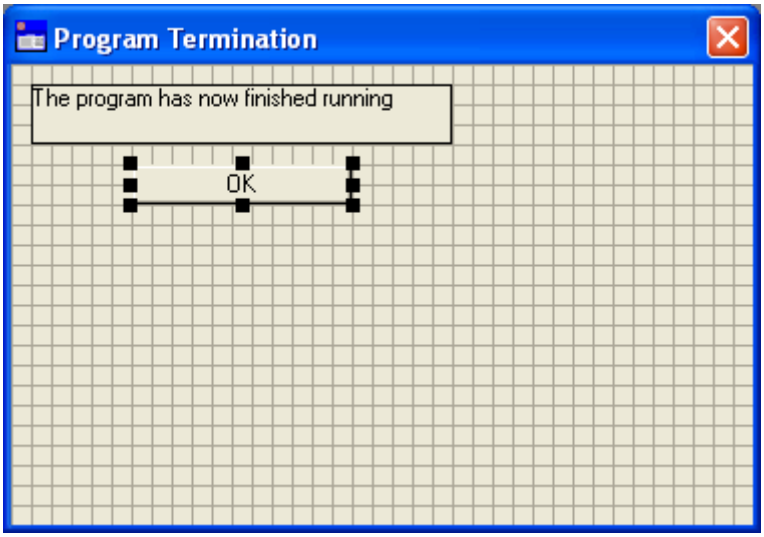


Figure 94 - The repositioned button control

Resizing Dialogs

Finally, we need to resize the dialog window itself. Move the mouse cursor until it is over the bottom-right hand corner of the dialog. The cursor should change to a diagonal (top-left to bottom-right) arrow. Click and hold down the left mouse button and drag the corner of the window until it is one square below the button control and one square beyond the right-hand end of the static control. After you release the mouse button, the dialog should appear as shown in *Figure 95*.

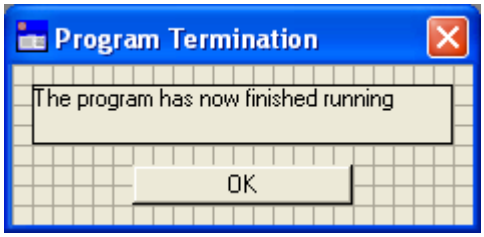


Figure 95 - The resized dialog

This can be done programmatically using the *size_dialog/2* predicate.

Exporting the Dialog Code

Export the dialog code to the I/E window, using the toolbox "Export" button.

```
notify_dialog :-
  _S1 = [ws_caption,ws_sysmenu,dlg_ownedbyprolog],
  _S2 = [ws_child,ws_tabstop,ws_visible,bs_pushbutton,bs_text,bs_center,bs_vcenter],
  _S3 = [ws_child,ws_visible,ss_center],
  wdcreate( notify_dialog,      `Program Termination`,                 260, 100, 236, 112, _S1 ),
  wccreate( (notify_dialog,1000), button, `OK`,                        60,  50, 110,  20, _S2 ),
  wccreate( (notify_dialog,11000), static, `The program has now finished running`, 10,  10, 210,  30, _S3 ).
```

Notice that the name of the program generated by the Dialog Editor and the name of the dialog used in the calls to *wdcreate/7* and *wccreate/8* has become "notify_dialog" (the name we set in the Dialog Style dialog). Now take this code and paste it into the previously created "Untitled-0" window.

When displayed (in the way outlined earlier), the notify_dialog window will appear as shown in *Figure 96*.

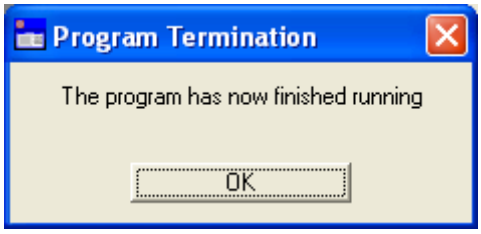


Figure 96 - The notify_dialog window

Quitting the Dialog Editor

When you have finished with editing dialogs, you can quit from the Dialog Editor by using the "Close" option on the toolbox system menu or by pressing *<Alt-F4>*. Quitting the Dialog Editor loses the dialog currently being edited, so you will be given the option to cancel this operation. Select the close option on the toolbox system menu and the message box, shown in *Figure 97,* will be displayed. Click on the "Yes" button and the Dialog Editor windows will be closed.

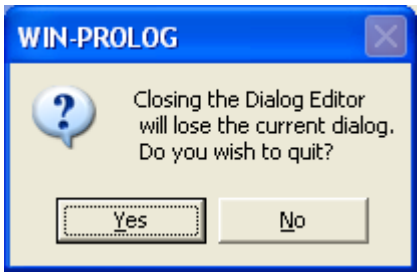


Figure 97 - Quitting the Dialog Editor

More Features

So far we've covered the process of creating dialogs and controls from new and how to change their sizes, positions, captions and styles. There are more editing features in the Dialog Editor, such as copying and deleting controls, but one of its major strengths is that it can import existing dialog code, so long as it is in the right form, and allow you to modify dialogs that you have already worked on. Let's suppose that we want to create a dialog with a similar appearance to our "notify dialog", but this time, with two buttons and a listbox. To demonstrate these extra features we first need to restart the Dialog Editor.

Restarting the Dialog Editor

The Dialog Editor can be re-started by selecting the "Run/Dialog Editor..." menu option. This will cause the Dialog Editor windows to re-appear as they did when the editor first started up.

Importing Dialog Code

Selecting the "Import" button in the toolbox makes an attempt to convert the contents of the I/E window into a graphically editable dialog in the scratch window. If it is successful, the dialog currently being edited will be lost, so you will be given the option to cancel this operation.

Copy the *notify_dialog/0* program from the "Untitled-<n>" source code window and paste it into the I/E window. Then select the "Import" button in the Dialog Editor toolbox. You should see the "Import warning", as shown in *Figure 98*.

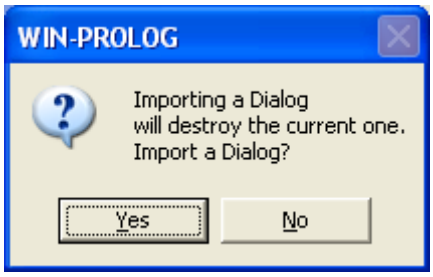


Figure 98 - The import warning dialog

Click on the "Yes" button and the dialog we created earlier will re-appear.

When importing dialog code, there are a few rules that need to be observed to ensure that the Dialog Editor can interpret the code successfully. These are as follows:

- There should be a single Prolog clause defining the dialog. If more clauses are placed in the Import\Export Window only the first clause will be used.
- The code should be correct Prolog syntax.
- There should be a single call to *wdcreate/7* followed by 0 or more *wccreate/8* calls.
- These calls must not contain variables and their arguments must be of the correct type, i.e. they would work correctly if called directly.
- The clause must be terminated by a full-stop followed by a new-line.
- Calls other than *wdcreate/7* and *wccreate/8* will be discarded.

If any errors are detected then the import will be aborted with an appropriate error message.

The Toolbox Grid Settings

Now that we have imported our dialog we'll have a brief look at some of the other options. The grid setting section of the toolbox controls the size and availability of the "snap to" grid in the scratch window. Changing the numbers in the edit boxes will change the grid size on the scratch window, the default being 10 pixels by 10 pixels. The grid checkbox toggles the grid visibility and snapping on and off. When the checkbox is selected, the grid is visible and any controls subsequently edited will be snapped to the grid.

Change the numbers in both the grid settings edit boxes to 8. You will see that the grid on the scratch window has immediately changed, as shown in *Figure 99*.

*Figure 99 - An 8 * 8 grid in the scratch window*

Any control subsequently placed on the scratch window will be snapped to this new grid setting.

Copying Controls

Another useful feature provided by the Dialog Editor is the ability to copy controls that have already been made. Selecting and then moving a control while the <ctrl> key is pressed, changes the cursor to an upward arrow and causes a copy of the selected control to be created at the new position.

To demonstrate this, we'll create a copy of the "OK" button and rename it "Cancel". To start with, move the "Ok" button (in the normal way) so that its left-hand side is in line with the left-hand side of the static field. Now shrink the right-hand side of the button by one square (this gives us the space to create a copy of the button), as shown in *Figure 100*.

Figure 100 - The repositioned "Ok" button

Now press the <ctrl> key and, while it is pressed down, click and drag the "OK" button to a new position so that its right-hand edge is in line with the right-hand edge of the static control. When you release the mouse button, a new control will be created on the right-hand side of the dialog, as shown in *Figure 101*.

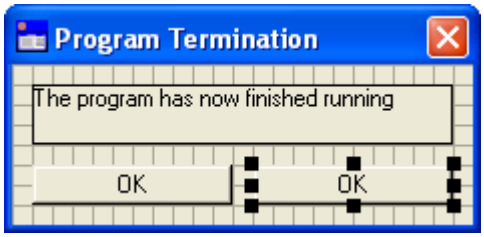


Figure 101 - A copy of the "OK" button

You can now double-click on the new "OK" button and change its caption in the "Push Button Style" dialog to "Cancel". You should end up with a dialog similar to that shown in *Figure 102*.

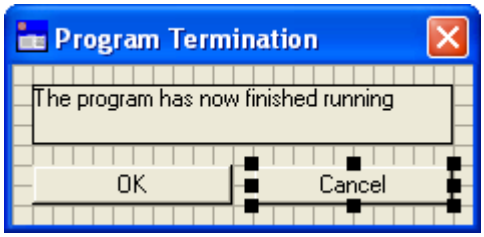


Figure 102 - The dialog showing the new "Cancel" button

Deleting Controls

Now we want to replace the static control with a combo box. To do this we first need to delete the static control. Controls are deleted using the <delete> key, which deletes the current selection. If nothing is selected, pressing the <delete> key has no effect.

Select the static control by clicking on it once with the mouse, its resize handles should appear, as shown in *Figure 103*.

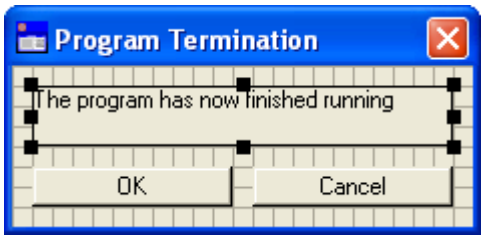


Figure 103 - Selecting the static control

Now press the <delete> key; the static control will be removed from the dialog, as shown in *Figure 104*.

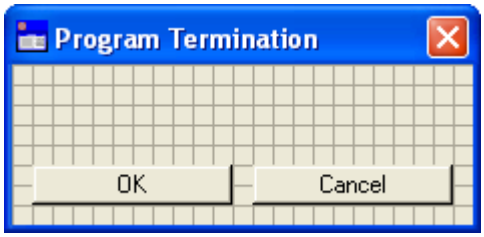


Figure 104 - The dialog after deleting the static control

Creating a Combo Box

Now we can put a combo box in place of the static control. Select the combo box tool from the toolbox; this is the one in the second column, third row, as shown in *Figure 105*.

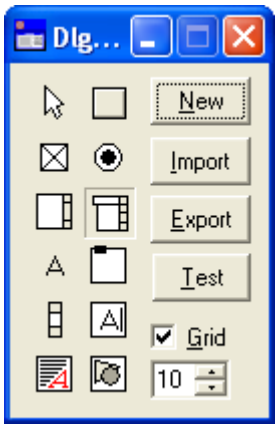


Figure 105 - Selecting the combo box tool

Now click and drag with the mouse in the scratch window to create a combo box that starts one square in from the left and top sides of the dialog and finishes one square in from the bottom and right sides of the dialog, as shown in *Figure 106*.

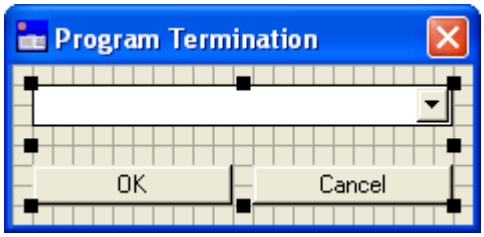


Figure 106 - The new combo box control

You will notice that the handles for the combo box actually extend beyond the picture that is drawn for it. This is because combo boxes are made internally from three controls: an edit control, a button and a list box. The combo box initially appears as a single edit control with a button beside it. When you click on the button, a list box appears beneath the edit control. When you create a combo box you are actually specifying the size of the combination of all three controls.

Selecting Combo Boxes

Because these controls tend to lie over the top of other controls, combo boxes are only sensitive to selection with the mouse in the edit control and button shown at the top of the control. This allows you to select the underlying controls, which would otherwise be difficult to get at. This is also true for group boxes and static controls.

Populating List/Combo Boxes

The process of populating a list or combo box with values can not be done within the Dialog Editor; this currently has to be done manually. The following program add the values "The", "Quick", "Brown" and "Fox" to the combobox, sets "Brown" as the current selection and then defines a window handler, which when you click on the "OK" button, writes the combo box's current selection to the Console window:

```
new_dialog :-
  _S1 = [ws_caption,ws_sysmenu,dlg_ownedbyprolog],
  _S2 = [ws_child,ws_tabstop,ws_visible,bs_pushbutton,bs_text,bs_center,bs_vcenter],
  _S3 = [ws_child,ws_visible,ws_tabstop,cbs_dropdown],
  wdcreate( new_dialog,     `New Dialog`,     467, 243, 236, 112, _S1 ),
  wccreate( (new_dialog,1000), button,  `OK`,     10,  50, 100,  20, _S2 ),
  wccreate( (new_dialog,1001), button,  `Cancel`, 120,  50, 100,  20, _S2 ),
  wccreate( (new_dialog,5000), combobox, `Combo1`,  10,  10, 210,  60, _S3 ).

test :-
  new_dialog,
  wcmbadd( (new_dialog,5000), -1, `The`, 0 ),
  wcmbadd( (new_dialog,5000), -1, `Quick`, 0 ),
  wcmbadd( (new_dialog,5000), -1, `Brown`, 0 ),
  wcmbadd( (new_dialog,5000), -1, `Fox`, 0 ),
  wcmbfnd( (new_dialog,5000), -1, `Brown`, Posn ),
  wcmbsel( (new_dialog,5000), Posn, 1 ),
  show_dialog(new_dialog),
  window_handler( new_dialog, new_dialog_handler ).

new_dialog_handler((new_dialog,1000),msg_button,_,_) :-
  wtext( (new_dialog,5000), Selected ),
  write(Selected),
  nl,
  wclose(new_dialog).

new_dialog_handler(Window,Message,Data,Result) :-
  window_handler(Window,Message,Data,Result).
```

You may like to take a look at example files in the LIBRARY and EXAMPLES directories, from where some of the above source code came.

Reference

The remaining sections in this guide give reference information on the various tools, style dialogs and menus available in the Dialog Editor.

The Toolbox Tools

In this section we give a brief description of the tools for creating control items that are available on the toolbox. Plus a short description of the style dialog for the controls they create. The tools are shown in *Figure 107*.

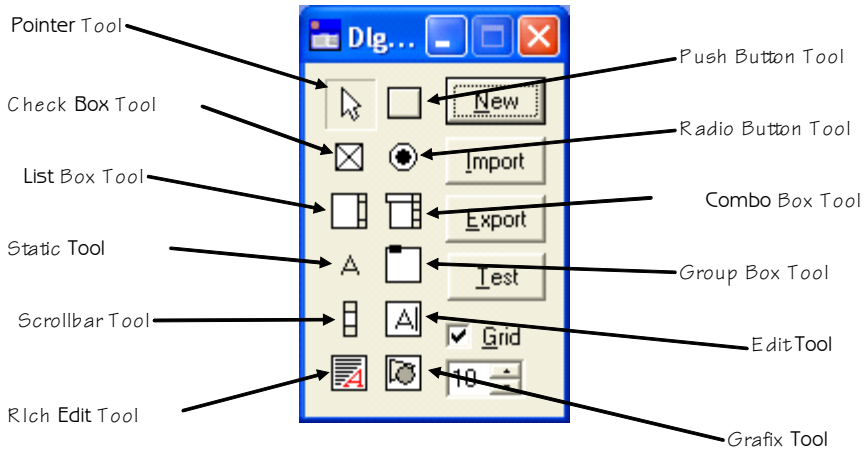


Figure 107 - The toolbox tools

The Pointer Tool

This tool does not create a control, but allows you to select, copy and edit previously created controls. Double-clicking with the left-hand mouse button on any control brings up the style dialog appropriate to that type of control.

Style dialogs usually contain a "Caption" field, for setting the text of a control, and an "ID" field, for setting the identifier for the control. In the style dialog for each control there is also a section for setting the styles of the control. All style dialogs allow you to set the styles, shown in *Table 6*.

Style	Description
ws_border	The created window will have a thin-line border.
ws_caption	The created window will have a title bar.
ws_clipchildren	Clips around the child windows of a control when painting occurs.
ws_clipsiblings	Clips child windows relative to each other when painting occurs.
ws_disabled	Determines if a control is disabled. If it is disabled it will appear greyed.
ws_dlgframe	The border of the created window will be in the style of those used by dialog boxes. Windows created with this style cannot have a title bar.

ws_group	Determines if a control is the start of a group. Group controls group things like radio-buttons together. So if you want two sets of radio-buttons for example, then the first radio in each group would be a group control and the control following the group would also be a group control. They act, in effect, like brackets.
ws_hscroll	The control will have a horizontal scroll bar.
ws_maximize	The created window will be initially maximised.
ws_maximizebox	The created window will have a maximise button.
ws_minimize	The created window will be initially minimised.
ws_minimizebox	The created window will have a minimise button.
ws_sysmenu	The created window will have a System menu box in its title bar.
ws_tabstop	Specifies a control that the <tab> key will stop on
ws_thickframe	The created window will have a sizing border.
ws_visible	Determines if a control is visible on the dialog
ws_vscroll	The control will have a vertical scroll bar.
ws_ex_acceptfiles	Creates a control that will accept drag-and-drop files.
ws_ex_appwindow	Forces a top-level window onto the taskbar when the window is minimised.
ws_ex_clientedge	The border of the control has a sunken edge.
ws_ex_contexthelp	Includes a question mark (?) in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message.
ws_ex_controlparent	Allows the user to navigate among the child windows of the window by using the <tab> key.
ws_ex_dlgmodalframe	Creates a window that has a double border.
ws_ex_left	Creates a window with 'left-aligned' properties. This is the default.

ws_ex_leftscrollbar	Places the vertical scroll bar (if present) to the left of the client area.
ws_ex_ltrreading	The created window will have text displayed using left to right reading order. This is the default.
ws_ex_mdichild	Creates an MDI child window.
ws_ex_noparentnotify	Turns off the WM_PARENTNOTIFY messages.
ws_ex_right	Creates a window with 'right-aligned' properties.
ws_ex_rightscrollbar	Places the vertical scroll bar (if present) to the right of the client area. This is the default.
ws_ex_rtlreading	The created window will have text displayed using right to left reading order.
ws_ex_staticedge	The control will have a three-dimensional border style. This style is intended for items that do not accept user input.
ws_ex_toolwindow	Creates a tool window. A tool window is intended to be used for floating toolbars.
ws_ex_topmost	The created control will be placed above all nontopmost controls and will stay above them.
ws_ex_transparent	The created control will be transparent.
ws_ex_windowedge	The created control will have a border with a raised edge.

Table 6 – Some common generic style dialog control styles

Another style which is common to most, but not all controls, is shown in *Table 7*.

Style	**Description**
ws_border	If a control has this style set then it will be displayed with a single line border.

Table 7 - Generic style dialog border style

The Push Button Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "pushbutton" control. Double clicking on a push button control will show the "Push Button Style" dialog. Push buttons are automatically assigned IDs between 1000 and 1999. In addition to the normal styles, the push button style dialog allows you to set the type of push button, as shown in *Table 8*.

Style	Description
bs_pushbutton	A normal push button
bs_defpushbutton	A push button with a thick border
bs_ownerdraw	A button that is drawn by the user

Table 8 - Push button type styles

The Check Box Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "checkbox" control. Double clicking on a check box control will show the "Check Box Style" dialog. Check boxes are automatically assigned IDs between 3000 and 3999. In addition to the normal styles, the check box style dialog allows you to set the type of check box, as shown in *Table 9*.

Style	Description
bs_3state	The checkbox has three states
bs_auto3state	The checkbox automatically cycles
bs_autocheckbox	The checkbox automatically cycles
bs_checkbox	The checkbox has two states

Table 9 - Check box control type styles

The Radio Button Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "radiobutton" control. Double clicking on a radio button control will show the "Radio Button Style" dialog. Radio buttons are automatically assigned IDs between 2000 and 2999. In addition to the normal styles, the radio button style dialog allows you to set the type of radio button, as shown in *Table 10*.

Style	Description
bs_autoradiobutton	
bs_radiobutton	

Table 10 - Radio button control type styles

The List Box Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "listbox" control. Double clicking on a list box control will show the "Listbox Style" dialog. List boxes are automatically assigned IDs between 4000 and 4999. In addition to the normal styles, the list box style dialog allows you to set the attributes of the list box, as shown in *Table 11*.

Style	Description
lbs_disablenoscroll	
lbs_extendedsel	Allows multiple selection of strings by clicking in combination with the shift-key or control-key
lbs_hasstrings	
lbs_multicolumn	A multi-column listbox
lbs_multiplesel	Allows the selection of multiple strings
lbs_nodata	
lbs_nointegralheight	If this is set the listbox is the exact height it is defined as, otherwise the listbox will be a size such that an integral number of lines of text will be shown
lbs_noredraw	
lbs_nosel	
lbs_notify	The list box notifies the parent whenever the user clicks or double clicks on a string
lbs_ownerdrawfixed	
lbs_ownerdrawvariable	
lbs_sort	Strings are sorted alphabetically
lbs_usetabstops	Tabs are expanded in strings
lbs_wantkeyboardinput	Keyboard input is consumed by the listbox
ws_hscroll	A horizontal scroll bar
ws_vscroll	A vertical scroll bar

Table 11 - List box control attribute styles

The Combo Box Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "combobox" control. Double clicking on a combo box control will show the "Combo Box Style" dialog. Combo boxes are automatically assigned IDs between 5000 and 5999. In addition to the normal styles, the combo box style dialog allows you to set the type of combo box, as shown in *Table 12*.

Style	Description
cbs_simple	A simple combobox - no dropdown
cbs_dropdown	A drop down combo box with editable text field
cbs_dropdownlist	A drop down combobox without an editable text field

Table 12 - Combo box control type styles

From the combo box style dialog you can also set the attributes for the combo box, as shown in *Table 13*.

Style	Description
cbs_autohscroll	
cbs_disablenoscroll	
cbs_hasstrings	
cbs_lowercase	
cbs_nointegralheight	If this is set, the combobox is the exact size of its definition. If this is not set then the combobox will be set to a size that will contain an integral number of lines of text.
cbs_oemconvert	
cbs_ownerdrawfixed	
cbs_ownerdrawvariable	
cbs_sort	Strings are sorted alphabetically
cbs_uppercase	

Table 13 - Combo box control attribute styles

The Static Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "static" control. Double clicking on a static control will show the "Static Style" dialog. Static controls are automatically assigned IDs between 10000 and 10999. In addition to the normal styles, the static style dialog allows you to set the type of static control, as shown in *Table 14*.

Style	Description
ss_bitmap	
ss_blackframe	A black frame
ss_blackrect	A black rectangle
ss_center	Centre justified label
ss_enhmetafile	
ss_etchedframe	
ss_etchedhorz	
ss_etchedvert	
ss_grayframe	A gray frame
ss_grayrect	A gray rectangle
ss_icon	Place holder for an icon
ss_left	Left justified label
ss_leftnowordwrap	No wrapping text
ss_ownerdraw	
ss_right	Right justified label
ss_simple	A Single line of text
ss_useritem	
ss_whiteframe	A white frame
ss_whiterect	A white rectangle
ss_centerimage	
ss_noprefix	

ss_notify

ss_realsizeimage

ss_sunken

ss_endellipsis

ss_noellipsis

ss_pathellipsis

ss_wordellipsis

Table 14 - Static control type styles

The Group Box Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "groupbox". Double clicking on a group box control will show the "Group Box Style" dialog. Group boxes are automatically assigned IDs between 11000 and 11999. Group box controls only have normal styles, as shown in *Table 6*.

The Scrollbar Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a vertical "scrollbar" control. Double clicking on a scrollbar control will show the "ScrollBar Style" dialog; A vertical "scrollbar" control can be converted to a horizontal "scrollbar" control via this dialog. Vertical scrollbars are automatically assigned IDs between 6000 and 6999. Horizontal scrollbars are automatically assigned IDs between 7000 and 7999.

In addition to the normal styles, the vertical scroll bar style dialog allows you to set the alignment of the scroll bar, as shown in *Table 15*.

Style	Description
sbs_free	Free alignment.
sbs_leftalign	Align to the left of the box.
sbs_rightalign	Align to the right of the box.

Table 15 - Vertical scroll bar control alignment styles

In addition to the normal styles, the horizontal scroll bar style dialog allows you to set the alignment of the scroll bar, as shown in *Table 16*.

Style	Description
sbs_free	Free alignment.
sbs_topalign	Align to the top of the box.
sbs_bottomalign	Align to the bottom of the box.

Table 16 - Horizontal scroll bar control alignment styles

The Edit Tool

When this tool is selected, clicking and dragging an area on the scratch window will create an "edit" control. Double clicking on an edit control will show the "Edit Style" dialog. Edit controls are automatically assigned IDs between 8000 and 8999. In addition to the normal styles, the edit style dialog allows you to set the justification of the edit control, as shown in *Table 17*.

Style	Description
es_left	Left justification.
es_right	Right justification.
es_center	Centered justification.
es_nocase	Insensitive Any case characters accepted.
es_uppercase	Characters typed are converted to upper case.
es_lowercase	Characters typed are converted to lower case.

Table 17 - Edit control justification styles

From the edit style dialog you can also set the line style for the edit control, as shown in *Table 18*.

Style	Description
es_multiline	Allow multiple lines.

Table 18 - Edit control line styles

You can add scrollbars to the edit control by setting one of the styles shown in *Table 19*.

Style	Description
ws_hscroll	A horizontal scroll bar.
ws_vscroll	A vertical scroll bar.

Table 19 - Edit control scroll bar styles

Setting the type of scrolling allowed in the edit control can be done by setting one of the styles shown in *Table 20*.

Style	Description
es_autohscroll	Automatically scroll the text being typed horizontally.
es_autovscroll	Automatically scroll the text being typed vertically.

Table 20 - Edit control auto scroll styles

There are a variety of other styles that you can set for an edit control, these are shown in *Table 21*.

Style	Description
es_password	Display "*" instead of typed characters. Not in a multiple line editor.
es_nohidesel	Keep displaying the selected text, even when not in focus. The default is to turn the selection off when the editor is not in focus.
es_readonly	Cannot edit the text.
es_wantreturn	The editor wants the return key not its parent dialog.
es_number	
es_oemconvert	

Table 21 - other Edit control styles

The Rich Edit Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "rich edit" control. Double clicking on a rich edit control will show the "Rich Edit Style" dialog. Rich Edit controls are automatically assigned IDs between 9000 and 9999. Rich Edit controls use many of the same styles as Edit controls; a variety of other styles that you can set for a rich edit control are shown in Table 22.

Style	Description
es_disablenoscroll	
es_noime	
es_nooledragdrop	
es_savesel	
es_selectionbar	
es_selfime	
es_sunken	
es_vertical	

Table 22 – additional rich edit control styles

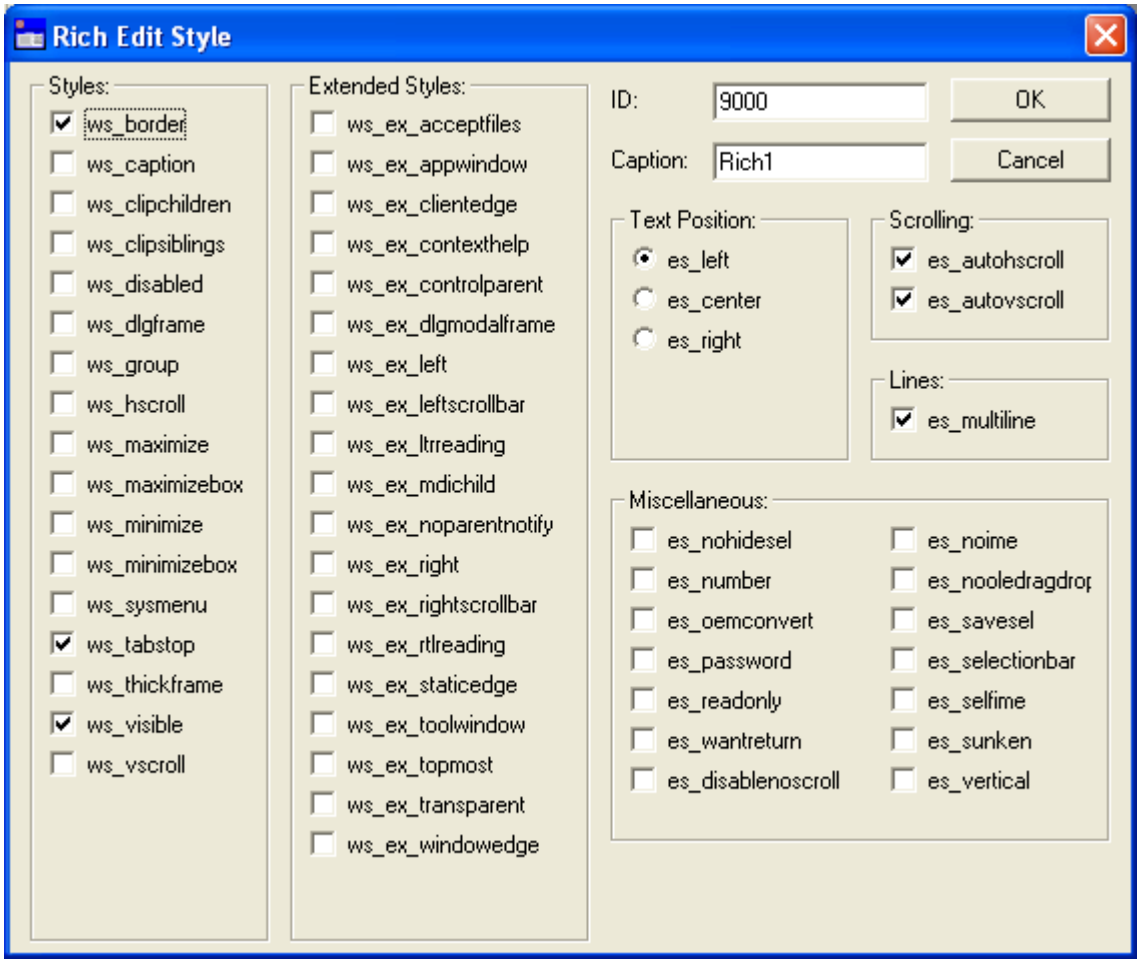


Figure 108 - The Rich Edit Style dialog

The Grafix Tool

When this tool is selected, clicking and dragging an area on the scratch window will create a "grafix" control. Double clicking on a grafix control will show the "Grafix Style" dialog. Grafix controls are automatically assigned IDs between 10000 and 10999. In addition to the normal styles, the grafix style dialog allows you to add a particular type of scroll bar to the grafix control, as shown in *Table 23*.

