MLWorks

User's Guide to the Environment

Restricted beta version, June 1996

harlequin

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

DRAFT

Contents

1

Using MLWorks interactively

1.1 The console

When you start MLWorks, it will draw the *console window* on your screen. This window will be present throughout your MLWorks session; closing the colsole with the **Tools > Exit** menu item will exit MLWorks. Other tools may be started by selecting them from the **Tools** menu of this or any other window.

1.2 Introducing the Listener

The first thing you will need to run MLWorks interactively is a Listener. This gives you an ML command line at which you can type expressions. To start a Listener, choose **Tools > Listener** from the MLWorks console (or any MLWorks window).

A Listener appears, with the prompt '`MLWorks> `'.

At the heart of the Listener is a *read-eval 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
```

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: 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
```

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 remains in the MLWorks environment, unless it is overridden by a later binding. It 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
```

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, invoked with the `Shell.Profile.profile` function.

1.3 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, or 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. When you press the Return key (or the **Evaluate** button on the Listener), MLWorks reads the current line.

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, perhaps modifying it, 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. (Pressing Return on a previous line which does not contain a complete declaration may lead to slightly odd results.)

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**.

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

Ctrl-A	move to start of line
Ctrl-E	move to end of line, or if at end of line, end of next line

Ctrl-B	move backwards one character
Ctrl-F	move forwards one character
Ctrl-N	move to next line
Ctrl-P	move to previous line
Alt-F	move to beginning of next word
Alt-B	move to beginning of previous word
Ctrl-V	scroll forwards one screen
Alt-V	scroll back one screen
Alt-<	move to start of Listener
Alt->	move to end of Listener
Ctrl-L	Redraw display
Ctrl-U	delete everything since last **`MLWorks>`** prompt
Ctrl-D	delete character right of insertion point
Ctrl-K	cut from cursor to end of line
Shift+cursor	move cursor, selecting text between start and end point
Delete	delete selected text
Ctrl-W	cut selected text
Ctrl-Y	paste
Alt-P	previous history item
Alt-N	next history item

Also, double- or triple-clicking will select an entire word or line, respectively.

A more flexible way of repeating declarations is to use the Listener's **History** menu. This is a list of all declarations previously accepted by the Listener; when you select one, it appears at the current prompt (at the bottom of the 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 cycle through the history list.

Note: if you use the mouse to paste text into the Listener window, remember it will be pasted at the position of the mouse pointer (not cursor). It will not automatically be pasted at the current prompt or at the current cursor position.

The **Clear** button clears the Listener window, but not the history list, which it leaves unchanged.

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 **Preferences > General** menu item on the MLWorks console.

1.4 Writing and reading 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 MLWorks console, and choosing **Context > Save Source As ...** MLWorks will give you a file dialog, and allow you to choose a directory and a filename in which to save the source. You might like to choose a filename with the suffix **`.sml`** to indicate that it contains Standard ML code. You can then edit the file as you would any other program, with a text editor.

Notice it will save *all* the code that has led to the current MLWorks context. In particular, if you have had more than one Listener, this option will record all code typed into all your Listeners, in the order in which it was evaluated. Since Listeners all share the same context, the context created by running this file will be the same as that existing in any of your Listeners at the point when you do the save.

If you have already used **Context > Save Source As** in this MLWorks session, you can now simply choose **Context > Save Source** 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; one is to use the Listener's **File > Use File** 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 will verify all the declarations in the file 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`**:

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

Note: The **File > Use File** 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 separate compilation system, documented in Chapter 2.

1.5 Error handling

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, etc (unhandled exceptions are dealt with in another way — see "Tracking down unhandled exceptions" on page 10).

Specifically, suppose you enter the following:

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

This line is wrong in two ways: the boolean `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.

Note: if you do not like the Error Browser, you may switch it off by selecting **Preferences > General** on the MLWorks console. 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 selecting **Action > Redo** on the Error Browser window. MLWorks will try to evaluate the line as amended, and the Error Browser will disappear.

The **Action > Close** menu item will close the Error Browser, copying the error report to the Listener and giving a new prompt.

The Error Browser's remaining menu item, **Action > Edit**, causes the part of the declaration where the error was detected to be underlined in the Listener window.

1.6 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 **Preferences > Editor ...** from the MLWorks console. The resulting dialog gives you three ways of specifying an editor:

- Select the Emacs server. This allows you to have a single Emacs session, and all files to be edited will appear as buffers in that session. Errors discovered by MLWorks are highlighted in the Emacs buffer.

  To use the server, you will need to load some Emacs Lisp code into your Emacs session. You can do this by copying the code in the file `env/emacs` in the directory where MLWorks is installed, into the file `.emacs` in your home directory.

  You can now make Emacs communicate with MLWorks by typing `M-x mlworks-server` in Emacs.

  Note: If you are running MLWorks across a network, remember to ensure your Emacs session is running on the same host.

- Select an Xterm editor. Use the text box to specify a command line. When you ask MLWorks to edit a file, it will start a new X 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.

- Select an X editor. This works in the same way as the Xterm editor, but instead of starting a new window, MLWorks will invoke the editor in the window from which you started MLWorks.

Note: Your choice of editor, like other preferences from the **Preferences** menu, can be saved by selecting **Preferences > Save Preferences**. MLWorks will write the details of your settings to a file called **`.mlworks_preferences`** in your home directory, which will be read in on starting subsequent MLWorks images.

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

