MLWorks™

User Guide

Windows Version 2.0

Copyright and Trademarks

MLWorks™ User Guide

July 1998

Part number: MLW-2.0.0-UG-WIN

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
U.K.

telephone +44 1223 873 800
fax +44 1223 872 519

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
U.S.A.

telephone +1 617 374 2400
fax +1 617 252 6505

Electronic Access:

http://www.harlequin.co.uk
http://www.harlequin.com

Contents

1

Using MLWorks interactively

1.1 Introduction

This chapter introduces the MLWorks interactive environment and many of the programming tools it provides.

1.2 Starting up the MLWorks interactive environment

Start the MLWorks interactive environment with the **`mlworks`** shortcut on the Windows **Start** menu (or wherever you installed it).

1.3 The podium

Figure 1.1 The MLWorks podium.

When you start MLWorks, it draws the *podium* or *console window* on your screen. This window provides a number of menus and a toolbar. You cannot type ML expressions into the podium. To do that you will need a *listener*, an

MLWorks programming tool that is explained in Section 1.4. You can create listeners and other tools from the **Tools** menu of this or any other MLWorks window.

The podium will be present throughout your MLWorks session; closing the podium with the **File > Exit** menu item exits MLWorks.

1.4 Introducing the listener

The first thing you need to run MLWorks interactively is a listener, a tool that is automatically createdwhen you run MLWorks. This tool gives you an ML command line at which you can type expressions.

You can also create a listener by choosing **Tools > Listener** from the podium. A listener appears, with the following prompt:

```
MLWorks>
```

Figure 1.2 The MLWorks listener.

You can exit any listener window with **File > Close** or with the function **`Shell.exit`**.

At the heart of the listener is a *read-eval-print loop* which reads input typed at the prompt, evaluates it, and prints the result. If you type an expression, the result is just the value of the expression:

```
MLWorks> (7 * 4) - 3;
val it : int = 25
MLWorks>
```

Remember to terminate each line of input with the Return (or Enter) key — in this example, after the semi-colon. MLWorks does not read the line until it is terminated in this way, or equivalently by pressing the **Evaluate** button at the bottom of the listener.

Note: The MLWorks system compiles *all* code, never interpreting as some other languages, such as Lisp, do. The listener, with its Lisp-like read-eval-print operation model, may seem like an interpreter, but in fact everything it accepts is evaluated by being compiled down to native code.

Note: In Standard ML, an expression encountered at top level is treated as an implicit declaration, binding the identifier **`it`** to the given expression. From here on, when we refer to a "declaration" typed in the listener, we include an expression typed at top level. The result of evaluating an expression, shown above, is in fact just a special case of the evaluation of a declaration, which we examine next.

If you type an explicit declaration in the listener, MLWorks binds the identifier to the given value and verifies the declaration and its type. For example, we can bind the identifier **`five`** to the value 5, declare an exception **`Fact`**, and define **`fact`** to be the usual factorial function:

```
MLWorks> val five = 5;
val five : int = 5
MLWorks> exception Fact;
exception Fact
MLWorks> fun fact 0 = 1
           | fact n = if n<0 then raise Fact
                      else n * fact (n-1);
val fact : int -> int = fn
MLWorks>
```

The last example shows another feature of the listener, namely that it accepts declarations typed over multiple lines. It reads and parses lines in turn as they are typed, but it does not attempt to evaluate a declaration until it encounters

the semi-colon that terminates the declaration. When MLWorks reads a line containing the terminating semi-colon, it interprets everything since the last evaluation as a declaration (or series of declarations).

Once a value has been bound to an identifier, that binding holds in your MLWorks session unless it is overridden by a later binding.

The binding can be used in subsequent declarations:

```
MLWorks> fact five;
val it : int = 120
MLWorks> val five = 6;
val five : int = 6
MLWorks> fact five;
val it : int = 720
MLWorks>
```

If you type a declaration in the window and press the **Time** button, MLWorks will evaluate the declaration as normal, and also give a line of information about how long the evaluation took. For much more complicated profiling of ML functions, you can use the MLWorks profiler tool, invoked with the **Profile** button. There is also programmatic interface to the profiler in the **`MLWorks.Profile`** and **`Shell.Profile`** structures: for details, see the *MLWorks Reference Manual*.

1.5 More on interacting with MLWorks

To quote from the *Definition of Standard ML* (Milner, R.; Tofte, M.; and Harper, R.; 1990, The MIT Press); "the Definition" hereafter:

> ML is an interactive language, and a *program* consists of a sequence of *top-level declarations*; the execution of each declaration modifies the top-level environment, which we call a *basis*, and reports the modification to the user.

In MLWorks, we call this top-level environment a *context* rather than a "basis", to avoid confusion with the Standard ML Basis library. Additionally, we reserve the term *environment* to describe the MLWorks interactive system, as invoked with the **`mlworks`** shortcut; we never use the term "environment" in the way the Definition uses it in the quotation above.

There are two kinds of contexts to distinguish between: the *interactive context* and the *batch context*. The interactive context is, essentially, what is available to you when you are running the MLWorks interactive environment, while the batch context is what is available to a program being compiled using the MLWorks batch compilation system, which allows you to compile ML code for execution outside the interactive environment — and hence, outside the interactive context. There are important differences between the two contexts which we discuss in Chapter 2, "Building Applications", which describes the batch compilation system.

1.6 Repeating and editing input

The listener has extensive support for editing and re-evaluating previous input. This can be done by moving about the listener window, and by using the listener's built-in history mechanism.

While you are typing a declaration, you can delete characters using the Delete key, and move the cursor using the cursor keys to insert or delete characters at any point in the current input. (Anything before the current **`MLWorks>`** prompt may not be modified.) When you press the Return key (or the **Evaluate** button on the listener), MLWorks reads the declaration and evaluates it.

You can use this to repeat an earlier one-line declaration, by moving the cursor to the line in the listener window containing the complete declaration and hitting Return (or, as usual, the **Evaluate** button). MLWorks copies the declaration to the current prompt at the bottom of the window and reads it.

Parts of the window may also be selected with the mouse and cut, copied or deleted using the relevant entries on the **Edit** menu. Cut or copied items may be pasted elsewhere by positioning the cursor and using **Edit > Paste**.

1.6.1 Keyboard shortcuts

Note that you can use a some common keystrokes to move around the window besides the cursor keys:

Home	If the insertion cursor is after the prompt, move it to the start of the current input. Otherwise move it to the beginning of the line.

End	If the insertion cursor is after the prompt, move it to the end of the buffer. Otherwise move it to the end of the line.
Delete	If the insertion cursor is after the prompt, delete the character to the right of the insertion cursor. If typing input to the standard input of an evaluating expression, and the insertion cursor is at the start of the last line, send an end-of-file indication to that stream. Otherwise do nothing.
Ctrl+X	An accelerator for the **Edit > Cut** menu item. Text before the current prompt may not be cut.
Ctrl+C	An accelerator for the **Edit > Copy** menu item.
Ctrl+V	An accelerator for the **Edit > Paste** menu item. Text may not be pasted before the current prompt.
Shift+cursor keys	Move cursor, selecting text between start and end point.
Esc+P	Replace the input with the previous history item (same as the **Previous** button).
Esc+N	Replace the input with the next history item (same as the **Next** button).
Tab	If the insertion cursor is after the prompt, complete the preceding item. If the item begins with a ", treat it as a file name; otherwise treat it as an identifier. If the insertion cursor is before the prompt, do nothing.
Page Up	Move the cursor up a screenful.
Page Down	Move the cursor down a screenful.

Return — If the insertion cursor is after the prompt, evaluate the current input. If the current input is not a complete top-level declaration, ending in a semicolon, do nothing. If the insertion cursor is before the prompt, copy the current line (excluding any prompt) to the end of the buffer and evaluate the current input.

Ctrl+Return, Ctrl+J (line feed) — Insert a new line without evaluating the input.

Also, double-clicking selects an entire word or line, respectively.

1.6.2 The History menu

A more flexible way of repeating declarations is to use the listener's **History** menu. This menu is a list of all declarations previously accepted by the listener; the most recent declaration is listed at the top.

When you select a declaration from the list, it appears at the current prompt (at the bottom of the listener window), as originally formatted — possibly over multiple lines. You may now edit it, before pressing Return, at which point it is evaluated.

Instead of using the menu, you can use the **Previous** and **Next** buttons at the bottom of the listener to move up and down the history list. You can control the maximum number of history items in the list for your listener with the **Usage > General Preferences > General** menu item.

The **Clear** button clears the current prompt (equivalent to the Ctrl+U keystroke), but not the history list, which it leaves unchanged. The **Abandon** button functions like **Clear**, but it stores the contents of the cleared prompt on the history list, from where it may be recovered later.

Finally note that you can use MLWorks' tab-completion to save typing. Pressing the Tab key when part-way through a word causes MLWorks to complete the word as an existing identifier, if this can be done uniquely. For example, consider the following:

```
MLWorks> val a_long_identifier = "Hello, World.";
val a_long_identifier : string = "Hello, World."
MLWorks> a_l
```

At this point on the line, we press the Tab key, and as there is only one identifier beginning '**a_l**', MLWorks completes it:

```
MLWorks> a_long_identifier
```

If there is more than one possible completion, MLWorks brings up a completion dialog with a line for each possible completion. You may select one, in which case it is inserted at the prompt in the usual way, or press the **Cancel** button on the dialog. Either action causes the dialog to disappear. Alternatively, if you simply carry on typing in the listener, the completion dialog will disappear.

If the identifier you start typing is a structure name, tab-completion will complete the structure name, allowing you to start typing the qualified name you need from within the structure. If it is a string, MLWorks will try to complete it as a filename; this is useful when using a function such as **use** (see the next section).

If completion dialogs are not to your taste, you can switch them off using the **Usage > General Preferences > General** menu item on the MLWorks podium.

1.7 Writing and reading source files

You may want to save the code you have so far typed into the listener to a file. You can do so by going to the menu bar, and choosing **File > Save Listener Input As...**. MLWorks pops up a file dialog with which you can choose a directory and a filename in which to save the source. Give the filename the suffix **.sml** to indicate that it contains Standard ML code; MLWorks will only treat files with the **.sml** suffix as source files. You can then edit the file as you would any other program, with a text editor.

If you have already used **File > Save Listnener Input As...** in this MLWorks session, you can now simply choose **File > Save Listener Input** instead, and MLWorks will update the file with any new source that you have entered since the last save.

Conversely, you may want to read ML code from a file in to your current context. There are various ways to do this; the simplest is to use the listener's **File > Read Into Listener** command. This produces a file dialog, and MLWorks will read code in from the file you choose as if it had been typed in to the listener

window. The listener verifies all the declarations in the file, which must have a **.sml** suffix, in the usual way. In the example below, the file **lists.sml** was used; it contains definitions of a couple of simple list functions, **length** and **flatten**:

```
MLWorks> use "lists.sml";
val it : unit = ()
Use: lists.sml
val length : 'a list -> int = fn
val flatten : 'a list list -> 'a list = fn
MLWorks>
```

Note: The **File > Read Into Listener** command, in conjunction with the error-handling and editing facilities detailed in the next section, provides a handy way of interactively managing fairly small pieces of ML code. However, for larger or more complex sets of ML files, you should use the project compilation system, documented in Chapter 2. The other entries on the listener's **File** menu are related to the project system.

You can save the state of your entire MLWorks session to a file with the Podium's **File > Save Session As...** command. MLWorks will prompt you for a filename in which to save the image, which is just a normal MLWorks image file. To restart the MLWorks session from Windows, use the **mlimage-console.bat** script. This script is installed by default in the **bin** subdirectory of your MLWorks installation.

```
mlimage-console <filename> [ <args> ]
```

The **<***filename***>** must be an MLWorks image (**.img**) file. The optional **<***args***>** are arguments to the image file, not to **mlimage-console.bat**.

1.8 Handling compilation errors

It may happen that you type in to the listener a declaration which causes the compiler to detect a fault. It could be a type mismatch, or an unbound identifier, and so on (unhandled exceptions are dealt with in another way — see "Tracking down unhandled exceptions" on page 22).

Specifically, suppose you enter the following:

```
MLWorks> val a = if b>7 then 9 else false;
```

This line is wrong in two ways: the integer **b** has not been defined, and the whole expression is ill-typed. MLWorks alerts you to the error by bringing up a new window, the error browser.

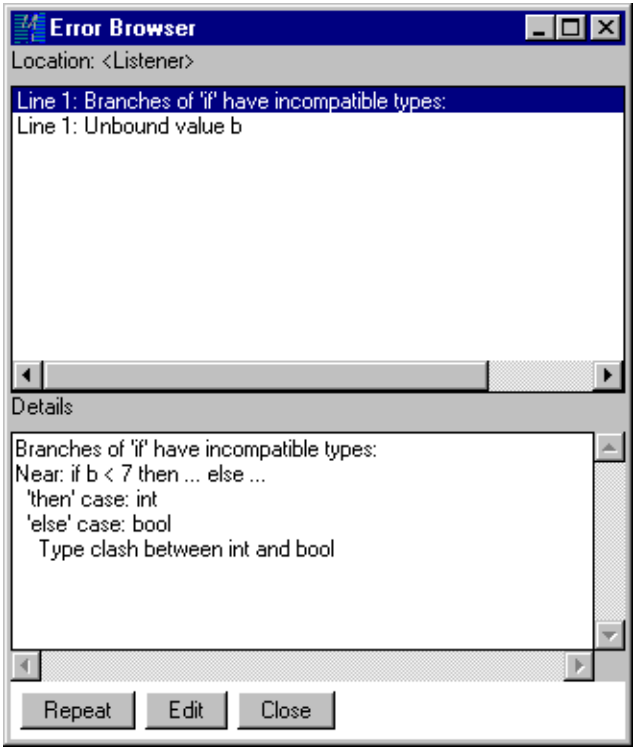


Figure 1.3 The MLWorks error browser.

Note: If you do not like the error browser, you may switch it off in the **Usage > General Preferences > General** menu on the MLWorks podium. Errors will then be reported directly in the listener window.

The error browser has two windows. The top window shows a list of the errors that have been encountered. Any of these may be selected, and the bottom window shows more details, if any are available, of the selected error. In this case, the error browser is initially showing details of the type clash it has detected, including what the two different types are that it cannot unify. If we select the other error in the top window (‘**`Unbound value b`**’), there is no more

to say about it, and the error browser simply repeats this information in the lower window.

You can thus look through all the errors in turn, editing the declaration in the listener window as you go, before trying to evaluate the declaration again. You can do this in the usual way (pressing the Return key or the **Evaluate** button), or by pressing **Repeat** on the error browser window. MLWorks tries to evaluate the line as amended, and the error browser disappears. The **Close** button closes the error browser, copying the error report to the listener and giving a new prompt.

The error browser's remaining button, **Edit**, causes the part of the declaration where the error was detected to be highlighted in the listener window.

1.9 Editing ML files

When MLWorks identifies an error in an ML file, one of the options it makes available is to load the file into your chosen editor at the line containing the error. To specify the editor you want to use, select **Usage > General Preferences > Editor ...** from the menu bar. The resulting dialog gives you several ways of specifying an editor:

- External editor. You can use any "one shot" editor such asthe Windows editor **`Wordpad`** (the default external editor), or any editor that uses DDE (Dynamic Data Exchange) to conduct ongoing communications with other applications, for example PFE 32.

  Use the External Editor text box to specify a command that would invoke a suitable editor if you typed it at the operating system command line (that is, in an MS-DOS window). When you ask MLWorks to edit a file, it starts a new window in which the given command line, with the name of the file to be edited appended, is executed. If the command line includes **`%l`**, MLWorks will substitute an appropriate line number.

  The Windows editor **`Wordpad`** is the default editor, and is assumed to be in **`c:\Program Files\Accessories\`**. This is the default location for the editor usder Windows 95, but Windows NT 4.0 users may have to specify **`c:\Program Files\Windows NT\Accessories\`** as the editor location.