Style	Description
ws_hscroll	A horizontal scroll bar.
ws_vscroll	A vertical scroll bar.

Table 23 - scroll bar control styles

The Scratch Window Menu

If you click with the right hand mouse button over the scratch window, the scratch window menu will appear as shown in Figure 109.

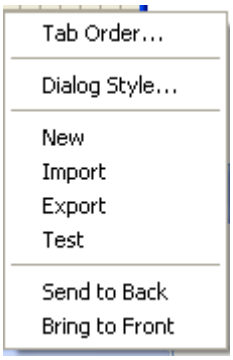


Figure 109 - The scratch window menu

The "Tab Order..." menu option allows you to alter the tab ordering of the controls.

The "Dialog Style..." menu option brings up the Dialog Style dialog; this menu option performs the same function as double-clicking on a control-free part of the scratch window.

The "New", "Import", "Export" and "Test" menu options perform the same functions as the four buttons within the Dialog Editor dialog.

The "Send to Back" menu option sends the currently selected widget to the bottom (i.e. behind all other widgets). This option has no effect if no widget is currently selected. This is an editing feature in the Dialog Editor to move things like large static widgets to the back to edit the underlying widgets. The actual ordering of the generated Prolog code is controlled by the tab-order.

The "Bring to Front" menu option brings the currently selected widget to the top (i.e. in front of all other widgets). This option has no effect if no widget is currently selected. The actual ordering of the generated Prolog code is controlled by the tab-order.

The Ordering of Controls

The ordering of the controls in the created dialog is determined by the order in which they are created. Newly created controls will be added to the end of the control order. However, there is a way to alter this ordering. If you click with the right hand mouse button over the scratch window, a menu will appear, the first item of which is "Tab Order..."; if you click on this menu option, the "Edit Tab Order" dialog will appear as shown in Figure 110.

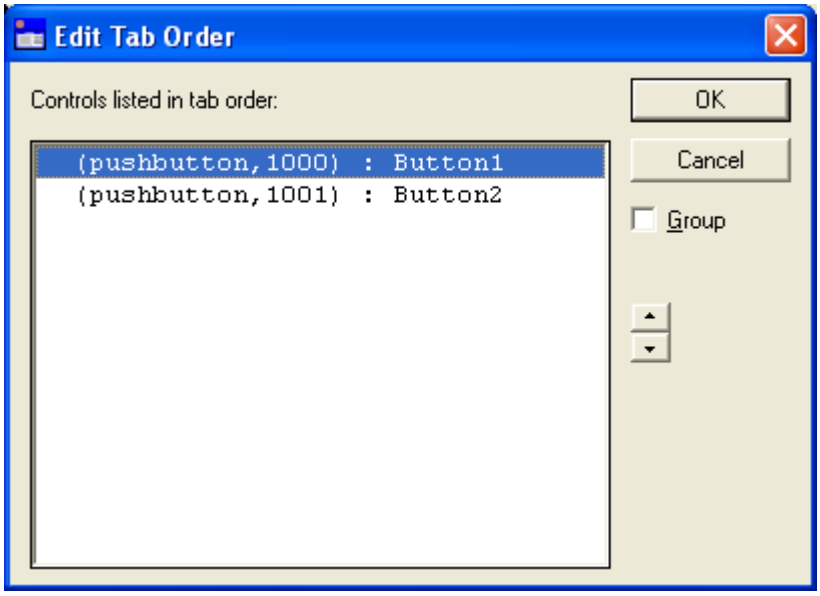


Figure 110 - The Edit Tab Order dialog

The tab position of an individual control may be promoted or demoted by highlighting the control in the dialog and clicking on either the up arrow or the down arrow one or more times until the control is in the required position.

Radio Button Groups

By default, all radio buttons in a single dialog created using the Dialog Editor will be part of the same group, regardless of what other types of control were created in between. To create two or more groups of radio buttons, you need to set the "Group" style for the first radio button in each group and in any control following the group. *Figure 111* shows a dialog with two groups of radio buttons.

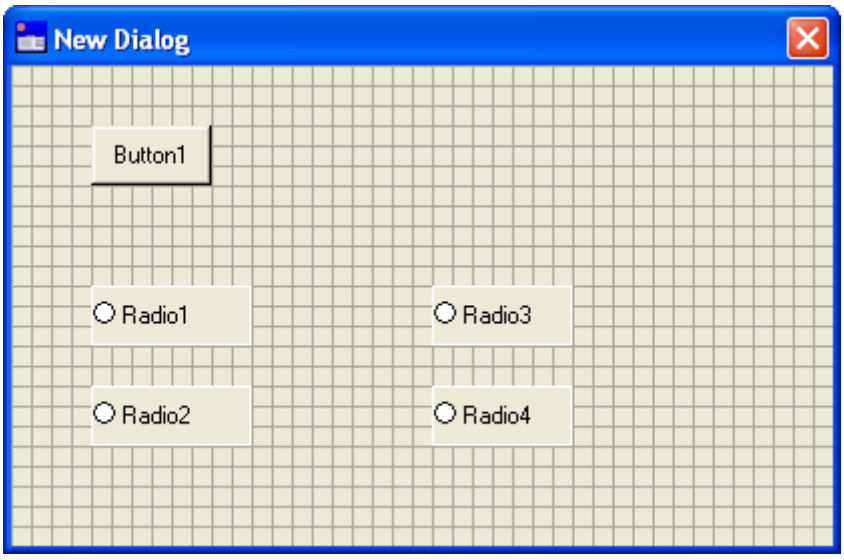


Figure 111 - Dialog with two groups of radio buttons

The exported code generated by the above would be something like:

```
new_dialog :-
   _S1 = [ws_caption,ws_sysmenu,dlg_ownedbyprolog],
   _S2 = [ws_child,ws_visible,ws_tabstop,bs_pushbutton],
   _S3 =
[ws_child,ws_group,ws_visible,bs_autoradiobutton,bs_text,bs_left,bs_vcenter
],
   _S4 = [ws_child,ws_visible,bs_autoradiobutton],
   wdcreate(  new_dialog,       `New Dialog`,      269,  74, 416, 272, _S1
),
   wccreate( (new_dialog,1000), button, `Button1`,  40,  30,  60,  30, _S2
),
   wccreate( (new_dialog,2000), button, `Radio1`,   40, 110,  80,  30, _S3
),
   wccreate( (new_dialog,2001), button, `Radio2`,   40, 160,  80,  30, _S4
),
   wccreate( (new_dialog,2002), button, `Radio3`,  210, 110,  70,  30, _S3
),
    wccreate( (new_dialog,2003), button, `Radio4`,  210, 160,  70,  30,
 _S4 ).
```

In order to have "Radio1" and "Radio2" in one group and "Radio3" and "Radio4" in another group, you need to set the "Group" style for each of the "Radio1","Radio3" and "Button1" controls. Another way to do this in the Edit Tab Order dialog, as shown in *Figure 112*.

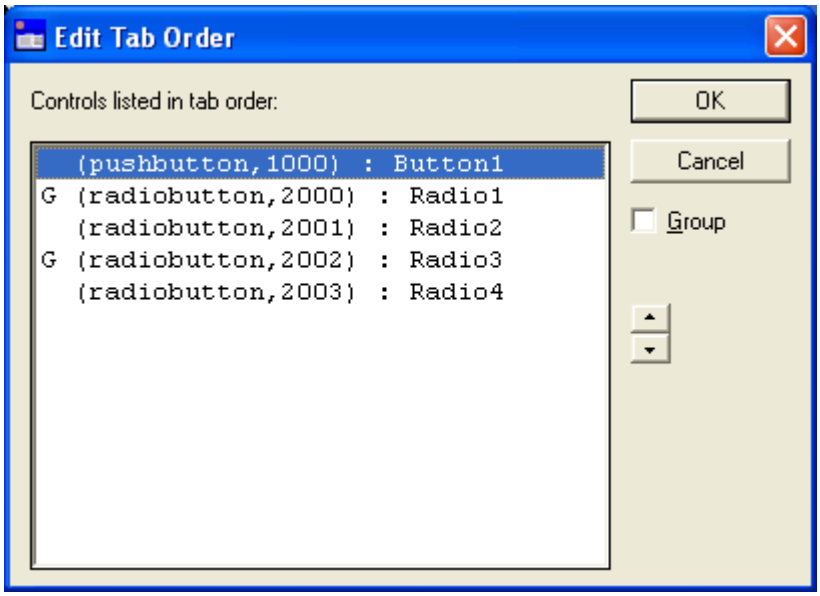


Figure 112 - Setting the 'Group' style

Getting Information Into and Out of a Generic Dialog

You are probably going to want to create a generic dialog and then customise it for a particular use. The following program shows how to customise an existing dialog without modifying the code that came out of the dialog editor. It shows how to:

- Change the text of a button, checkbox, radio, static control, group box, edit control or rich edit control.
- Add entries to a listbox or combobox.
- Set the range and default value for a scrollbar.
- Add a user-defined window handler.
- Repaint a graphics control via a user-defined window handler.

It also shows how to get the information back that the user entered into the dialog; the list it returns contains the following entries:

- ButtonText – the text of the button
- CheckText – the text alongside the checkbox
- CheckState – the state of the checkbox
- SelectedRadioEntry – the text for the selected radio entry
- SelectedListboxEntries – a list of selected listbox entries
- ComboboxText – the selected text in a combobox
- StaticText – the text of a static control
- GroupText – the text of a group box control
- ScrollbarValue – the value of a scrollbar
- EditText – the text in an edit control
- RichText – the text in a rich edit control

```
test( Results ) :-
  new_dialog,
  wtext( (new_dialog,1000), `OK` ),
  wtext( (new_dialog,3000), `On` ),
  wtext( (new_dialog,2000), `On` ),
  wlstadd( (new_dialog,4000), -1, `The`, 0 ),
  wlstadd( (new_dialog,4000), -1, `Quick`, 0 ),
  wlstadd( (new_dialog,4000), -1, `Brown`, 0 ),
  wlstadd( (new_dialog,4000), -1, `Fox`, 0 ),
  wcmbadd( (new_dialog,5000), -1, `Bill`, 0 ),
  wtext( (new_dialog,11000), `Some Text` ),
  wtext( (new_dialog,12000), `My Group` ),
  wrange( (new_dialog,6000), 0, 1, 100 ),
  wthumb( (new_dialog,6000), 0, 50 ),
  wtext( (new_dialog,8000), `This is some text` ),
  wtext( (new_dialog,9000), `This is some text` ),
  window_handler( new_dialog, new_dialog_handler ),
  call_dialog( new_dialog, Results ).

new_dialog :-
  _S1 = [ws_sysmenu,ws_popup,ws_caption,dlg_ownedbyprolog],
  _S2 = [ws_child,ws_visible,ws_tabstop,bs_pushbutton],
  _S3 = [ws_child,ws_visible,ws_tabstop,bs_autocheckbox],
  _S4 = [ws_child,ws_visible,bs_autoradiobutton],
  _S5 = [ws_child,ws_visible,ws_border,ws_tabstop,ws_vscroll,lbs_sort],
  _S6 = [ws_child,ws_visible,ws_tabstop,cbs_dropdown],
  _S7 = [ws_child,ws_visible,ss_left],
  _S8 = [ws_child,ws_visible,bs_groupbox],
  _S9 = [ws_child,ws_visible,sbs_vert],
  _S10 = [ws_child,ws_visible,ws_tabstop,ws_border,es_left,es_multiline,
es_autovscroll,es_autohscroll],
  _S11 = [ws_child,ws_border,ws_visible],
  wdcreate( new_dialog,      `New Dialog`,      198, 36, 176, 352, _S1 ),
  wccreate( (new_dialog,1000), button,   `Button1`, 90, 10, 70, 20, _S2 ),
  wccreate( (new_dialog,3000), button,   `Check1`,  10, 40, 70, 20, _S3 ),
  wccreate( (new_dialog,2000), button,   `Radio1`,  90, 40, 70, 20, _S4 ),
  wccreate( (new_dialog,4000), listbox,  `List1`,   10, 70, 70, 50, _S5 ),
  wccreate( (new_dialog,5000), combobox, `Combo1`,  90, 70, 70, 50, _S6 ),
  wccreate( (new_dialog,11000), static,  `Static1`, 10, 130, 70, 20, _S7 ),
  wccreate( (new_dialog,12000), button,  `Group1`,  90, 120, 70, 30, _S8 ),
  wccreate( (new_dialog,6000), scrollbar, `Scroll1`, 30, 160, 20, 70, _S9 ),
  wccreate( (new_dialog,8000), edit,     `Edit1`,   90, 160, 70, 70, _S10 ),
  wccreate( (new_dialog,9000), rich,     `Rich1`,   10, 240, 70, 70, _S10 ),
  wccreate( (new_dialog,10000), grafix,  `Grafix1`, 90, 240, 70, 70, _S11 ).

new_dialog_handler( (new_dialog,10000), msg_paint, _, _ ) :-
  gfx_begin((new_dialog,10000)),
  gfx(ellipse(10,10,70,70)),
  gfx_end((new_dialog,10000)).
```

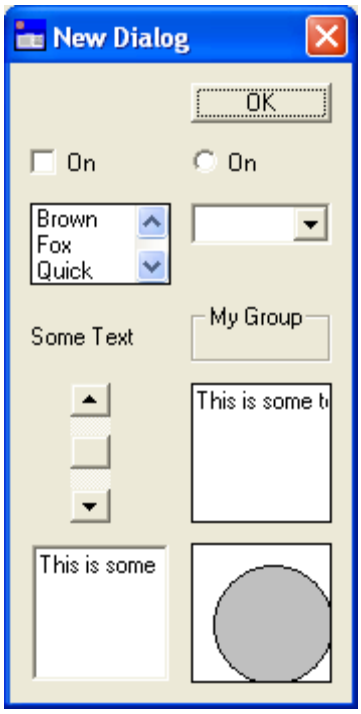
```
new_dialog_handler( (new_dialog,1000), msg_button, _, Results ) :-
  wtext( (new_dialog,1000), ButtonText ),
  wtext( (new_dialog,3000), CheckText ),
  wbtnsel( (new_dialog,3000), CheckState ),
  ( findall( SelectedRadioEntry,
          ( integer_bound( 2000, RadioEntryPos, 2000 ),
            wbtnsel( (new_dialog,RadioEntryPos), RadioState ),
            (  RadioState = 1
            -> wtext( (new_dialog,RadioEntryPos), SelectedRadioEntry )
            ;  fail
            )
          ),
          [SelectedRadioEntry]
        )
  ; SelectedRadioEntry = ``
  ),
  findall( SelectedListboxEntry,
         ( integer_bound( 0, Pos, 3 ),
           wlstsel( (new_dialog,4000), Pos, State ),
           (  State = 1
           -> wlbxget( (new_dialog,4000), Pos, SelectedListboxEntry )
           ;  fail
           )
         ),
         SelectedListboxEntries
       ),
  wtext( (new_dialog,5000), ComboboxText ),
  wtext( (new_dialog,11000), StaticText ),
  wtext( (new_dialog,12000), GroupText ),
  wthumb( (new_dialog,6000), 0, ScrollbarValue ),
  wtext( (new_dialog,8000), EditText ),
  wtext( (new_dialog,9000), RichText ),
  Results = [ ButtonText,
            CheckText,CheckState,
            SelectedRadioEntry,
            SelectedListboxEntries,
            ComboboxText,
            StaticText,
            GroupText,
            ScrollbarValue,
            EditText,
            RichText
          ].

new_dialog_handler( A, B, C, D ) :-
  window_handler( A, B, C, D ).
```

After typing in and compiling the above program, execute the following from the *WIN-PROLOG* command line:

```
?- test( Results ). <enter>
```

The following dialog will appear:



Clicking on the dialog's OK button will cause Results to be instantiated:

```
Results = [`OK`,`On`,0,``,[],``,`Some Text`,`My Group`,50,`This is some
text`,`This is some text`]
```

Try executing *test(Results)* again, changing the entered information and submitting.

Chapter 18 - The Call-Graph Plug-in

The call-graph plug-in allows you to view the outline of a Prolog program, by showing the graph of predicate calls. The call-graph is bi-directional – it will show predicates which call a particular predicate and predicates called by the same particular predicate. This chapter tells you how to run the call-graph, and how to find your way around the call-graph to get information on Prolog programs.

Running the Call Graph

When you run *WIN-PROLOG* an additional "Run/Call Graph..." menu option is available. The call-graph is run by selecting this option, although it is disabled if no currently defined predicates that it can view are available. Start up *WIN-PROLOG* now, if you haven't already done so. Select the "File/Open..." menu option and go into the *EXAMPLES* folder, then select the *MEALS.PL* example and open that. Now select the "Run/Compile" menu option to compile the file. Now select the "Run/Call Graph..." option. You will see the "Call Graph: Root" dialog, as shown in *Figure 113*.

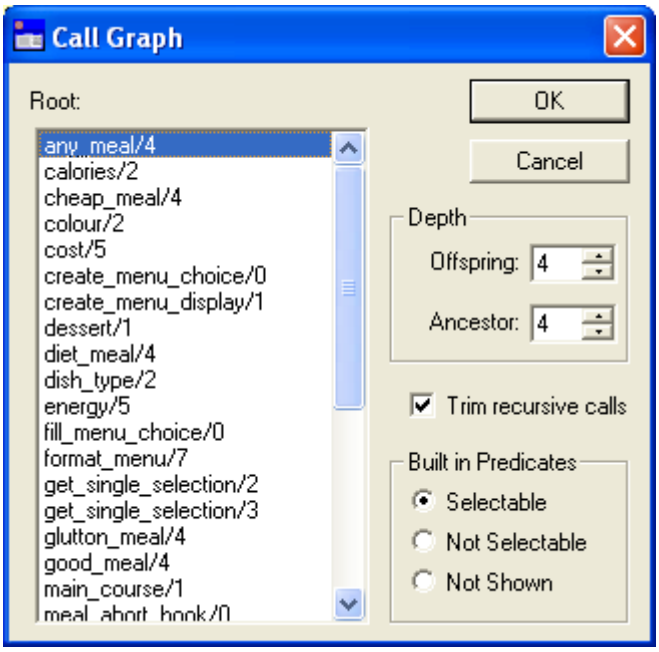


Figure 113 - the "Call Graph : Root" dialog

This dialog shows a list of currently defined predicates (i.e. those in MEALS.PL) that can be viewed using the call graph plug-in.

Displaying the Call Graph

Now select the *glutton_meal/4* predicate from the listbox, select the Built in Predicates' "Not Shown" radio option and then click on "OK". The call graph for the *glutton_meal/4* predicate will then be displayed (you may need to re-size and re-position the window view in order to show the whole of the graph).

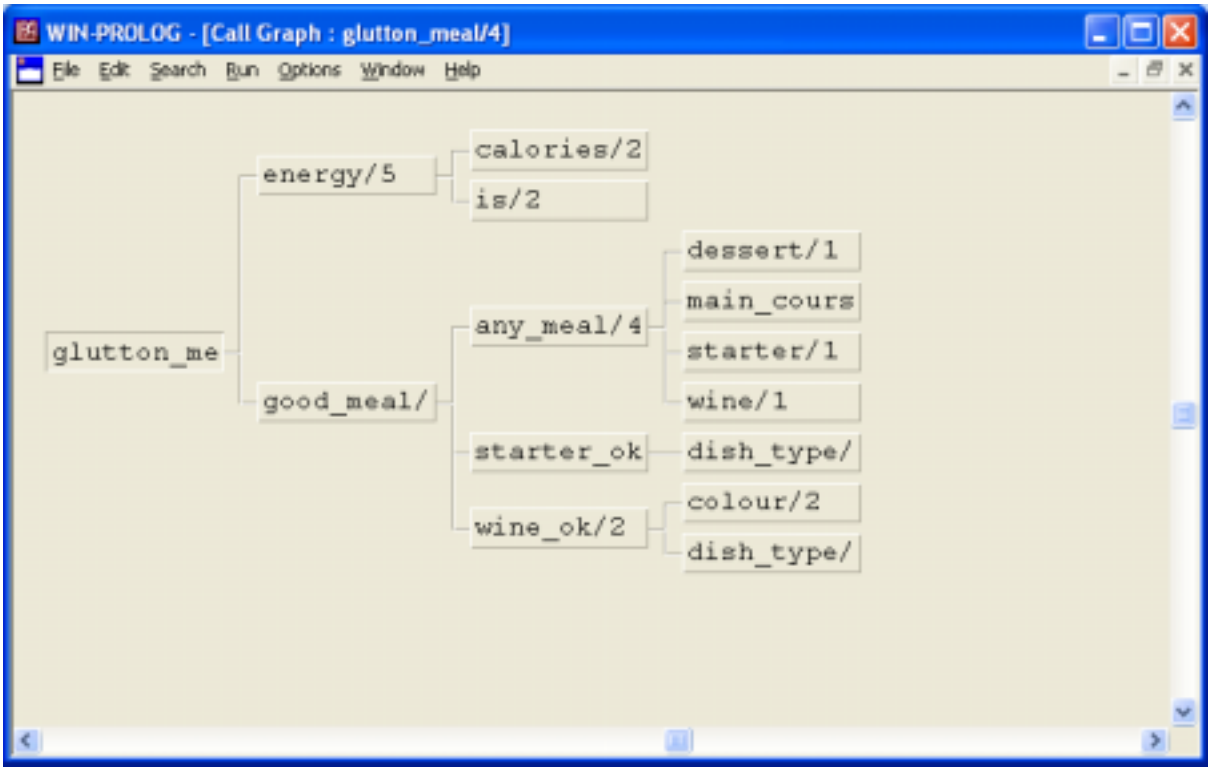


Figure 114 - the glutton_meal/4 call graph

The Call Graph

The representation that you see in the "Call Graph" window shows the "root" predicate on the left-hand side of the screen linked by lines to the predicates that it calls shown in the next column. These predicates in turn are linked by lines to the predicates that they call shown in the next column. At the moment these four levels are all that you will see; if you need to see more, a greater depth can be set in the "Call Graph : Root" dialog.

If the "Selectable" or "Not Selectable" radio options in the "Built in Predicates" section had been selected, built-in predicates such as *>/2* would also be displayed. With "Not Selectable" selected, they would be shown as if they were sunken into the surface; this is because such built-in predicates do not allow you to navigate further into the call graph.

Moving the Call Graph

The entire call graph can be moved within the window by clicking and holding down the left mouse button whilst over a non-selectable predicate and then simply moving the mouse.

Navigating the Call Graph

To navigate around the call graph you simply select the predicate you want to appear as the new "root" of the graph with the left mouse button. To demonstrate this, click on the *good_meal/4* predicate. The call graph view will then be replaced with the one shown in *Figure 115*.

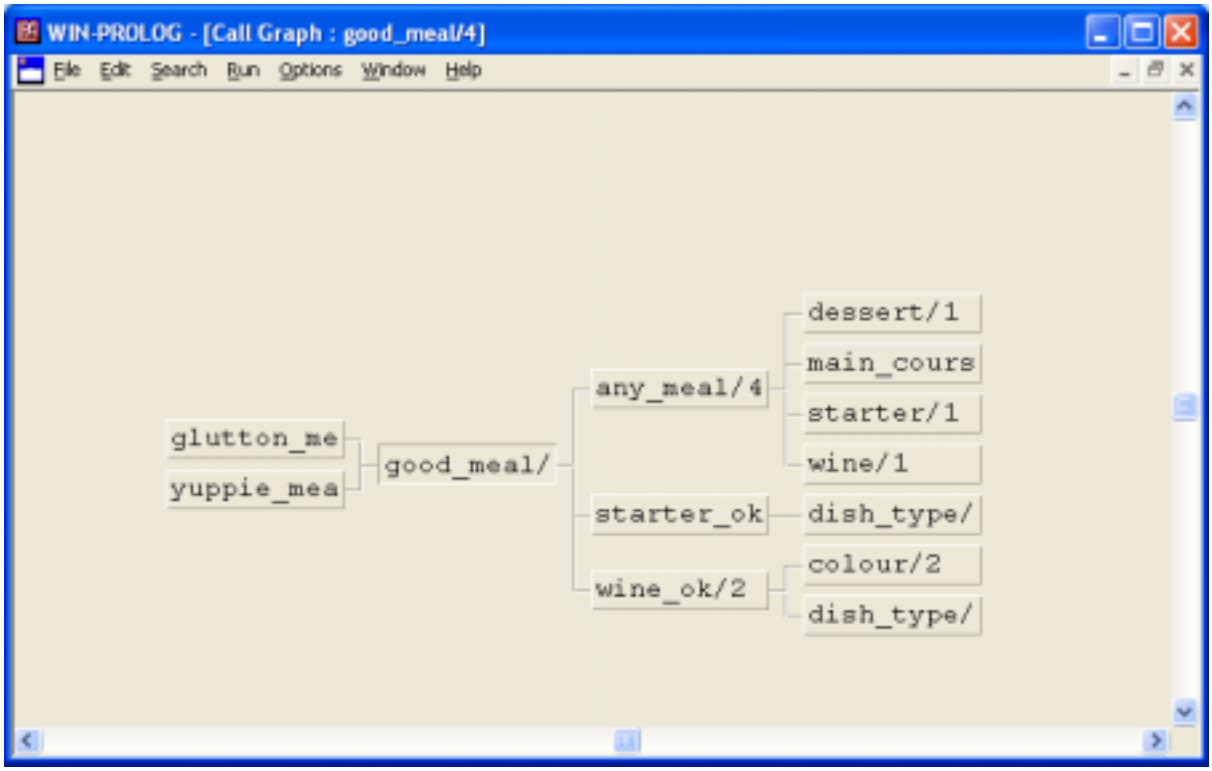


Figure 115 - the good_meal/4 call graph

Now select the *wine/1* predicate and this will become the new "root", as shown in *Figure 116*.

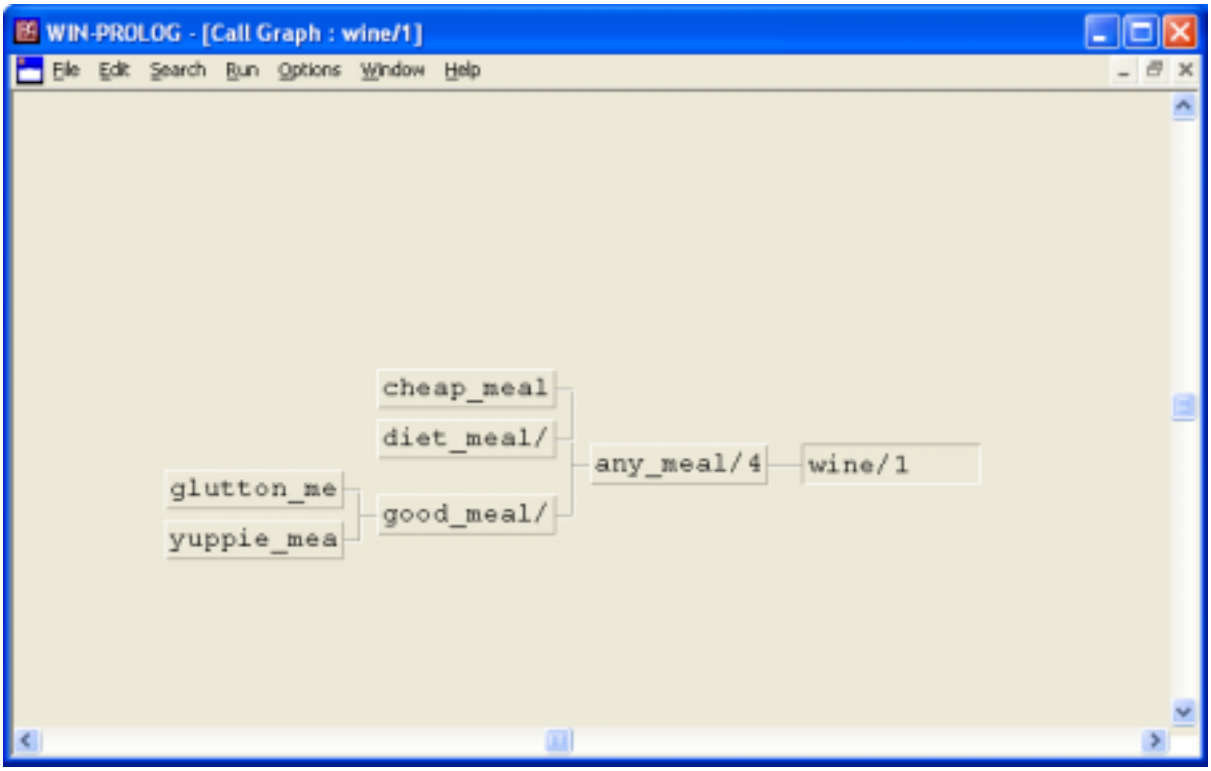


Figure 116 - the wine/1 call graph

After selecting this predicate you will see that there is no way further forward, indicating that this predicate makes no calls and is therefore a "fact". The only way to go now is back up the call tree.