```
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 **Action > Edit** menu item 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.

1.7 Examining ML objects

You can look at the details of an ML value or function using the Inspector. To see it in action, define an object as follows:

```
MLWorks> val walrus = ("hello", [[1,2], [3,4] @ [5]]);
```

Now choose **Action > Inspect**. MLWorks brings up an Inspector.

The Inspector contains details of the item most recently defined in the Listener where it was invoked. It has several fields: the Source field shows the declaration MLWorks read which defined the object 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 object. Currently this is not very informative; however, it can be expanded to show the structure of the object. Double-click on the object 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 your window manager.

By invoking the Inspector from the Listener you can only inspect the most recently defined item. You can inspect an older item, however, from the Context Browser. To do this, select **Tools > Context Browser**. The Context Browser appears.

It contains a list of everything defined in the current MLWorks context. To inspect any item in the context, select the item by clicking on it in the list, and then choose **Action > Inspect** from the Context Browser.

For more details of the Browser and Inspector, see Chapter 3, "Examining Objects".

1.8 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. Similar tools are sometimes called 'debuggers', but unfortunately they do not actually remove bugs; they merely supply information which helps track them down. Besides, the word 'debugger' has a vulgar taste in the mouth. So, as the purpose of the tool is to browse the stack, it is called the Stack Browser.

Before using the Stack Browser, it is important to turn 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 **Preferences > Mode** from the console. In the Mode Options dialog, select the 'Debugging' check-box and press **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.

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 will bring up more information about that function call in the middle pane of the Stack Browser. The **Abort** button will close the Stack Browser.

For more details of the Stack Browser, see Chapter 4, "Debugging and tracing".