- Custom editor. Custom editors are defined in the same way as the External Editor, but MLWorks allows you to record them in a table

keyed by custom editor names that you supply. You can choose a custom editor in the Custom Editor Name box by its name. See Section 1.9.1.

Note: Your choice of editor, like other settings and preferences selected from the **Usage** menu, can be saved by selecting **Usage > Save Preferences**. MLWorks writes the details of your preferences to a file called `.mlworks_preferences` in your home directory (your home directory is specified by the environment variable `HOME`. To see or change this environment variable, see the System view in the control panel). The preferences file will be read in on starting subsequent MLWorks images.

Suppose that, having set your editor to something acceptable, you invoke **File > Read Into Listener** to run a file which contains an error. MLWorks reports the error in the listener, and brings up the error browser.

```
MLWorks> use "error.sml";
val it : unit = ()
Use: error.sml
error.sml:3,9 to 3,11: error: Unbound value boo
```

All errors in the first erroneous declaration in the file will be found, and they may be browsed in the usual way. The actions available are the same as when any error is encountered, with one exception: the **Edit** button on the error browser will start up your chosen editor, passing it the line number containing the error currently selected in the error browser.

Alternatively, you can position the cursor on the line in the listener where the location of the error is given, and select **Edit > Edit Error**. Again, MLWorks will start up your chosen editor at the line containing the error. This method allows you to edit a file containing an error even when you have switched off error browsers.

Editor settings and preferences can be controlled programmatically by the `Shell.Editor` and `Shell.Options.Preferences` structures. For full details, see the *MLWorks Reference Manual*.

1.9.1 Custom editors

As stated above, custom editors are similar to external editors because they are defined with a command string. However, a custom editor also has:

- a name
- a *connect dialog*

You can use the custom editor's name in the **Usage > General Preferences > Editor** dialog to specify that you want to use it. The *connect dialog* is not a window-system dialog, but a list of commands that are sent to the editor when MLWorks establishes contact with it. When MLWorks wants to connect to an (existing) remote editor, it first establishes contact and then sends the editor a sequence of commands down the channel that has been established. These commands typically direct the editor to load a source file at a specific position.

Note that connect dialogs are not necessary for all custom editors you define; most editors cannot accept commands issued in this way.

See the section on the **`Shell.Editor.Custom`** structure in the *MLWorks Reference Manual* for details of building scripts to access custom editors.

1.10 Examining ML values

You can look at the details of an ML value using the *inspector*. To see it in action, define a value as follows:

```
MLWorks> val bar = ("hello", [[1,2], [3,4] @ [5]]);
val bar : (string * int list list) = ("hello", [[1,2], [3,4, 5]]);
MLWorks>
```

Now choose **Usage > Inspect**. MLWorks brings up an inspector.

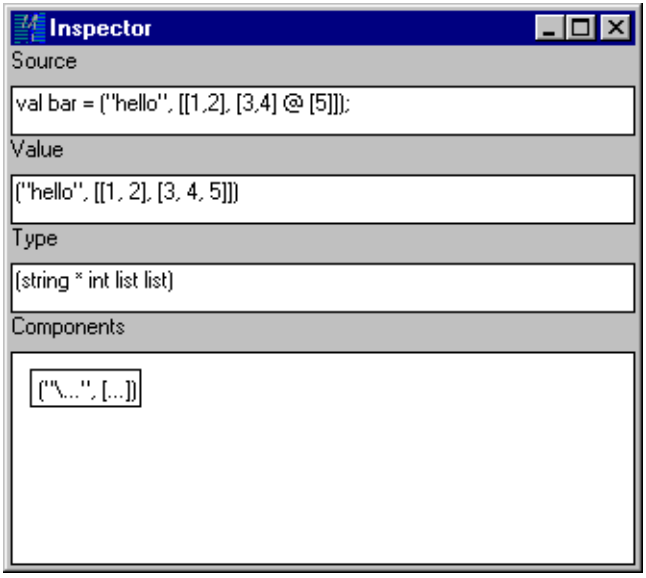


Figure 1.4 The MLWorks inspector.

The inspector contains details of the item last defined in the listener before it was invoked.

The inspector has several fields: the Source field shows the declaration MLWorks read which defined the value being examined, and the Value and Type fields show its value and type, as you would expect.

The Components field shows a graphical representation of the value. In the current example this is not very informative; however, it can be expanded to show the structure of the value. Double-click on the value to expand it one level, showing the components. The second component is a list, and can be further expanded in the same way. It may be convenient to resize the inspector window using the window manager.

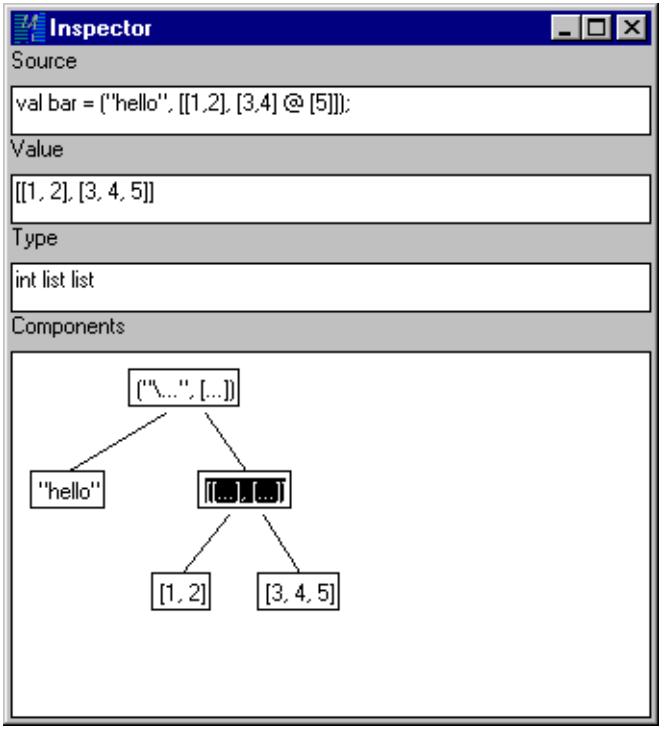


Figure 1.5 The inspector's graphical representation of a value.

By invoking the inspector from the listener you can only inspect the most recently defined item. Thus, if you already had an inspector on screen, it would not instantly inspect the declaration of **bar**.

One way to fix this is to check **Usage > Auto-Update Tool**. This makes the inspector show values as they are defined from then on. (So you would still have to re-evaluate the declaration of **bar** to get it to appear.)

Another way to inspect an older item is to look at it in the *context browser*. To do this, select **Tools > Context Browser**. The context browser appears.

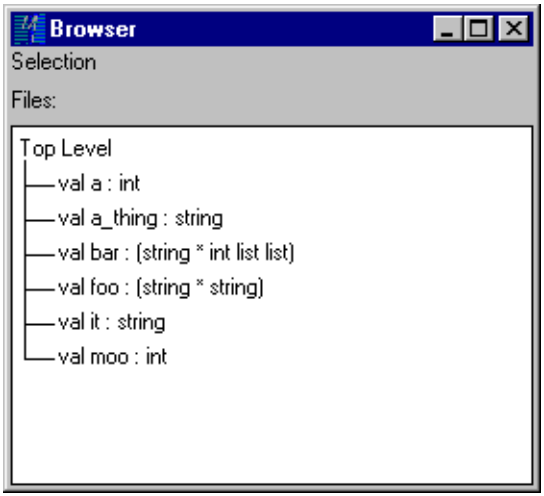


Figure 1.6 The MLWorks context browser.

It contains an up-to-date list of all the bindings you have created in the interactive environment. To inspect any item, click on it in the list, and then choose **Usage > Inspect** from the menu bar.

Similar to the context browser is the *system browser*. With this tool you can browse the built-in parts of MLWorks. To do this, select **Tools > System Browser**. The system browser appears.

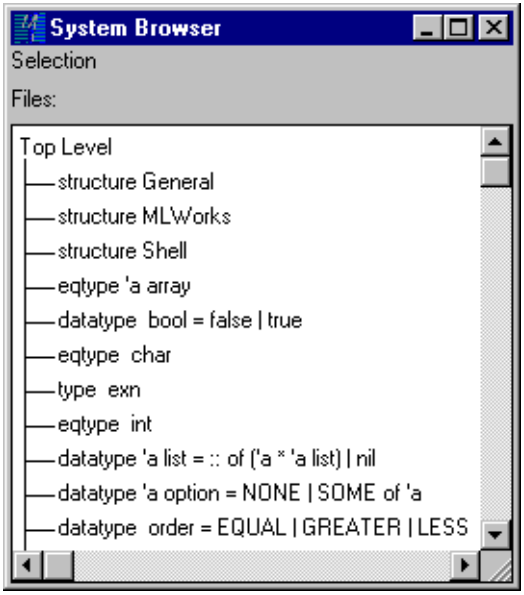


Figure 1.7 The MLWorks system browser.

With the system browser you can browse the built in **MLWorks** and **Shell** structures, and the Standard ML Basis library **General** structure. There are other identifiers available unqualified at top level, such as the **bool** datatype and the ***** function, which are as defined by the Standard ML Basis library — with the exception of the function **use**, which we saw in Section 1.7 as the function invoked by the listener's **File > Read Into Inspector** menu item. The **use** function is defined by **Shell**.

If you want to see what is inside a particular structure, double-click on it. The system browser shows you the contents of the structure. To see this in action, double-click on **Shell**.

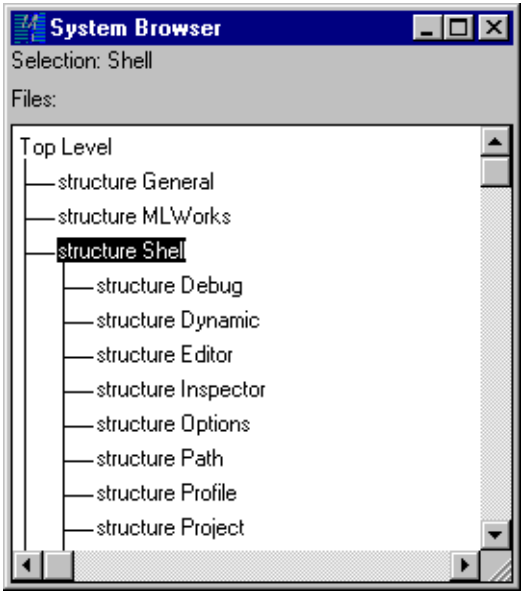


Figure 1.8 Browsing the **`Shell`** structure.

You can browse the substructures of **`Shell`** by further double-clicking.

Every time you click on an item, the system browser puts the full qualified name of that item in the Selection field. If you want to use the full qualified name of an item somewhere, you can save typing it out by selecting it in the system browser and copying it with **Edit > Copy**, or with mouse selection, for subsequent pasting in another tool such as a listener.

For more details of the system browser, context browser, and inspector, see Chapter 4, “Examining Objects”.

1.11 Searching for ML identifiers

Sometimes, you know the name of an identifier that you want to use, but you do not know where in a library to look for it. Other times you want to see if a library contains an item that will do what you want, but you can only guess at a name for the item. These situations are typical when working with large libraries that are organized into a hierarchy of many structures.

For instance, you may know that there is a function called **use** built in to MLWorks, but you do not know where it resides in the large **MLWorks** and **Shell** structures.

The listener's Search dialog is useful here. To invoke the Search dialog on the system browser, ensure that it has the focus and choose **Usage > Find** from the menu bar. The Search dialog appears.

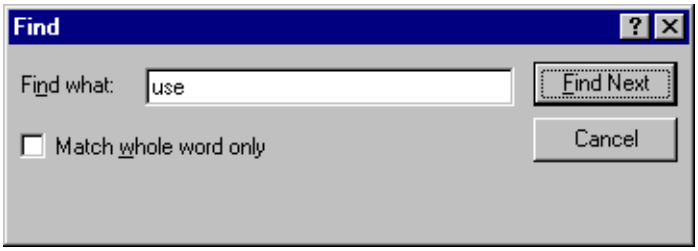


Figure 1.9 The listener's Search dialog.

You can enter the identifier you are interested in in the **Find what** text box. When you click **Find Next**, MLWorks searches the interactive environment and comes up with a list of matches. As well as returning exact matches, MLWorks will also return identifiers that start with what you entered. Note that trailing whitespace is counted as part of the search item.

MLWorks returns the results of the search in the tool being serched. The following dialog was generated by a search for **use**:

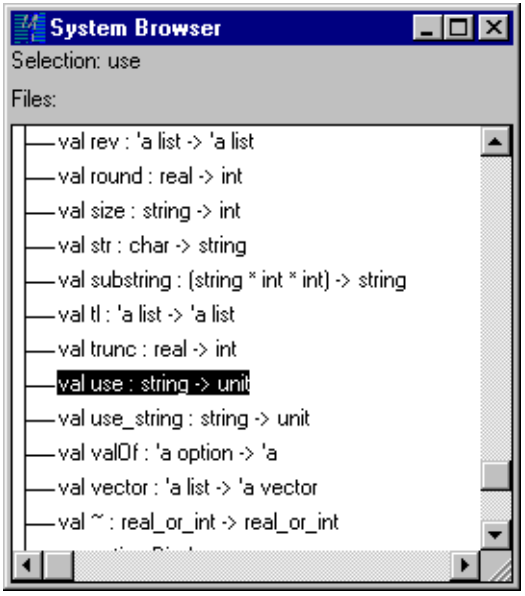


Figure 1.10 Results of a search for the identifier **use**.

Ensure the listener has the focus, and select **Usage > Find**. Note that this time the search dialog has some extra options.

Figure 1.11 The listener search dialog.

You can tailor the search by checking relevant options in the search dialog. Checking **Signatures** ensures that signatures are searched, and checking **Functors** ensures functors and structures are searched.

You can get more information about a matching identifier by setting the **Display Types of Entries** option. This option makes entries in the Matches dialog appear with their full ML type information.

The final two checkboxes control the following aspects of your search:

Search System Context

Search for identifiers in the built-in structures (**`MLWorks`**, **`Shell`**, **`General`**, and if the basis library is loaded, **`basis`**) and in the built-in identifiers available unqualified at top level (for example, the **`real`** function). The search area is equivalent to what is visible in the system browser.

Search User Context

Search for identifiers created in the current interactive session. This includes those in any libraries you have loaded into the MLWorks interactive environment. The search area is equivalent to what is visible in the context browser.

If you know where your identifier is likely to be found, you can reduce the search time significantly by choosing only one of the **Search User Context** and **Search System Context** options.

1.12 Tracking down unhandled exceptions

This section gives a very brief introduction to the MLWorks *stack browser*. The stack browser appears whenever the compiled code causes an unhandled exception to be raised; it allows you to see the function calls on the stack at the point when the exception occurred.

Before using the stack browser, it is important to turn on debugging mode; this will ensure that the necessary debugging information is generated when code is compiled, so that it can later be passed to the stack browser. To turn on debugging mode, choose **Usage > General Preferences > General** from the menu. In the mode options dialog, check the **Use debugger** and **Always use window debugger** check boxes and click **OK**.

To invoke the stack browser, you need to raise an unhandled exception. Define and call the following less useful variation on the factorial function:

```
MLWorks> exception Fract;
exception Fract
MLWorks> fun fract 0 = raise Fract | fract n = n * fract (n-1);
val fract : int -> int = fn
MLWorks> fract 5;
```

On the call to **`fract 5`**, MLWorks enters the stack browser.