Getting Predicate Information

To get information on one of the predicates displayed you click on the predicate, this time with the right mouse button. Select the *good_meal/4* predicate with the right mouse button and a small pop-up information window will appear.

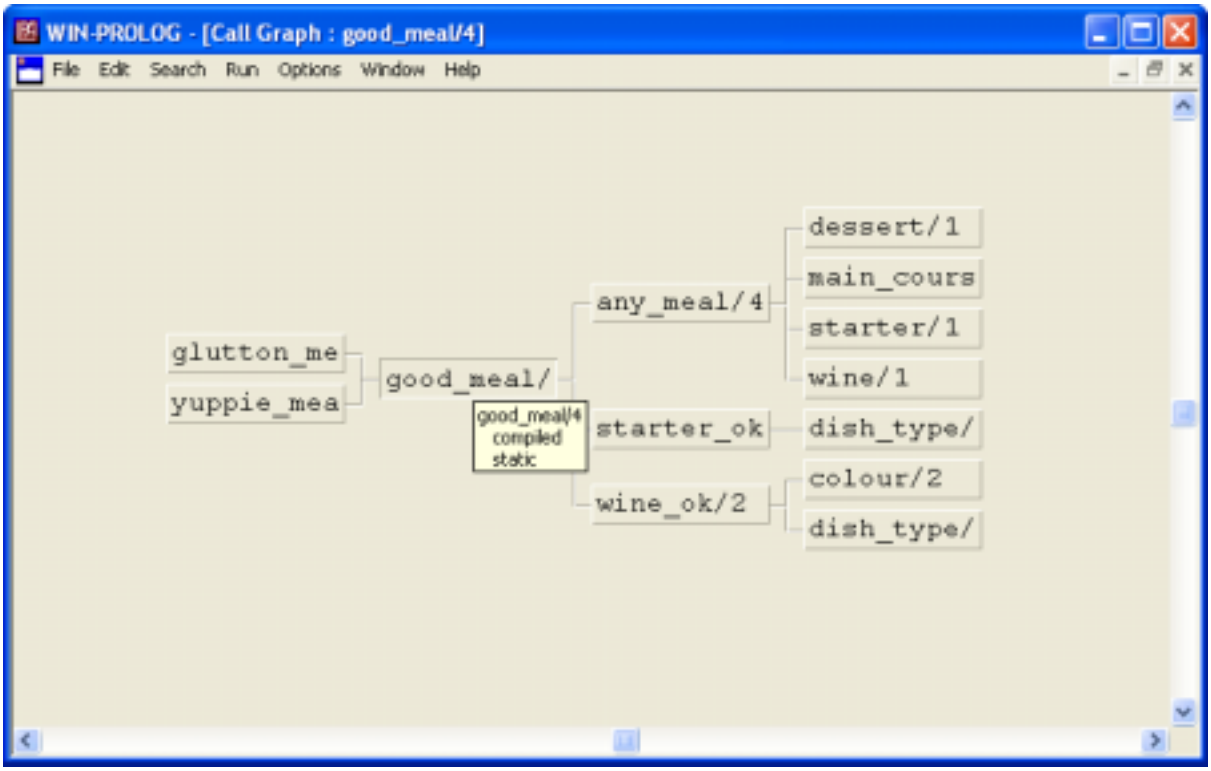


Figure 117 - good_meal/4 predicate information

This information shows that the predicate is compiled and static. This means that it has been compiled, but not optimised, and that it is not dynamic and therefore cannot be modified using *assert/1* and *retract/1*.

If you were to click on a built-in predicate, for example, *>/2*, with the right mouse button you would see that it is a built-in optimised predicate, which is why it cannot be selected as the root.

Showing More Levels

We've shown how the call graph can be navigated around using the mouse, but what if we want to see the further levels of the call graph all on the screen at the same time. You can do this by changing the "Depth" of display in the "Call Graph : Root" dialog. Select the "Run/Call Graph..." menu option and increase the "Offspring Depth" or "Ancestor Depth" by 1 then click on "OK". This time more of the call graph will be shown on the screen (again you will need to re-size the window to show all the call graph).

You can still "focus" in on parts of the call graph by clicking on a selectable predicate with the left mouse button.

The Call Graph Menu

Clicking with the right mouse button over a call graph window will bring up the call graph menu.

The 'Colours...' option leads to the Call Graph Colours dialog where the individual colours used for the call graph can be changed.

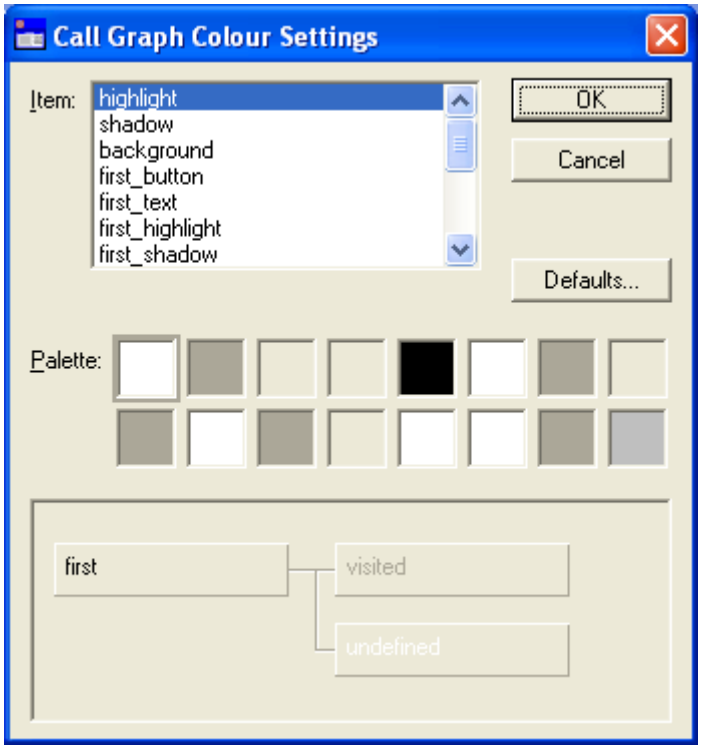
The 'Font...' option leads to the *WIN-PROLOG* Font dialog where the font used by the call graph can be changed.

The 'Call Graph...' option leads to the Call Graph: Root dialog mentioned earlier.

The 'Settings...' option leads to the Call Graph Settings dialog.

The Call Graph Colours Dialog

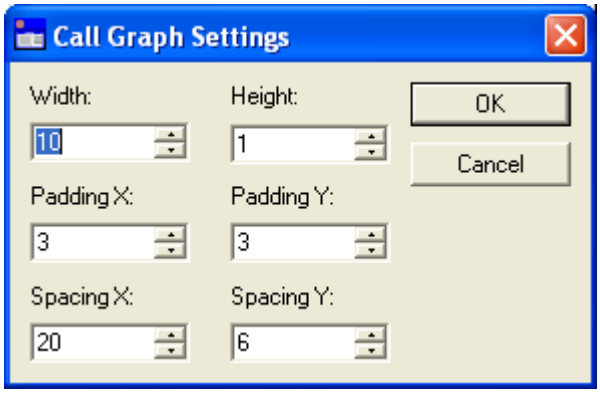
The Call Graph Colours dialog allows the colours of the Call Graph to be altered.



Select the item that you wish to change the colour for from the Item: listbox; a 'box' will automatically become selected in the Palette: section. Click, with the right button, on the selected box in the Palette: section to go to the Change Colour dialog.

The Call Graph Settings Dialog

The Call Graph Settings dialog allows you to alter the size, padding or spacing of a Call Graph predicate's rectangle box 'on the fly'.



Changing the Exclusion Settings

By selecting either the 'Not Selectable' or 'Not Shown' radio option in the 'Built in Predicates' section of the Call Graph dialog, you can disable or exclude built-in system predicates from the call graph.

Exporting the Call Graph

The call graph can be exported as an Enhanced MetaFile (.WMF) via the File\Export... menu option.

Printing the Call Graph

The call graph can be printed out via the File\Print... menu option.

Chapter 19 - The Cross-Referencer Plug-in

This chapter tells you how to run the cross-referencer and what type of information it returns.

Running the Cross Referencer

The cross-referencer plug-in generates information about the Prolog calls and definitions found in all the ***WIN-PROLOG*** program windows currently open (but not necessarily compiled).

Prior to selecting this option we need some source code to actually run the cross referencer on! Again we will use the *MEALS.PL* example, select the "File/Open..." menu option and go into the *EXAMPLES* folder, then select *MEALS.PL* and click on "Open" to open that file.

Having opened the *MEALS.PL* example we can now show the typical output from the cross-referencer. Select the "Run/Cross Reference..." menu option; the Cross Reference dialog will be displayed asking you to select all the program edit windows required to be cross referenced.

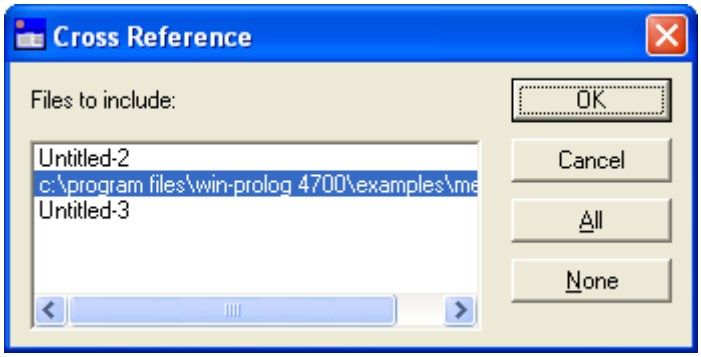


Figure 118 – The Cross Reference dialog

Just click on "OK" as only MEALS.PL will be present and selected. A "Cross Reference" window will then be generated containing the text of an analysis of the *MEALS.PL* example. The analysis is divided into several sections. We will describe each of these in turn.

The Cross Reference Header

The header appears at the top of the "Cross Reference" window this simply shows the time and date of the cross reference. A typical header is as follows:

```
Cross Reference at 15:02:02 on Thu 08 Aug 2002
```

Program Windows in Cross Reference

The next section gives the names and locations of the files involved in the current cross-reference. In our example this will be the single file *MEALS.PL*:

```
c:\program files\win-prolog 4900\examples\meals.pl
```

Should any file involved in the current cross-reference contain an error, the relevant error number will precede its names and location:

```
**42** c:\program files\win-prolog 4900\examples\meals.pl
```

Built-in Predicates Called Directly

The next section gives the names and arities of all the built-in predicates called in the files involved in the cross-reference. As you can see the *MEALS.PL* example calls a few built-in predicates:

```
!/0
->/2
;/2
</2
=/2
=../2
>/2
\+/1
abort/0
atom_string/2
call/1
call_dialog/2
cat/3
fail/0
fwrite/4
halt/0
halt/1
is/2
msgbox/4
nl/0
not/1
putb/1
repeat/0
show_dialog/1
sttbox/2
wait/1
wccreate/8
wclose/1
wdcreate/7
wfocus/1
wfont/2
wlbxadd/3
wlbxfnd/4
wlbxget/3
wlbxsel/3
wndhdl/2
write/1
wshow/2
wtext/2
~>/2
```

Predicates both Defined and Called

The next section gives the names and arities of all the predicates defined in the files involved in the cross-reference that are also called in those files. The *MEALS.PL* defines and calls the following predicates:

```
any_meal/4
calories/2
colour/2
cost/5
create_menu_choice/0
create_menu_display/1
dessert/1
dish_type/2
energy/5
fill_menu_choice/0
format_menu/7
get_single_selection/2
get_single_selection/3
good_meal/4
main_course/1
menu/0
menu/1
menu_choice/1
menu_display/2
price/2
starter/1
starter_ok/2
wait_dialog/1
wine/1
wine_ok/2
```

Predicates Defined, but Not Called

The next section gives the names and arities of all the predicates defined in the files involved in the cross-reference that are not called in those files. These include the top-level goals to run the program, handlers for dialogs, system hooks and programs that are called in a meta-logical way. The *MEALS.PL* produces the following output:

```
cheap_meal/4
diet_meal/4
glutton_meal/4
meal_abort_hook/0
meal_main_hook/0
yuppie_meal/4
```

The predicates *cheap_meal/4*, *diet_meal/4*, *glutton_meal/4* and *yuppie_meal/4* are all examples of meta-called predicates. The predicates *meal_abort_hook/0* and *meal_main_hook/0* are examples of system hooks (used when running *MEALS.PL* as a stand-alone application).

Predicates Called, but Not Defined

This is a section that will not appear initially when doing a cross reference of the *MEALS.PL* example. Predicates appear in this section when they are called but not actually defined. If we were to go to the predicate *menu/1* in the *MEALS.PL* example and change its definition to the following:

```
menu :-
  foo,
  menu_choice( Meal ),
  menu( Meal ).
```

A subsequent cross reference will report the following predicate in this section:

foo/0

Predicates with Unknown Meta Calls

The final section of the cross-reference deals with predicates that contain meta-calls that do not become explicit until run-time. The *menu/1* predicate is an example of this:

```
menu/1
```

Chapter 20 - Flex for Windows

This chapter describes the use of flex, an expert system toolkit, with ***WIN-PROLOG***. Flex for Windows is a ***WIN-PROLOG*** toolkit, and as such, inherits many of the features of this extensive programming language. These include access to a large number of Windows Graphical User Interface (GUI) functions, allowing you to create polished flex for Windows applications.

What's in flex for Windows?

Flex provides the capabilities of an expert system shell combined with the potential of the fully developed programming language Prolog. Flex gives you the ability to define expert systems using "frames", forward and backward chaining "rules" and built-in "questions".

The Windows version adds to flex an easy-to-use pull down menu programming environment, multiple edit windows, automatic syntax colouring, source level debugging, comprehensive text search and replace facilities and various browsing facilities. All of the GUI features used by the environment, and more beside, are directly available to flex via Prolog programs, allowing customised environments to be built and shipped as part of an application.

Please note that flex is written in ***WIN-PROLOG*** and runs within the ***WIN-PROLOG*** environment. You can create, edit and run flex and Prolog programs at the same time within the ***WIN-PROLOG*** environment, and even get them to call each other if you so wish.

About this Chapter

In addition to the standard ***WIN-PROLOG*** environment, flex for Windows includes a "flex" menu to provide easy access to information on your flex programs. This chapter describes how to use the flex specific aspects of the environment, including the debugging of forward chaining, browsing frame hierarchies (both by dialog and graphically) and setting the inheritance algorithm for frame hierarchies. Throughout this guide, examples show you how to perform basic actions; these examples have been kept as simple as possible: you will find more detailed information about the system itself in the *Flex Tutorial* and the *Flex Reference* manual.

Running Flex for Windows

Assuming that you have successfully installed ***WIN-PROLOG*** and flex, you will have entries on the Windows Start Menu to launch ***WIN-PROLOG*** or ***WIN-PROLOG*** + flex. You can run flex by selecting the "Programs\WIN-PROLOG 4900\flex 4900" menu item on the Start Menu.

As Flex is merely an extension to *WIN-PROLOG*, you can, from within *WIN-PROLOG* itself, load Flex by executing the following command at the Console window prompt:

?- **ensure_loaded(system(flexenv)).** *<enter>*

You will know that flex has been loaded by the appearance of a new Flex menu on the menu bar.

The flex menu item on the Start Menu is, in fact, a shortcut containing the "ensure_loaded(system(flexenv))." text in its command line.

Creating Your Own Flex Shortcut

Should you need to create a shortcut for flex yourself, the procedure is as follows: Go to the folder in which you've installed *WIN-PROLOG* and then create a shortcut to *WIN-PROLOG*. This is done by clicking on the PRO386W.EXE (or RUN386W.EXE) application with the right mouse button and selecting "Create Shortcut" from the menu subsequently displayed. Once you've created a shortcut you can then drag it to the location you require, e.g. the "desktop" or "Start" menu. The command line of the shortcut can be edited by clicking on it with the right mouse button and selecting "Properties" from the menu subsequently displayed. The "Shortcut" tab in the "Properties" dialog contains a "Target" edit field that is equivalent to the *WIN-PROLOG* command line. Enter the following command line (assuming "C:\Program Files\Win-Prolog 4900" is the home folder for your copy of *WIN-PROLOG*) all on one line:

"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(system(flexenv)).

Associating .KSL Files With Flex

Should you wish to associate .KSL files with flex, and launch flex whenever you double click upon one, you may like to read *Appendix D - Associating* WIN-*PROLOG Files With The Application*.

Flex Help File

A flex help file, Flexhelp.hlp, is provided in the supplied Docs directory.

Chapter 21 – Flex and Existing Menus

This chapter describes the modifications made by flex to ***WIN-PROLOG***'s menus, covering the creating, opening, editing and saving of flex files, and their relationship to edit windows. You should refer to ***Chapter 5 - the File Menu*** on page 66 as some of the options on the "File" menu remain the same. **Note that this chapter only applies if you have purchased the flex for Windows toolkit.**

The File/New... Option

The "File/New..." option allows you to create a new, untitled program edit window. Edit windows are simply work areas in which you use the standard Windows editing features to write, modify and otherwise maintain flex source code.

Start up flex for Windows, and when you see the ***WIN-PROLOG*** application window, select the "File/New..." option. When you click the mouse or press *<enter>*, a dialog will appear asking you to select the class of edit window required; select "flex Source Files (*.ksl)" and press "OK", whereupon an edit window will be created with the name "Untitled-<n>".

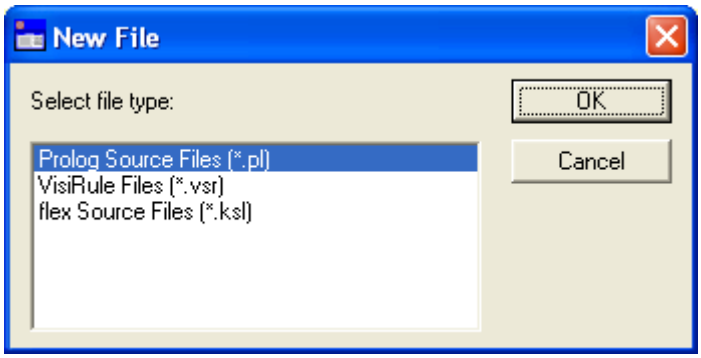


Figure 119 – The New File dialog

Try entering the following simple flex KSL example:

```
relation likes(brian, ADrink)                    <enter>
        if  ADrink is some drink                 <enter>
        and the type of ADrink is alcoholic .    <enter>

frame drink                                      <enter>
        default type is non_alcoholic .          <enter>

instance water is a drink .                      <enter>

instance beer is a drink                         <enter>
        type is alcoholic .                      <enter>

instance wine is a drink                         <enter>
        type is alcoholic .                      <enter>
```

In this, and all other examples, type the text, and press the named key for anything bracketed in <*italics*>.

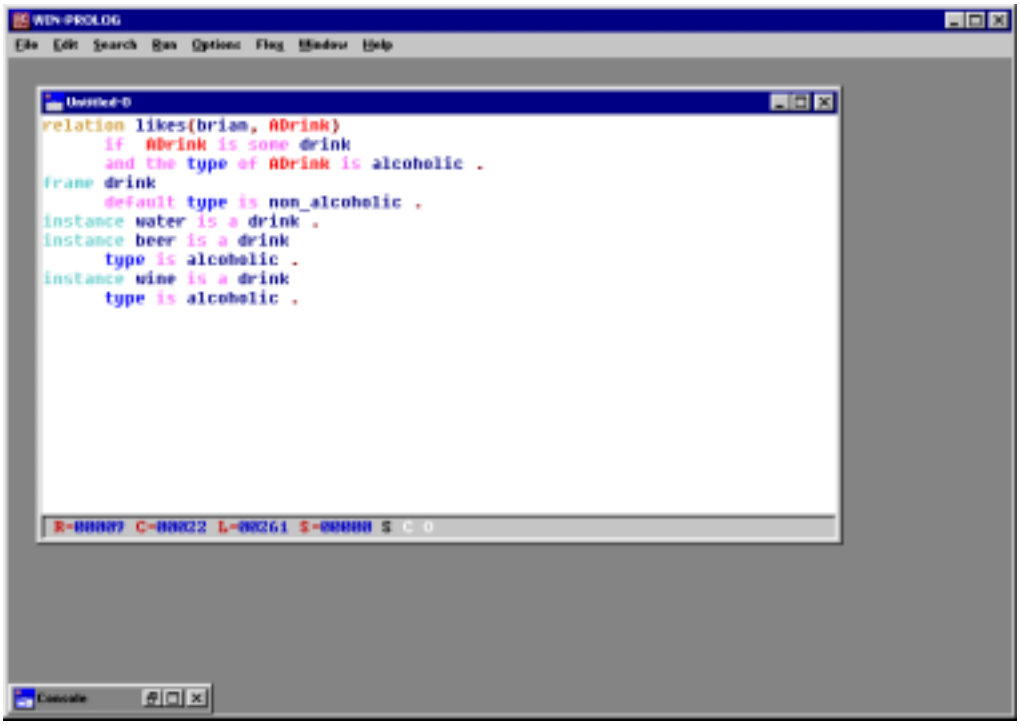


Figure 120 – The KSL program syntax-coloured in a rich edit window

The Run/Compile Option

Before you can run a flex program, you need to compile it via the "Run/Compile" menu option. After you click the mouse on the "Run/Compile" menu option or press *<enter>*, messages about the types of flex sentences being compiled will be printed to the Console window. If a syntax error occurs and the flex menu's Analyse Syntax Errors option is ticked, syntax analysis will begin to take place on your flex code. For further information on syntax analysis of flex code, see the next chapter.

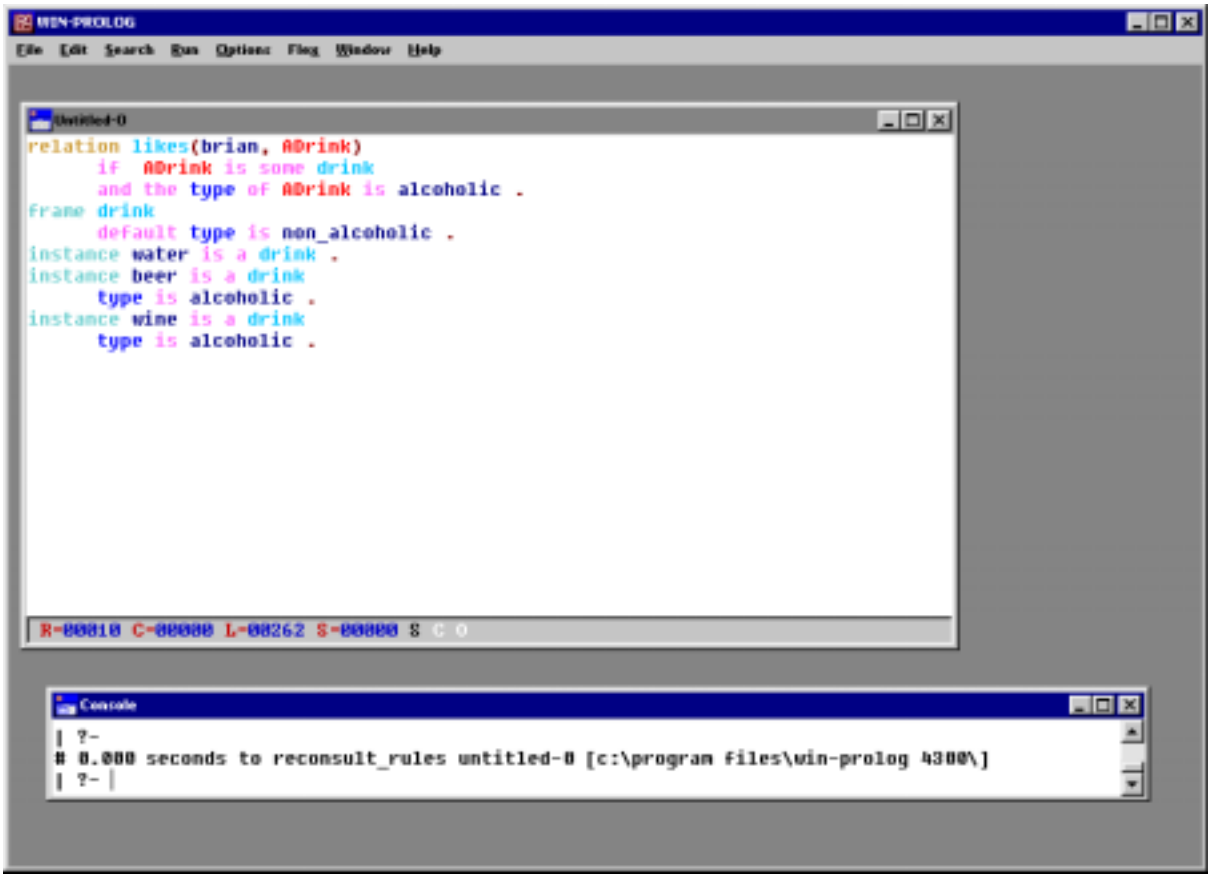


Figure 121 – Compiling your flex program

Query

Once your flex program has been compiled, you will be able to run it. Change focus to the Console window; you can use the usual Windows keys (such as <*ctrl-tab*>) to switch between ***WIN-PROLOG*** edit windows. Now let's try out your program by typing the following at the command line:

```
?- likes(brian,What).                    <enter>
What = beer ;                            <space bar>
```

The first solution should be displayed; if you press the <*space bar*>, or click the left mouse button, each next solution will be displayed in turn.

```
What = wine                              <space bar>
?-
```

Notice that ***WIN-PROLOG*** does not wait for another <*space bar*> or mouse click when it knows there are no more solutions to a query, it automatically returns to the "?-" prompt.

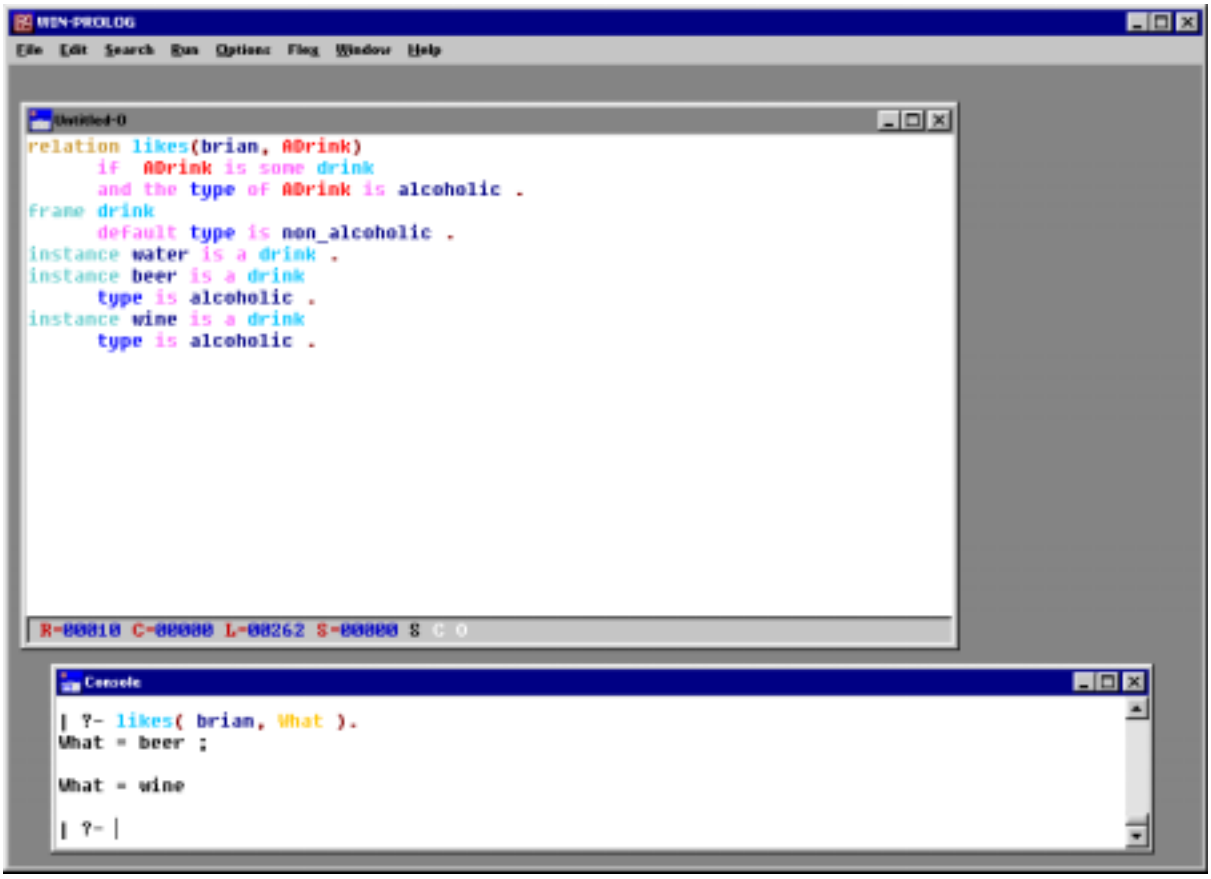


Figure 122 – Executing likes/2

The File/Close Option

The "File/Close" option works in the same way for a KSL edit window except that it may leave KSL code in the flex workspace; to remove all KSL code from the flex workspace, you can enter the following command at the ***WIN-PROLOG*** command line prompt:

?- **initialise.** *<enter>*

The Options/Spypoints... Option

The Options/Spypoints... menu option is greyed out for a KSL edit window. Use the Flex/Watch Points... menu option instead.