1.9 The MLWorks start-up file

When MLWorks starts up, it looks for a file called **`.mlworks`** in your home directory, and if this exists, evaluates its contents as ML declarations and modifies the context accordingly. You could use this file to pre-load function 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;
```

2

The separate compilation system

2.1 Introduction

MLWorks has a ‘separate compilation’ system (rather similar to UNIX’s **`make`** command), which allows you to compile files of ML code or load them into your environment. The files may depend on one another in complex ways, in which case when you ask MLWorks to compile a file, it will also compile all the files that your file depends on. However, a given file is only compiled by MLWorks if it has been changed since the last time it was compiled.

When it compiles a file of code (source file), MLWorks can do either of two things: it can output an object file, which contains an internal representation of the objects defined by the code; or it can load those objects (functions, values, and so on) into the current environment. (Of course, it can also do both.) If an object file is already available, 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 separate compilation system does this tracking for you.

2.2 Expressing dependencies in files

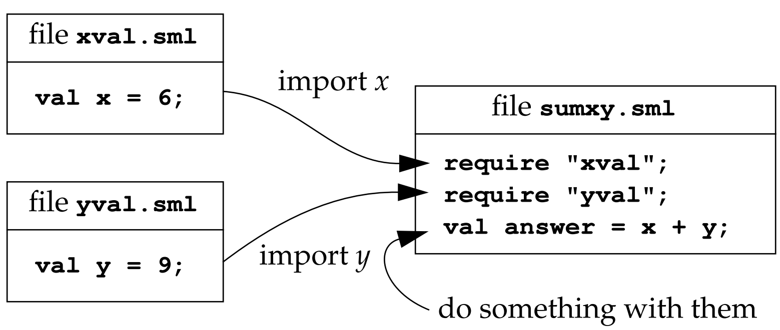
To allow efficient separate compilation of source files, there must be some way to express dependencies between files. MLWorks accomplishes this by extending standard ML with an additional keyword, the **`require`** keyword, which can be used to make a top-level declaration of the form

```
require "<source>";
```

to tell the compiler that the present file uses values or functions defined in <*source*>, which should therefore be read in first.

The **`require`** keyword takes a single string argument, which is the name of an MLWorks unit (for units see Section 2.5; for the moment, you can think of a unit name as being the name of a source file without the **`.sml`** extension). Any **`require`** statements *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:



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*:

file `xval.sml`	file `zval.sml`
`require "zval";` `val x = z div 7;`	`val z = 42;`

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` keyword is rather like ML's `local` **<***declarations***>** `in ... end` construction.

2.3 The compilation mechanism

2.3.1 Registering units

For the moment, we shall assume the files `xval.sml`, `yval.sml`, `zval.sml` and `sumxy.sml` reside in a common directory, from which MLWorks was started. For details of how to make ML search other directories for `require` statements, see Section 2.4.

Once you have expressed file dependencies in your source files using `require` statements, you can compile the files using the Compilation Manager. To get a Compilation Manager, choose **Tools > Compilation Manager** from any MLWorks window.

The top pane of the Compilation Manager shows a list of the units currently known to it; each unit corresponds to a file of source code, and has the same name as the file without the `.sml` prefix.

Initially, the list is empty. To register `sumxy` on the list, choose **File > Read Dependencies**. MLWorks brings up a standard file dialog; select the file `sumxy.sml`, and click **OK**.

MLWorks adds `sumxy` to its list of units. It also reads the require statements in `sumxy.sml` and adds whatever further units are required by `sumxy`, and it checks those files for require statements, and so on. In this example it has found four units.

Choosing **View > Show Graph** will give a new window showing the current dependencies of the units in graphical form.

So far, MLWorks has not read the declarations in the files, and in particular it has not compiled them or loaded them into the environment. It has only read the require statements and calculated the dependencies.

Incidentally, you could remove a unit from the list by choosing **Build > Delete**, or clear the list with **Build > Delete All**. These operations only affect the list in the Compilation Manager; they do not delete any files or any objects from the environment. You could also edit the file corresponding to a unit by choosing **Build > Edit**; this will put you into your preferred editor, as detailed in Section 1.6, "Editing ML files".

2.3.2 Compiling files

MLWorks can compile a unit into an object file, which contains a representation of the objects defined in that unit. The object file can later be loaded into the environment very quickly — it is much faster than re-evaluating all the declarations in the file, for example. You can see this in action by compiling `xval`. In the Compilation Manager, select `xval` in the list of units, and choose **Build > Compile**. The command is echoed in the lower pane of the Compilation Manager:

```
Compile xval
Compiling /u/ldisk/markw/ml/zval.sml
Compiling /u/ldisk/markw/ml/xval.sml
```

Notice that, though you only asked for `xval` to be compiled, MLWorks has compiled `zval` too, as `xval` depends on `zval`. MLWorks brings up a confirmation dialog when the compilation is finished. It has now created object files `xval.mo` and `zval.mo` in the same directory as the source files.

MLWorks only compiles those files that do not already have up-to-date object files. You can see this in action if you now compile `sumxy`, by choosing it from the list and choosing **Build > Compile**:

```
Compile sumxy
Compiling /u/ldisk/markw/ml/yval.sml
Compiling /u/ldisk/markw/ml/sumxy.sml
```

MLWorks compiled `yval` (on which `sumxy` depends), but did not re-compile `xval` and `zval`, as up-to-date object files already existed for these units. If the files had been changed since they were last compiled, MLWorks would have recognised this and re-compiled them. For example, if you now choose **Query > Compile**:

```
Check compile of sumxy
No units need compiling
```

The Query menu item does not actually try to compile the unit; it merely looks to see which non-existent or out-of-date object files would need to be re-generated if this file were compiled. None of the source files for `sumxy` or the units it depends on have changed, so all the object files are still current. Now choose **Touch > Source**. This changes the date stamp on the source file (as if the source had been edited), making the object file for `sumxy` look out of date. Choosing **Query > Compile** again gives:

```
Check compile of sumxy
Units to compile: sumxy
```

`sumxy.mo` is now out of date, but none of the the other object files it depends on need re-compiling.

The **Touch > Source** menu item can be used to force re-compilation of a unit, even when the source file has not actually been changed.

2.3.3 Loading objects into the environment

So far you have compiled source files to object files. No objects have been loaded into the MLWorks 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 object file for `sumxy` into the current environment, choose **Build > Load Objects**.

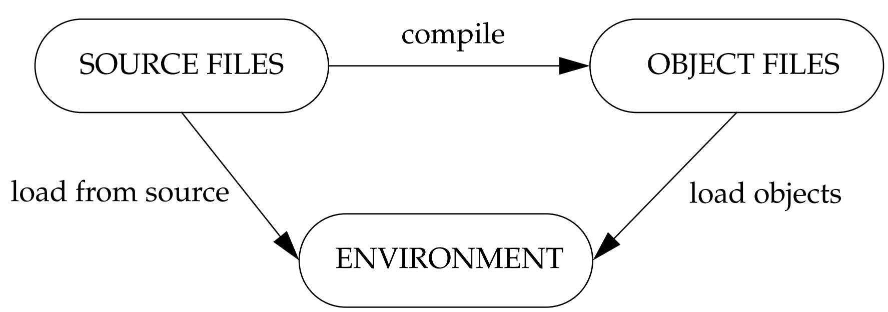
```
Load sumxy
Loading mo module zval
Loading mo module xval
Loading mo module yval
Loading mo module sumxy
processing result
val answer : int = 15
```

As the compilation manager reports, the value of **`answer`** is now available in the environment. Notice, however, that the values of x, y and z are not available; to make them available, you must explicitly load the units where they are defined.

The **Build** menu also has a **Load from Source** item. Choosing this causes MLWorks to read the source file for the selected unit (and any units on which it depends), evaluate it, and load the objects into the environment without going through an intermediate object file. As loading from source, like compiling, involves evaluating ML source code, it in general takes much longer than just loading from an object file. The **Compile and Load** item, as you might expect, evaluates the source and both loads the result into the environment and writes out an object file.

2.3.4 Summary of compilation operations

MLWorks converts between source files, object files and loaded objects in the environment as shown by the following diagram:



All three operations load or compile the selected unit and all units on which it depends. No unnecessary loading or compiling is ever done: that is, units are loaded or compiled only if the relevant objects do not already exist, or are out of date. Objects can be marked out of date using the Touch menu: **Touch > Source** updates the time stamp on the source file, which will force re-evaluation if a unit is compiled or loaded from source. **Touch > Loaded** removes the time stamp on the objects loaded in the environment, which will force them to be re-loaded if the unit is loaded from source or object file. **Touch > All Loaded** marks all loaded objects out of date.

2.4 The source path

Until now we have supposed that MLWorks looks for all files in the directory where it started up. However, this can be changed using the **Paths** menu on the MLWorks console. MLWorks keeps a notion of the source path, a list of directories which it should search, in order, for files and subdirectories.

Paths > Set Source Path shows a list of the directories on the current path, and allows you to edit it by adding and removing items. By default, the initial source path on starting MLWorks contains only the start-up directory. However, the source path may be changed by using the `Shell.Path.setSourcePath` function, which takes an argument of type `string list`; each string should be an absolute pathname, or a relative pathname to be interpreted relative to MLWorks' 'current directory' (initially the directory from which it is started up). If you want a particular source path set on start-up, you could put a call to this function in the MLWorks start-up file (see Section 1.9, "The MLWorks start-up file"). Alternatively you can set the desired source path in the `MLWORKS_SRC_PATH` environment variable before invoking MLWorks.

2.5 MLWorks units and compounds

MLWorks has a notion of units separated into compounds. Compounds can contain subcompounds; units and compounds are analogous to files and directories.

In the above example it was assumed that all source files were in the directory from which you invoked MLWorks (or other directories on the source path). Suppose instead that `zval.sml` is in a subdirectory `values` of that directory. If you now choose **File > Read Dependencies** and select `zval.sml` in the file chooser dialog, the corresponding unit will show up as `values.zval`.

MLWorks considers the unit to be in a compound called **values**. If **xval**, a top-level unit, still wants to require **values.zval**, the require statement must be changed:

file **xval.sml**