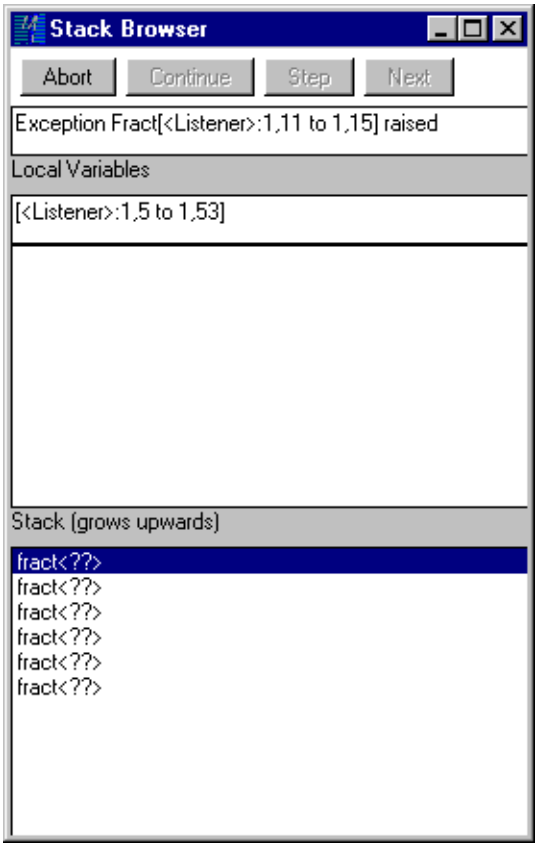


Figure 1.12 The MLWorks stack browser.

The box at the top contains an error message which tells you what exception was raised, and the bottom window contains a list of the function calls leading up to the unhandled exception. Selecting one of these with the mouse brings up information about that function call in the middle pane of the stack browser.

The **Abort** button closes the stack browser and returns control to the listener.

For more details of the stack browser, see Chapter 5, “Debugging”.

1.13 The MLWorks start-up file

When MLWorks starts up, it looks for a file called **.mlworks** in your home directory (which is given by the environment variable **HOME** — see saving preferenced in Section 1.9), and if this exists, evaluates its contents as ML declarations and modifies the context accordingly. You could use this file to pre-load libraries, for instance, so that you do not need to load them by hand at the beginning of each session.

MLWorks will print the result of evaluating the contents of the file at the command line where it was invoked:

```
$ mlworks
Use: .mlworks
val twice : int -> int = fn
```

Here, the **.mlworks** file defines a function **twice**:

```
fun twice n = 2 * n;
```

1.14 Printing extra information in the listener

As we have seen, the listener has a read-eval-print loop, meaning that it always prints the results of evaluating the expressions you type into it. Evaluation always causes a given identifier or **it** to be bound to the value computed, thus the result printed for all evaluations is the value and the type of the value bound.

MLWorks allows you to control the form of this printed result, and also to print more information in the result of each evaluation, via the **Usage > Tool Settings** menu. Other tools in the interactive environment also use this menu which allows you to change the view in those tools.

As an introduction to these controls, we now look at how to change the form of a certain kind of result: sequence values. To do this we must go to the dialog produced by the **Usage > Tool Settings > Value Printer** menu item. The dialog presents a column of text and check boxes with which you can control various settings of the *value printer* that is used to print out results. The settings in this dialog can also be controlled using the structure **Shell.Options.ValuePrinter**, which is built in to the interactive context. For

full details of this (and other, similar options), see the *MLWorks Reference Manual*.

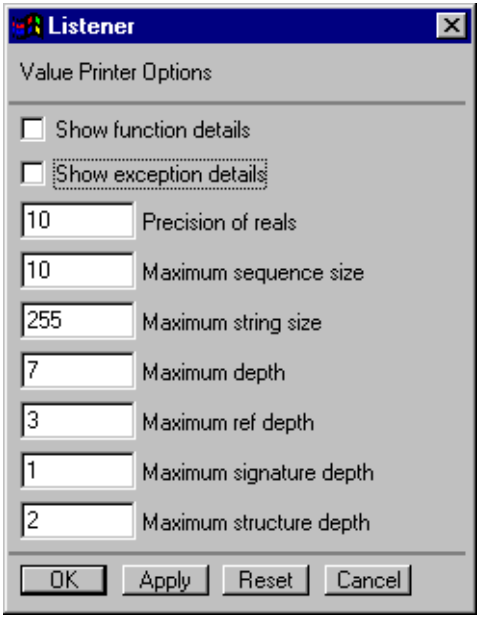


Figure 1.13 The Value Printer Options dialog for the listener.

After changing value printer settings in the Value Printer Options dialog, you can apply them with the **Apply** button. **Reset** returns the settings to what they were when you last clicked **Apply**. If you have not yet clicked **Apply**, it returns the settings to what they were when you invoked the dialog.

The **Maximum sequence size** box controls the maximum number of elements to print for sequence values. Sequences longer than this are elided. The sequence size must be an integer; the default is 10.

With **Maximum sequence size** set to 3:

```
MLWorks> val l = [ 1, 2, 3, 4, 5, 6 ];
val l : int list = [ 1, 2, 3, ..]
MLWorks>
```

You can also invoke the Value Printer Options dialog in the inspector and in the compilation manager (which we shall see in Chapter 2, “Building Applica-

tions"). Notice that when you do so, the title of the dialog shows which tool owns it.

This concept of ownership means that changes to values do not take place throughout the GUI environment — different tools keep their own value-printing settings. Thus, if you set **Maximum sequence size** to 3 in the listener, the value-printing behavior for sequence values will be not changed in any other tools you have created.

The inspector is a little different: though you may make duplicates of it (with **Usage > Duplicate Tool**), MLWorks considers there to be only one inspector tool, and so all copies of it use the same value-printing values. What is more, the inspector does not inherit the value-printing values of the listener from which it was invoked, so inspecting list `l` by choosing **Usage > Inspect** in a listener with elided list value printing does not produce an elided list in the inspector's Value field.

2

Building Applications

2.1 Introduction

MLWorks provides a powerful project compilation system which, in conjunction with the programming tools we have seen so far, helps to support a variety of application development styles. This chapter explains how to construct and manage application projects with MLWorks.

2.1.1 Different styles of application development

MLWorks supports a range of application-development styles. You can work solely in the interactive environment in its GUI and TTY forms. The interactive environment, as we have seen, features a number of different programming tools for profiling, debugging, value inspection, and so on. By contrast, you can work entirely outside the interactive environment and use MLWorks as a batch compiler system from the MS-DOS command line with the **`mlbatch.bat`** script.

You can also invoke the compiler in batch mode from within the interactive environment. This means you can use a mixture of development styles during the same project, because whether you are working in the interactive environment, or solely with the batch compiler, the compilation system at the heart of MLWorks functions in the same way. There is, however, one important dis-

tinction between the batch and interactive contexts here, which we will see in Section 2.6.

2.1.2 Introducing the MLWorks project system

Application sources are typically decomposed into a collection of separate source files. These files may depend on one another in complex ways, in which case when you ask MLWorks to compile a file, it also compiles all the files that your file depends on. MLWorks handles this using a project file which is used to compile ML source files to object files in an organized and flexible way.

When it compiles a source file, MLWorks produces an object file, which contains an internal representation of the objects defined by the code; and it can then load those objects into the interactive environment. If an object file already exists, MLWorks does not need to compile the source file; it can just load the objects directly from the object file.

Using object files is much faster than re-compiling every file whenever you change it or re-start MLWorks. However, it would be inconvenient if you had to keep track of object files and remember which ones needed re-compiling. The project system does this tracking for you, only compiling a given source file if it has been changed since the last time it was compiled, or if it has never been compiled before.

The MLWorks project system also provides two different notations for describing the locations of source and object versions of your application's files. Using relative path names allows you to compile different versions of the application files stored in matching directory structures by simply copying the project file to a different location, whereas using absolute path names provides a fixed reference to the application files, regardless of where the project file is saved. Both relative and absolute path name specifications are discussed in greater detail in Section 2.3.2.

2.1.3 Building a standalone version of your application

Once development is over you can build a fast, compact, standalone executable version of your application. The *delivery* mechanism offers the usual advantages of a batch-compiled language such as C, even though you can

invoke the delivery mechanism from within the interactive environment. The delivery mechanism links a compact ML runtime into an executable with your application code.

2.2 Expressing source dependencies

To allow efficient compilation of the source files that make up an application, there must be some way to express dependencies between the files. MLWorks accomplishes this by extending Standard ML with an additional reserved word, **`require`**, which can be used in source files to make a top-level declaration of the form

```
require "<source-file>";
```

This tells the compiler that the present file uses values or functions defined in *<source-file>*, which should therefore be compiled first. Note that require statements no longer specify any path or location information, just the name of the required file without the **`.sml`** extension. As a result of this, require statements are not used to find source files and these can only be found by searching the list of files specified as part of the project. See Section 2.3.2, “Setting the project’s required files” to find out how to specify required file directory information.

Note: In future versions of MLWorks the **`require`** function will be deprecated in favor of an automated system for determining the dependencies of a set of source files using the project system.

The **`require`** reserved word takes a single string argument, which is the name of an MLWorks *unit* (for the moment, you can think of a unit name as being the name of a source file without the **`.sml`** extension). Any **`require`** declarations *must* be at the start of the file, before any other top-level declarations, and the argument must be a string constant, not just an ML expression of type **`string`**.

The effect of a **`require`** declaration is to import the declarations from the specified file into the current file. For example, suppose you have three files, **`xval.sml`**, **`yval.sml`** and **`sumxy.sml`**, all in the same directory, as follows:

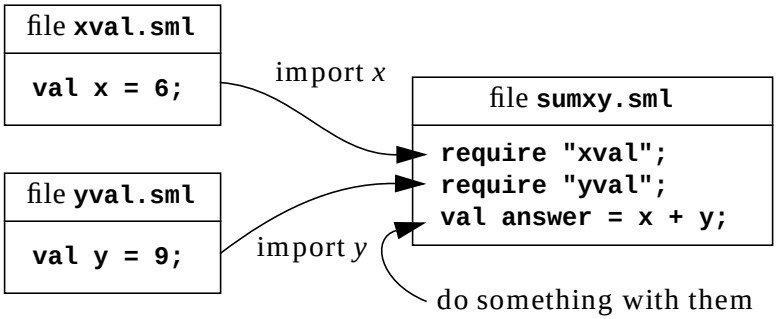


Figure 2.1 A program comprised of three files.

Then the effect of the file **sumxy.sml** is to bind the identifier **answer** to the value 15.

Note: A file may use declarations from files it imports with **require**, but this relation is not transitive. Suppose you change **xval.sml** to require another file, **zval.sml**, to compute its value of *x*:

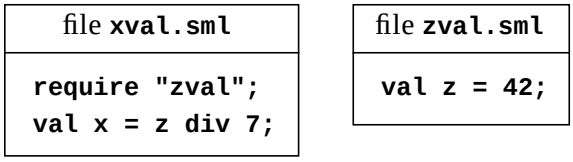


Figure 2.2 Another dependency is introduced.

Then **sumxy.sml** could still use the value of *x*, but not the value of *z* from **zval.sml**, as it does not explicitly require the latter file. In this respect, the MLWorks **require** extension is rather like the ML construction

```
local <declarations> in … end
```

2.3 Basic use of the MLWorks project system

This section introduces the use of the MLWorks compilation mechanism. It concentrates on how to use the GUI's *project system*, but equivalent program-

matic access to the same features is available from the **`Shell.Project`** structure. See the *MLWorks Reference Manual* for more details.

2.3.1 Creating a project file

Before we can set about compiling a set of ML code files, we need to set up a new project using the *project workspace*. Choose **File > New Project** from the menu bar. You are prompted for an initial working directory for the new project — choose **`examples\projects`** for the purposes of this example — and then the project workspace appears.

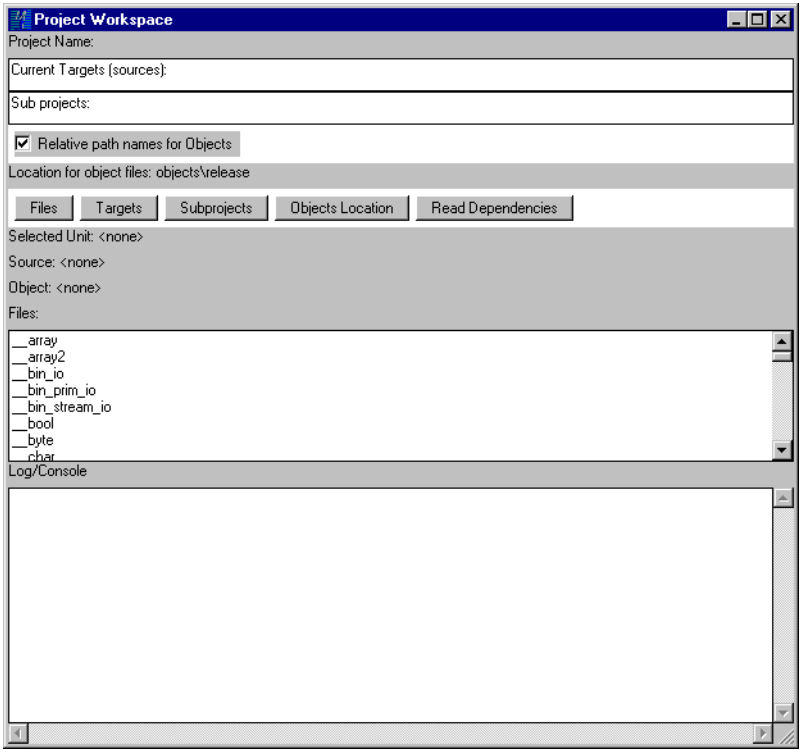


Figure 2.3 The MLWorks project workspace.

The project workspace consists of three main areas. The top third contains details of the various directories and files the project system refers to during the compilation process, and includes a set of buttons that call up dialogs for

setting these directories and files. Note that the text areas listing the targets and subprojects are read only — these values are actually set using project properties dialogs. The text areas can be selected and copied, however, and the scroll bar can be used to view the entire contents of the fields if they extend beyond the border of the project workspace window.

The middle third of the project workspace provides the details of individual object and source files, and includes a list of these files. Currently this list is empty, as we have not yet added any target source files.

The bottom third of the project workspace contains a log of the actions that have taken place in the project system so far.

Save the new (empty) project in the **`examples\projects`** folder by selecting **File > Save Project** from the menu bar to call up the Save As dialog, and navigate to the relevant folder. Specify the name **`myproject`** in the file name text area and click on **OK**. MLWorks automatically adds the **`.mlp`** suffix for you — all valid MLWorks project files must have this suffix. From now on you can save any changes to your project file by selecting **File > Save Project** from the menu bar.

2.3.2 Setting the project's required files

To begin with we need to specify the required files for the project. This tells the project manager where the target source files for the project are located. Click on **Files**. The "Project Properties - Files" dialog appears.

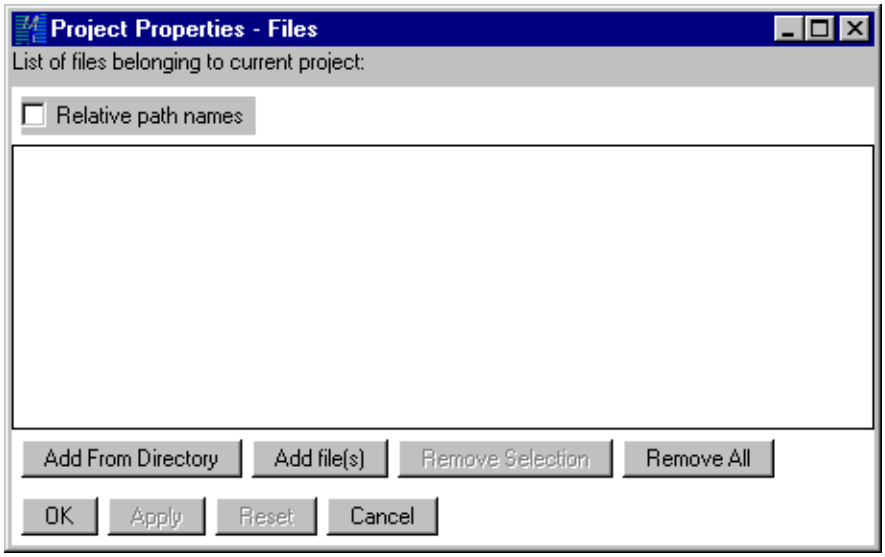


Figure 2.4 Project Properties - Files.