Spying a Rule

To spy a flex rule, you need to execute *spy_rule/1* from the ***WIN-PROLOG*** command line:

```
?- spy_rule( check_bread_and_butter ). <enter>
yes
?- spied_rule( check_bread_and_butter ). <enter>
yes
?- nospy_rule( check_bread_and_butter ). <enter>
yes
?-  spied_rule( check_bread_and_butter ). <enter>
no
```

This can also be done from the Watch Points dialog.

Spying a Slot

A slot can be spied by executing *spy_slot/2*:

```
?- spy_slot( temperatuure, _ ). <enter>
yes
?- spied_slot( temperature, _ ). <enter>
yes
```

This can also be done from the Watch Points dialog.

Spying a Relation

When the source level debugger window appears, you will find that the code you are looking at is not KSL but the Prolog code that the KSL has been mapped onto. You can view the Prolog code that the KSL code has been mapped onto at any time by executing *listing/0* from the ***WIN-PROLOG*** command line:

```
?- listing. <enter>
```

As the KSL relation:

```
relation likes(brian, ADrink)
if  ADrink is some drink
and the type of ADrink is alcoholic .
```

maps onto the Prolog code:

```
likes( brian, A ) :-
  equality( A, B : some_instance(drink,B) ),
  equality( type @ A, alcoholic ).
```

you can spy such a relation by executing the following from the ***WIN-PROLOG*** command line:

```
?- spy( likes/2 ). <enter>
```

Spying a Fact

You can spy a fact by executing *spy_fact/1*:

```
?- spy_fact( likes ) . <enter>
yes
?- spied_fact( likes ). <enter>
yes
```

This can also be done from the Watch Points dialog.

Spying the Forward Chaining Engine

The forward chaining engine can be spied by executing *spy_chain/1*:

```
?- spy_chain. <enter>
yes
?- spied_chain. <enter>
yes
```

This can also be done from the Watch Points dialog.

Chapter 22 - The Flex Menu

This chapter describes the uses of the "Flex" menu of *WIN-PROLOG*, covering the analysis of flex syntax errors; the inheritance settings for frame hierarchies; the classification of frames according to their attributes; the testing of forward chaining rules; the resetting of dynamic data; the browsing of frames, rules and relations; the graphical presentation of frame hierarchies and the setting of watch points. **Note that this chapter only applies if you have purchased the flex for Windows toolkit.**

Start up flex for Windows, as described in ***Running Flex for Windows*** on page *213*, and after the environment has finished loading, you will see that the "Flex" menu appears between the "Options" menu and the "Window" menu.

The Analyse Syntax Errors Option

The first option on the "Flex" menu sets the analysis of syntax errors when compiling KSL code. Selecting this option toggles syntax analysis between on and off. When this option is ticked syntax analysis is turned on.

When the syntax analyser is turned on, it looks at the code for any KSL sentence that causes a syntax error and makes suggestions about how you might change the code so that it will compile successfully.

In order to demonstrate the "Flex/Analyse Syntax Errors" option, you should now load a file into an edit window. Assuming you worked through the examples in **Chapter 21 - Flex and Existing Menus**, you will have created two files in *WIN-PROLOG*'s *EXAMPLES\FLEX* folder, *LIKES.KSL* and *ROBOT.KSL*. Start up *WIN-PROLOG*, and when you see the Main window, open *LIKES.KSL* into an edit window: select "File/Open..." and when you click the mouse or press *<enter>*, the "Open" dialog will appear. Choose the *EXAMPLES\FLEX* folder from the "File listing" box, then click "Open" or press *<enter>*. From the "File listing" box, select *LIKES.KSL* and click "Open" or press *<enter>*. An edit window will appear containing a copy of your example program.

Switch to the *LIKES.KSL* window and delete the full stop in the frame "drink" just after the word "non_alcoholic". Now switch to the Console window and then select the "Run/Compile All" option, and when you click the mouse or press *<enter>*, a question dialog will be displayed on the screen and the following message will be displayed in the Console window:

```
Cannot parse the sentence ...
************************************************************
frame drink
        default type is non_alcoholic
```

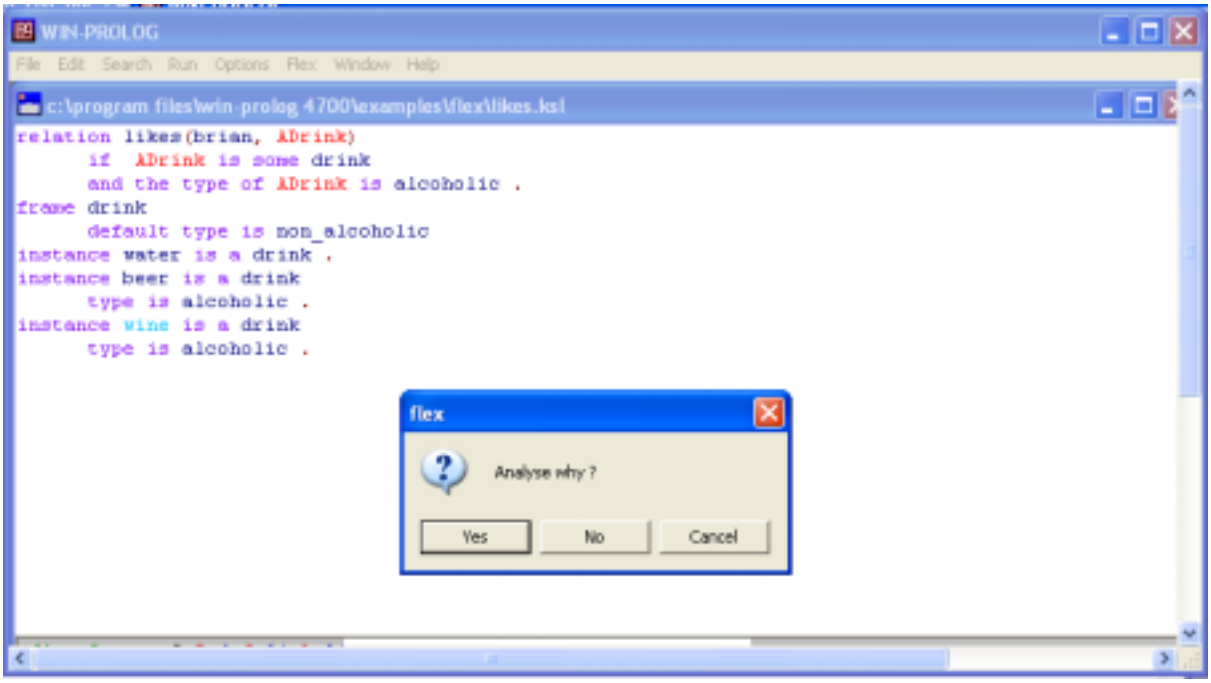


Figure 123 – "Analyse why" message box

The question dialog asks if you want to analyse why the syntax error has occurred; if you select the "Yes" option another dialog will be displayed telling you that there is a "missing full stop between... non_alcoholic and instance". If you put the full stop back then the program will parse correctly.

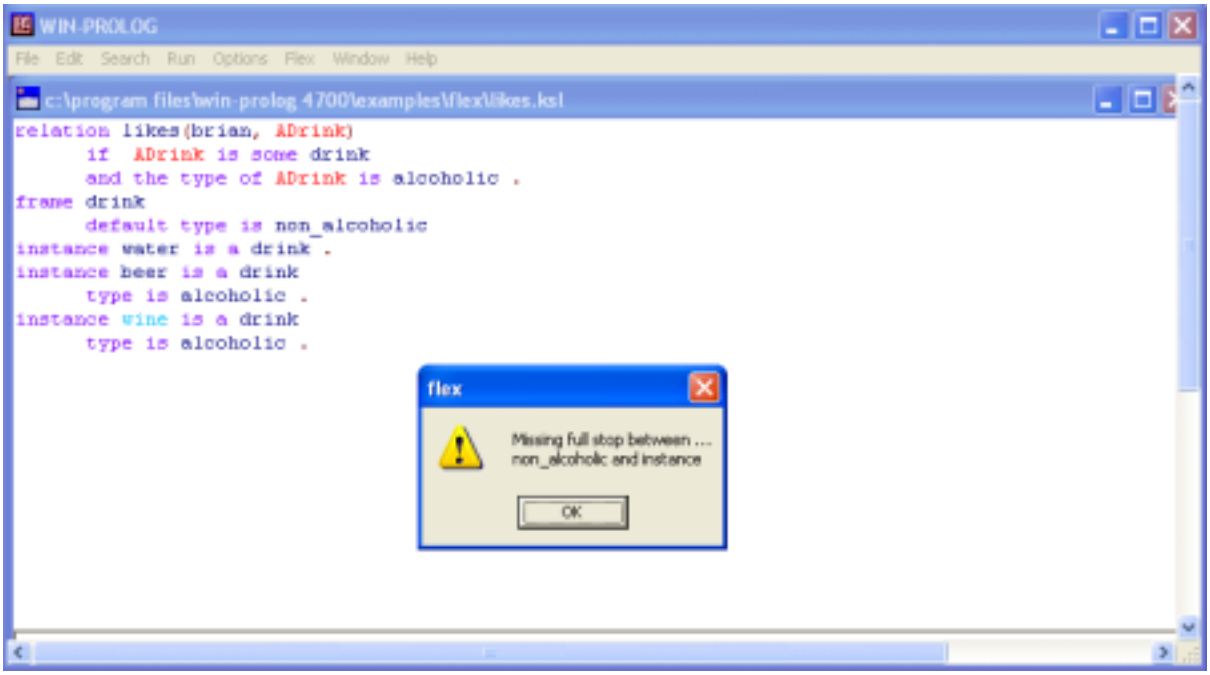


Figure 124 - Suggestion for a possible fix

You will then be asked if you want to continue parsing. If you do, you can always refer to the Console window for information about each syntax error reported.

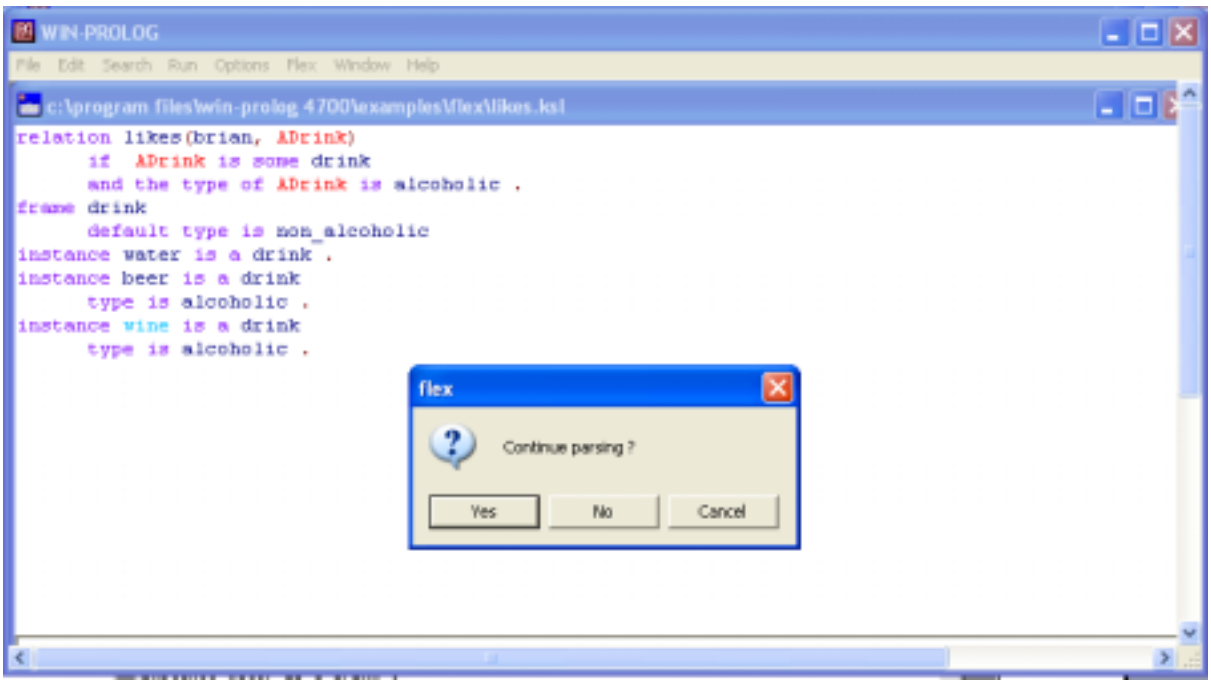


Figure 125 – "Continue parsing" message box

The Inheritance Settings... Option

This option allows you to globally change the way that information is inherited in flex frame hierarchies. The inheritance search algorithm can be varied in the following ways. You can change the inheritance search procedure from depth first to breadth first. You can specify that the attributes contained in the "universal" or "root" frame are considered first or last in the inheritance search (the "root" frame is a global frame to which all others are directly attached).

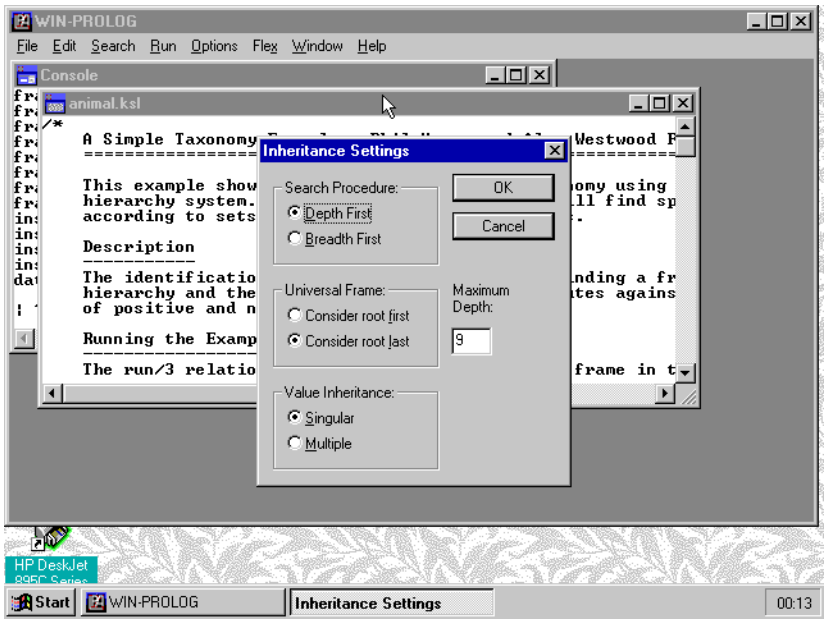


Figure 126 – "The Inheritance Settings" dialog

You can set the inheritance search mechanism to return either singular or multiple values. You can set the maximum depth of the inheritance search, which determines how many levels of the hierarchy will be considered when finding the value for an attribute.

Use of the "Flex/Inheritance Settings..." option will be shown later in the section on the "Flex/Browse Frames..." option.

The Classify by Example... Option

This option will find a number of frames and instances that all have the positive attributes and values that you specify and not the negative ones.

In order to demonstrate the "Flex/Classify by Example..." option, you should now load another file into an edit window. This time the *ANIMALS.KSL* file: select "File/Open..." and when you click the mouse or press *<enter>*, select the *ANIMALS.KSL* file from the "Open" dialog then click "Open" or press *<enter>*. An edit window will appear containing a copy of the example program. To compile the program, select the "Run/Compile" option.

Now select the "Flex/Classify by Example..." option and when you click the mouse or press *<enter>*, you will see the "Classification by Example" dialog. This dialog contains two sections: the positive clues and the negative clues. Each section allows you to specify a "clue" which consists of an attribute-value pair.

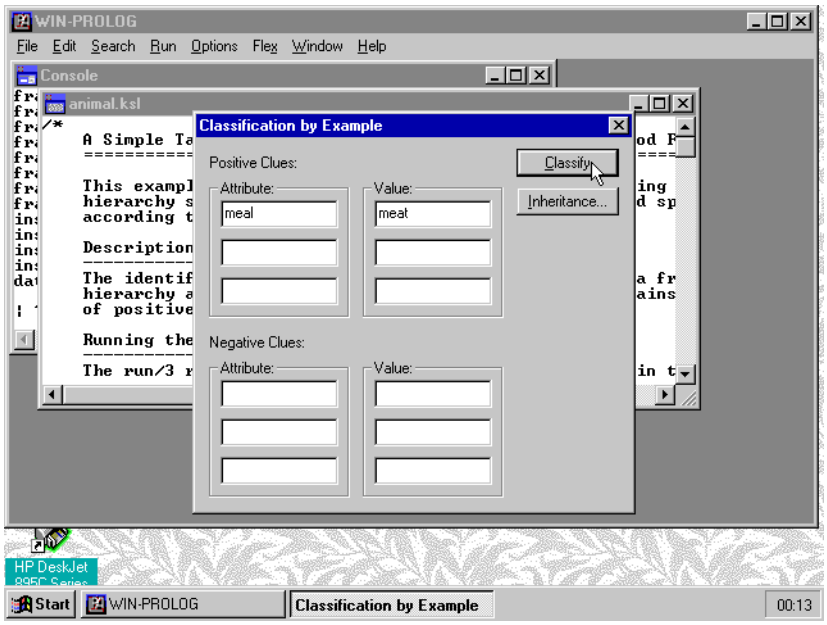


Figure 127 - The "Classification by Example" dialog

Type the word "meal" in the first field of the positive clues section, then enter the word "meat" in the next field on the right and click on the "Classify" button. The following output should appear in the Console window:

```
Criteria: A ^ (lookup(meal,A,meat),true)
Classifying frames ...

carnivore
feline
cat
manx
moggy

Classifying instances ...
No (more) frames/instances fit classification
```

The output shows the frames that have the attribute and value that you specified. Now return to the "Classification by Example" dialog and fill in the first field of the negative clues section with the word "tail" and click on the "Classify" button. The following output should appear in the Console window:

```
Criteria: A ^ (lookup(meal,A,meat),\+ lookup(tail,A,_21496),true)
Classifying frames ...
carnivore
manx
Classifying instances ...
No (more) frames/instances fit classification
```

This output shows all the frames that have the meals = meat attribute and value but do not have a tail attribute.

The Forward Chaining... Option

You can use this menu option to help you test out the currently defined forward chaining rules.

In order to demonstrate the "Flex/Forward Chaining..." option, you should now open another file, this time the *WATER.KSL* file, into an edit window.

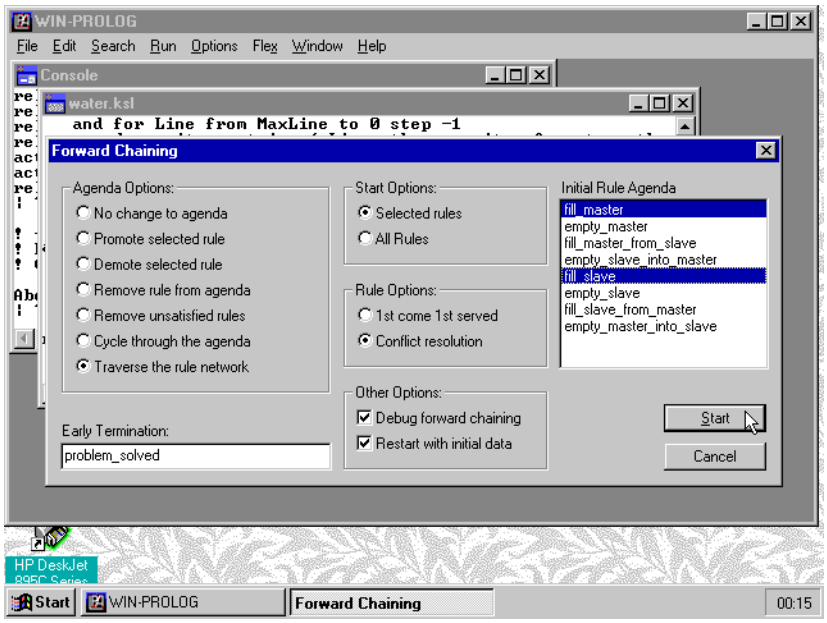


Figure 128 - The "Forward Chaining" dialog

The *WATER.KSL* program provides a solution to the problem of trying to get four litres of water in a jug given two water jugs with a capacity of five and seven respectively. To test the forward chaining rules with the "Flex/Forward Chaining..." option you will need to add the following relation to the *WATER.KSL* file window:

```
relation problem_solved
        if the contents of any container is 4 .
```

this relation defines the termination criterion for the forward chaining rules. You can now compile the WATER.KSL program by selecting the "Run/Compile" option.

The Forward Chaining Dialog

When you select the "Flex/Forward Chaining..." option the "Forward Chaining" dialog will be displayed. This dialog has six sections, each of which corresponds to a different aspect of the forward-chaining process.

The Agenda Options Section

"Agenda Options:" this section deals with the updating of rule agendas during forward chaining.

The Early Termination Section

"Early Termination:" this section specifies the termination criteria that stops the forward chaining process.

The Start Options and Initial Rule Agenda Sections

"Start Options:" this section is associated with the "Initial Rule Agenda:" section and in combination these sections specify the rules that are used for the first cycle of the forward chaining process.

The Rule Options Section

Rule Options:" this section specifies the way that the rules are selected from the rule agenda for firing.

The Other Options Section

"Other Options:" this allows you to set the debugging of forward chaining and the type of data used prior to forward chaining.

To set the "Forward Chaining" dialog to be equivalent to the "water_jugs" ruleset in the *WATER.KSL* file you will need to set the following parameters.

The "Agenda Options:" should be set to "Traverse the Rule Network", this makes use of the rule groups defined in *WATER.KSL*.

You should type the word "problem_solved" in the "Early Termination:" edit field this makes use of the relation that you typed in earlier which specifies that the termination criteria will be true if the contents of any container is four.

Set the "Start Options:" to "Selected rules" and select the rules "fill_master" and "fill_slave" in the "Initial Rule Agenda" listbox. This sets the initial rule agenda to contain the "fill_master" and "fill_slave" rules.

You should set the "Rule Options:" to "Conflict Resolution" which sets forward chaining to choose the rule that has the highest score and whose conditions are satisfied, during the selection phase of the forward chaining process.

The last option that you need to set is the "Other Options:" "Debug Forward Chaining" checkbox. Set this option so that the names of the candidate rules and the rules that are selected will be displayed as the program goes through the forward chaining cycle.

Finally you should click on the "Start" button or press *<enter>* to start forward chaining. The output from this forward chaining session will appear in the Console window in the following form (this shows the last part of the forward chaining cycle only):

```
Current rule agenda ...
empty_master_into_slave, fill_slave_from_master, fill_slave, empty_slave

Conflict resolution selection phase !
candidate rule ... empty_master_into_slave
candidate rule ... fill_slave_from_master
Conditions of this rule are satisfied !
Best score so far is 10
candidate rule ... fill_slave
Conditions of this rule are satisfied !
candidate rule ... empty_slave
Conditions of this rule are satisfied !
This is the best rule !

fill slave from master
|       |
|       |
|       |     |#####|
|#####|     |#####|
|#####|     |#####|
|#####|     |#####|
|#####|     |#####|
|_____|     |_____|
master       slave

Termination criterion is now satisfied !
```

at each stage the current rule agenda is printed, these are the rules from which the selection must be made. Then the conflict resolution selection phase is displayed, this shows the forward chaining mechanism deciding which rule to fire and what its score is. Finally, the fact that the termination criterion has been reached is output to the Console window.

The Browse Frames... Option

The Browse Frames... option allows you to browse the currently defined frames.

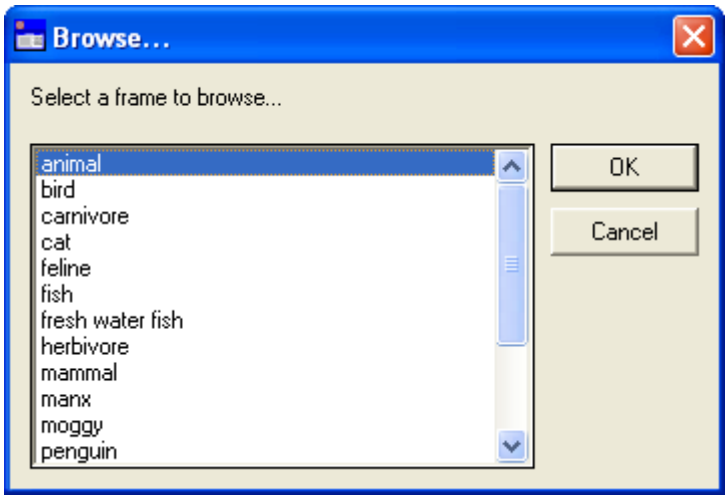


Figure 129 - Choose which frame to browse

You will see the name of the frame that is being browsed, its local values, its inherited values, its parent frames and its child frames. Notice that the local value for the "animal" frame is "blood = warm".

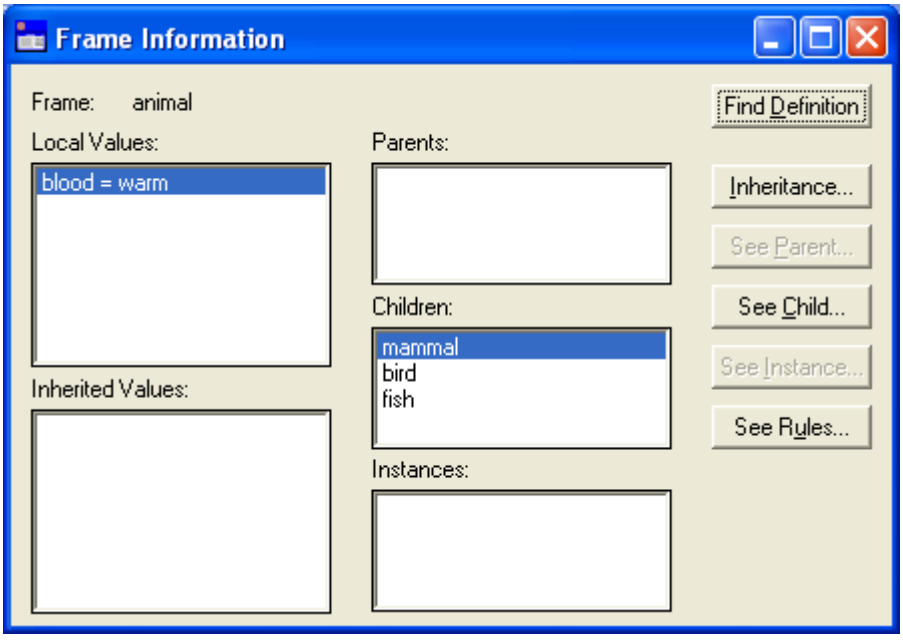


Figure 130 – Browsing the animal frame

Now select the "mammal" child frame and click on the "See Child..." button. The frame that is now being browsed is the "mammal" frame. Notice that the parent frame is "animal" and that the inherited value is "blood = warm". It is possible to browse through a frame hierarchy in this way checking the local and inherited values.

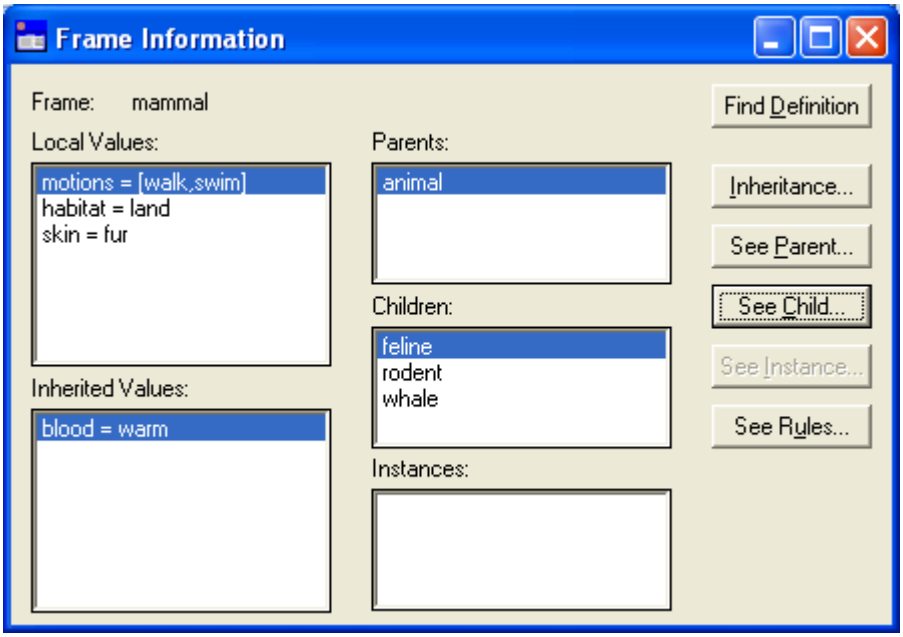


Figure 131 - Browsing the frame "feline"

If you select the "feline" child frame and click on the "See Child..." button you will be browsing the "feline" frame. Take note of the local value "tail = [long,furry]. Now select the "cat" child frame and click on the "See Child..." button to browse the "cat" frame. Scroll the inherited values until you see the "tail" attribute to show that it has been inherited correctly from the "feline" frame. Now select the "manx" child frame and again click on the "See Child..." button to browse this frame. This time, if you search through the inherited values, you will notice that there is no definition for the "tail" atttribute. To see why this is click on the "Find Definition" button. You will be taken to the definition of the "manx" frame in the *ANIMAL.KSL* window. Looking at the code you will see that the definition of the "manx" frame explicitly excludes the inheritance of the "tail" attribute.