```
require "values.zval";
val x = z div 7;
```

file **zval.sml**
in directory **values**

```
val z = 42;
```

Require declarations are interpreted relative to the current compound. In this example, **xval** is in the top-level compound, so MLWorks will search the source path for a directory **values** containing a file **zval.sml**. If this require statement appeared in a unit **maths.xval**, MLWorks would look for the unit **maths.values.zval**.

The **^** character can be used to specify the parent compound: if the dependency between **xval** and **values.zval** were the other way round, **zval.sml** might be changed to read

file **zval.sml**

```
require "^.xval";
val z = x * 7;
```

A unit imported with **require** is searched for along the entire source path, so in this example MLWorks would search the entire path for **xval.sml**; it would not look only in the directory above **values**.

The **$** character denotes a notional root compound, so if a unit **A.B.C.foo** contained the line

```
require "$.bar";
```

it would denote the unit **bar**, not the unit **A.B.C.bar**.

2.6 Delivery using MLWorks

It is possible to deliver an image file or executable file from MLWorks. An image file contains an image of a particular function inside MLWorks. This is read by the run-time code, which recreates and evaluates the function. In the useful case this function has side-effects, which are the output of the program being delivered. An executable file is similar, but includes the run-time code as well, so it is free-standing.

Programs are delivered using the MLWorks 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;
- a boolean, which should be **`true`** for an executable file, **`false`** for an image file.

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 () =
  let open MLWorks.IO in output (std_out, "Hello, World.\n") end;
```

By invoking **`MLWorks.Deliver.deliver`** from a Listener, you can now deliver the function **`hello`** as an image file:

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

or as an executable file:

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

The image file is invoked using the **`mlrun`** command, part of the MLWorks distribution:

```
mlrun -load hello.img
Hello, World.
```

The executable file, considering what it does, may seem quite large. However, this is because it must contain a copy of the run-time code. This is a constant overhead (somewhat under 300 KB), so is less significant for larger programs. By contrast, the overhead for an image file is only about 6 KB.

3

Examining Objects

There are various tools in MLWorks that you can use to look more closely at the values and functions defined in your environment. Two of them we have already met: the browser and the inspector. Here we shall look at these in more detail.

3.1 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 **Action > 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.