The source files for the example are in the **`examples\projects`** folder, so click on **Add From Directory**, and select this folder from the file selection dialog. This adds all the **`.sml`** files in this directory to the project as source files. Clicking on **Add File(s)** button allows you to add individual files as opposed to a whole directory.

If the **Relative path names** checkbox is unchecked, then the full path name for the **`examples`** folder is given in the dialog's path name list, and the target source files will be taken from this directory regardless of where the project file is saved. If the **Relative path names** checkbox is checked, then path names are set relative to the folder in which the project file is saved. The advantage of using relative path names is that you can move the project file to a different place where the directory structure is the same, and hence use the project to build the application using different source files, for example ones which print diagnostics. The advantage of using absolute path names is that you can move the project file anywhere, and it will still refer to the same source files.

In our example the project file is saved in the **`examples\projects`** folder, so checking the checkbox collapses the path name of the **`examples\projects`**

folder to a '.'. For our example it does not matter whether relative path names are used or not.

Now that the source path has been selected, click **OK** in the dialog. The project source path becomes visible in the project source path field of the project workspace.

Note that you can add as many path names to the project source path as you need. This allows you to compile interdependent source files located in as many different folders as you like.

2.3.3 Adding target source files

Now that we have set the source path of the project we need to specify the target source files. These are the top-level **`.sml`** files that contain the ML code to be compiled. In this example we use the files **`xval.sml`**, **`zval.sml`**, **`yval.sml`** and **`sumxy.sml`** as described in Section 2.2. They can be found in the **`examples\projects`** folder in the MLWorks installation folder. In our example, only one target source file needs to be specified, namely **`sumxy.sml`**. Click on **Targets** to call up the "Project Properties - Targets" dialog.

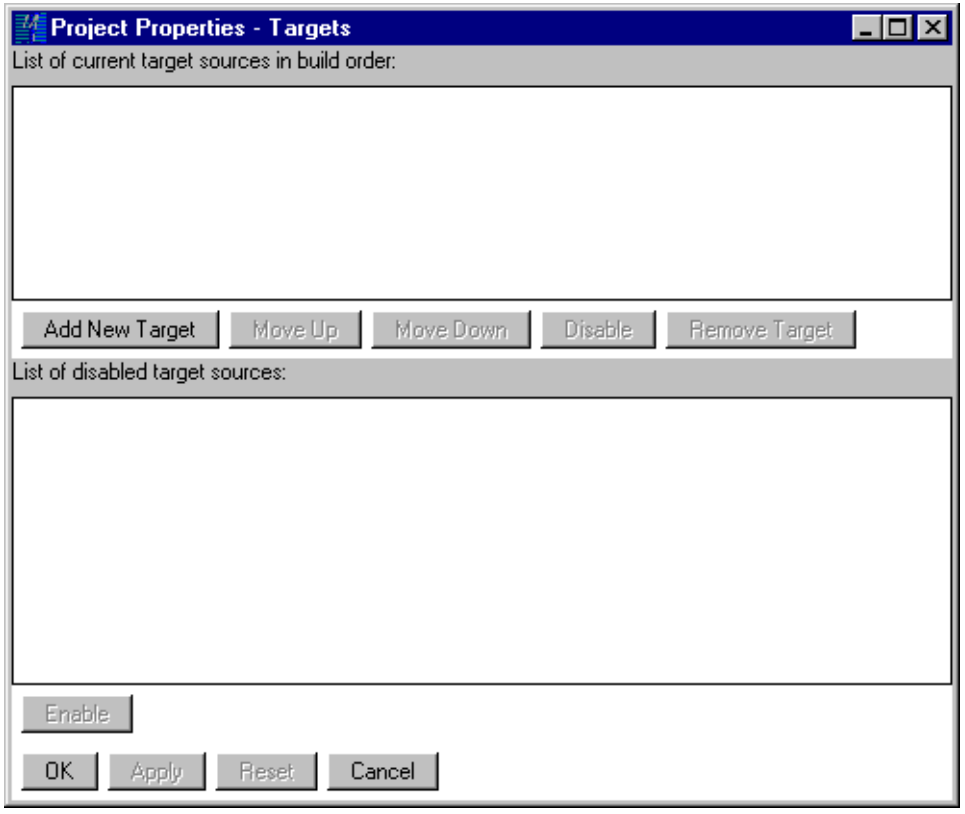


Figure 2.5 Project Properties - Target Sources.

Click on **Add New Target** and select **`sumxy.sml`** from the list of files in the **`examples\projects`** directory.

The target source dialog allows you to move selected target sources up and down the list, which varies the order in which they are compiled. For compilation purposes it is also possible to disable the compilation of specific target sources by selecting them and clicking on **Disable.** Disabled target sources are shown in the list of disabled targets, and when selected can be enabled by clicking on **Enable**.

Click on **OK**. The target source file **`sumxy.sml`** is visible in the current target field of the project workspace as the current program.

2.3.4 Setting the object path and mode

MLWorks compiles the source files into object files, which contain a representation of the objects defined in that file. Object files have the suffix **`.mo`**. The object files can later be loaded into the environment very quickly — this is much faster than re-evaluating all the declarations in the source file, for example.

The base location where the created objects are placed is specified by clicking the **Objects Location** button in the project workspace. This calls up a file selection dialog that allows you to chose a folder. By default, the base location is the same as the project file location. For this example we will use the default directory, which is the **`examples\projects`** folder.

The **Relative path names for Objects** checkbox in the project workspace has the same effect as the checkbox in the "Project Properties - Target Sources" dialog — it specifies whether the folder in which to place the generated object files is relative to the project file folder, or an absolute path name.

There is an added complication to the object files location. Object files are not put in the base object location, but rather in a folder one level deeper. the name of this folder is specified by the *mode* in which the files are compiled. Choose **Project > Properties > Modes** to call up the "Project Properties - Modes" dialog, which is shown in Figure 2.6.

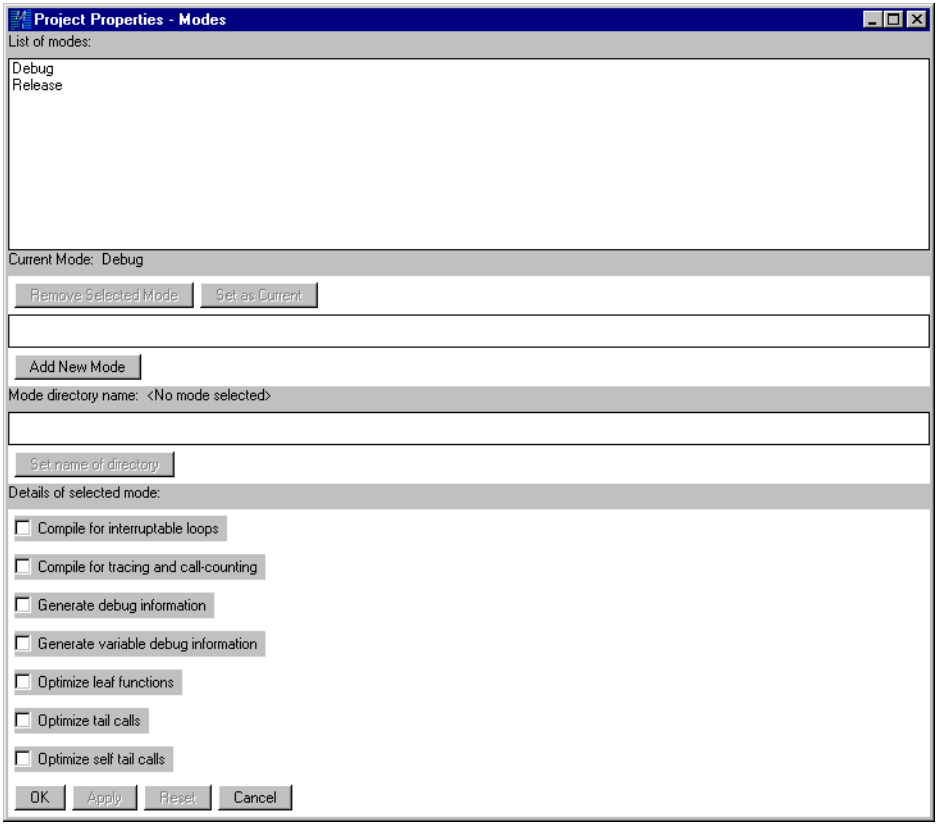


Figure 2.6 Project Properties - Modes.

By default the name of the folder in which the object files are placed is the same as the name of the mode in which they are compiled. For example, if the base location for object files is the **`examples\projects`** folder, and the target source files are compiled in Debug mode, then the object files are placed in **`examples\projects\Debug\`**.

Compilation can occur with the following different settings enabled or disabled:

- Compile for interruptible loops
- Compile for tracing and call-counting
- Generate debug information

- Generate variable debug information
- Optimize leaf functions
- Optimize tail calls
- Optimize self tail calls

A given mode has different selections of these options enabled and disabled. To compile under a given mode, select that mode from the list and click on **Set as Current**. Note that it is important to click on the button — merely selecting the desired mode and clicking on **OK** or **Apply** does not choose it.

Two modes are provided by default — the Debug mode and the Release mode. The Debug mode has the first four options checked, whereas the Release mode has all the optimization options checked. More modes can be added, each with their own settings, in the following manner:

1. Set the required options by checking the relevant option checkboxes
2. Type in a new mode name into the text field above the **Add New Mode** button
3. Click on the **Add New Mode** button.

Modes can also be set to have a folder name different to the mode name. Select the mode from the list of modes and enter the new folder name into the text field above the **Set Name of Directory**, and then click on the button.

For our example we will use the Debug mode, which will place our compiled object files into the `examples\projects\Debug\` folder.

Note: If you want to place your compiled object files in the `projects/` directory instead of the `projects/Debug` directory, create a new mode and specify nothing as its location.

2.3.5 Compiling the target sources into object files

We are now ready to compile the target source, `sumxy.sml`, and all its dependencies into object files. Select **Project > Compile Target Sources**. The Log / Console shows us line by line what it occurring. When all the target sources have been compiled they are displayed in the list of files. Select `sumxy` from this list.

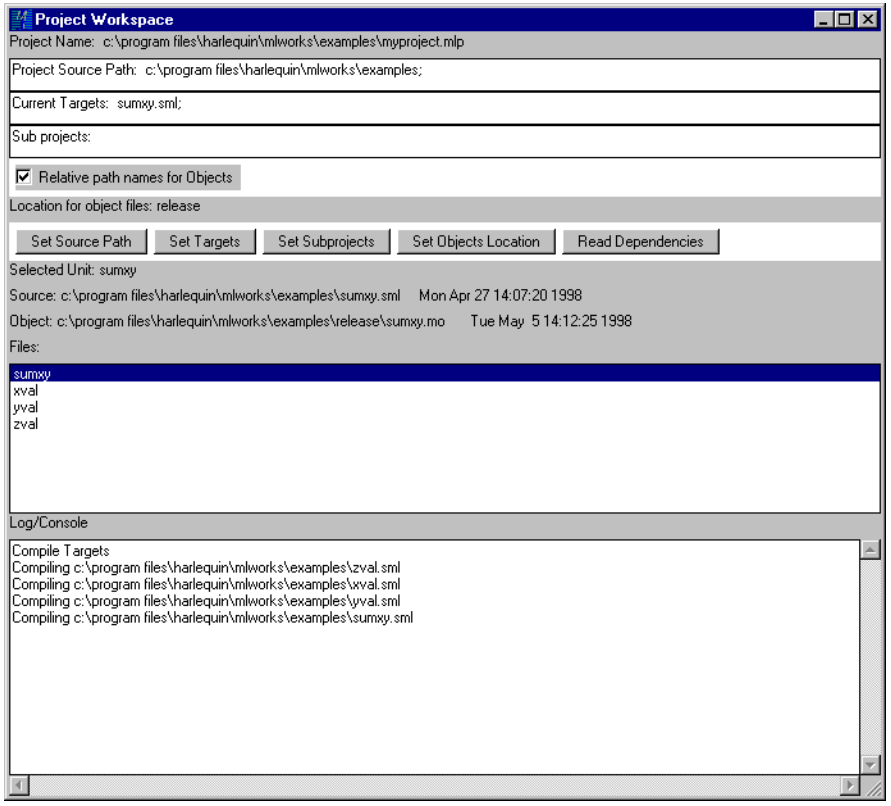


Figure 2.7 The compiled target source files.

The middle third of the project workspace gives us information on the selected unit, and consists of four parts:

Selected Unit — This area shows the name of the unit selected in the file list, and also indicates if the unit is loaded (see Section 2.3.6, “Loading code into the environment”) and visible.

Source — This area shows the full path name and filename of the target source file (`.sml` file) for the unit, together with its date stamp.

Object	This shows the full path name and filename for the unit's object file (**.mo** file), together with its date stamp.
Files	This text area shows the names of all the units which were compiled in the process of compiling the target source files.

2.3.6 Loading code into the environment

So far you have compiled source files into object files. No compiled objects have been loaded into the interactive environment; for instance, if you type the following at a listener prompt:

```
MLWorks> answer;
```

it generates an **Unbound value** error, even though **sumxy.sml** assigns a value to **answer**.

To load the compiled targets into the listener, select **Project > Load Targets**. This loads **sumxy.mo** into the listener, and makes *answer* available in the interactive environment. Notice, however, that the values of *x*, *y* and *z* are not available. To make values defined in files not explicitly required at the top-level of the project, you must select the relevant file from the file list, and choose **Project > Load Selection**. For example, selecting **zval** and choosing to load the selection makes *z* available in the interactive context.

2.3.7 Clearing the compilation message log

You can clear the compilation manager window that logs compiler messages with **Project > Clear Log**.

2.3.8 Removing, recompiling, and reloading units

For recompilation purposes it is possible to remove a selected unit with **Project > Remove Selected Unit**, and all units can be removed from the file list with **Project > Remove All Units**. The project system only recompiles source files into new object files if the source file has changed since the last compilation unless **Project > Force Compile of Selection** is used. Similarly, changed object

files can be reloaded into the interactive environment using **Project > Force Load of Selection**.

2.3.9 Summary of the basic compile and load operations

The MLWorks compilation system converts between source files, object files, and loaded objects in the interactive environment as shown in Figure 2.8.

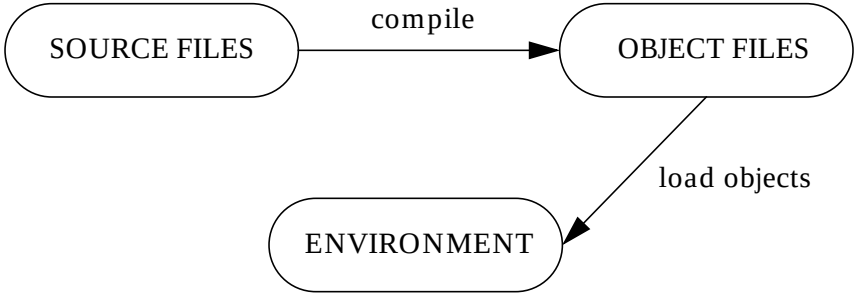


Figure 2.8 Relationship between source and object files in MLWorks.