The inherited values you have seen so far have been found using the default inheritance settings. We will now see the effect of using the "Inheritance Settings" dialog on the values that are found. Select the "Flex/Browse Frames..." menu option. This time select the "salmon" frame from the listbox in the "Browse..." dialog. The "Frame Information" dialog will be displayed for the "salmon" frame showing the following inherited values:

```
blood=cold
habitat=sea
motions=[swim]
skin=scale
```

now click on the "Inheritance..." button and you will be shown the "Inheritance Settings" dialog. Change the "Value Inheritance:" option from "Singular" to "Multiple" and click on "OK" or press *<enter>*. This time the "Frame Information" dialog for the "salmon" frame will show the following inherited values:

```
blood=cold
blood=cold
blood=warm
blood=warm
habitat=sea
habitat=river
habitat=water
habitat=water
motions=[swim]
motions=[swim]
skin=scale
skin=scale
```

notice how most of the options are shown twice; this is because the "salmon" frame has two parents and the inherited values are being returned from both. Now click on the "Inheritance..." button again and this time change the "Maximum Depth:" option in the "Inheritance Settings" dialog from "9" to "1". and click on "OK" or press *<enter>*. This time the "Frame Information" dialog for the "salmon" frame will show the following inherited values:

```
habitat=sea
habitat=river
```

The Browse Rules... Option

The Flex/Browse Rules... menu option allows you to browse the currently defined rules.

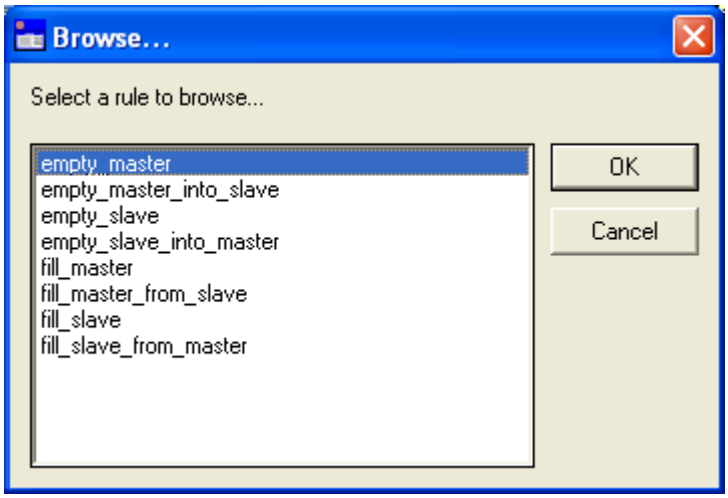


Figure 132 – Selecting a rule to browse

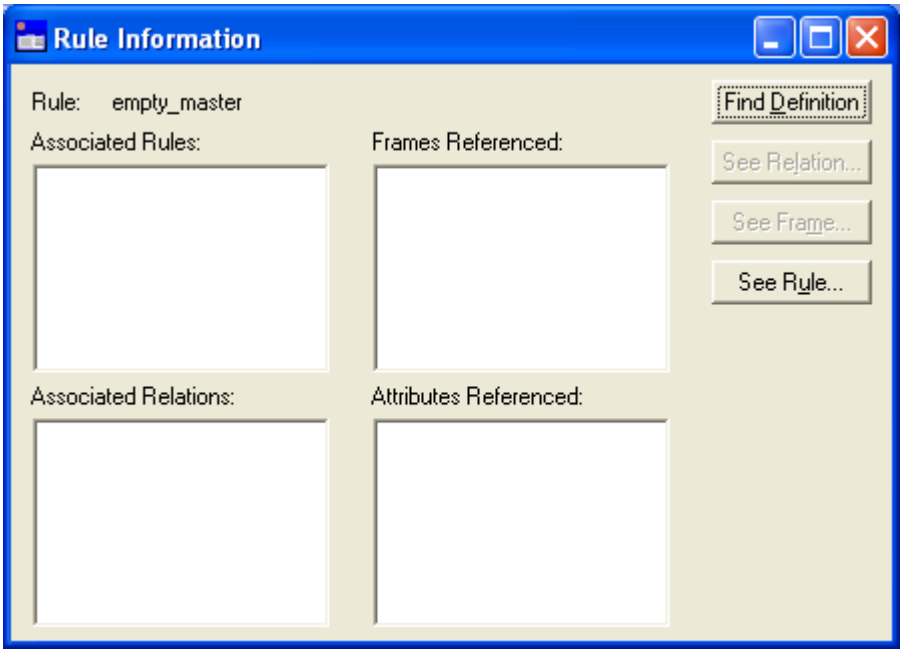


Figure 133 – The Rule Information dialog

The Browse Relations... Option

The Flex/Browse Relations... menu option allows you to browse the currently defined relations.

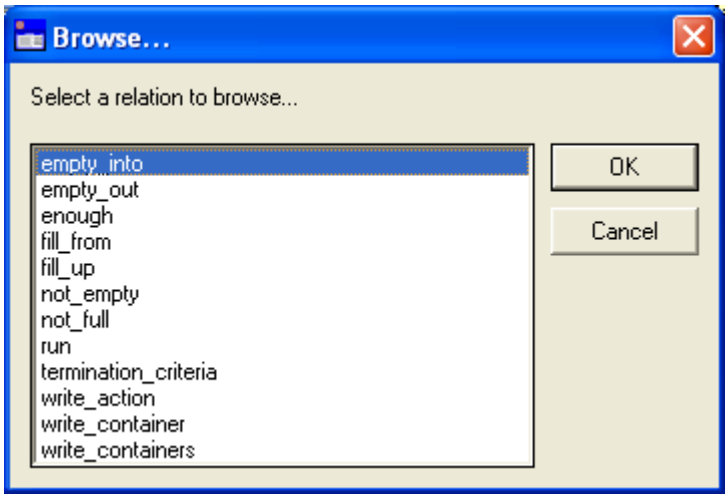


Figure 134 – Selecting a relation to browse

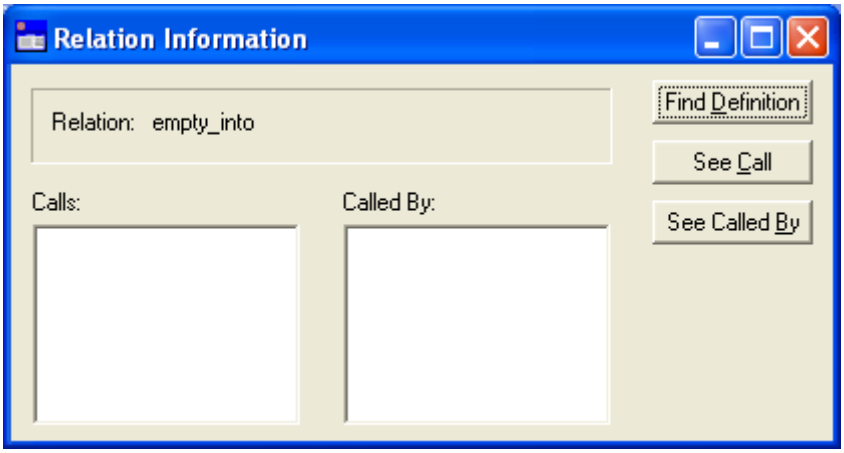


Figure 135 – The Relation Information dialog

The Graph Frames... Option

The Graph Frames... option allows you to browse a frame hierarchy graphically. You can graphically browse the "animal" frame hierarchy by first selecting the "Flex/Graph Frames..." option, then selecting the "animal" frame from the listbox in the "Frame Graph" dialog and clicking on the "OK" button or pressing *<enter>* to accept the dialog.

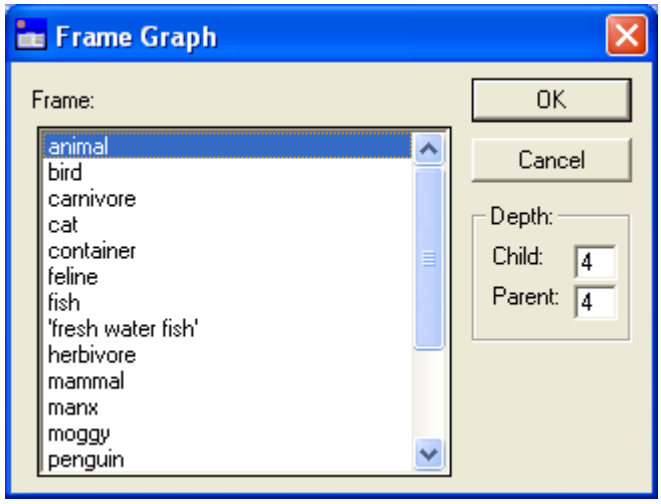


Figure 136 – The Frame Graph dialog

You will now see a graphical representation of the *ANIMAL.KSL* frame hierarchy with the "animal" frame as root. Each frame is represented by a button that contains the name of the frame (you can click on this to make it the root of a new graph, and therefore navigate around a large tree, or you can right-click on this to get information on the frame). The frames are connected by lines that show the inheritance links.

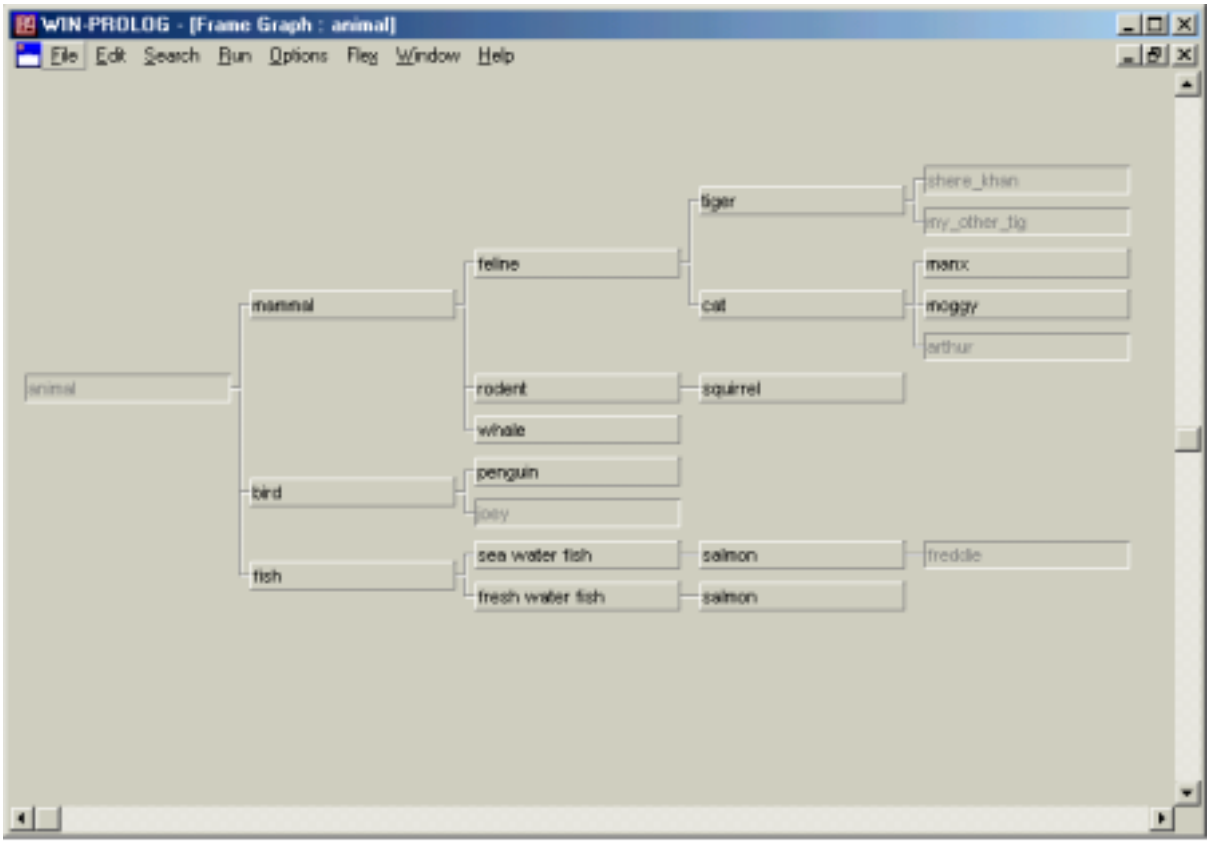


Figure 137 - Frame graph of the frame "animal"

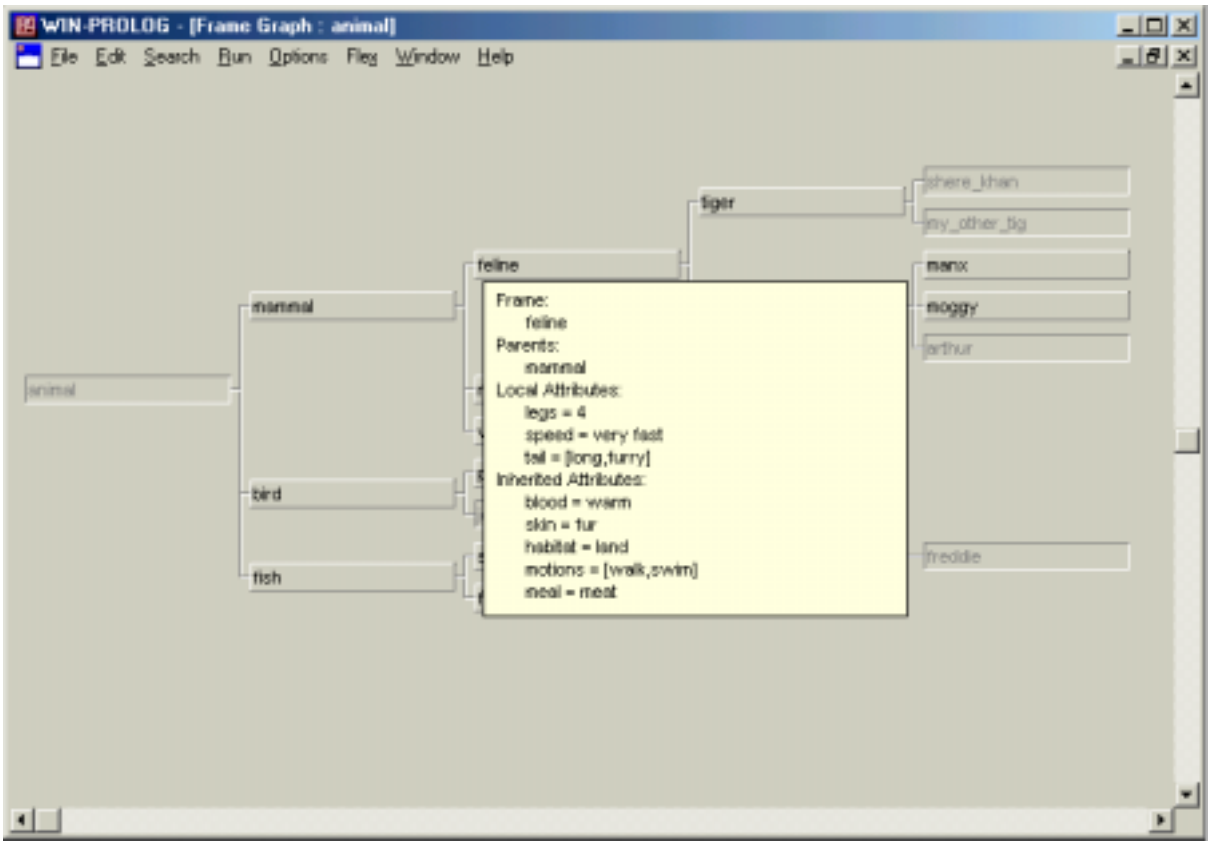


Figure 138 – Getting information on the"feline" frame

The frame at the middle left of the screen, is the current "root" frame. Notice that it is indented, this represents the very top of the browse tree. It has direct inheritance links with all the other displayed frames. The frames displayed to its right are its children. The currently selected frame in this case is the "animal" frame. Frames are drawn as normal buttons while instances are drawn as indented buttons (which you cannot select as there is no tree beneath them) .

Clicking on a frame will draw the frame hierarchy from that frame. If you click on the "sea water fish" frame's button the hierarchy will be redrawn with "sea water fish" as the root frame. If you now click on the "sea water fish" frame's button the hierarchy will again be redrawn this time with "animal" as the currently selected frame, thus clicking on a selectable root button moves up the hierarchy while clicking on any other button moves down. You can move to any place in the frame hierarchy in this manner.

Right clicking on a frame's button will show a "Frame Information" popup.

The Graph Relations... Option

The Graph Relations... option allows you to browse a relation hierarchy graphically.

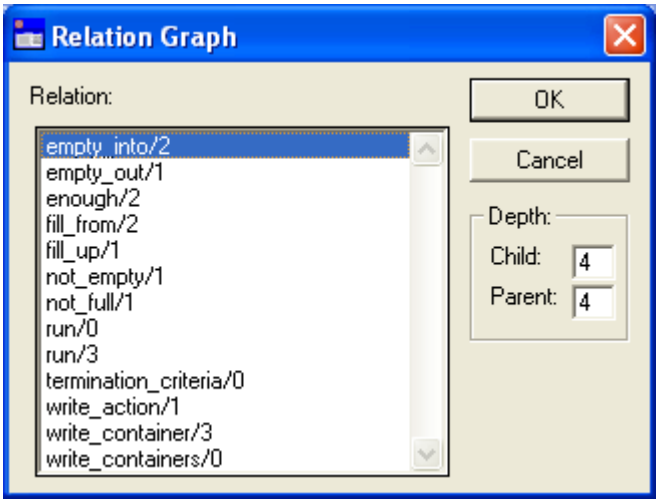


Figure 139 – The Relation Graph dialog

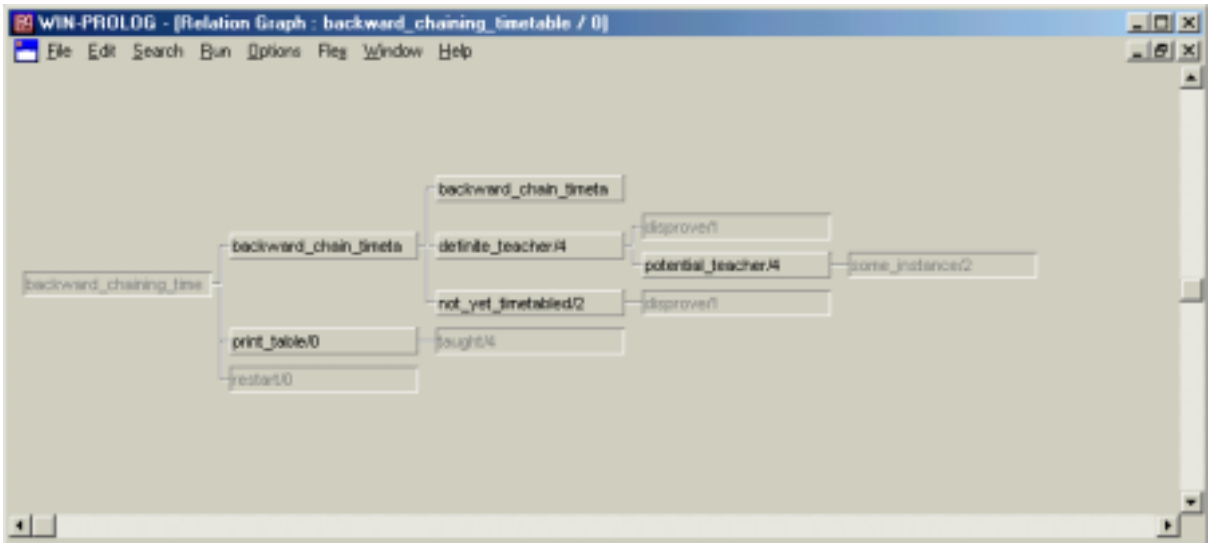


Figure 140 – A relation graph for "backward_chaining_timetable/0"

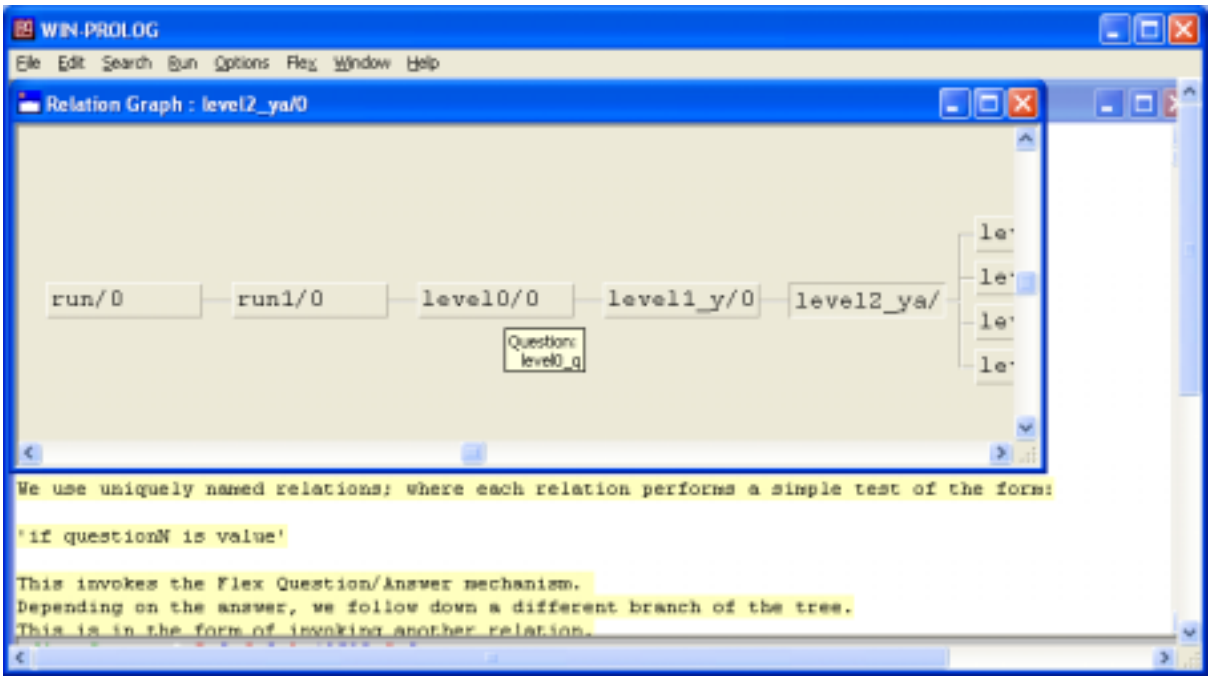


Figure 141 – Getting information about a relation

The Watch Points... Option

The Watch Points... option leads to the Watch Points dialog. Watch points are to flex what spy points are to Prolog. The Watch Points dialog allows you to spy flex facts, rules, frames and/or slots; you can also spy the forward chaining engine.

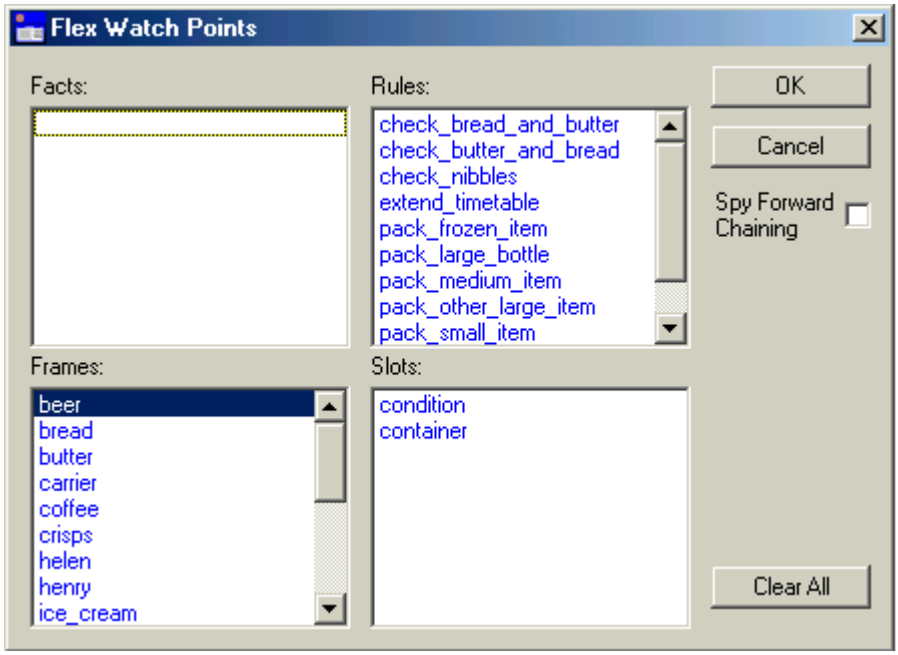


Figure 142- WatchPoints dialog

Chapter 23 – Combining Flex and Prolog

This chapter describes an example which uses both Flex and Prolog code. There are many ways of combining Flex (.KSL code) files with Prolog (.PL) files. This is because Flex is actually implemented in Prolog; and the Flex environment is just some extra functionality loaded on top of the standard Prolog environment. This means you can still use your Prolog system from within Flex.

Generally, LPA advise that you keep you Prolog code and Flex code in separate files. However, in the v4.9 release, there is in addition an example of how to incorporate Prolog code DIRECTLY within a Flex file – this is very convenient if you just want to incorporate a small piece of Prolog logic withn a Flex program.

Flex comes with a simple to use but limited run-time GUI system for data capture from the end-user. This is integrated into the question and answer mechanism that Flex provides as part of the question definition concept. In this example, we use the user-defined question mechanism in Flex to define a dialog in Prolog (rather than restrict ourselves to the simple built-in GUI screens that Flex offers). Thereby, the control passes from Flex to Prolog and then back to Flex.

The supplied example consists of two files - MULTIKSL.KSL and MULTFLEX.PL. It also uses the supplied modal dialog utility (LIBRARY\DLGCALL.PL) which is dynamically loaded using *ensure_loaded/1*.

To run the program, execute *run/0* from the command line prompt:

```
?- run. <enter>
```

run/0 is defined as a flex action clause in MULTIKSL.KSL:

```
action run ;
   do   ensure_loaded( examples('flex/multflex.pl') )
   and  new_dialog( person`s name, person`s food,
                                   person`s drink )
   and  ask preferences
   and  new_person( preferences )
   and  for every ThisPerson is some person
        do  write( 'Person: ' )
        and writeq(  the name of  ThisPerson -
                     the food of  ThisPerson -
                     the drink of ThisPerson
                  )
        and nl
        end for .
```

The *new_person/1* action is defined as:

```
action new_person( Values ) ;
    do  Person is a new person
    and forall(
           member( Attribute-Value, Values ),
```

```
            update_person( Attribute, Person, Value )
          ) .
```

Where *forall/2* is a built-in Prolog routine. The *update_person/1* action updates the slots for the new instance and is defined as:

```
action update_person( Attribute, Person, Value ) ;
    do  atom_string( ValueAtom, Value)
    and new_slot( Attribute, Person, ValueAtom ) .
```

A frame, person, is defined as:

```
frame person
   default name  is 'Fred'  and
   default food  is 'Pasta' and
   default drink is 'Wine' .
```

A question, preferences, is defined using the 'answer is' construct. This tells flex that the 'answer is' code will be supplied by a user routine named, *call_the_dialog/*. This could be defined in flex or in Prolog (or indeed in VB, Java etc). In this example, it is defined in Prolog:

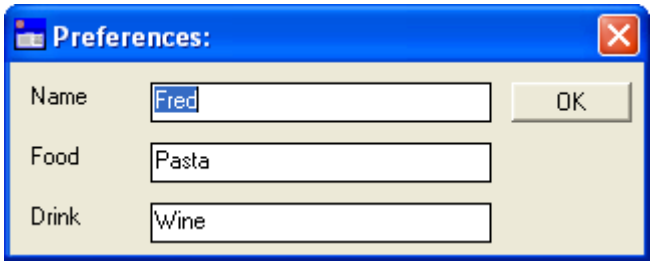
```
question preferences
   answer is Preferences such that
          call_the_dialog( Preferences ) .
```

The output from *call_the_dialog/1* is passed back through a logical variable called, Preferences; so that whenever the preferences question is encountered, this answer to the question will become bound to whatever value is in the Preferences variable.

The *call_the_dialog/1* predicate is defined in Prolog and called by the flex question, preferences. It displays the dialog via a call to *call_dialog/3* (defined in DLGCALL.PL). The output from *call_dialog/3* is composed into a list used *findall/3,* a very powerful built-in Prolog routine:

```
call_the_dialog( Result ) :-
   Dialog = new_dialog,
   call_dialog( Dialog, _, Terms ),
   findall( Label-Value,
           ( member( ((Dialog,FieldID),Value),Terms),
             entry( FieldID, Label ) ),
            Result
            ).
```

If you call *call_dialog/3* directly, you will see what the new_dialog dialog looks like and what data structure gets returned:




```
     ?- call_dialog( new_dialog, X, Y ).
     X = ok ,
     Y = {(new_dialog ',' 8000) ',' `Diane`, (new_dialog ','
8001) ',' `Cake`, (new_dialog ',' 8002) ',' `Coffe`}
```

The *entry/2* facts (defined in MULTIFLEX.PL) are a local database of entries to help correlate answers to labels; if you were to add a new input field to user_dialog you would need to add a corresponding *entry/2* fact as well:

```
     entry( 8000, name  ).
     entry( 8001, food  ).
     entry( 8002, drink ).
```

The code, in MULTIFLEX.PL, for defining user_dialog came almost straight out of the Dialog Editor:

```
new_dialog( Name, Food, Drink ) :-
     atom_string( Name, Edit1 ),
     atom_string( Food, Edit2 ),
     atom_string( Drink, Edit3 ),
        _S1 =
        [ws_caption,ws_sysmenu,dlg_ownedbyprolog,ws_ex_dlgmodal
        frame],
         _S2 =
     [ws_child,ws_visible,ws_tabstop,ws_border,es_left,es_mult
     iline,es_autovscroll,es_autohscroll],
         _S3 =
     [ws_child,ws_tabstop,ws_visible,bs_pushbutton,bs_text,bs_
     center,bs_vcenter],
   _S4 = [ws_child,ws_visible,ss_left],
     wdcreate(  new_dialog,          `Preferences:`,
                                338, 211, 322, 128, _S1 ),
     wccreate( (new_dialog,8000),  edit,    Edit1,
                                70,  10, 170,  20, _S2 ),
     wccreate( (new_dialog,8001),  edit,    Edit2,
                                70,  40, 170,  20, _S2 ),
     wccreate( (new_dialog,8002),  edit,    Edit3,
                                70,  70, 170,  20, _S2 ),
     wccreate( (new_dialog,1000),  button, `OK`,
                              250,  10,  60,  20, _S3 ),
     wccreate( (new_dialog,10000), static, `Name`,
                                10,  10,  50,  20, _S4 ),
     wccreate( (new_dialog,10001), static, `Food`,
                                10,  40,  50,  20, _S4 ),
```

```
wccreate( (new_dialog,10002), static, `Drink`,
                              10,  70,  50,  20, _S4 ).
```

After running the program, the new instances created can be seen using the flex frame browser.