3.1.1 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);
MLWorks> ilist 100;
```

Now choose **Action > 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 **View > 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 > 0**

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

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.

3.1.2 Display controls

The Inspector has another dialog affecting the graph: the **View > 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 **Atomic entries** box is not checked, tree-nodes consisting of a single atom (integers, reals, strings, etc) 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 **Equal values shared** 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.

3.1.3 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 **View > Value Printer** dialog. 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 **Action > Edit** to open the file in your preferred editor at the definition of `ilist`. For details of defining a preferred editor, see Section 1.6, "Editing ML files". Debugging mode is switched on using **Preferences > Mode** from the MLWorks console.

The Show exception details check-box has a similar effect when a value being inspected is an exception, and in general the options on this dialog all affect the contents of the Inspector's Value window in the same way as the corresponding options in the Listener.

3.2 Examining the environment: the Browser

Just as the you can examine a particular object with the Inspector, so you can look at the environment as a whole with the Browser. The Browser comes in two flavours: the System Browser lets you look at the functions and values built in to MLWorks, while the Context Browser lets you look at objects you have defined yourself.

As we have already seen in Chapter 1, one use of this is to invoke an Inspector on any object in the environment. To see more clearly the functions of a Browser, select **Tools > System Browser** from any MLWorks tool.

The Browser shows all the pervasive (built-in) objects that exist at top level: signatures, structures, types, values, and so on. A signature contains a list of further objects with their types, etc.; you can use the Browser to look inside a signature. Click on the `VECTOR` signature to select it, and click again to expand it.

The graph expands to show the types, exceptions and values specified in the `VECTOR` signature. The pervasive structure called `Vector` is defined to have this signature, and you can expand the `Vector` structure to see its contents. Since the Browser only shows the types of objects, not their values, this is of course the same as expanding the `VECTOR` signature.

If this seems unnecessarily repetitive, you can remove signatures (for example) from the Browser window. Select **View > Filters**, turn off the 'Show signatures' check-box and click **Apply**; all signatures will vanish from the Browser window. The other check-boxes on the filter dialog can be used to control exactly what is included in the Browser view.

There is also a **View > Graph Layout** dialog, whose entries have the same functions as the corresponding entries on an Inspector.

At the top of the Browser is a 'Selection' window, which contains the identifier of whatever object is currently selected. (An object is selected by clicking.) If the object is inside a structure, this window contains the full qualified name of the object. The contents of the window can be cut and pasted (into a Listener, for example), either using **Edit > Copy**, or with the mouse.

3.2.1 Inspecting objects from the Context Browser

Tools > Context Browser brings up the Context Browser, which is like the System Browser, but shows user-defined objects rather than built-in ones. You can invoke an Inspector from the Context Browser to inspect a value you have defined. Select the object and choose **Action > Inspect**. An inspector appears, showing details of the value in the usual way.

4

Debugging and tracing

We have already seen the Stack Browser in operation briefly in Chapter 1. Now we shall take a longer look at what it does and at the various ways in which it can be invoked.

Note: First ensure that debugging mode is on, and optimising mode is off. You can do this by invoking MLWorks with the `-debug-mode` option, or from the MLWorks console's **Preferences > Mode** dialog. (The **Preferences > 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 1.6. Finally, if your preferred editor is the Emacs server, you should make sure you have Emacs running and have started the server with the `M-x mlworks-server` command.

4.1 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 > Use File**.

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.

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 **Action > Edit** from the Stack Browser will put you into your preferred editor, passing the line number of the definition of `efact`.

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 **Action > Edit** from the Stack Browser to enter your preferred editor, passing the line number of the currently selected function.

4.2 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, choose **Break / Trace > Step Mode** from the MLWorks console. Now the next 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. For instance, having selected step mode (and with the definitions from the last section still in force), type

```
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.

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`. **Continue**, at any point, will switch off step mode and continue until the end of the evaluation.

4.3 Adding break points

Sometimes, when tracking down a bug, you might want to be able to stop a computation whenever a particular function is called. The Breakpoint Manager is for just this purpose. It is invoked from the MLWorks console with **Break / Trace > 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, type the function name into the text box and click **Add Name** followed by **Apply**.

4.4 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 pressing Ctrl-C. 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.

4.5 Tracing

Beside the Stack Browser, MLWorks has another facility to help the debugging process: tracing. It is controlled from the Trace Manager, invoked from the MLWorks console with **Break / Trace > Trace Manager**. The Trace Manager works in exactly the same way as the Breakpoint Manager: it contains a list of functions to be traced whenever they are called. You can add or remove functions in the list.

Whenever a function listed in the Trace Manager 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
```