In summary, the following steps are required to compile and load an application from scratch using the project system:

1. Create and save a new project with **File > New Project** and **File > Save Project As...**
2. Specify the source path of the target source files by clicking on **Set Source Path** in the project workspace
3. Specify the names of the current target source files by clicking on **Set Target Sources** in the project workspace
4. Specify a location for the object files that are built on compilation by clicking on **Set Objects Location** in the project workspace
5. Specify a mode of compilation by choosing **Project > Properties > Modes**
6. Compile the application by choosing **Project > Compile Target Sources**
7. Load the application into the listener by choosing **Project > Load Targets**

2.4 Further uses of the MLWorks project system

In Section 2.3, “Basic use of the MLWorks project system” we saw how to use the project workspace to set up a project file for compiling a set of dependent source files into object files, and how to load those files into the interactive environment. In this sections we examine some further uses of the project system, namely how to display source dependencies, how to specify libraries and subprojects for a project, and how to set compilation for different configurations of, for example, the application platform.

2.4.1 Determining file dependencies

If the dependencies of the target sources have not yet been read, clicking on **Read Dependencies** in the project workspace. This lists the referenced source files in the file list. If you have previously compiled the files this step is unnecessary as MLWorks has already determined the file dependencies.

In our example, select **sumxy** from the list of files, and choose **Project > Show > Dependencies of Selection**. A dependency graph for **sumxy** appears.

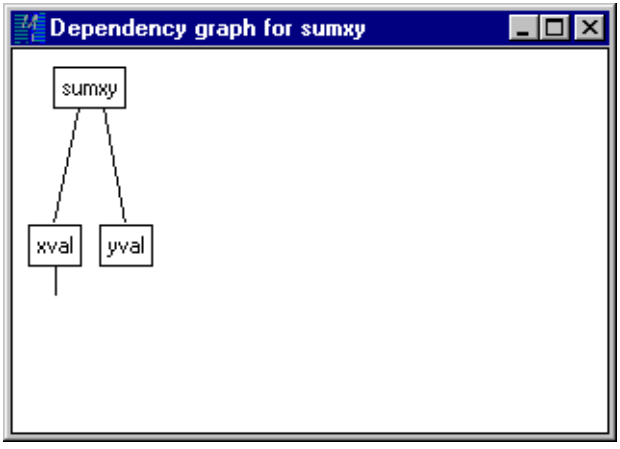


Figure 2.9 Dependency graph for **sumxy**.

Note that this graph only shows the immediate dependencies for **sumxy**. Double-click on an unexpanded element expands the graph to show further

dependencies. For example, double-clicking on **xval** reveals its dependence on **zval**.

2.4.2 Setting the library path

Your application may depend on a number of precompiled object files from another source. Such object files are known as library files. The object files of the Basis library are an example of a library. You can inform the project of the location of library files by choosing **Project > Properties > Library Path** from the project workspace menubar. The "Set Library Path" dialog appears:

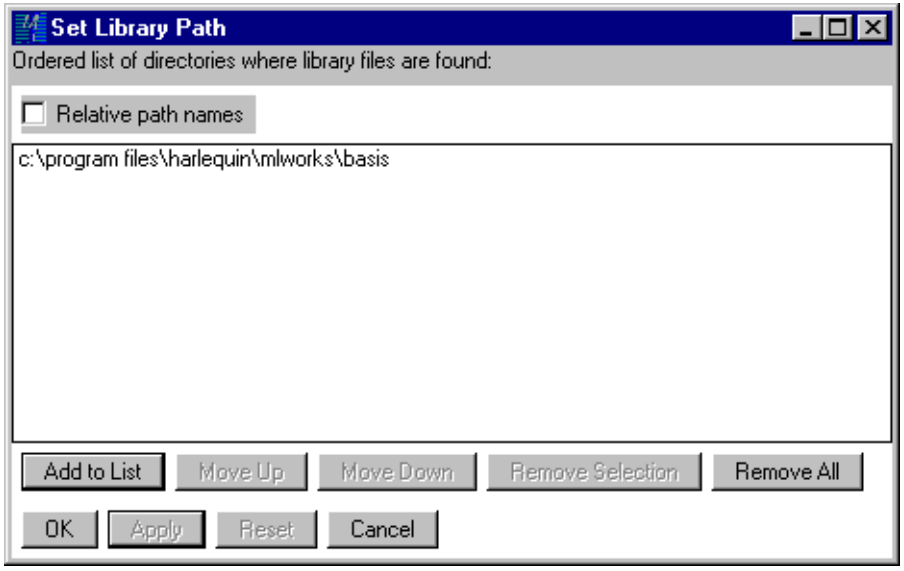


Figure 2.10 Setting the library path.

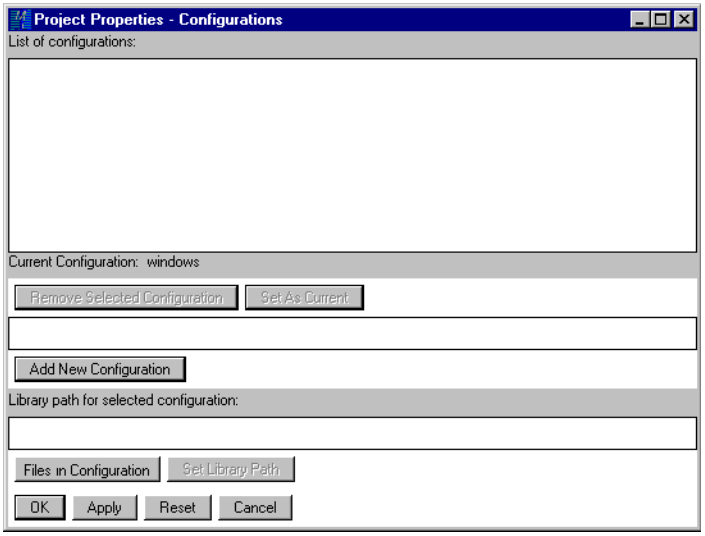
For example, imagine that our application needs some of the functions defined in the Basis library. We can point the project system to the object files defining these functions by clicking on **Add to List** and selecting the **basis** folder from the file selection dialog that appears. As with other dialogs for specifying path names there is a checkbox for using relative path names, and the choices for the library path are not added to the project file until you click on **Apply** or **OK**.

2.4.3 Configurations

Using configurations allows you to use the same MLWorks project to build the same application for a different platforms. This is done by proving a simple mechanism for switching between extensions to the source and library paths. The location of the compiled objects is also changed.

Choose **Project > Properties > Configurations** to call up the “Project Properties - Configurations” dialog.

Figure 2.11 Project Properties - Configurations



At the top of the configurations dialog the list of available configurations is displayed. The current configuration is shown beneath this list.

You can add a new configuration by typing in a name for it in the text field above the **Add New Configuration** button, and then clicking on the button. The new configuration name appears in the list of configurations.

The library path for a given configuration can be set or changed as follows:

1. Select the configuration by clicking on it in the list of configurations
2. Click on **Set Library Path** to set the library path. The configuration path dialog appears.

3. You can add a new folder to the configuration library path by clicking on **Add to List** and choosing it from the file selection dialog that appears.
4. Once you have added all the required folder to the path dialog click on **OK**. The new path is displayed in the relevant text area in the "Project Properties - Configurations" dialog.

You can add extra source files to be used by the project when it is in the current configuration by clicking on **Files in Configuration** in the configuration dialog, and adding extra files using the standard project file dialog that appears. This dialog also allows you to remove source files from the current configuration.

A current configuration is set by selecting a configuration from the list of configurations, and clicking on **Set As Current**. Selecting a current configuration has the following effect when compiling the project:

- As well as the target source files specified in the main project workspace, source files specified in the configuration are used during compilation.
- Library files are searched for in the configuration library path instead of the main library path.
- Object files are placed in a folder defined as the base object location with the configuration name appended to it, and the mode name appended to that. For example, if the base object location is **`examples\projects\`**, the configuration name is **`I386`**, and the mode is **`Debug`**, then the object files are placed in a folder called **`examples\projects\I386\Debug\`**.

2.4.4 Subprojects

You can have one or more MLWorks projects as subprojects of a main project. This allows applications to be organized into manageable parts which depend on each other. Note that circular subprojects are not permitted — for example, you cannot have two projects defined as subprojects of each other. Subprojects are transitive, which means that if you add a subproject to a project, you are also adding all of the subproject's subprojects too.

Subprojects are added to the main project by clicking on **Set Subprojects** in the project workspace. This calls up the “Project Properties - Subprojects” dialog. Added subprojects are shown in the Subprojects text field of the project workspace.

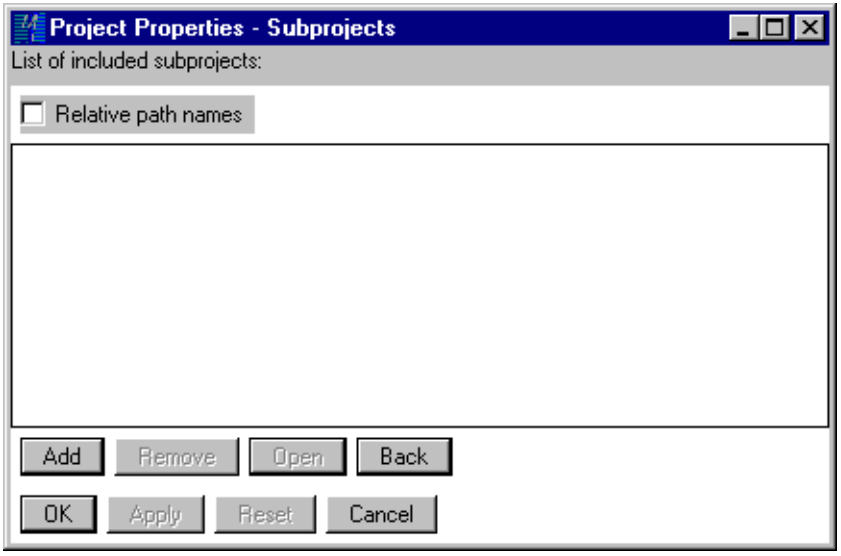


Figure 2.12 Project Properties - Subprojects

In our example, click on **Add** to call up a file selection dialog, and add the project **`fac.mlp`**, in the **`examples\projects\subproject`** folder. Once you have added this subproject to the main project, you can open it in the project workspace by clicking on **Open**. The focus moves to the project workspace. Note that the “Project Properties - Subprojects” dialog remains open, but now shows any of the subprojects of **`fac.mlp`**. Clicking on **Back** closes the subproject and returns you to the main project in the project workspace.

You should be aware of the following features of subprojects:

- A folder which is on the source path of a subproject does not need to be included in the source path of the main project, unless there are source files in the folder that are used by the main project but not by the subproject.

- Each subproject keeps the mode it was saved with, regardless of the mode of the main project. For example, if a subproject is saved with its mode set to Release, all of the subproject's object files will be placed in the Release folder on compilation of the main project, even if the main project has its mode set to Debug.
- In the project workspace only those files that are used by the main project are displayed. None of the subproject files are displayed. This allows you to concentrate on the dependencies within the main project (although on compilation the project system checks the files in all the subprojects too, and brings them up to date if this is necessary).

With **`fac.sml`** added to **`myproject.mlp`** as a subproject, the target source files of the main project (such as **`sumxy.sml`** and **`xval.sml`**) can refer to the **`fac`** function defined in the target source files of the subproject.

A note of warning concerning the use of different configurations with subprojects: setting the main project to have a specific configuration affects the subprojects in one of two ways:

- If the subproject does *not* have a configuration with the same name defined, then the subproject is compiled with no configuration information.
- If the subproject has a configuration with the same name as the configuration of the main project, it is compiled with that configuration set as the current one. Note that the subproject uses its own configuration source and library path information, and not the configuration information of the main project.

2.5 Delivering applications with MLWorks

MLWorks allows you to produce executables from ML applications. This process is known as *delivery*.

Applications can be delivered using the function **`MLWorks.Deliver.deliver`**. This function takes three arguments:

- A string, which is the filename of the file to be output
- A function of type **`unit -> unit`**. This is the function to be evaluated when the image is loaded

- Either MLWorks.Deliver.CONSOLE or MLWorks.Deliver.WINDOWS, which specifies whether your executable is a console executable or has a user interface.

To be specific, suppose you have defined the following function. It may have been defined in a file that was compiled and loaded, or in a file that was called with **use**, or by typing directly in the listener.

```
fun hello () = print "Hello, world.\n";
```

The **print** function is available at top-level in MLWorks, so you do not need to read any library code into MLWorks, nor **require** any libraries, to make this example work. For a delivery example that does use other libraries, see Section 3.5 on page 49, which delivers a simple program that uses the Standard ML Basis library for I/O.

By invoking **MLWorks.Deliver.deliver** from a listener, you can now deliver the function **hello** as a console application:

```
MLWorks> MLWorks.Deliver.deliver ("hello.img", hello,
MLWorks.Deliver.CONSOLE);
```

or as a GUI application:

```
MLWorks> MLWorks.Deliver.deliver ("hello", hello,
MLWorks.Deliver.WINDOWS);
```

The executable file **hello**, considering what it does, may seem quite large. However, this is because it must contain a copy of the MLWorks runtime. This is a constant overhead (somewhat under 300 KB), so is less significant for larger programs..

2.6 The MLWorks batch compiler

You can compile source files without running the MLWorks interactive environment by invoking the MLWorks compiler as a batch process. MLWorks provides a script called **mlbatch.bat** for this purpose. This script takes a series of unit names and produces object files from the unit sources.

However, you do not need to use **mlbatch.bat** to compile files that are suitable for use outside the interactive environment. All files you compile in the interactive environment are compiled equivalently: every unit you compile

down to an object file is compiled ready for execution outside the interactive environment. You can pass it to the MLWorks runtime system with the **`mlpervasive.bat`** script.

But if you do want to execute applications you developed in the interactive environment outside the interactive environment, you must bear in mind that there are important differences between what we introduced in Section 1.5 as the *batch context* and the *interactive context*.

Recall that the interactive context is what is available to you in the interactive environment. We distinguish it from the batch context, which is what is available to a file being batch-compiled. The interactive and batch contexts both contain:

1. The Standard ML language.
2. The *pervasive* library, which is a built-in library available at top level in MLWorks. It is provided in the top-level structure **`MLWorks`**.
3. The Standard ML Basis library's **`General`** structure, and various top-level functions, exceptions, and types that the **`General`** structure defines. It is also built in to MLWorks at the top level.

The interactive context contains two further items:

4. The **`Shell`** structure, which is a collection of functions and options connected with the MLWorks interactive environment. Like the **`MLWorks`** and **`General`** structures it is always available at the top level.
5. Any modifications you make to the interactive context by causing declarations to be evaluated.

   For instance, binding the identifier **`x`** to the value 5 counts as a modification of the interactive context: it introduces a new binding to the top level. You modify the interactive context similarly whenever you load units from source or object files.

The batch context does not include the last two items for good reason. The bindings you create in the interactive context are typically transient, and should not be allowed to affect how the compilation of a set a permanent application sources should turn out. The **`Shell`** structure is not ephemeral, but since it relates to details of the interactive MLWorks environment — like the

value-printer settings, editor preferences, and how to start running the GUI — it makes no sense to make it visible in the batch context.

3

Using the MLWorks Libraries

3.1 Introduction

MLWorks is distributed with a set of libraries. This chapter explains how you can explore these libraries in the interactive environment and how you can utilize them in your applications.

3.2 The MLWorks libraries

The MLWorks libraries are supplied in two ways:

1. As mixtures of source and object files
2. As image files

The libraries are:

- The Basis library. This library is an implementation of the Standard ML Basis library developed by the SML community. It provides a basic SML toolkit for general programming tasks such as I/O; extensions to the SML type system; internationalization; and Posix support. Part of the Basis library, the **`General`** structure, is available permanently at top level in MLWorks.

Object and source files for the Basis library are in the **basis** compound, an installation subdirectory. (Some Basis object files are in the **system** subdirectory.) An image file including the Basis library is in **images\basis.img**.