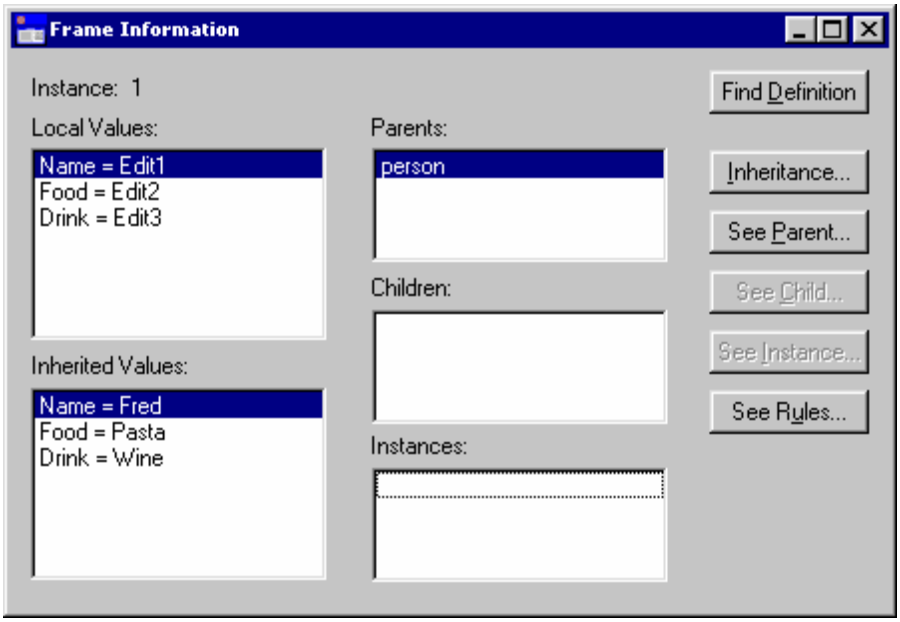


Figure 143- Updated Frame Information dialog

Chapter 24 - Prolog++ for Windows

This chapter describes Prolog++ for Windows, an object-oriented extension for ***WIN-PROLOG***. Prolog++ inherits all of the features of the extensive ***WIN-PROLOG*** programming language including access to a large number of Windows Graphical User Interface (GUI) functions, allowing you to create polished Prolog++ for Windows applications. **Note that this chapter only applies if you have purchased the Prolog++ for Windows toolkit. You should go read the Prolog++ manual which explains how to program using Prolog++**

What's in Prolog++ for Windows?

Prolog++ gives you the potential of an object-oriented programming language combined with the underlying capabilities of ***WIN-PROLOG***. Prolog++ gives you the ability to define objects that can be joined together to form hierarchies. The objects in the hierarchies can inherit both methods (programs) and attributes (data).

The Windows version adds to Prolog++ by providing an easy-to-use pull down menu programming environment, sporting multiple edit windows, incremental and optimised compilation, source level debugging, and comprehensive text search and replace facilities. All of the GUI features used by the environment, and more beside, are directly available to Prolog++, allowing customised environments to be built and shipped as part of an application.

About this Chapter

In addition to the standard ***WIN-PROLOG*** environment, Prolog++ for Windows includes a "Prolog++" menu to provide easy access to the information contained in your Prolog++ programs. This chapter describes how to use the Prolog++ specific aspects of the environment, including the display of attribute information, the sending of messages, the optimisation of objects, the initialisation of the object workspace, the browsing of objects and getting information on the Prolog++ system itself.

Throughout this guide, examples show you how to perform basic actions; these examples have been kept as simple as possible: you will find more detailed information about the system itself in the "Prolog++" manual. Please enjoy this guide, and have fun with Prolog++ for Windows!

Running Prolog++ for Windows

Once you have installed Prolog++, a Shortcut is automatically created for ***WIN-PROLOG*** to run Prolog++. Should you need to do this manually, the procedure is as follows:

Go to the folder in which you've installed *WIN-PROLOG* and then create a shortcut to *WIN-PROLOG*. This is done by clicking on the PRO386W.EXE application with the right mouse button and selecting "Create Shortcut" from the menu subsequently displayed. Once you've created a shortcut you can then drag it to the location you require, e.g. the "desktop" or "Start" menu. The command line of the shortcut can be edited by clicking on it with the right mouse button and selecting "Properties" from the menu subsequently displayed. The "Shortcut" tab in the "Properties" dialog contains a "Target" edit field that is equivalent to the *WIN-PROLOG* command line. Enter the following command line (assuming "C:\Program Files\Win-Prolog 4900" is the home folder for your copy of *WIN-PROLOG*) all on one line:

"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(system(pppenv)).

Chapter 25 - The Prolog++ Menu

This chapter describes the uses of the "Prolog++" menu of ***WIN-PROLOG***, covering the setting of spypoints, the optimisation of objects, the initialisation of the object workspace and the browsing of objects and information on the Prolog++ system. **Note that this chapter only applies if you have purchased the Prolog++ for Windows toolkit. You should go read the Prolog++ manual which explains how to program using Prolog++.**

Start up Prolog++ for Windows, as described in ***Running Prolog++ for Windows*** on page *243*, and after the environment has finished loading, you will see that a "Prolog++" menu appears in addition to ***WIN-PROLOG***'s standard menus, between the "Options" menu and the "Window" menu.

The Set Spypoint... option

This option allows you to set spypoints on all the methods in a class. When you select this option you are presented with a dialog that contains a listbox showing all the currently defined classes. You can select any number of the classes from the listbox and when you click on the "OK" button, spypoints will be set on all the methods of those classes.

The Goto Definition... Option

This option will take you to the source code for a currently defined class. When you select this option you are presented with a dialog that contains a listbox showing all the currently defined classes. You can select any one of the classes from the listbox and when you click on the "OK" button the source for that class will be brought into view.

The Browse Option

This option allows you to browse the attributes and methods of the currently defined classes. When you select this option you are presented with a dialog that contains a listbox showing all the currently defined classes.

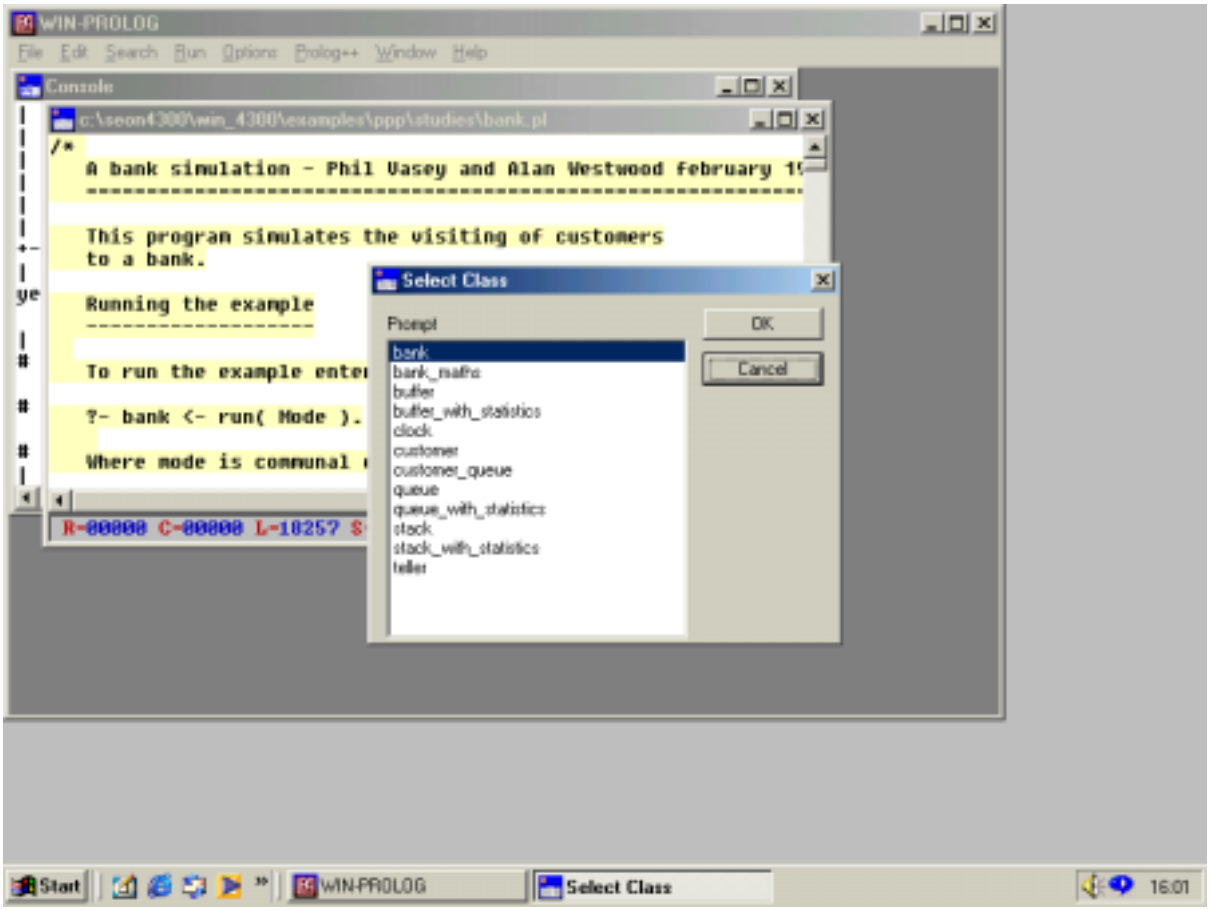


Figure 144 – The Select Class dialog

You can select any one of the classes from the listbox and when you click on the "OK" button the 'Prolog++ Browser' dialog will be shown for that class. This dialog gives various types of information on the class you have selected.

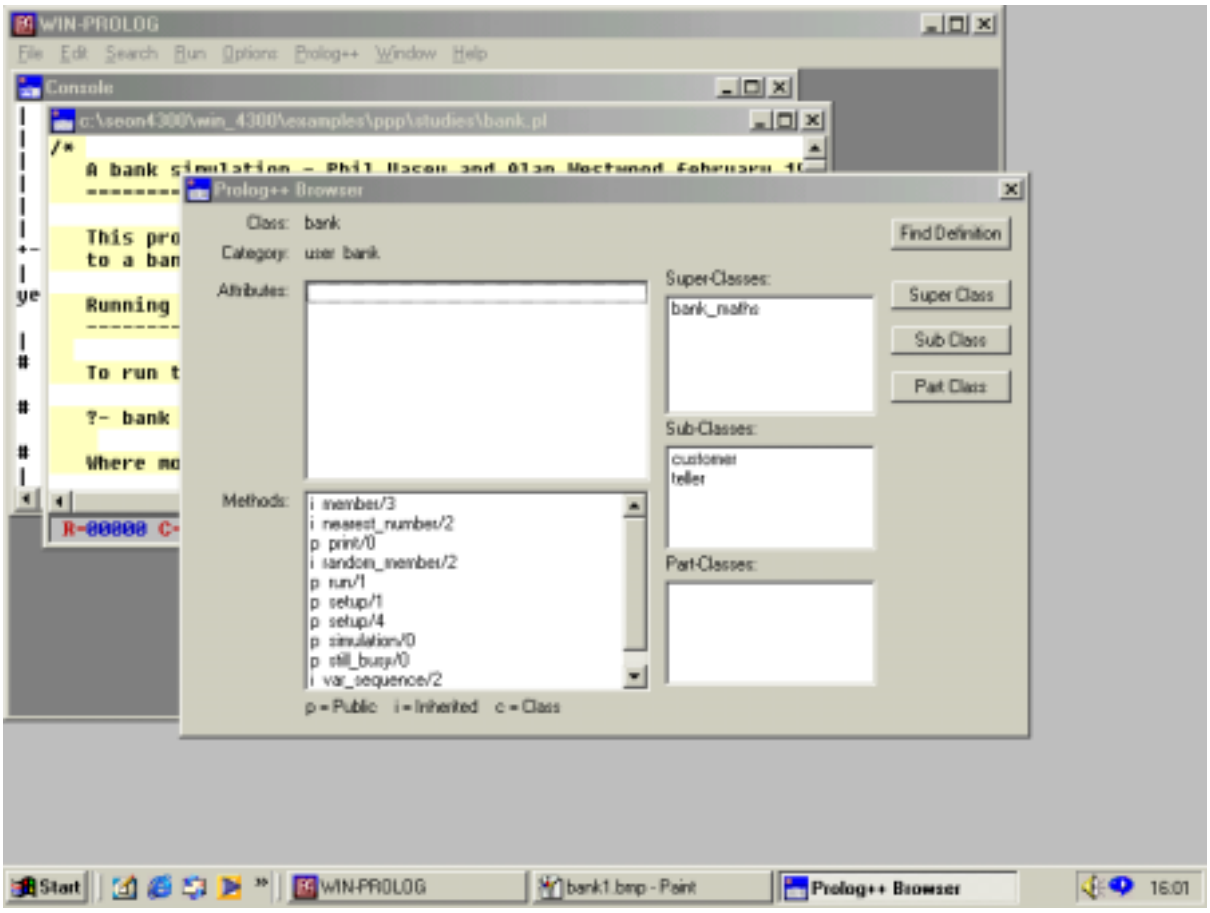


Figure 145 – The Prolog++ Browser dialog

The Optimize Option

This option allows you to optimise the code for a class hierarchy. When you select this option you are presented with a dialog that contains a listbox showing all the currently defined classes. You can select any one of the classes from the listbox and when you click on the "OK" button the object-oriented code for that class will be optimised.

The Optimize All... Option

This option allows you to optimise the code for all the currently defined classes. When you select this option the code for all the classes will be optimised.

The Initialize Option

You can use the "Prolog++/Initialize" option to clear out all the currently defined objects in the Prolog++ workspace.

Chapter 26 – Flint

This chapter describes the use of the Flint toolkit with *WIN-PROLOG*. **Note that this chapter only applies if you have purchased the Flint toolkit. You should go read the Flint manual which explains how to use Flint and the fuzzy editor.**

What is Flint?

Given a rule:

rule1: if A & B then C

there are 3 potential areas for uncertainty.

- Uncertainty in data (how true are A and B)

- Uncertainty in the rule (how often does A and B imply C)

- Impreciseness in general

The first two can be handled using techniques based on probability theory and the third using fuzzy logic.

Flint supports various modes of uncertainty handling, namely:

- Fuzzy Logic

- Bayesian updating

- Certainty factors

These are presented in a uniform and consistent way, using a common set of access routines.

These are made available in the context of a powerful AI programming environment supported by text editing facilities, compilers, debuggers and a variety of GUI tools.

This means you can integrate uncertainty with databases, spreadsheets, expert systems, graphics, etc.

Rules with uncertainty can be represented in both a native Prolog syntax, or where flex is present, in the KSL syntax that flex uses.

Chapter 27 – ProData Database Interface

This chapter describes the use of the Prodata toolkit with *WIN-PROLOG*. **Note that this chapter only applies if you have purchased the ProData toolkit. You should go read the ProData manual which explains how to use ProData and relational databases.**

What is ProData?

ProData is an interface between *WIN-PROLOG* and an ODBC data source, allowing *WIN-PROLOG* to access Microsoft Access data sources, SQL Server data sources, etc. Prodata allows database tables to be accessed from *WIN-PROLOG* as though they existed within *WIN-PROLOG*'s environment as facts. This facilitates the use of Prolog rules over the contents of the data source, with no need to download any part of the data source, as all data source accesses are done 'on-the-fly'. Backtracking, cut, call, not and all other standard Prolog mechanisms work identically over the data source accesses and the internal database, thus achieving the highest level of transparency possible.

The Prodata Prolog code can be loaded into *WIN-PROLOG*, and its DLL (LPADBW.DLL) launched, by executing the following at the *WIN-PROLOG* command line:

```
?- ensure_loaded( system(dblink) ). <return>
yes
```

Should you wish to launch *WIN-PROLOG* and Prodata together via a shortcut, the procedure is as follows. Go to the folder in which you've installed *WIN-PROLOG* and then create a shortcut to *WIN-PROLOG*. This is done by clicking on the PRO386W.EXE application with the right mouse button and selecting "Create Shortcut" from the menu subsequently displayed. Once you've created a shortcut you can then drag it to the location you require, e.g. the "desktop" or "Start" menu. The command line of the shortcut can be edited by clicking on the shortcut with the right mouse button and selecting "Properties" from the menu subsequently displayed. The "Shortcut" tab in the "Properties" dialog contains a "Target" edit field that is equivalent to the *WIN-PROLOG* command line. Enter the following command line (assuming "C:\Program Files\Win-Prolog 4900" is the home folder for your copy of *WIN-PROLOG*) all on one line:

```
"C:\Program Files\WIN-PROLOG 4900\PRO386W.EXE" ensure_loaded(
system(dblink) ).
```

An example program, NWIND.PL, which interacts with the Northwind data source as supplied with Microsoft Access, can be found in the EXAMPLES\PRODATA folder. Just make sure you have set NORTHWIND.MDB up as a data source under the name 'NWIND' within the ODBC Administrator. The comments at the top of the NWIND.PL file will direct you further in running this example program.

Chapter 28 – LPA Intelligence Server

This chapter describes the use of the Intelligence Server toolkit with *WIN-PROLOG*. **Note that this chapter only applies if you have purchased the Intelligence Server toolkit. You should go read the Intelligence Server manual which explains how to use LPA Intelligence Server with a variety of programming languages.**

What is the Intelligence Server?

The *WIN-PROLOG* "Intelligence Server" provides a full-featured Prolog server for applications written using almost any Windows programming language or visual development system. It not only lets applications use the inferencing power of Prolog in the background, but also offers 'call-back' facilities which allow the Prolog code to request any additional input needed to complete its computations.

The Intelligence Server interface provides a number of features that directly benefit the development of distributed intelligent applications, these can be summarised as follows:

Text-based interface – you can use this clean and safe interface between languages with radically different data types.

Language independence – you can develop the Prolog component independently of the language used to build the client component.

Multiple engines – you can load as many instances of the *Intelligence Server* (from a single application) as you need. So, for example, in a multi-threaded application each thread can have its own instance of the *Intelligence Server*.

Extendibility – you can use any Windows language that supports 32-bit DLL calls as the client.

Prolog backtracking – each solution to a multiple solution Prolog goal is returned individually.

Clean and safe development – because the Prolog code for the server utilises pure text input and output, you can easily develop it in isolation. This means that your Prolog code can be fully tested and debugged using the extensive features of the *WIN-PROLOG* development environment.

Call-backs – these allow a Prolog goal to request additional information from your client code and then to continue with the computation.

Running the Examples

An adaptation of the WIN-PROLOG *MEALS.PL* example, where all of the GUI handling has been replaced by plain Teletype output, is supplied as an Intelligence Server client application. In fact, four versions of this adaptation are supplied, one of each in C, Delphi, Java and Visual Basic.

All four use the same Prolog code, *CLIENT.PL* ; four identical copies of this file have been supplied in the C, Delphi, Java\SunJava and VB folders within C:\Program Files\WIN-PROLOG 4900\EXAMPLES\SERVER.

Now view the folder for one of the examples.

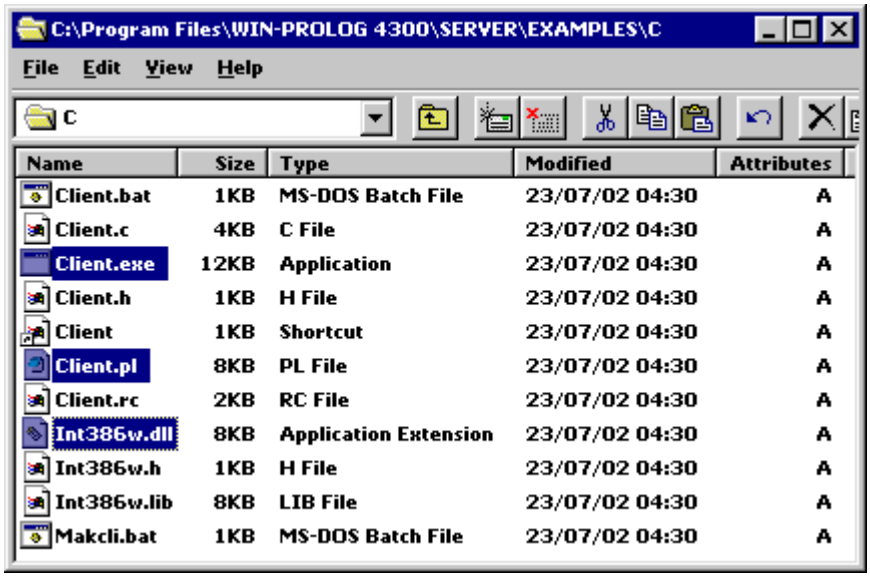


Figure 146 – The SERVER\EXAMPLES\C folder

Create a new folder on your hard disk and copy the *CLIENT.PL*, *CLIENT.EXE* and *INT386W.DLL* files from C:\Program Files\Win-Prolog 4900\Server\Examples\<chosen_language> into it. Then copy *PRO386W.EXE* and *INT386W.OVL* from your WIN-PROLOG home directory into it also.

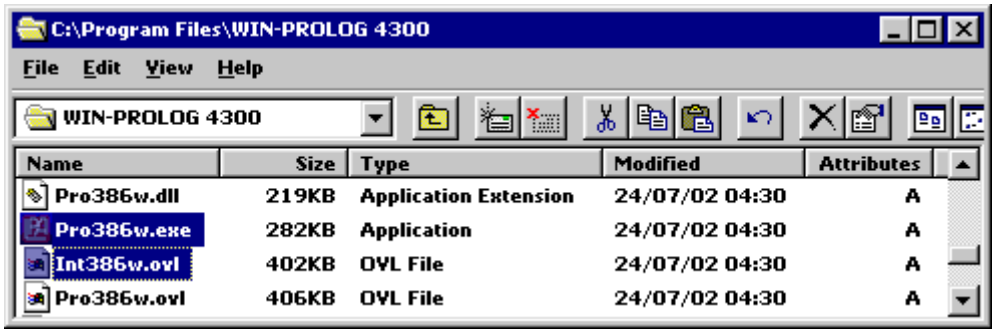


Figure 147 – The WIN-PROLOG home directory

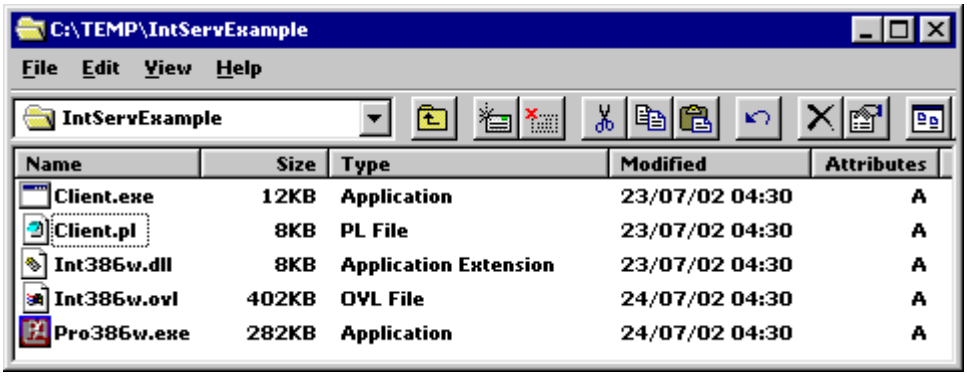


Figure 148 – The five files copied to their new folder

To run the chosen example, double-click on the *CLIENT.EXE* program in your newly created folder. The CLIENT dialog will appear. If you are also watching the Windows Task Manager dialog, you will see that PRO386W.EXE is also launched. Click on the Init Goal button in the CLIENT dialog. Another dialog will appear asking you to select a meal type; click on one of the buttons. The first solution will appear in the CLIENT dialog. Click on the Call Goal button to get the next solution. Click on the Exit Goal button. When you close the CLIENT dialog, the associated PRO386W.EXE task running in memory is terminated.

If you want to launch CLIENT.EXE from a DOS prompt, make sure you are already in its home directory:

```
C:\>CD Temp\IntServExample <enter>
C:\Temp\IntServExample>Client.Exe <enter>
```

otherwise an "error 20 *menu/0* predicate not defined" message may appear in the CLIENT dialog because the CLIENT.PL file can't be found.

The same is true when launching CLIENT.EXE from another process; if you were to launch CLIENT.EXE from *WIN-PROLOG* via a call to *exec/3*, make sure you are already in the folder containing CLIENT.EXE:

```
?- chdir( 'c:\temp\intservexample' ). <enter>
yes
?- chdir( X ). <enter>
X = 'c:\temp\intservexample'
?- exec( 'client.exe', '', _ ). <enter>
yes
```

It is the fact that CLIENT.PL can't be found that is causing the problem. So whatever folder you are currently in when you launch CLIENT.EXE, make sure CLIENT.PL resides in this same (i.e. current) folder.

Chapter 29 – ProWeb Server

This chapter describes the use of the ProWeb Server toolkit with *WIN-PROLOG*. **Note that this chapter only applies if you have purchased the ProWeb Server toolkit. You should go read the ProWeb Server manual which explains how to use ProWeb Server and the WWW.**

What is the ProWeb Server?

ProWeb is a specialised extension to a Hypertext Transport Protocol (HTTP) server. ProWeb is written in Prolog and supports the development, testing and deployment of intelligent, dynamic server-based applications on intranets and the Internet. ProWeb basically provides a link between an HTML page displayed on your client's web browser and your application (developed in *WIN-PROLOG* and associated toolkits) running on your server. When a client views HTML pages in the traditional way, there is normally no 'intelligent' interaction between the client and the server - the client simply asks for the next HTML page via its Uniform Resource Locator (URL) and it is dutifully returned by the server. With ProWeb, your clients can effectively 'talk' directly to your ProWeb-based application running on your server, using HTML pages as the means to pass back and forth the many interactions generated during such conversations.

The LPA ProWeb Server allows web sites to use the powerful reasoning capacity of *WIN-PROLOG* and associated toolkits completely in the background with HTML, Javascript, Java and other standard tools providing the user interface. Prolog is a very powerful technology, ideal for building intelligent solutions. The Web is saturated with information; low-level languages do not have the capacity to sift, sort and make sense of such a huge volume of data in a meaningful way. ProWeb brings the power of the industry-leading *WIN-PROLOG* to the Web, and so provides an ideal platform for building intelligent agents, smart search engines, diagnostic systems and data mining utilities.

ProWeb hides the complexity of CGI programming and HTML forms from the developer; ProWeb automatically generates all the required HTML and CGI code needed for your application to run smoothly on the Web. By handling all the communication between the forms and the application logic, ProWeb relieves the developer of all the tiresome book-keeping. This provides an intuitive and robust way for developers to build 'live', dynamic applications.

Chapter 30 – WebFlex

This chapter describes the use of the WebFlex toolkit with *WIN-PROLOG*. **Note that this chapter only applies if you have purchased the WebFlex toolkit. You should go read the WebFlex manual which explains how to use WebFlex and the WWW.**

What is WebFlex?

WebFlex is a version of the flex expert system toolkit for the web.

There are specific instructions for how to install WebFlex as a web server application and there is a separate manual which explains how you can extend your Flex code to use HTML and Javascript.

Appendix A - Initialisation and Switches

The WIN-PROLOG system can be configured in two main ways, namely through an *"initialisation file" (.INI file)* and by a number of *"command line switches"*. In fact, the initialisation file can contain default settings for these switches, which can then be overridden as needed by the user.

Configuring *WIN-*PROLOG Memory Usage

Being a 32-bit program, ***WIN-*PROLOG** can use up to 2Gb (2048Mb) of memory. It is entirely up to you how much memory you want to allocate to ***WIN-*PROLOG**, and how much you want to leave for other Windows applications (and Windows itself). By default, ***WIN-*PROLOG** uses around 3-4Mb of memory, allocated amongst its six internal data areas: you can change the size of one or more of these data areas using command line switches. If you need to change one or more of ***WIN-*PROLOG**'s default settings, you will need to either add these to the PRO386W.INI file or create your own shortcut containing the required command line switches.

Command Line Syntax

The WIN-PROLOG command line can be typed at and executed from the Windows command line or embedded in a Windows shortcut file.

The WIN-PROLOG command line consists of three optional sections. The first is an "overlay override", which uses the <escargot> character ("@") to specify an overlay (.OVL file) to use in place of the standard one, "pro386w.ovl". This is followed by zero or more switches, each of which consists of a <slash> character ("/") followed by one of the letters A..Z or a..z, which in turn may be followed by an unsigned integer: switch letters are not case-sensitive ("/A" and "/a" are synonymous). There must be no spaces within a switch, and there must be at least one space between neighbouring command line entries. The final section of the command line contains zero or more Prolog goals which will be executed once the development system has been initialised. For example, the following command initialises WIN-PROLOG to use an overlay called "FOO.OVL", with the "A" switch set to "0" and the "B" (backtrack stack) switch set to 128 and the term, "write(`Hello World`)." (which must be terminated by a full stop) offered as a goal:

```
pro386w @foo /A /B128 write( `Hello World` ).
```

Merging .INI and Typed Command Lines

The "PRO386W.INI" file can contain a command line with all the features described above (see below for more information about the .INI file), and this is merged with any actual command line elements as follows:

overlay	*an overlay override on the command line takes precedence over one in the .INI file*
switches	*any switches in the .INI file and not on the command line are used together with those on the command line*
goals	*any goals in the .INI file are executed first, followed by any on the command line*

The effect of these rules is to allow default settings to be stored in the .INI file, without losing the ability to change individual settings in any given session. By allowing .INI file goals to be executed in addition to command line goals, it is easy to load libraries or current projects automatically.