- The MLWorks pervasive library. This library is MLWorks' own general-purpose library containing facilities for I/O, creation of standalone applications, multiprocessing, profiling, and so on. There is some overlap between the facilities provided by the pervasive library and those provided by the Basis library.

  The pervasive library literally pervades the MLWorks environment, as the permanently available structure **MLWorks**. Object and source files for the pervasive library are also available in the **pervasive** compound, an installation subdirectory. An image file can be found in **images\pervasive.img**.

- The MLWorks foreign interface library. This library is MLWorks' own library for interfacing to code written in languages other than Standard ML. C is the only foreign language supported at present.

- The MLWorks Win32 interface library. This library is MLWorks' own library for GUI programming through Win32.

  Object files for the Win32 interface library are in the directory **winsys**; the image file is **images\windows.img**.

- The interactive environment library. The MLWorks interactive context provides this library in the permanently available structure **Shell**. This library is not available in the batch context — see Section 2.6, "The MLWorks batch compiler".

3.3 How the libraries are distributed

As we saw in Chapter 2, in the project management system we can consider source and object files to be *units*. All library units (that is, all the library object files) are stored in the **objects/Release** directory.

Each unit in an MLWorks library contains either a structure, a functor, or a signature. The unit names are based on the names of the structure, functor, or sig-

nature that they contain. The prefix of the unit name indicates which of the three kinds of SML module it contains:

Structure	__ (two underscores)
Functor	_ (one underscore)
Signature	no prefix

The rest of the unit name is the name of the structure, functor or signature, converted to lower case, with word breaks indicated by underscores. For example:

signature LIST_UTILS ***becomes*** **list_utils.mo**

structure ListUtils ***becomes*** **__list_utils.mo**

Beware of swapping singular names for plural (and vice versa).

Some units contain signatures or structures that are sub-components of other signatures or structures. In these cases, any "." separators in the SML identifier are replaced by single underscores. For example:

structure OS.Process ***becomes*** **__os_process.mo**

The object-file versions of units have the usual extension **.mo**, and the source versions (if any) have the extension **.sml**. Typically, structures and functors are distributed in object-file form only, and signatures are distributed in both source-file and object-file form. The availability of the signature in source form permits you to use the structure or functor it defines, or indeed to extend the library by writing new structures and functors using the signature.

(This distribution method is similar to the way C programming libraries are often distributed: "implementation" files (**.c**) are usually only distributed in compiled form (**.o**), but header files (**.h**), which specify the interface to the library, are distributed in source form. However, in MLWorks the "header" files are distributed in object form as well.)

The image files for the libraries, which contain pre-loaded versions of the compiled units, are stored in the directory **images**.

The following is a complete list of compounds (subdirectories of the top-level installation directory):

basis	Compound for most Standard ML Basis signature files.
bin	Compound for scripts to run MLWorks runtime and images. It also contains some DLLs.
documentation	Contains the online reference and user guides.
examples	Contains example files demonstrating the use of projects, threads, the basis library, and so on.
images	Contains the MLWorks images.
pervasive	Compound for MLWorks pervasive library (**MLWorks** structure) units and the Standard ML Basis library's **General** structure, both of which are necessary to run all code compiled with MLWorks. The **General** structure is also available in **basis**.
utils	Compound for mutexes signatures.
winsys	Compound for MLWorks Win32 interface library units.

The following is a complete list of library images:

basis.img	An image of the Standard ML Basis library.
pervasive.img	An image of the MLWorks pervasive library.
windows.img	An image of the MLWorks Win32 interface library

Note: This is not the complete list of images, just that of library images: there is also an image for the batch compiler and the MLWorks interactive environment itself. The batch compiler image is **batch.img**. The interactive environment with the pervasive library (**MLWorks**), the Standard ML Basis library's **General** structure, and the interactive environment library (Shell) is in **gui.img**, and a copy of the interactive environment including the entire Standard ML Basis library is in **guib.img**.

3.4 Using the libraries in your applications

You can load the object file versions of the libraries into the MLWorks environment using the listener or compilation manager **File > Load Objects** item. See Section 2.3 on page 30. The **`Shell.Project.load`** functions can also be used. And if you started MLWorks with **`mlworks-basis`**, the Basis library is already loaded.

You can refer to the libraries in your application source files using a **`require`** declaration. See Section 2.2 on page 29. It is best to begin **`require`** declarations for library units with the root symbol (**`$`**).

The source path mechanism controls where to search for library units, and the default installation sets an appropriate initial source path.

3.4.1 Portability and the Standard ML Basis library

Some parts of the Standard ML Basis library are specific to the host platform. These are compiled Basis library units for structures and functors. Harlequin's implementations for UNIX and Windows both store these compiled units in a directory called **`system`** so that you can make portable references to them, as follows:

```
require "system.unit";
```

3.5 Using the libraries in your applications: an example

The following example shows how to build a simple application executable that uses part of the Standard ML Basis library.

3.5.1 The application

The application is a trivial program which reads a line of input and then prints it out prefixed by "The output is: ".

```
fun get_value () = TextIO.inputLine TextIO.stdIn;

fun show_value x = TextIO.output (TextIO.stdOut, x);
```

```
fun run () =
  let val v = get_value()
      val s = String.concat ["The output is: ", v]
  in show_value s end;
```

As you can see, this program uses the **TextIO** and **String** structures from the Basis library.

3.5.2 Requiring TextIO and String in the application sources

The application sources must **require** the **TextIO** and **String** structures. Suppose the application is contained by a single file, **app.sml**. All we need to do is to add the following **require** declarations to the start of **app.sml**. They tell the compiler that the file depends on the **TextIO** and **String** structures; the compiled units for those structures in turn explain their own dependencies.

```
require "basis.__text_io";
require "basis.__string";
```

Now, when you read **app.sml** into a version of MLWorks without the Basis library loaded, its dependencies will be apparent and MLWorks will know how to satisfy them during compilation.

3.5.3 Compiling and loading the application

Compile the application and load it into MLWorks using the compilation manager. The steps are: **File > Read Dependencies** and then **Build > Compile and Load**. The compilation manager will load the parts of the Basis library that it needs automatically if they are not already loaded.

See Section 2.3 on page 30 for more details.

3.5.4 Delivering a standalone executable version of the application

The final step is to deliver the application as an executable,

```
MLWorks.Deliver.deliver("app", run, MLWorks.Deliver.CONSOLE);
```

You can now run the **app** executable:

```
> app
This is my typed line of input.
The output is: This is my typed line of input.
>
```

4

Examining Objects

4.1 Introduction

There are various tools in MLWorks that you can use to look more closely at the values and functions defined in the interactive environment. Some of them we have already met: the context browser, the system browser, and the inspector. Here we shall look at these in more detail. We shall also look at the history tool, a tool based on the listener history mechanism described in Chapter 1

4.2 Examining values: the inspector

Let us take a closer look at the inspector. In a listener window, type the following definition of a binary tree (MLWorks' verification output is omitted):

```
MLWorks> datatype btree = leaf of int | node of (btree * btree);
MLWorks> node (leaf 1, node (node (leaf 2, leaf 3), leaf 4));
```

Now select **Usage > Inspect**. An inspector appears with details of the tree just defined. The bottom pane of the inspector includes a graphical representation of the tree, showing only the root node; clicking on this expands the graph to show the children. (You may want to re-size the inspector using your window manager.) You can click on any node to select it, and clicking again will further expand that node.

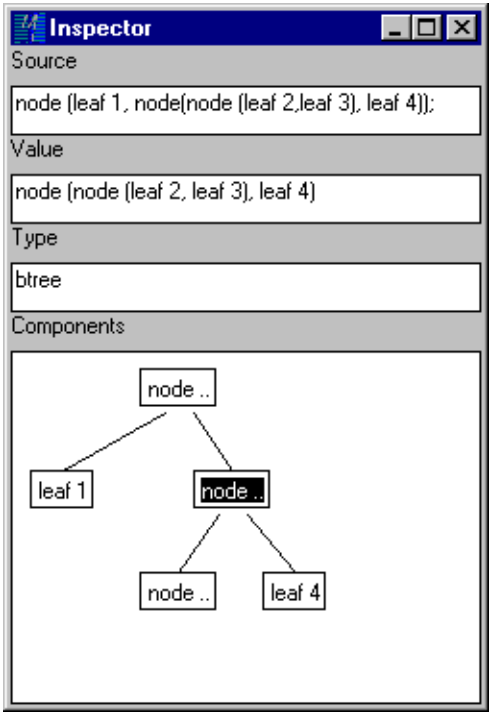


Figure 4.1 The MLWorks inspector.

4.2.1 Changing the graph layout

Different data structures have different shapes, which can affect how best to represent them graphically. Type the following two lines into the listener:

```
MLWorks> fun ilist 0 = nil | ilist n = n::ilist (n-1);
val ilist : int -> int list = fn
MLWorks> ilist 100;
val it : int list = [ 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, ..]
MLWorks>
```

Now choose **Usage > Inspect** and click on the root node to see all the children of this hundred-element list.

The current representation is not, perhaps, the most useful for inspecting this value. Select **Usage > Tool Settings > Graph Layout...** to get a dialog from which the appearance of the graph can be changed. On this dialog, select:

- **Orientation > Vertical**
- **Line Style > Step**
- **Child Position > Next**
- **Horizontal Spacing > 1**

Click **Apply**. The appearance of the graph changes to reflect the new options.

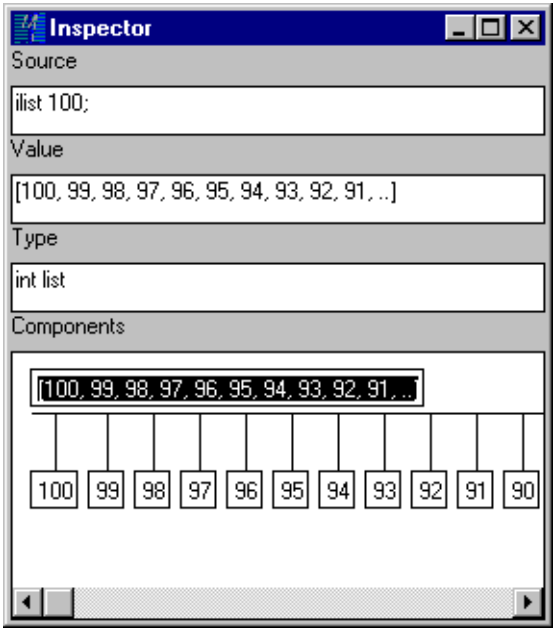


Figure 4.2 The inspector graph after setting changes.

The many parent-child links are easier to distinguish because you have specified steps instead of straight lines. The **Child Position** setting ensures that the list itself aligns with its first few entries, rather than the entries in the middle, and the decrease in horizontal spacing allows more values to fit in the inspector window.

The **Orientation** setting affects whether successive generations appear below their parents, or to the right of them. Change this setting to **Horizontal**, and set **Vertical Spacing** to zero, to get a tall thin graph instead of a short wide one.

Of course, with sufficiently large structures, no amount of settings will make it possible to view an entire value at one time, but it is generally possible to see the important information by judicious expansion and contraction of parts of the graph. The layout options are present to provide further flexibility.

4.2.2 Display controls

The inspector has another dialog affecting the graph: select the **Usage > Tool Settings > Inspector...** menu item to call up the Inspector: Display Controls dialog.

Selecting the **Type** radio button (and, as usual, clicking **Apply** or **OK**) changes the representation of objects in the inspector's graph window. Instead of showing the objects' values, the graph shows their types.

If the **Show atomic entries** box is not checked, tree-nodes consisting of a single atom (integers, reals, strings, and so on) will not be shown when the graph is expanded.

The **Expand all children** check box expands the current tree to the fullest extent, and does the same for subsequent objects when they are first inspected. If **Atomic entries** is not checked it will, of course, omit atoms.

If the **Share equal values** check box is checked, a single value will not be drawn twice in the inspector's graph window; instead, MLWorks will show which values are shared by drawing a graph where objects have multiple parents.

4.2.3 Value and Type fields

The inspector has Value and Type fields which show a text representation of the current object. This is not necessarily the whole object being inspected, but the component selected in the graph window. Single-clicking selects an element without expanding or collapsing the subtree below that element.

If you select a component which is a function, the Value field does not give the ML code for the function. For instance, try evaluating and inspecting the function `ilist` itself. The Type window gives the type of the function, but the Value window says only that it is a function. It can be made more informative using the **Usage > Tool Settings > Value Printer...** dialog. The dialog is in general

the same as described in Section 1.14 on page 24 for the listener, except that its effects are confined to printing in the inspector.

On this dialog, check the **Show function details** box and click **Apply**. Now re-select **ilist** on the graph by clicking. The inspector's Value field now reads

```
fn[ilist[<listener #1>:1,5 to 1,44]]
```

It gives the name of the function (**ilist**), and where it was defined: in this case, in the listener. If it had been defined in a file, and if debugging mode had been on when that file was read in, you could now use **Edit > Edit Source** to open the file in your preferred editor at the definition of **ilist**. For details of defining a preferred editor, see Section 1.9, "Editing ML files". Debugging mode is switched on using **Listener > Properties > Mode** from the MLWorks podium.

4.3 Examining the interactive context: the browser tools

4.3.1 Inspecting values from the system browser

As we have already seen in Chapter 1, one use of this is to invoke an inspector on any value in the environment. To see more clearly the functions of the system browser, select **Tools > System Browser** from the menu bar.

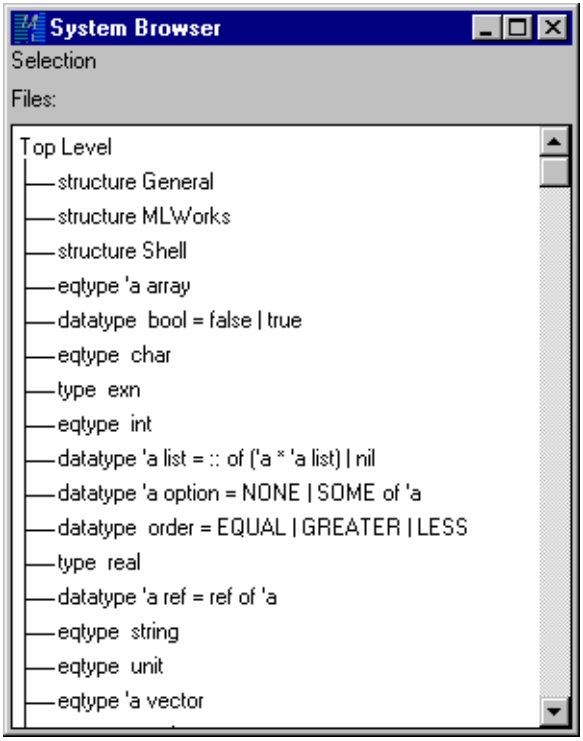


Figure 4.3 The MLWorks system browser.

The system browser shows all the built-in values that exist at top level in MLWorks: structures, types, values, and so on. A structure contains a list of further values with their types, and so on; you can use the system browser to look inside a structure. Click on the **`Shell`** structure to select it, and click again to expand it.

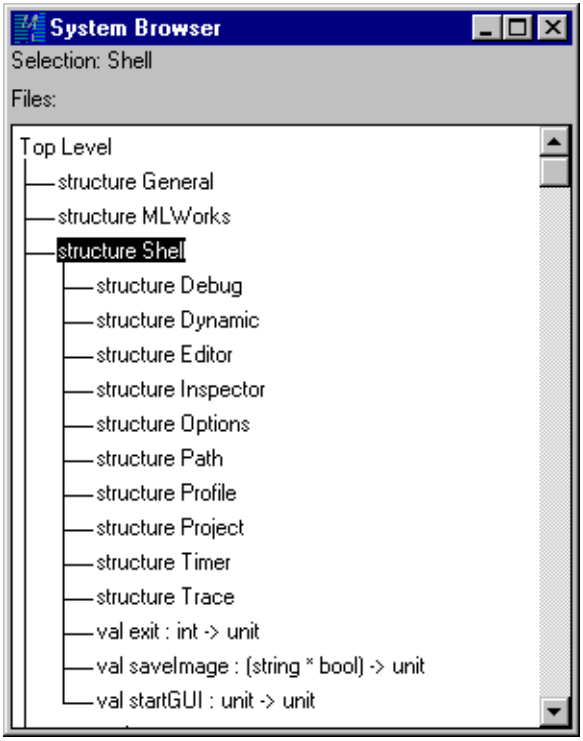


Figure 4.4 The system browser showing the contents of structure **`Shell`**.

The graph expands to show the types, exceptions and values specified in the **`Shell`** structure.

You can remove different kinds of ML entity from the system browser window using the **Usage > Tool Settings > Browser Filters...** dialog. The dialog contains options for showing and hiding structures, types, exceptions, variables, and so on.

If you select a value in the system browser, and use **Edit > Copy** and **Edit > Paste** to paste it into a different tool (for example, a listener), the full qualified name of the value is pasted.

4.3.2 Inspecting values from the context browser

Tools > Context Browser brings up the context browser, which is like the system browser, but shows user-defined values rather than built-in ones. Notice how, once displayed, a context browser automatically keeps itself up to date with any new bindings you make. Try declaring a new function in the listener: it is added to the Entries pane automatically.

Like the system browser, the context browser also allows filtering through its own **Usage > Tool Settings > Browser Filters...** dialog.

In addition to the features of the system browser, the context browser allows you to inspect a value you have defined. Select a value in the context browser and choose **Usage > Inspect**. An inspector appears, showing details of the value in the usual way.

4.4 Examining and repeating listener input: the history tool

We saw the listener's **History** menu in Section 1.6 on page 5. Recall that the menu lists the last *n* items evaluated by that listener. (The value of *n* is taken from the setting of **Usage > General Preferences > General**'s **Maximum history length** option.)

The history tool lists *all* items accepted by *all* listeners, and also all items loaded into the environment. The items are listed in the order in which they were added to the environment.

Figure 4.5 The MLWorks history tool.

Like the context browser, the history tool keeps itself up to date with newly evaluated input automatically.

Tools > File Viewer or double-clicking on an iten shows the source code for that item. For an item that was loaded from the filesystem rather than typed into the listener, this shows the load operation which brought the item into MLWorks, including the file or module that contained the item.

Usage > Remove Duplicates searches for items that occur more than once and deletes all but one of them from the list.

Edit > Copy allows you to copy an item from the list in order to paste it into some other MLWorks tool; **Edit > Delete** deletes the currently selected item; **Edit > Delete All** deletes all of them.

Note that the history tool has an effect on garbage collection. If you evaluate expressions that have extremely large return values (for example, proofs) and bind these return values to `it`, say, then when you rebind `it`, the values are not garbage collected; instead they are added to the history tool list. Values are only garbage collected when they drop off the end of the history list. Memory can be reclaimed in these instances by explicitly emptying the history list using **Edit > Delete All** or removing duplicate values using **Usage > Remove Duplicates**.

5

Debugging

5.1 Introduction

The GUI interface to the MLWorks debugger is provided by the error browser and the stack browser, both of which we saw in operation briefly in Chapter 1. The error browser deals with compile-time errors, while the stack browser deals with run-time errors. In this chapter, we shall take a longer look at what the stack browser does and at the various ways in which it can be invoked. The error browser does not need further treatment.

We shall also look at the facilities for tracing evaluations, setting execution breakpoints on functions, and how to step through evaluations call by call.

Note: Before running any of the examples in this chapter, ensure that debugging mode is on, and optimizing mode is off. You can do this by invoking MLWorks with the **`-debug-mode`** option, or from the MLWorks podium's **Listener > Properties > Mode** dialog. (The **Usage > Save Preferences** menu item will save any preferences in force and use them in the future whenever you start up MLWorks again.) You should also make sure that you have set your preferred editor; see Section 1.9.

5.2 Viewing files

Associated with the stack browser is the ability to view different parts of the source code corresponding to the various functions that have been called. To see this in action, create a file called, say, **debug.sml**, containing the following curious code defining factorial functions for odd and even numbers:

```
exception Fact;
fun ofact 0 = raise Fact | ofact n = n * efact (n-1)
and efact 0 = 1 | efact n = n * ofact (n-1);
```

Now read this file from a listener with **File > Read Into Listener...**.

The somewhat inefficiently coded functions **efact** and **ofact** happily compute factorials of even and odd integers, respectively. However, invoking one of them on an integer of the wrong parity raises an exception, and enters the stack browser. In the listener window, type

```
MLWorks> ofact 6;
```

The stack browser appears, with a list of the function calls leading up to the uncaught exception.

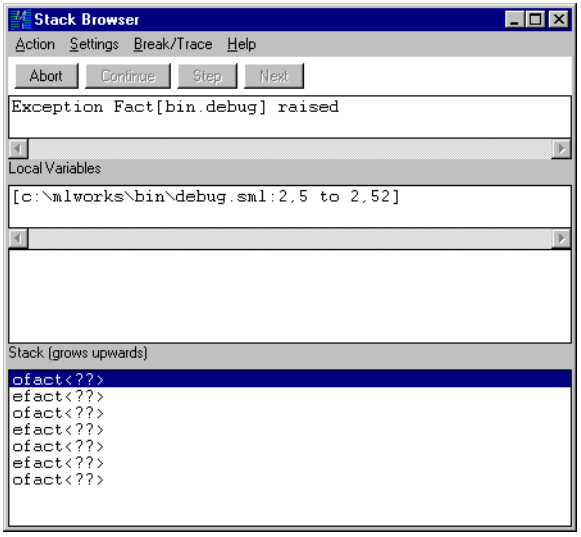


Figure 5.1 The MLWorks stack browser.

At the same time, a view of the file **debug.sml** appears in one of two ways:

- If you have set your preferred editor as the Emacs server, and have an Emacs server running, the source appears in your Emacs window. The definition of the last function called before the exception was raised, namely **efact**, is highlighted, but you can move about the source and edit it.
- Otherwise, it appears in an MLWorks *file viewer*. The definition of **efact** is highlighted. Choosing **Edit > Edit Source** from the stack browser puts you into your preferred editor, passing the line number of the definition of **efact**.

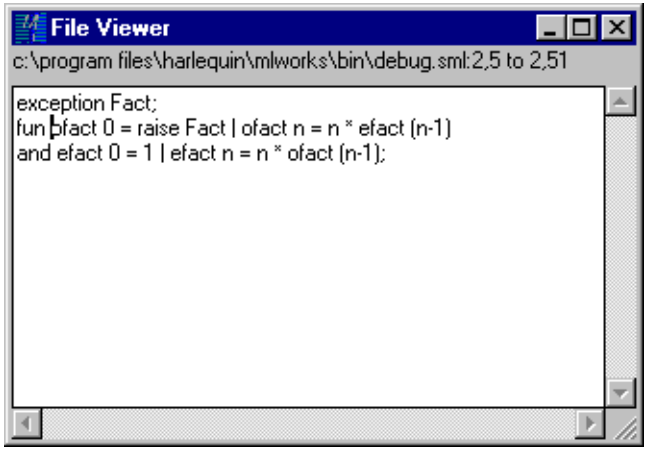


Figure 5.2 The MLWorks file viewer.

In the bottom pane of the stack browser are the function calls leading to the uncaught exception; they are alternately **efact** and **ofact**. You can select any function call by clicking. Selecting a function call has two effects:

- It changes the other information in the stack browser, specifically the details of the local variables passed to the function in the upper pane of the stack browser.
- It causes the Emacs server or file viewer to change view, highlighting the definition of the function you have selected. If you have a file viewer, you can at any point select **Edit > Edit Source** from the stack

browser to enter your preferred editor, passing the line number of the currently selected function.

5.3 Step mode

It is not necessary to raise an exception in order to use the stack browser; this and the next sections present more ways of invoking it. This section looks at running a function in step mode.

To use step mode, make sure the listener is in debug mode (**Listener > Properties > Mode...**) and type a declaration in the usual way at the listener prompt. Then press the listener's **Step** button. The declaration typed into the listener will be evaluated one step at a time, entering the stack browser each time a new function call is entered. Only calls to functions that were compiled in debugging mode appear are stepped.

For instance, with the definitions from the last section still in force, type the following and press **Step**:

```
efact 10 + ofact 11;
```

The stack browser appears, and the text field at the top shows this is because **`efact`** has been entered. The lower pane shows the details of the function call, and the upper pane the value of any arguments passed; in this case there was only one argument.

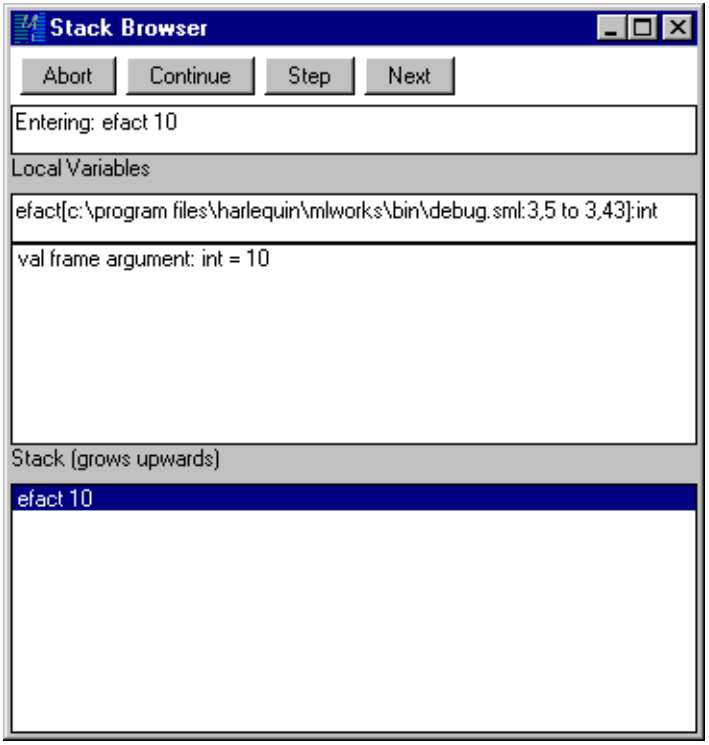


Figure 5.3 Stack browser invoked in step mode.

As well as the usual **Abort** button, there are now active buttons marked **Continue**, **Step** and **Next**. The **Step** button will continue the evaluation until another function call is entered — in this case, `ofact`. At each step the stack browser shows the local variables and function arguments to the top function on the stack, and the file viewer (or Emacs server) moves to the definition of the relevant function, which is highlighted.

The **Next** button will continue the evaluation until the function currently at the top of the stack returns: in the present case, it will finish the evaluation of `efact 10`, and stop again at the call to `ofact 11`. At any point, pressing **Continue** will switch off step mode and continue until the end of the evaluation.

5.4 Adding breakpoints

Sometimes, when tracking down a bug, you might want to be able to stop a computation whenever a particular function is called. The trace and breakpoint manager is for just this purpose. It is invoked from the menu bar with **Tools > Break / Trace**.

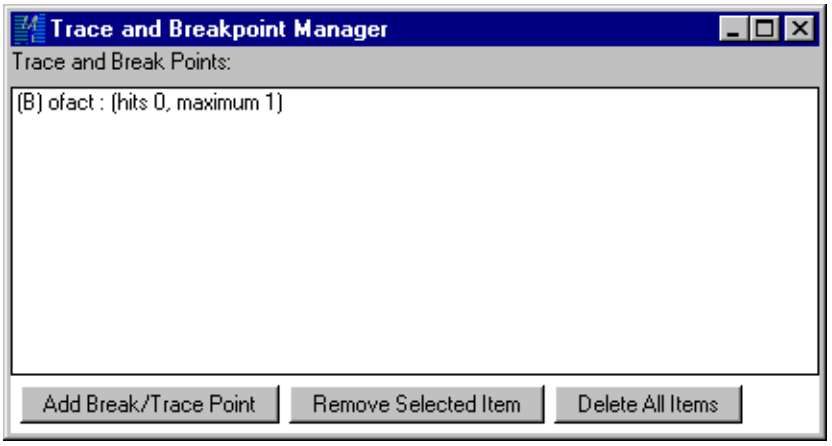


Figure 5.4 The MLWorks breakpoint manager.

The breakpoint manager allows you to specify a number of function names. Whenever one of these function names is called, MLWorks enters the stack browser. To specify a break point, click on **Add Break/Trace Point** and type the function name into the **Name** text box and click **OK** or **Apply.** The function must have been compiled in debug mode for the breakpoint to take effect.

5.5 Interrupting an evaluation

Finally, the stack browser may be entered by interrupting an evaluation. While MLWorks is evaluating some declaration, you can interrupt it by clicking the  button on the podium. MLWorks scans the stack and enters the stack browser.

Note: The stack may be very large if you are in the middle of a long recursive computation, in which case scanning it will take a correspondingly long time. Please be patient waiting for the stack browser to appear.

5.6 Local variables

As well as showing the sequence of function calls and their arguments, the stack browser shows the values of local variables in force during a function call. You can see this in action by typing the following function definition into the listener:

```
fun f[] = raise Div | f(h::t) = let val y:int = h*h in y+f t end;
```

The function **f** is supposed to calculate the sum of squares of a list of integers, but will fail because it raises an exception on the empty list. Now evaluate **f[3,4]**. The stack browser appears, showing a stack of three function calls. If you select, say, the middle of these, MLWorks shows all the local variables in force during that function call.

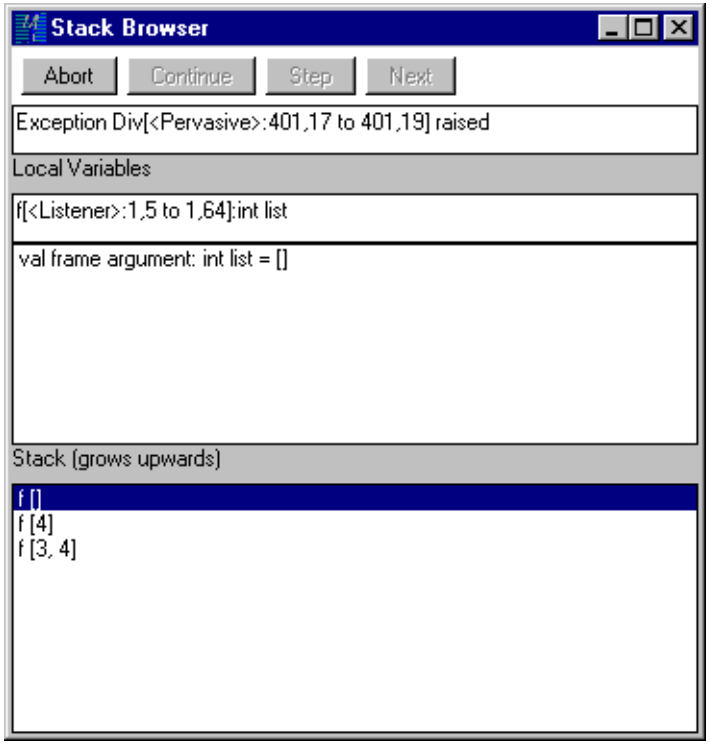


Figure 5.5 Local variables in the stack browser.

It shows that, during the call to **f[4]**, **h** and **t** (defined by pattern-matching) had the values **4** and **[]**, respectively, and **y** (defined explicitly using **let**) had the value **16**.