Memory Settings

Several switches are used to allocate the amount of memory used in each of the nine configurable ***WIN-PROLOG*** memory areas: each one specifies a number of kilobytes of memory to use (see Appendix H in the Technical Reference Manual for a detailed discussion of memory management). If any given switch is not specified, its value will be set to "-1"; if it is specified without a numerical argument, or with an argument of zero, its value will be set to "0": in any of these cases, it is treated as the default setting. The memory switches are:

Switch	Meaning	Default Size (kb)
/B	backtrack stack	64
/L	local stack	64
/R	reset stack	64
/H	term heap	256
/T	text heap	2048
/P	program heap	8192
/S	system stack	64
/I	string input buffer	256
/O	string output buffer	256

Miscellaneous Settings

A handful of switches are used to set certain other aspects of WIN-PROLOG behaviour. If any given switch is not specified, its value will be "-1"; if it is specified without a numerical argument, or with an argument of zero, its value will be "0": in any of these cases, it is treated as the default setting. These switches are:

Switch	Meaning	Settings
/D	dynamic code calls	0 = direct
		1 = logical
/V	banner flag	0 = show banner
		1 = hide banner

Initialisation File Entries

The "PRO386W.INI" file contains two main classes of section. The first comprises a single heading, and is processed by the WIN-PROLOG kernel very early in the boot process: this "[pro386w]" section can contain one of two main entries. The first entry, "profile=", can be used to redirect all further processing to another .INI file, and allows network administrators to position such files on users' hard disks, private directories, current working directories, centrally, or whatever. If present, it renders the rest of the "PRO386W.INI" file redundant, giving complete precedence to its named file. Note that this entry is only recognised in "PRO386W.INI" itself: this limits redirection to a single step, and therefore avoids "wild goose chases" or even endless looping.

The second entry, "command=", contains any default command line, as discussed above. Consider the following .INI file:

```
[pro386w]

command=/h512 /t4096 /p10000 consult( library(trace) ).
```

A command or Windows shortcut such as:

```
pro386w /h /p10000 /s128 write( hello ), nl.
```

will start WIN-PROLOG, returning the "term heap" allocation to its default, rather than the 512kb indicated in the .INI file, further increasing the "program heap" allocation to 10000kb, setting the "system stack" to 128kb, and using the .INI file setting of 4096kb for the "text heap". It will then execute the goal, "consult(library(trace)).", before in turn executing "write(hello), nl.".

Some toolkits use additional entries in the "[pro386w]" section, but these are ignored by WIN-PROLOG itself, and so are beyond the scope of this appendix.

The second class of section includes a variety of headings, and lists various persistent settings that are saved whenever the "save_settings" Prolog flag has been set to "on" (see prolog_flag/3). Note that even if this flag is set to "off", any settings in the .INI file will still be loaded at startup: the only way to restore defaults is to delete this file (or, at the very least, its offending entries) prior to starting WIN-PROLOG. Here are some typical entries:

```
[prolog_flag]
save_settings=on
context=supervisor
...etc
[window]
console=0 0 814 452 3
main=0 0 1020 696 1
```

Appendix B - *LPA-*PROLOG and Memory Management

One of the black arts of Prolog programming is the management of memory. By its very nature, Prolog hides the low-level allocation and deallocation of memory from the user: this is a feature to be expected of such a high-level language; on the other hand, it can be a cause of confusion and program inefficiency if not reasonably well understood.

This document attempts to shed light on the matter in a practical way, while touching on some theoretical aspects of this complex subject.

Memory: Stacks, Heaps and Buffers

LPA-*PROLOG**, both in its ***WIN- and DOS- guises, provides sophisticated, automatic memory management, with the added flexibility that the user can change default memory settings when required for any particular application. One of the problems with this, however, is that it is not always apparent what such settings should be used!

***LPA-*PROLOG** uses three types of memory area, generally known as "stacks", "heaps" and "buffers". We will look at these in more detail in the following sections.

*LPA-*PROLOG: Stacks

Stacks are used to handle the flow of control during program execution, and are common to all programming languages which support functions, procedures or subroutines. In C, for example, the function call:

```
foo(a) ;
```

uses a stack, first to save the parameter "a", and then to save the address of the next thing to do after calling "foo" (the "return address"). Then it jumps to the code for "foo" itself. Upon completion, "foo" picks up the return address from the stack, and jumps to that location to continue processing; the "a" parameter is also "popped" off the stack to restore its status to that immediately before the function call.

Prolog's execution model is more complex than that of C or most other languages; in particular, there are effectively two return addresses for each "function" (or rather, "predicate") call: where to continue after success and where to continue after failure. This suggests that at least two independent stacks must exist, and this is indeed the case: the "local" stack is used to store "on success" addresses and associated data, while the "backtrack" stack is used to store "on failure" addresses and associated data. For example, consider the program:

```
foo :-
  bar,
  sux.

foo.
```

At the point at which "bar" is being called by "foo", there are two things that might need to be done subsequently. If "bar" succeeds, the next thing that needs to be called is "sux". If it fails, the next thing to be called is the second clause of "foo". The address of "sux" is therefore stored on the local stack, and that of the second clause of "foo" on the backtrack stack.

But even this is not quite sufficient. As well as knowing where to go on failure, Prolog must also know how to "undo" any variable bindings that were made before a given predicate call failed. The information needed to do this is stored on yet another stack, the "reset" stack. Entries on this stack do not occur every time a variable is bound, because many variables simply cease to exist on failure. Again, consider the program:

```
foo :-
  X = 123,
  bar( X ).

foo.
```

Should "bar(X)" fail, the whole first clause of "foo" will fail, and the variable "X", which only occurs in that clause, will vanish. As such, there is no need for Prolog to create a "reset" entry for it. However, consider a variation of the above program:

```
foo :-
  ( X = 123
  ; X = 456
  ),
  bar( X ).

foo.
```

This time, if "bar(X)" were to fail, an attempt is going to be made to bind "X" to "456" before trying "bar(X)" again. Patently, if "X" still had the value "123", any attempt to say "123 = 456" would fail at once, so Prolog has to convert "123" back to the variable "X" before continuing. It prepares for this eventuality by creating an entry on the reset stack for the variable "X" before binding it to "123". Notice that if "bar(X)" does fail, and "X" is reset, it is not necessary to create a new reset entry before binding the variable to "456": if "bar(X)" were to fail a second time, the whole first clause for "foo" would fail, and "X" would vanish.

In brief, "local" stack entries are used to record where to continue in the current clause after a successful predicate call, "backtrack" stack entries are used to record where to continue earlier in the current clause, or elsewhere in the current predicate, after a failed predicate call, and "reset" entries are used to store the previous values of any variables bound, if those variables will still be visible after a failed predicate call. All three of these stacks, together with a fourth one (the "system" stack), will be discussed in greater depth below.

*LPA-*PROLOG: Heaps

Unlike stacks, which are ordered memory areas onto which data and addresses can be "pushed", and later "popped", a "heap" is a disordered repository of memory units, called "cells". In *LPA-*PROLOG, heaps are used to store the individual elements of compound terms and lists, as well as strings and atoms, and even segments of compiled Prolog code.

Stacks do not need special memory management, just so long as pushes and pops are performed symmetrically; heaps, on the other hand, require additional handling if disused memory is to be recycled. Consider the program:

```
foo :-
  bar( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,0] ).
```

Each time "foo" calls "bar(X)", the latter program creates a ten-element list structure by allocating and linking ten cells from the data heap. A recursive call is then made to "foo", requiring no use of any stacks because there is no "next thing to do" either for success or failure.

Moreover, the list that has been created can no longer be seen, because neither was it passed out of "foo" as a parameter, nor is there any way back into "foo" through failure such that it might need to be seen again. The ten cells of heap memory that the list occupied is simply severed from the computation, becoming ten cells of "garbage".

Each new recursive call uses up a further ten cells of heap space, and eventually the entire heap will be full of garbage. At this point, a special internal routine, the "garbage collector", is invoked. It works by checking which bits of the heap are still in use, and then collecting up all the remaining cells into a new allocation list. The garbage collector is called automatically whenever the heap fills with garbage, and is extremely quick, so in effect the heap behaves as an endless source of memory cells. It is, of course, possible to exhaust this "endless" supply, simply by creating a massive data structure; for example, consider:

```
foo( List ) :-
  bar( X ),
  foo( [List|X] ).
```

```
bar( [1,2,3,4,5,6,7,8,9,0] ).
```

This variation of the program not only uses ten cells per iteration for the list returned by "bar(X)", but also an eleventh of linking to the existing passed-in list. The main point, however, is that the data is not "lost" on the recursive call, and so does not become garbage, cannot be collected, and will quickly fill the heap to capacity.

The space used to store strings and atoms, the "text" heap, is handled in exactly the same way as data heap, except that the cells are somewhat larger and designed for storing text rather than Prolog data. The third, or "program" heap, is used to store the clauses and predicates of compiled Prolog code. While still a true heap, with specially designed cells, it does not need to be garbage collected in the conventional sense. The reason is that its data items (pieces of Prolog code) are persistent, and have to be explicitly removed by programs (using *retract/1*, *abolish/1*, etc). Whenever these predicates delete code, they explicitly return the freed memory to the program heap.

*LPA-*PROLOG: Buffers

Only a few words need to be said about buffers. These are simply flat memory blocks used for special purposes. *LPA-*PROLOG features a powerful "string" data type, which can be used among other things as an input or output "device". For example, the call:

```
?- listing ~> String.
```

creates a string called "String" containing the text of a program listing. Two buffers, the "input" and "output" buffers, are used to support this feature. The above example requires output buffer space, while:

```
?- read( Term ) <~ `foo(X). `.
```

reads from a string, requiring input space. *WIN-*PROLOG additionally supports user buffers, which can be created and deleted at will, and are primarily used for interfacing to the Windows API; these are, however, outside the scope of the current document.

Controlling Stack Usage

In virtually all cases, the default size settings for the four stacks enable programs to run comfortably, with ample "spare" space. It is almost always true that "Backtrack Stack Full", "Local Stack Full", "Reset Stack Full" and "System Stack Full" errors are due to a bug in an application's code, and not because the program concerned genuinely requires more resources. The next few sections show some examples of how to fill - or avoid filling - each of the four stacks.

The Backtrack Stack

As noted above, this stack is used to store "on failure" addresses introduced by choice points in an evaluation. The simplest way to fill it up, whatever its size, is to write a programs such as:

```
foo :-
  foo.

foo.

bar :-
  sux,
  bar.

sux.

sux.
```

On entry to the first clause for "foo", a backtrack stack entry is made pointing to the second clause, because this would need to be called in the event of failure. In the case of "bar", there is just one clause, but a backtrack entry is made at the point "sux" is called, because this has two clauses. The only way to avoid filling the backtrack stack is to ensure that, at the point of recursion, there are no choice points left in the current clause or relation. The following programs do not use up space on the backtrack stack:

```
foo :-
  foo.

bar :-
  sux,
  bar.

sux.
```

The "foo" case only has one clause, so no choice points are introduced, and the backtrack stack is not used; similarly, the "bar" case calls a version of "sux" with only one clause, and again, no backtrack entries are needed.

It is possible to force "determinism" on programs which otherwise would have choice points, by using the "cut" (!/0) predicate. This has the effect of removing backtrack stack entries for any choice points earlier in the current clause and elsewhere in the current predicate. The following programs run forever without filling the backtrack stack:

```
foo :-
  !,
  foo.

foo.
```

```
bar :-
  sux,
  !,
  bar.

sux.

sux.
```

The cut in "foo" removes the backtrack stack entry pointing to the second clause, while the one in "bar" removes the choice point introduced by the call to "sux". Similarly, the following program will run indefinitely:

```
bar :-
  sux,
  bar.

sux :-
  !.

sux.
```

In this case, the "sux" program is itself deterministic thanks to its use of cut.

Another feature of *LPA-PROLOG* which helps create determinism is "first argument indexing". This enables the execution mechanism to "ignore" clauses whose first arguments do not match the type, or in many cases, the value of a call. This subject is discussed at length elsewhere, but for present purposes, consider the following programs:

```
foo :-
  sux( 1 ),
  foo.

bar :-
  sux( _ ),
  bar.

sux( 1 ).

sux( 2 ).
```

The "foo" program calls "sux(1)", specifying the first argument: because there are no other clauses of "sux(...)" which can match the type or value "1", no choice points are created, and the program will run forever without using the backtrack stack. The "bar" program, however, calls "sux(_)" with a variable, which means that it cannot use first argument indexing to isolate just one of the clauses of "sux(...)", so choice points are created on the backtrack stack. Now consider:

```
foo :-
  sux( 1 ),
  foo.

sux( 1 ).

sux( _ ).
```

Here, even though the call in "foo" is to "sux(1)", because the second clause for "sux(...)" contains a variable, which could also match the value "1", choice points are once again needed and the backtrack stack will be quickly used up. Finally, consider:

```
foo :-
  sux( 1, _ ),
  foo.

bar :-
  sux( _, 1 ),
  bar.

sux( 1, 1 ).

sux( 2, 2 ).
```

Even though both "foo" and "bar" specify one value of "1" and one variable ("_") in their respective calls to "sux(...)", only "foo" is deterministic; "bar" creates choice points and requires backtrack stack space. The reason is that "foo" specifies the first argument as "1", in the call "sux(1,_)", while "bar" specifies the second argument, in the call "sux(_,1)". As its name implies, first argument indexing only works when the first argument is specified!

The Local Stack

As already described, this stack is used to store "on success" addresses. In theory, every predicate call needs to know where to continue upon successful completion, but in practice, the final predicate call in any relation saves stack space by not creating an entry: upon completion of such a call, the next thing to do is whatever the "parent" program is supposed to do next. Consider the program:

```
foo :-
  bar,
  sux.

bar :-
  you,
  too.
```

When "foo" calls "bar", a local stack entry is made pointing at "sux". Now "bar" calls "you", and another local stack entry is made, this time pointing at "too". When "you" succeeds (it might also have made local stack entries, as might any programs it called), the topmost entry is popped from the local stack, telling Prolog to continue execution at "too". This time, because "too" is the last call, no local stack entry needs to be made. Upon success, the topmost local stack entry is popped, which is our original one, pointing at "sux", so this is where execution continues.

It follows that a "tail recursive" program, whereby the recursive call is the final one in the clause, does not use local stack:

```
foo :-
  bar,
  foo.
```

If the recursive call is not the last call, we have a non-tail recursive (or "head recursive") program, such as:

```
foo :-
  foo,
  bar.
```

Such a program will use up local stack very quickly. Unlike the backtrack stack, it is not possible to use "cut" to save local stack space:

```
foo :-
  !,
  foo,
  bar.
```

requires just as much local space as the previous example. The reason is that cut is designed for controlling backtracking on failure, and has no power to control success. Note that the cut itself is treated as a predicate call, and a common mistake is to place one at the end of every clause in a program; this converts tail recursion into head recursion:

```
foo :-
  bar,
  foo,
  !.
```

will use up the local stack, because Prolog must store the call to "!" as the next thing to do after calling "foo".

The Reset Stack

This stack is used to store variables that need resetting during failure. Only variables "older" than the current choice point, as stored on the backtrack stack, need resets to be created during binding. This is because "newer" ones simply vanish on failure. Consider the program:

```
foo :-
  X = 123,
  bar( X, Y ),
  Y = 456.

bar( X, Y ).

bar( X, Y ).
```

Both variables, "X" and "Y", are "declared" at the point "foo" begins to execute. They have the same "age" as whichever choice point (backtrack stack entry) is most recent. As "X" is bound to "123", a test is made to see if the variable is older than the current choice point: it is not, so no reset needs to be made. A call is then made to "bar(X,Y)", which has two clauses, and therefore introduces a choice point, with its entry on the backtrack stack. Next, "Y" is to be bound to "456", and a test is once again made on the variable's age. This time, because a new choice point has been introduced since the variable was declared, a reset stack entry must be made before the variable can be bound.

If a failure subsequently occurs, the most recent choice point is popped off the backtrack stack, returning control to the second clause for "bar(X,Y)". Before execution proceeds, however, the variable "Y" must be reset to its unbound state, by popping the reset stack entry. This way, "bar(X,Y)" will see, as expected, "X" still bound to the value "123", and "Y" as an unbound variable. Because the backtrack stack has been popped, and there are no further choice points in "bar(X,Y)", when control returns to "foo", and "Y" is once again to be bound to "456", the age test will now show that "Y" is the same age as the current choice point, and a reset will this time not need to be made. This is because should a second failure occur, both "X" and "Y" will cease to exist entirely, because "foo" will have failed, and there will have been no point in trying to reset these variables.

In practise, the reset stack can be thought of as vaguely linked to the backtrack stack. If the latter does not grow, then neither will the former; however, it is quite possible to grow the latter without the former.

The System Stack

A few brief words should be said about the system stack. This is the direct equivalent of the stack used by "C", and is used internally by *LPA-PROLOG*'s engine to recurse into structures and call functions. The only time that it is ever an issue is when excessively deep Prolog structures are passed to built-in predicates. For example, the call:

```
?- X = [X], assert( foo(X) ).
```

creates an infinitely deep list structure, and then tries to assert it. In attempting to compile this structure, *assert/1* runs out of system stack space. The other time system stack space can be seen to run out is during term output: when items being written are too deep, the output routine "cheats" by writing three dots and skipping to the tail. Thus the call:

```
?- X = [X].
```

will not generate an error, but will output something like:

```
X = [[[[[[[[[[[[[[[[...]]]]]]]]]]]]]]]
```

Several thousand brackets, rather than just sixteen, normally print before the ellipsis.

Setting Stack Sizes

As mentioned above, the default sizes of the four stacks, at 64kb each, are virtually always sufficient for any non-buggy Prolog program. As a rough guide, 64kb on the backtrack stack allows up to about 1400 choice points to be active at any one time; 64kb on the local stack allows up to about 2100 head recursions before filling up. In the former case, any program with 1400 outstanding choice points is going to take a vast amount of time to complete: each choice point represents at the very least a binary choice, meaning this program is non-deterministically searching a solution space of 2 ^ 1400 or greater complexity! It is essential to add cuts where necessary to cut down the search space to manageable proportions, and in the process, free up the backtrack (and reset) stacks.

As for the latter case, it is always possible to write programs to avoid incrementally creating local stack entries. For example, consider the "traditional" implementation of the factorial function:

```
fact( 0, 1 ).

fact( Number, Fact ) :-
  Less is Number - 1,
  fact( Less, Partial ),
  Fact is Number * Partial.
```

While perfectly "correct", this implementation requires local stack space because the recursive call to "fact(Less,Partial)" is followed by a call to is/2. An alternative implementation uses an additional argument to allow the last two calls to be reversed:

```
fact( Number, Fact ) :-
  fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ).
```

```
fact2( Number, Sofar, Fact ) :-
  Less is Number - 1,
  Next is Number * Sofar,
  fact2( Less, Next, Fact ).
```

The trick here is to pass an initial value ("1") into the program at the start, and to perform the computation on this, simply echoing back the final result. Because the recursive call to "fact2(Less,Next,Fact)" is the last call in the second clause for "fact2", no local stack is used.

It is tempting to make sure that "fact" (or "fact2") is deterministic, by adding a cut. This will prevent endless backtracking after failure, but it is essential to place the cut correctly. In the above program, you could write:

```
fact( Number, Fact ) :-
  fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ) :-
  !.

fact2( Number, Sofar, Fact ) :-
  Less is Number - 1,
  Next is Number * Sofar,
  fact2( Less, Next, Fact ).
```

This would avoid backtracking by ensuring that once the "0" end case was found, no alternatives would be tried. It is not helpful in this case to place a cut in the second clause of "fact2":

```
fact( Number, Fact ) :-
  fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ).

fact2( Number, Sofar, Fact ) :-
  Less is Number - 1,
  Next is Number * Sofar,
  !,
  fact2( Less, Next, Fact ).
```

There are no non-deterministic calls in this clause, and no subsequent clauses, so this use of cut is a waste of time (but no worse). It is, however, distinctly wrong to write:

```
fact( Number, Fact ) :-
  fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ).
```

```
fact2( Number, Sofar, Fact ) :-
  Less is Number - 1,
  Next is Number * Sofar,
  fact2( Less, Next, Fact ),
  !.
```

Not only does this do nothing to help avoid backtracking, but it also makes the recursive call to "fact2(Less,Next,Fact)" head recursive, once again using one local stack frame per iteration.

Ironically, factorial is a bad example for demonstrating local stack usage, because it will cause an "Arithmetic Overflow" error for any values of "Number" greater than 170, and so cannot get near to using up the 2100 or so available stack frames! Other programs however, such as those which add up list elements, can benefit from rewriting with the extra argument:

```
total( [], 0 ).

total( [Head|Tail], Total ) :-
  total( Tail, Rest ),
  Total is Rest + Head.
```

adds up a list such as "[1,2,3]" to return the value "6", but needs one local stack frame for each list element, limiting it to lists of less than about 2100 entries. The same job can be attained without any such limits by adding the extra argument:

```
total( List, Total ) :-
  total2( List, 0, Total ).

total2( [], Total, Total ).

total2( [Head|Tail], Sofar, Total ) :-
  Next is Head + Sofar,
  total2( Tail, Next, Total ).
```

This program runs without using local stack space, and its only limit is the size of list that can be squeezed into the current heap. Other examples include "naive reverse", which is traditionally written:

```
reverse( [], [] ).

reverse( [Head|Tail], List ) :-
  reverse( Tail, Partial ),
  append( Partial, [Head], List ).
```

As well as being inefficient because of the use of append/3, this is not tail recursive, and therefore uses local stack. The three-argument version of "reverse" is preferred:

```
reverse( Data, List ) :-
  reverse2( Data, [], List ).
```

```
reverse2( [], List, List ).

reverse2( [Head|Tail], Sofar, List ) :-
  reverse2( Tail, [Head|Sofar], List ).
```

Here, the extra argument is initialised with an empty list ("[]"); this program is not only tail recursive, requiring no local stack space, but is also linear in performance with respect to the length of the input list, compared with "naive reverse", whose performance relates to the square of the length of the input list. Cuts are completely unnecessary here, thanks to first argument indexing which can distinguish the two cases of "reverse2".

The summary of this section is that, other than in completely exceptional circumstances, it is never necessary to increase the sizes of the backtrack, local, reset or system stacks from their default settings of 64kb each.

Setting Heap Sizes

Unlike the stacks, which should never usually overflow except when application programs contain bugs, heaps are used to store real data that can legitimately grow in size. The main data heap, or just plain "heap", has a default size of 256kb. This is sufficient to hold a single list with about 26100 elements, or ten each with 2610 elements, and so forth. In other words, it can hold a lot of data. While most programs never even get close to requiring this much dynamic term storage, there are valid exceptions. In natural language programs, compilers, chess algorithms and similar, parse trees can be of a considerable size; while portions of these are passed around, extended, manipulated and so on, the overall data burden can grow rapidly. Similarly, predicates such as findall/3, setof/3 and bagof/3, when applied to queries with large solution sets, can use up considerable quantities of heap space.

If a "Heap Space Full" error occurs, the first question that should be considered is, "does this application really need over twenty-six thousand list cells in order to operate?" The answer might be "yes", in which case *LPA-PROLOG* should be run with a higher heap setting, but often the answer will be "no", and the search for "heap traps" should commence. Consider the program:

```
foo :-
  bar( X ),
  sux( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).

sux( X ).
```

At each iteration of "foo", a call is made to "bar(X)", which generates a list of twenty elements, requiring twenty cells, or two hundred bytes, or heap space. A call is then made to "sux(X)", which here simply ignores the data. As the recursive call is made to "foo", all references to the twenty-element list are lost, since there are no choice points, and the list is not being passed as a parameter. The list therefore becomes garbage, and can be collected and recycled in due course as needed. Now consider the program:

```
foo :-
  bar( X ),
  sux( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).

sux( X ).

sux( X ).
```

The only difference is an extra clause for "sux(X)". As we saw earlier, this means that "foo" now uses up backtrack stack entries, as well as local stack entries. But it also uses up heap cells, twenty at a time, because even though the list is not passed as a parameter to the recursive call to "foo", it needs to be kept in case of a subsequent failure, which will cause backtracking into the second clause of "sux(X)". Here is another case where a simple placement of a cut saves on memory usage, including both the above-mentioned stacks and the heap:

```
foo :-
  bar( X ),
  sux( X ),
  !,
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).

sux( X ).

sux( X ).
```

Here, the cut before the recursive call to "foo" removes the backtrack stack entry pointing to the second clause of "sux(X)". Alternatively, the cut could be placed in the first clause of "sux(X)":

```
foo :-
  bar( X ),
  sux( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).
```

```
sux( X ) :-
  !.

sux( X ).
```

This time, "sux(X)" itself prevents backtracking to subsequent clauses. Placing a cut in the clause for "bar([...])" would have no effect, and neither would a cut in the second clause for "sux(X)". Placing one after the recursive call to "foo" would both fail to help backtracking, and require use of the local stack by making this call head recursive.

The text heap is used to store atoms and strings. In the former case, each atom's text is only stored once, and all subsequent occurrences of the atom refer to this same piece of stored text. The first eight characters of a new atom require two cells, or 32 bytes of storage in the text heap, the remaining space after these characters being used for system housekeeping purposes. Each successive 12 bytes require another cell, or 16 bytes of storage. As atoms get longer, up to their limit of 1024 characters, their storage requirement tends towards 4/3 of their length.

Unlike atoms, strings are stored as text each time they occur. The first eight bytes of a string require one text heap cell, or 16 bytes, and then successive characters are stored, as with atoms, at 12 characters per 16-byte text heap cell. Again, as their length increases, the space required to store strings is approximately 4/3 of their length.

Because atoms are stored once each, programs with large numbers of long, but re-used atoms, take up less text heap space than might be imagined.

The same programs, using strings in place of atoms, would require more text space. If the majority of atoms are unique, then the same program would be more space-efficient if it used strings rather than atoms. The real reasons for choosing between atoms and strings should be governed by their intended uses, rather than their storage requirements. Atoms are slower to create than strings, but once created, are more efficiently indexed and stored, and can be used as predicate names, file names, and so on. Strings are very fast to create, but are not used in first-argument indexing; their intended uses are for input and output streams, storage of text messages and prompts, and for handling the contents of files.

As with the (data) heap, unreferenced atoms and strings in the text heap become garbage, and can be collected automatically for recycling of their memory. The only difference between these two heaps in this respect is that atoms and strings may be referenced in programs as well as in dynamic data, and their memory can only be recycled if the respective program clauses or relations are first deleted.

The final heap is for program code itself, and this one is easy to understand. Each clause that is added to a dynamic program, and each entire optimised predicate, occupies one or more 128-byte cells in the program heap. This heap is not "garbage collected" as such, because when code is deleted, its memory is directly recycled there and then.

In general, text and program heap space needs to be allocated in step: larger programs tend to contain larger numbers of unique atoms and strings. The default settings of 512kb and 2048kb respectively for these two heaps allows for some fairly large programs to be loaded and run, but these allocations are by no means excessive, and should be freely increased as needed. To see how much space a program requires, a call can be used such as:

```
?- gc( 1 ),
  free( _, _, _, _, T0, P0, _, _, _ ),
  consult( foo ),
  gc( 1 ),
  free( _, _, _, _, T1, P1, _, _, _ ),
  TextUsed is T0 - T1,
  ProgUsed is P0 - P1.
```

The calls to "gc(1)" invoke the garbage collector manually, simply to ensure that the returned values of free text space in the calls to "free(...)" are maximized for measurement purposes. The amount of text and program space available both before and after loading the program are returned, and their differences computed. Whatever results are returned for "TextUsed" and "ProgUsed" reflect the minimum text and program space required by the file "foo" itself, not taking into account the requirements of any other modules that might have been loaded, or of the system itself. The predicate free/9 is matched with another, called total/9, that returns the total settings, so an overall text and program heap memory requirements can be computed with the call:

```
?- gc( 1 ),
  total( _, _, _, _, T0, P0, _, _, _ ),
  free( _, _, _, _, T1, P1, _, _, _ ),
  TextUsed is T0 - T1,
  ProgUsed is P0 - P1.
```

The returned values here are the absolute minimal required to load all current program modules and the system itself. Setting the text and program heap size to about 256kb and 1024kb above these values respectively should give most applications ample working space.