5.7 Tracing

Beside the stack browser, MLWorks has another facility to help the debugging process: tracing. It is also controlled from the trace and breakpoint manager, invoked from the MLWorks podium with **Tools > Break / Trace**. The trace options of the trace and breakpoint manager work in exactly the same way as the break points: they contain a list of functions whose execution must be traced each time they are called. You can add or remove functions in the list. Again, functions that are to be traced must have been compiled in debug mode.

A trace of a function listed in the trace and breakpoint manager proceeds as follows: whenever the function is called, MLWorks reports that it has been called, and also reports the result of the call when the function returns.

If you now add **efact** to the trace manager and type **efact 6** in the listener, MLWorks will trace all the recursive calls to **efact**:

```
MLWorks> efact 6;
efact 6
  efact 4
    efact 2
      efact 0
      efact returns 1
    efact returns 2
  efact returns 24
efact returns 720
val it : int = 720
```

You can tell the difference between trace points and breakpoints in the trace and breakpoint manager through their prefixes - trace points start with (T), whereas break points start with (B).

6

Profiling

6.1 Introduction

MLWorks has a sophisticated profiler for investigating the CPU time used by a function call, the amount of space it allocates, how many times it is called during a given evaluation, and so on. This chapter discusses the GUI interface to the profiler, the profile tool.

In passing this chapter makes reference to the programmatic interface to the profiler, provided by the structures **`MLWorks.Profile`** and **`Shell.Profile`**, which are built in to the interactive environment. **`MLWorks.Profile`** presents a full interface that is substantially more flexible than that of the GUI profile tool, while **`Shell.Profile`** provides the means to invoke the GUI profile tool but with different profiler settings, such as stack scanning frequency and the type of profiling to be performed (which can be any or all of time profiling, space profiling, and call-counting). You can find out more about these structures in the *MLWorks Reference Manual*.

You can also access the profiler through the MLWorks runtime executable, **`main.exe`**. See the output of **`main.exe -help`** for more details.

6.2 A note on compiling code for profiling

You do not need to compile code in debugging mode to profile it. Indeed, profiling works on fully optimized code. However, optimizing can complicate profiler output by introducing new code items.

The only kind of profiling that requires special compilation is call counting: to produce useful results in call-count profiling. Ensure that the listener's preferences dialog **Compile for tracing and call-counting** check box is checked (**Listener > Properties > Compilers** calls up the dialog), or that the **`Shell.Options.Compiler.generateTraceProfileCode`** option is set to **`true`**.

6.3 Profilable entities

We say that the profiler produces execution profiles for function calls. More exactly, the profiler produces execution profiles for all *code items*. For more on code items, see the section on **`MLWorks.Profile`** in the *MLWorks Reference Manual*. For now, note that a code item most closely corresponds to an instance of a lambda function (a **`fn`**) in the source. So in:

```
val f = fn x => x + 10;
```

The **`fn x => x+10`** code item is profilable: an application of **`f`** to an integer value could be profiled, though in practice its execution would probably be completed so quickly that the profiler would not have time to record anything about the application.

6.4 The profile tool

To enter the profile tool, type a declaration at the listener prompt and, instead of hitting Return, press the **Profile** button. You can also enter the profiler tool by passing your declaration to the **`Shell.Profile.profile`** function.

We shall use the function **`pyramid`**, defined in terms of several other functions, in our examples:

```
local
  fun a 0 l = l
    | a x l = a (x-1) (x::l);
  fun len (n,[]) = n
    | len (n,x::xs) = len(n+1, xs);
  fun addlens (a,[]) = a
    | addlens (a,x::xs) = addlens(a+len (0,x), xs);
  fun p (0,l) = l
    | p (n,l) = p(n-1,a n l);
  fun b (0,l) = addlens (0,l)
    | b (n,l) = b(n-1,p(n,[])::l);
in
  fun pyramid n = b (n, []);
end;
```

These functions, written in order to demonstrate the profiler, exhibit a range of execution time and allocation characteristics. The **`pyramid`** function computates three-dimensional triangular numbers.

Below is shown the result of profiling **`pyramid 200`** with the definitions above in force. Note that exact figures will vary from machine to machine, and from one evaluation to another, so if you profile this function call yourself you should not expect to see identical results.

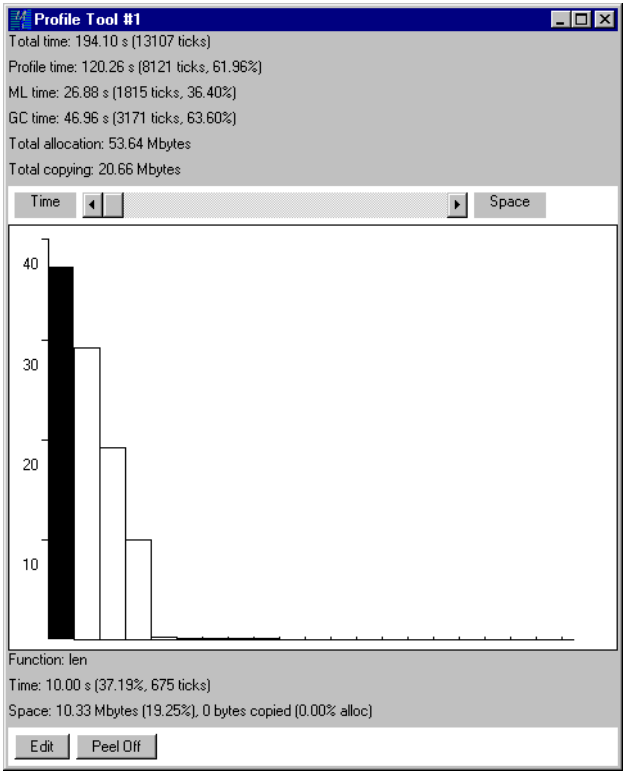


Figure 6.1 The MLWorks profile tool.

At the top of the profile tool you can read the overall profile of the evaluation. At the very top is printed the total CPU time taken by the evaluation, in seconds. This total includes the number of *ticks* that occurred; the CPU time for each function called in the evaluation is measured by interrupting evaluation every tick (10 milliseconds) and identifying the top function on the stack. (You can change the tick rate if you use the built-in **`MLWorks.Profile.profile`** or **`Shell.Profile.profileFull`** functions.)

Beneath the total time is the profile time. This is the amount of CPU time during the evalation that was taken by the larger-scale profiling operations.

Beneath the profile time are the total ML and garbage collection (GC) times. These totals show what amounts of the total time (after profile time has been subtracted) were taken up by execution of the ML code itself and by garbage

collection. ML time includes some of the profiler operations done at a very fine granularity which are not timed separately.

Beneath ML and GC time you can read the total amount of space allocated by the evaluation. Beneath that, the total amount of that space that was copied by the garbage collector during evaluation is shown. This latter value is therefore a measure of how much work the garbage collector did.

"Total copying" is also a measure of the ephemerality of the allocated data; if a program allocates 9000K but only 180K is copied, then we can conclude that the rest (8820K) died before the program reached a garbage collection point. However, note that the garbage collector will look at long-lived data more than once, so it is possible for "Total copying" to exceed "Total allocation".

Beneath the overall profile is a graph pane. Each bar represents a function, and the height of the bar corresponds by default to how much processor time that function used. The vertical axis is calibrated in percent of total time taken. The Time / Space slider above the graph can turn this into a weighted average of the time and allocation factors. Selecting a bar gives the figures for the function it represents below the graph. Note the following points:

- CPU time is measured by interrupting the evaluation at every tick (10 milliseconds), and identifying the top function on the stack.
- There are two separate bars for the function **a**. It is a curried function, that is it takes two arguments, and MLWorks measures separately time taken in applications to the first argument and to the second argument.
- As well as the bars for the functions you call, there are usually two other bars on the profiler, for functions called **wrapper** and **<Setup>**. These are internal MLWorks functions and can be ignored. They generally have negligible cost, both in time and space.

As well as the time information, the profiler shows how much space each function allocated. The height of the bars can be made to reflect allocation rather than CPU time. Moving the Time / Space slider on the profiler alters the bars to reflect a weighted average of the two factors, depending on how far you move the slider along. The figure below shows the same profile as before, but the slider has been moved to the 50% mark.

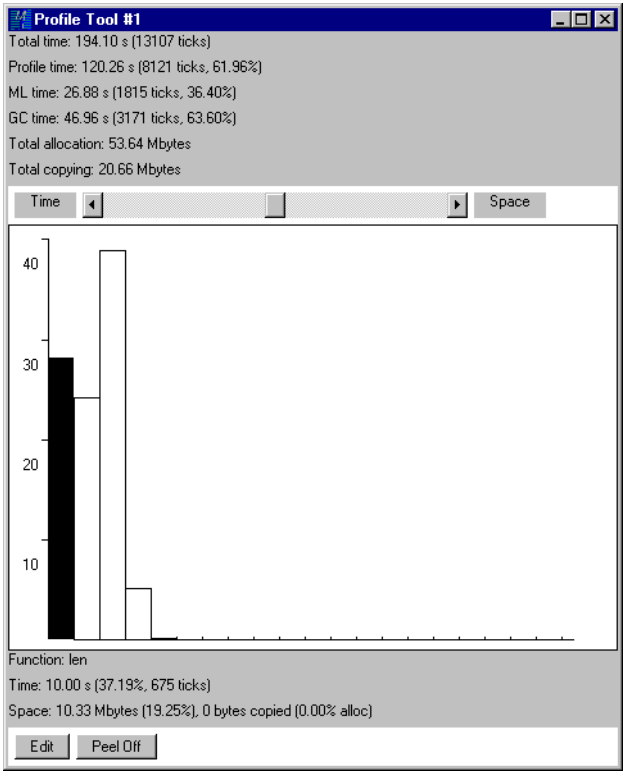


Figure 6.2 A profile with equal weighting of allocation and time.

If you choose **Usage > Sort**, the profile tool sorts the bars in decreasing order of size. Figure 6.3 shows what Figure 6.2 looks like after sorting.

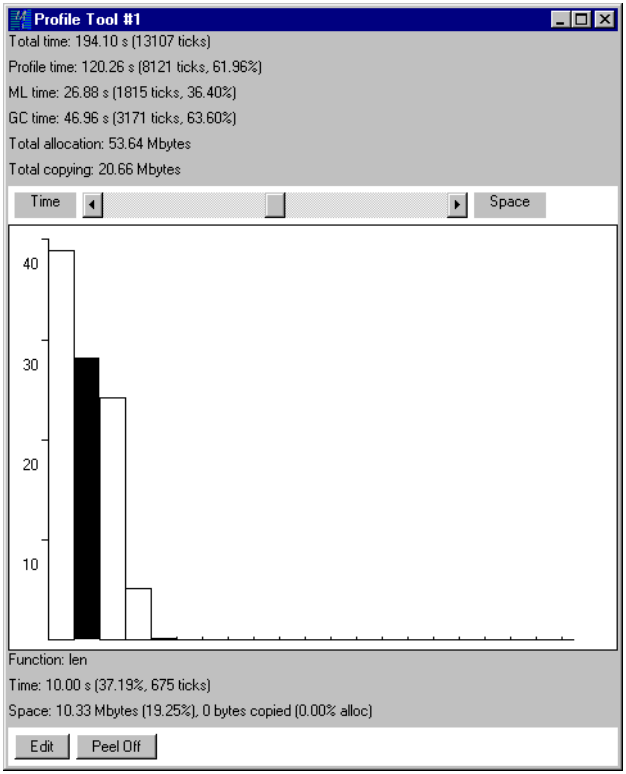


Figure 6.3 The profile graph after being sorted.

When you use the profile tool, the results will appear in the same window each time, replacing the previous profile information. However you can take a "snapshot" of the profile tool at any point with the **Usage > Duplicate** menu item.

Additionally, with the **Peel Off** button, you can create a dialog containing a text copy of the profile for the function currently selected.

Both the profile tool and the function profile dialog have an **Edit** button which allow you to edit the source code for the current function in your preferred editor.

6.4.1 Controlling the layout of the profiler graph

The layout of the profiler graph can be adjusted with **Usage > Tool Settings > Profiler Layout...**. There are four settable parameters:

- Maximum number of bars. The maximum number of bars that the graph may show at any one time. Note that if this number is smaller than the number of functions profiled in your evaluation, the graph will favour the most significant functions first, leaving out those that produce the smallest potential bars at the current Time / Space slider setting. The default maximum is 20 bars.
- Bar width. The default maximum is 20 units.
- Maximum tick spacing. Frequency with which the vertical axis is marked to represent ticks. The default maximum is 100.
- Ideal label spacing, in pixels. The default is 100.

7

Using MLWorks in TTY Mode

7.1 Introduction

The MLWorks interactive environment can be run in TTY mode as well as in GUI mode. While the graphical tools are no longer available, there is a teletype listener, debugger, and inspector, and the interactive context contains the same items as in the GUI: the **`MLWorks`**, **`General`**, and **`Shell`** structures. Most of the functionality in TTY mode is the same as in GUI mode. This chapter documents some interface differences.

7.2 Starting MLWorks up in TTY mode

To run MLWorks in TTY mode, execute **`mlconsole.bat`**, which can be found in the **`bin`** folder of your MLWorks installation folder.

7.3 Debugging in the TTY interface

The TTY interface provides its own interface to the debugger. The commands available are:

`<`	Go to the earliest frame in the stack

>	Go to the latest frame in the stack
i	Next frame into the stack or next later frame (callee)
i <*n***>**	Next *n* frames into the stack
o	Next frame out of the stack or next earlier frame (caller)
o <*n***>**	Next *n* frames out of the stack
b	Do a backtrace of the stack
f	Show full frame details
e	Edit definition
h {<*name***>}**	Hide the given types of frame or list hidden frame types if none given
r {<*name***>}**	Reveal the given types of frame or list the revealed frame types if none given
p	Print values of local and closure variables
c	Continue interrupted computation
s	Step through computation
s <*n***>**	Step through computation to nth function call
n	Step over the current function to the start of the next one
trace <*name***>**	Set trace on function entry at <*name*>
breakpoint <*name***>**	Set breakpoint on function entry at <*name*>

untrace <*name***>**

Unset *trace on function entry at <name>*

unbreakpoint <*name***>**

Unset breakpoint on function entry at <*name*>

breakpoints Display the list of breakpoints

ignore <*name***>** *n*

Ignore the next n hits on the breakpoint on function <*name*>

help Display this help info

? Display this help info

c Continue interrupted code

q Return to top level

7.4 Inspecting values in the TTY interface

The TTY interface provides its own interface to the inspector, though it can also be invoked in the GUI environment via the listener.

The TTY inspector inspects the value of **it**, whatever that may be at the time the inspector is invoked. To invoke the inspector, call the built-in interactive environment function **Shell.Inspector.inspectIt**. For example:

```
MLWorks> [["a", "b"], ["c", "d"]];
val it : string list list = [["a", "b"], ["c", "d"]]

MLWorks> Shell.Inspector.inspectIt();
Entering TTY inspector - enter ? for help
Value: [["a", "b"], ["c", "d"]]
Type: string list list
0: ["a", "b"]
1: ["c", "d"]
Inspector>
```

The TTY inspector allows you to navigate the value being inspected recursively. As you can see, at each stage it prints a numbered list of subvalues of the current value. It supports the following commands:

<*field name***>**	Inspect named field of current value
p	Inspect parent value of current
q	Quit.

To continue our example:

```
Inspector> 0
Value: ["a", "b"]
Type: string list
0: "a"
1: "b"
Inspector> 1
Value: "b"
Type: string
Inspector> p
Value: ["a", "b"]
Type: string list
0: "a"
1: "b"
Inspector> q
val it : unit = ()
MLWorks>
```

The TTY inspector supports inspector methods. Inspector methods allow you to apply a function whenever a value of a particular type is inspected. You could, for example, control the printing style for values of certain types your application defines. Inspector methods must be compiled in debugging mode. For more on inspector methods, see the section on **`Shell.Inspector`** in the *MLWorks Reference Manual*.

Index