Command Line Switches

Having discussed the stacks and heaps in *LPA-PROLOG*, and explained why most of them should be left at their default sizes, it is time to describe how their memory allocations are changed. A series of nine command line switches refer to each of the four stacks, three heaps and two buffers, as listed below:

Switch	Description	Size	Notes
/Bnnn	Backtrack Stack	64kb	normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug

/Lnnn	Local Stack	64kb	normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug
/Rnnn	Reset Stack	64kb	normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug
/Hnnn	(Data) Heap	256kb	this can be increased modestly, but seldom needs to exceed this size (most applications can run with a far smaller heap); the apparent need for a huge (>1024kb) heap should be considered as an indication of possible application bugs
/Tnnn	Text Heap	512kb	this should be increased in line with the size of the application and its data set, and as a rough guide should be about 25% the size of the program heap
/Pnnn	Program Heap	2048kb	this should be increased in line with the size of the application and its data set, and as a rough guide should be about 400% the size of the text heap
/Snnn	System Stack	64kb	normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug
/Innn	Input Buffer	64kb	this is the maximum size of string that can be used as an input stream, and can be adjusted to whatever size suits the application
/Onnn	Output Buffer	64kb	this is the maximum size of string that can be used as an output stream, and can be adjusted to whatever size suits the application

When *DOS-PROLOG* starts up, the current settings (in kb) for these nine memory areas, minus any system overheads, can be viewed on the banner:

```
LPA DOS-PROLOG 3.600 - S/N 0000000000 - 29 Aug 1997
Copyright (c) 1997 Logic Programming Associates Ltd
This Special Version Is For The Personal Use Of BDS
B=64 L=64 R=64 H=255 T=500 P=2017 S=63 I=64 O=64 Kb
```

Summary

In this document, we have looked at a number of aspects of the "black art" of Prolog memory management, and discovered that, apart from two of the three heaps, in most cases, default settings for stack and heap sizes should be left unchanged. Before increasing stack or data heap sizes, application should be checked for inefficient structures and other bugs. For each of the following errors, the corresponding checks should be made:

Error	Check
Backtrack Stack Full	make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls
Local Stack Full	make sure there are no unwanted head recursions: search for any recursive calls which are followed by further calls, and rearrange code as necessary
Reset Stack Full	make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls, and if still a problem, try to bind variables before the introduction of choice points in a given clause
Heap Space Full	decide whether the application really needs tens of thousands of data elements: if not, make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls, and avoid passing large list parameters around if not necessary
Text Space Full	after checking that this is not simply a heap related problem, restart Prolog with an increased text heap in line with the complexity of the application, perhaps using the free/9 and total/9 predicates to obtain a measure of how much space is genuinely required
Program Space Full	restart Prolog with an increased program heap in line with the complexity of the application, perhaps using the free/9 and total/9 predicates to obtain a measure of how much space is genuinely required
System Stack Full	check the application for incorrect unifications which result in infinitely deep structures, and correct them
Input Space Full	too long a string has been used as an input stream: modify the application or restart Prolog with an input buffer which reflects the requirements

Output Space Full	too long a string is being created as an output stream: modify the application or restart Prolog with an output buffer which reflects the requirements

Hopefully, the explanations, examples and "cookbook" solutions above will help demystify memory management under ***WIN-PROLOG***, and help readers write code which runs comfortably without the need for huge and wasteful memory resources.

Appendix C - Types of Compilation

This appendix discusses the differences between the three types of compilation, namely incremental, hashed and optimised compilation in *WIN-PROLOG*, and indicates why and when each of these types of compilation should be used.

Incremental Compilation: Clause by Clause

The term "incremental compilation" is used to describe a process where each individual program clause is compiled independently, without reference to other clauses. In *WIN-PROLOG*, incremental compilation is performed by handwritten 80386 assembler code, and is extremely fast. First argument indexing and unification instructions are highly optimised, even in the incremental compiler, making it ideal for use in small to medium sized data relations (collections of facts), whether or not these need to be modified dynamically at run time.

Program relations (collections of rules, with or without recursion) are less efficiently compiled, but still run considerably faster than would be possible using an interpreter.

A special feature of incrementally compiled programs is that they can be incrementally decompiled, back into their original source form; this allows full support not only of *assert/1* (compilation), but also *clause/2*, *retract/1*, and so forth. Apart from this flexible run-time support, incremental compilation also fully supports the *WIN-PROLOG* debuggers.

Hashed Compilation: Instant Access

The term "hashed compilation" is a slight misnomer, as hashing is actually a reversible post-process applied to incrementally compiled programs. Introduced in version 4.200 of *WIN-PROLOG*, this feature allows the creation of a highly optimised hash table for any incrementally compiled predicate that has no variable or variable-headed structure cases in its first argument. The benefit of hashing on small to medium sized relations is relatively slight, but it really comes into its own on large relations containing thousands, or even millions, of clauses.

When an incrementally compiled predicate is hashed, each of its unique first arguments is counted and built into a table with a user-defined amount of headroom, specified by a "fudge factor", or percentage excess. The greater the fudge factor, the fewer the number of mishits are likely to occur during hashing, resulting in a play-off between efficiency and memory requirements. A default setting of 100% gives excellent performance in most circumstances. Because hashing is reversible, it is possible to try different settings on any given relation quickly to find out an optimal fudge factor. Although a hashed relation cannot be modified with *assert/1* or *retract/1*, it is still fully accessible through *clause/2*, and so forth; furthermore, because hashing can be removed as easily as it has been added, a relation revert to its incrementally compiled status for any such modifications, before (optionally) rehashing. Hashed compilation also fully supports the ***WIN-PROLOG*** debuggers.

Optimised Compilation: Relation by Relation

The term "optimised compilation" is used to describe a process where each entire relation (collection of clauses for a given predicate and arity) is compiled into a single piece of code. In ***WIN-PROLOG***, optimised compilation performed by a program is written in Prolog which, although somewhat slower at compiling than the incremental compiler, carries out sophisticated analysis on data and control flow within the program, eliminating redundant data transfers and optimising data traffic in general. The optimising compiler can perform multiple argument indexing, and special high speed jump instructions can convert tail recursion into a conventional program loop. Data relations do not normally benefit from being optimised, unless multiple argument indexing is used, but program relations are invariably faster and more space efficient.

Because the optimised compiler converts an entire program relation into a single, monolithic piece of executable code, it is not able to support *assert/1*, *clause/2*, *retract/1*, or the ***WIN-PROLOG*** debuggers. On the other hand, since optimised code cannot be decompiled, it provides a level of security for program source code.

First Argument Indexing

The argument indexing used in ***WIN-PROLOG*** is fairly extensive, but there are some minor differences between the incremental, hashed and optimised compilers in this respect. Firstly, the incremental compiler handles a few more indexing cases than the hashed compiler, which in turn handles a few more than the optimising compiler; secondly, on the other hand, the optimising compiler can index on multiple arguments, which the incremental and hashed ones cannot.

All three compilers index on the type of the first argument; further, if the argument is an atom or an integer, they index on the value. When the first argument is any kind of compound term (tuple, list, conjunction or disjunction), the incremental compiler indexes on the type of its first element, and if this is an atom, on the value. The optimising compiler only performs indexing on the value of an atom in the first element of a compound term, and does not index on the other types when at the head of a term. The following table summarises the indexing capabilities:

Indexing Type	Inc	Hsh	Opt
argument type	yes	yes	yes
argument atom value	yes	yes	yes
argument integer value	yes	yes	yes
tuple head type	yes	yes	no
tuple head atom value	yes	yes	yes
list head type	yes	yes	no
list head atom value	yes	yes	yes
conjunction head type	yes	yes	no
conjunction head atom value	yes	yes	yes
disjunction head type	yes	yes	no
disjunction head atom value	yes	yes	yes
indexing on 1st argument	yes	yes	yes
indexing on other arguments	no	no	yes

The Comparison: Head to Head

The three compilers in ***WIN-PROLOG*** provide complementary, rather than competing services; in an application, it will typically be desirable to use a mix of all types of compiled code. A key consideration is that both incremental and optimising compilers implement indexing through a form of case statement, where successive table entries are checked sequentially, while the hashed compiler implements a true hash table that can give direct access to the clause required. In small or medium relations, the performance improvement is fairly small, and may be offset by other considerations (such as the desire to modify the data relation dynamically, or to disguise the source code by optimising it); in large data relations, the improvement offered by hashing can be very dramatic. The following table summarises the main features and properties of the three compilers:

Feature	Inc	Hsh	Opt
suitable for small data relations	yes	no	yes
suitable for medium data relations	yes	yes	no
suitable for large data relations	no	yes	no

suitable for fast program relations	no	no	yes
supports assert/retract	yes	*	no
supports clause/decompile/listing	yes	yes	no
supports debuggers	yes	yes	no
builtnoin to run time kernel	yes	yes	no
provides source code security	no	no	yes

* because it is possible to remove and reapply hashing at any time, a hashed relation can be modified simply by converting it back into an incremental relation, performing the desired updates, and then rehashing it.

Appendix D - Associating *WIN-*PROLOG Files With The Application

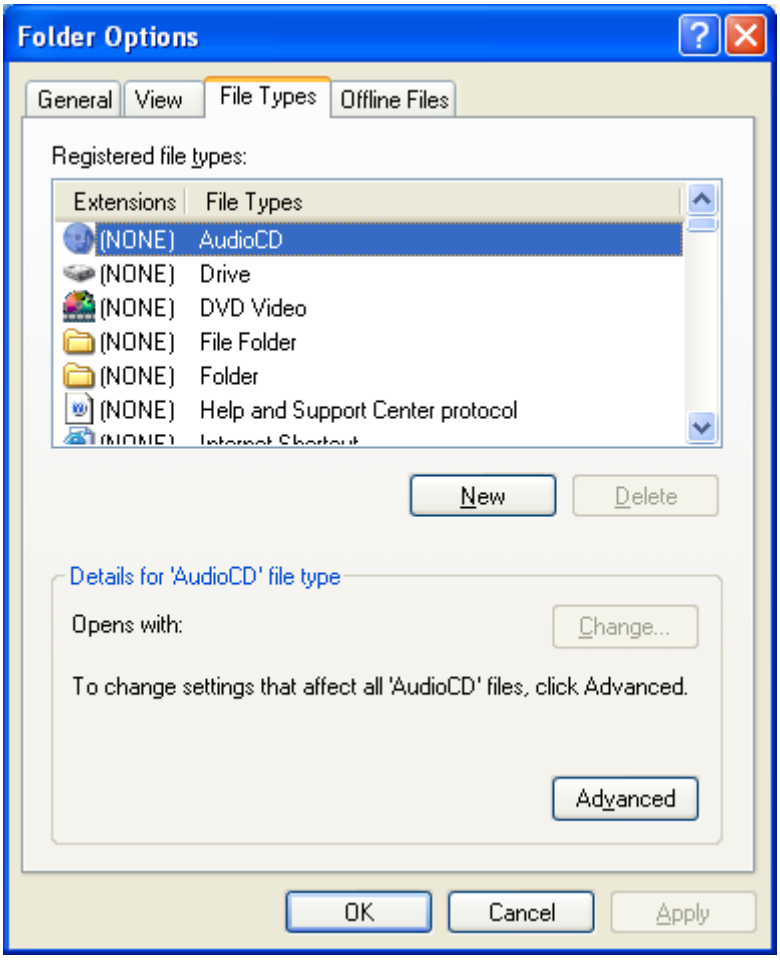
This appendix shows you how to set up Windows so that whenever you double-click on any .PL, .PC, .KSL or .VRL file, it is opened or loaded into *WIN-*PROLOG, flex or VisiRule.

Select "Options..." from the "View" menu

Double click on the "My Computer" icon. From the window that opens (either the "My Computer" folder or the "Windows Explorer" window) open the 'Tools' menu and select the "Folder Options..." entry.

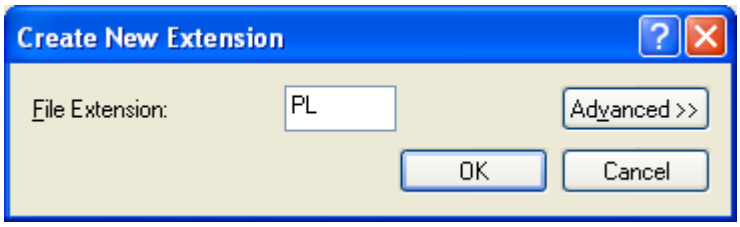
Select the "File Types" Tab

From the "Folder Options" dialog that appears, select the "File Types" tab.

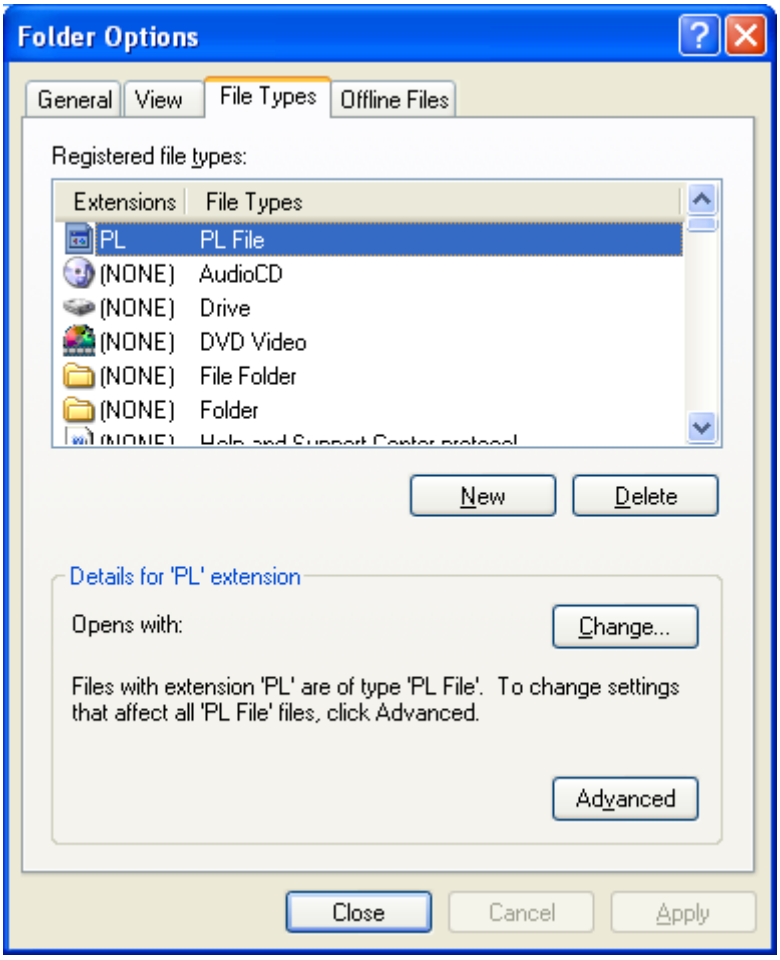


Create a New File Type Entry For .PL (Prolog Source) Files

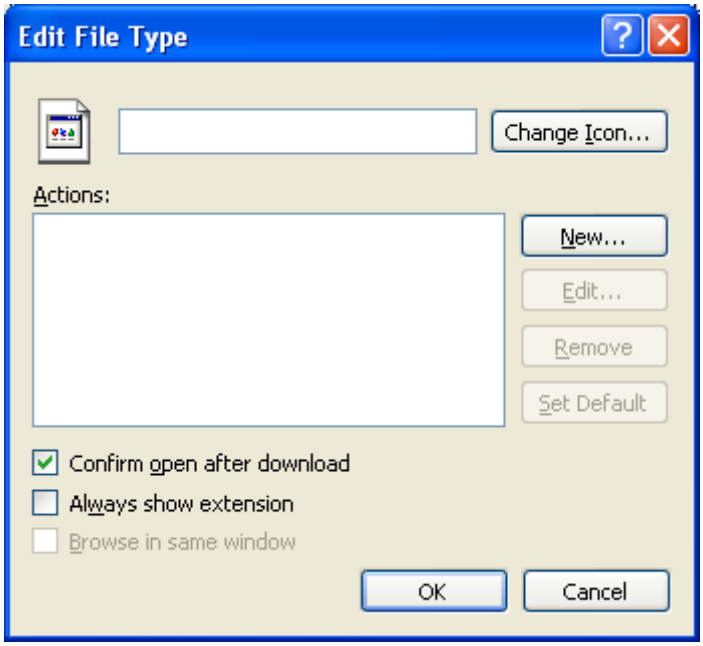
Within the Folder Options dialog, click on the "New" button to create a new file type entry for .PL files. When the "Create New Extension" dialog appears, enter "PL" in the "File Extension:" field and then click OK.



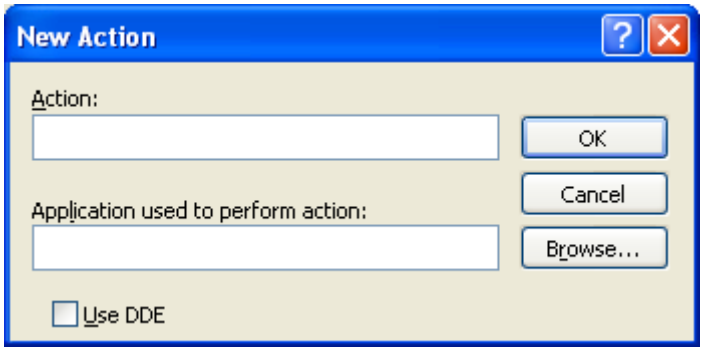
Within the Folder Options dialog, locate and select the PL entry in the 'Registered file types:' list.



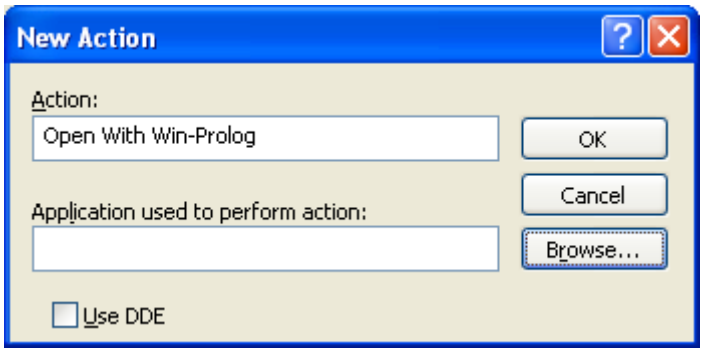
Click on the Advanced button to display the Edit File Type dialog.



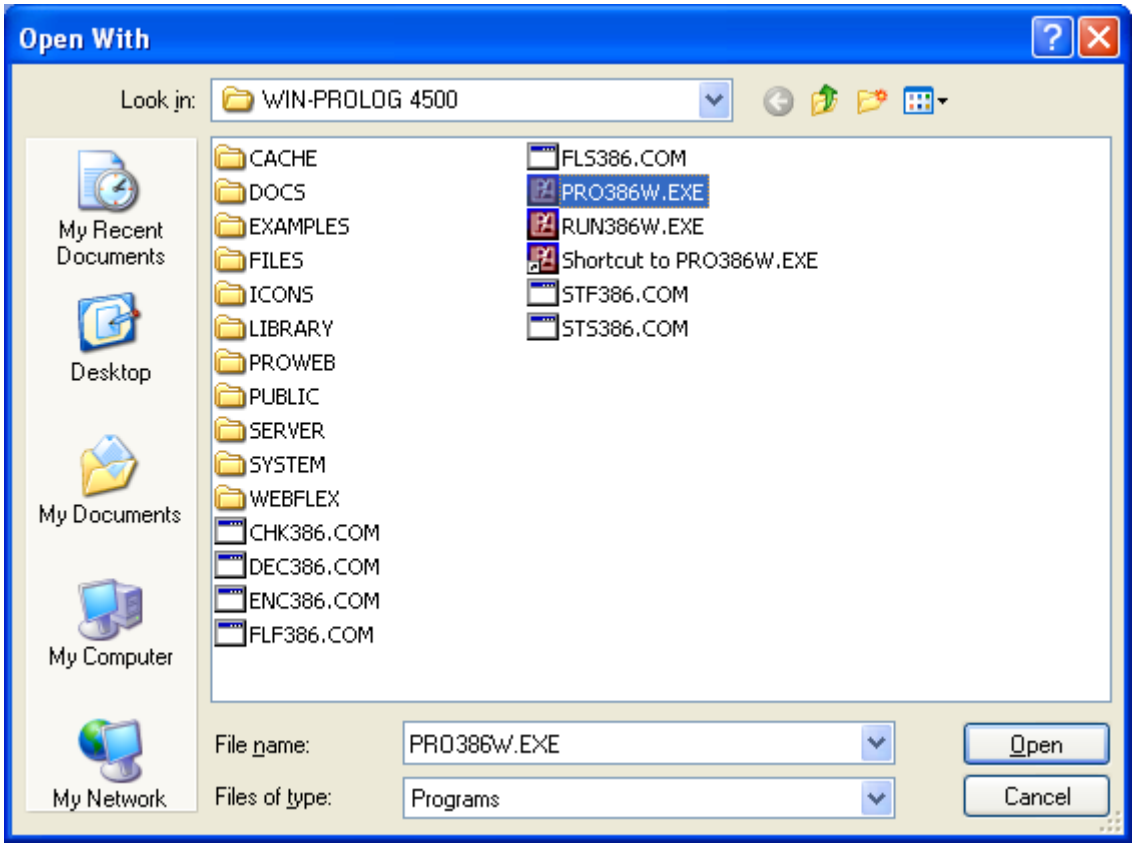
Click on the 'New...' button to display the 'New Action' dialog.



Enter into the 'Action:' field something like 'Open With Win-Prolog', 'Load With Win-Prolog' or whatever you want the textual name of the action to be; this will eventually become an entry in the menu that appears when you right-click over a .PL file.



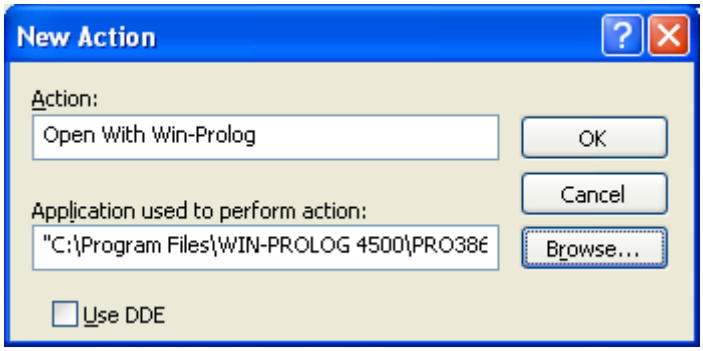
Click on the 'Browse' button to locate and select the PRO386W.EXE file.



You should now have:

"c:\program files\win-prolog 4900\pro386w.exe"

in the 'Application used to perform action:' field.



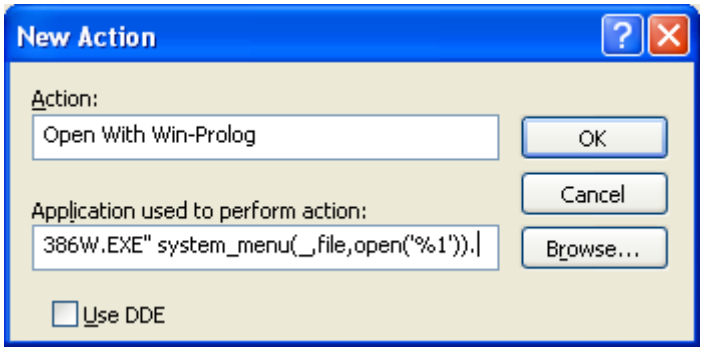
Click in the 'Application used to perform action:' field and navigate to the end of the line and append one of the following to the existing text:

```
system_menu(_,file,open('%1')).
```

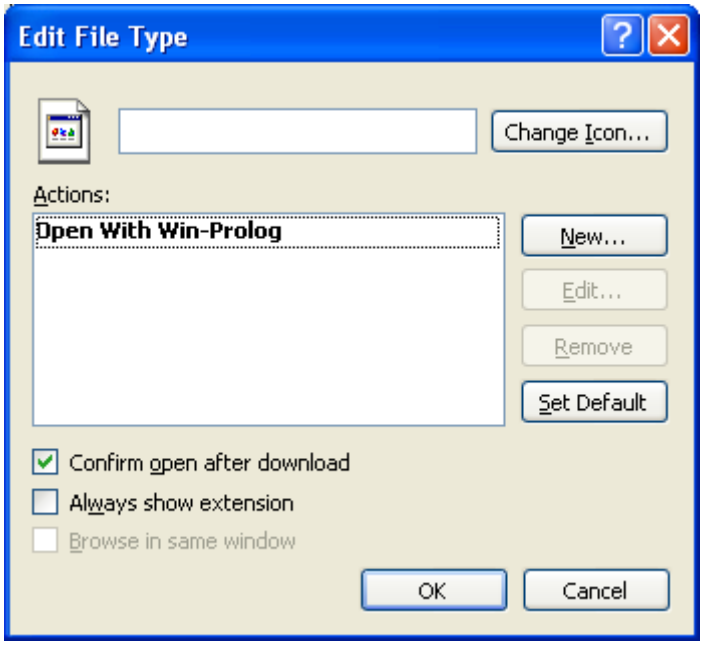
(if you want to open the .PL file into an edit window within the *WIN-PROLOG* development environment) or

```
system_menu(_,file,load('%1')).
```

(if you want to load the .PL file directly into *WIN-PROLOG*'s internal memory) ensuring that you leave a space character between the bit in double quotes and the Prolog goal.



The bit in double quotes (i.e. "c:\program files\win-prolog 4900\pro386w.exe") is the path to your installation of *WIN-PROLOG*. The double quotes must be included if spaces are present in the path or file name. The "%1" in "system_menu(_,file,open('%1'))." will get replaced by the full pathname of the .PL file you later double click on. Click the "OK" button. The action you have just added should appear in the 'Actions' list on the Edit File Type dialog.



If you want to add another 'action' such as the 'Load With Win-Prolog' one as well as the 'Open With Win-Prolog' one, click on the New button.

When finished, click OK on the Edit File Type dialog. Click Close on the Folder Options dialog.

Create a New File Type Entry For .PC (Prolog Object) Files

Within the Folder Options dialog, click on the "New" button to create a new file type entry for .PC files. When the "Create New Extension" dialog appears, enter "PC" in the "File Extension:" field and then click OK. Within the Folder Options dialog, locate and select the PC entry in the 'Registered file types:' list. Click on the Advanced button to display the Edit File Type dialog. Click on the 'New...' button to display the 'New Action' dialog. Enter into the 'Action:' field something like 'Load With Win-Prolog' or whatever you want the textual name of the action to be; this will eventually become an entry in the menu that appears when you right-click over a .PC file. Click on the 'Browse' button to locate and select the PRO386W.EXE file. You should now have:

```
"c:\program files\win-prolog 4900\pro386w.exe"
```

in the 'Application used to perform action:' field. Click in the 'Application used to perform action:' field and navigate to the end of the line and append the following to the existing text:

```
system_menu(_,file,load('%1')).
```

which will load the .PC file directly into *WIN-PROLOG*'s internal memory, ensuring that you leave a space character between the bit in double quotes and the Prolog goal. The bit in double quotes (i.e. "c:\program files\win-prolog 4900\pro386w.exe") is the path to your installation of *WIN-PROLOG*. The double quotes must be included if spaces are present in the path or file name. The "%1" in "system_menu(_,file,load('%1'))." will get replaced by the full pathname of the .PC file you later double click on. Click the "OK" button. The action you have just added should appear in the 'Actions' list on the Edit File Type dialog. Click OK on the Edit File Type dialog. Click Close on the Folder Options dialog.

Create a New File Type Entry For .KSL (Flex) Files

Within the Folder Options dialog, click on the "New" button to create a new file type entry for .KSL files. When the "Create New Extension" dialog appears, enter "KSL" in the "File Extension:" field and then click OK. Within the Folder Options dialog, locate and select the KSL entry in the 'Registered file types:' list. Click on the Advanced button to display the Edit File Type dialog. Click on the 'New...' button to display the 'New Action' dialog. Enter into the 'Action:' field something like 'Open With Flex', 'Load With Flex' or whatever you want the textual name of the action to be; this will eventually become an entry in the menu that appears when you right-click over a .KSL file. Click on the 'Browse' button to locate and select the PRO386W.EXE file. You should now have:

```
"c:\program files\win-prolog 4900\pro386w.exe"
```

in the 'Application used to perform action:' field. Click in the 'Application used to perform action:' field and navigate to the end of the line and append one of the following to the existing text:

```
ensure_loaded(system(flexenv)), system_menu(_,file,open('%1')).
```

(if you want to open the .KSL file into an edit window within the *WIN-PROLOG* development environment) or

```
ensure_loaded(system(flexenv)), system_menu(_,file,load('%1')).
```

(if you want to load the .KSL file directly into *WIN-PROLOG*'s internal memory), ensuring that you leave a space character between the bit in double quotes and the Prolog goal. The bit in double quotes (i.e. "c:\program files\win-prolog 4900\pro386w.exe") is the path to your installation of *WIN-PROLOG*. The double quotes must be included if spaces are present in the path or file name. The "%1" in "system_menu(_,file,load('%1'))." will get replaced by the full pathname of the .KSL file you later double click on. Click the "OK" button. The action you have just added should appear in the 'Actions' list on the Edit File Type dialog. If you want to add another 'action' such as the 'Load With Flex' one as well as the 'Open With Flex' one, click on the New button. When finished, click OK on the Edit File Type dialog. Click Close on the Folder Options dialog.

Create a New File Type Entry For .VRL (VisiRule) Files

Within the Folder Options dialog, click on the "New" button to create a new file type entry for .VRL files. When the "Create New Extension" dialog appears, enter "VRL" in the "File Extension:" field and then click OK. Within the Folder Options dialog, locate and select the VRL entry in the 'Registered file types:' list. Click on the Advanced button to display the Edit File Type dialog. Click on the 'New…' button to display the 'New Action' dialog. Enter into the 'Action:' field something like 'Open With VisiRule' or whatever you want the textual name of the action to be; this will eventually become an entry in the menu that appears when you right-click over a .VRL file. Click on the 'Browse' button to locate and select the PRO386W.EXE file. You should now have:

```
"c:\program files\win-prolog 4900\pro386w.exe"
```

in the 'Application used to perform action:' field. Click in the 'Application used to perform action:' field and navigate to the end of the line and append the following to the existing text:

```
ensure_loaded(system(visirule)), system_menu(_,file,open('%1')).
```

which will load the .VRL file into a VisiRule window within the *WIN-PROLOG* development environment, ensuring that you leave a space character between the bit in double quotes and the Prolog goal. The bit in double quotes (i.e. "c:\program files\win-prolog 4900\pro386w.exe") is the path to your installation of *WIN-PROLOG*. The double quotes must be included if spaces are present in the path or file name. The "%1" in "system_menu(_,file,open('%1'))." will get replaced by the full pathname of the .VRL file you later double click on. Click the "OK" button. The action you have just added should appear in the 'Actions' list on the Edit File Type dialog. Click OK on the Edit File Type dialog. Click Close on the Folder Options dialog.

INDEX