Program Development under Inferno

Roger Peppé
rog@vitanuova.com

Introduction

Inferno provides a set of programs that, used in combination, provide a powerful development environment in which to write Limbo programs. *Limbo*(1) is the compiler for the Limbo language; there are versions that run inside and outside the Inferno environment. *Acme*(1) is an integrated window system and editor, and the preferred source-code editing tool within Inferno. The Limbo debugger, *wm-debug*(1), allows interactive inspection of running Limbo programs. *Stack*(1) allows a quick inspection of the execution stack of a currently running process.

Getting started

This document assumes that you have already managed to install Inferno and have managed to obtain an Inferno window, running the Inferno window manager, *wm*(1). The document ''Installing Inferno'' in this volume has details on this. If running within emu, it is worth giving Inferno as large a window as possible, as it cannot be resized later. This paper assumes that you are using a three-button mouse, as it is not feasible to use Acme without a three-button mouse. (if you have a two button mouse with a ''mouse wheel'', the wheel can be used as the middle button). The first thing to do is to get Acme going. By clicking on the Vita Nuova logo at the bottom left of the window, you can display a menu naming some preconfigured commands. If this has an ''Acme'' entry, then just clicking on that entry will start acme. If not, then click on the ''Shell'' entry, and type

```
acme
```

to start it up. The Acme window should then appear, filling most of the screen (the window manager toolbar should still be visible).

Acme basics

For a general overview and the rationale behind Acme, see ''Acme: A User Interface for Programmers'', elsewhere in this volume, and for detailed documentation, see *acme*(1). The basics are as follows:

Acme windows are text-only and organised into columns. A distinctive feature of Acme is that there are no graphical title bars to windows; instead, each window (and additionally each column, and the whole Acme window itself) has a textual *tag*, which can be edited at will, and is initially primed to contain a few appropriate commands.

An Acme command is just represented by text; any textual command word may be executed simply by clicking with the middle mouse button on the word. (See ''Acme mouse commands'', below). If Acme recognizes the word that has been clicked on as one of its internal commands (e.g. Put, Undo), then it will take the appropriate action; otherwise it will run the text as a shell command. (See *sh*(1)).

Acme mouse commands

Mouse usage within Acme is somewhat more versatile than in most other window systems. Each of the three mouse buttons has its own action, and there are also actions bound to *chords* of mouse buttons (i.e. mouse buttons depressed simultaneously). Mouse buttons are numbered from left (1) to right (3). Button 1 follows similar conventions to other window systems - it selects text; a double click will select a line if at the beginning or end of a line, or match brackets if on a bracket character, or select a word otherwise. Button 2, as mentioned above, executes an Acme command; a single click with button 2 will execute the single word under the click, otherwise the swept text will be executed. Button 3 is a general ''look'' operator; if the text under the click represents a filename, then Acme will open a new window for the file and read it in, otherwise it will search within the current window for the next occurrence of the text. Clicking button 2 or button 3 on some text already selected by button 1 causes the click to refer exactly to the text selected, rather than gathering likely-looking characters from around the click as is the default.

There are two mouse chord sequences which are commonly used in Acme (and you will find that some other programs in the system also recognise these sequences, e.g. *wm-sh*(1)). They are both available once some text has been selected by dragging the mouse with button 1, but before the button has been released. At this point, touching button 2 will delete the selected text and save it in Acme's *snarf* buffer; clicking button 3 replaces the selected text with the contents of the snarf buffer. Before button 1 has been released, these two buttons reverse each other's actions, so, for example, selecting some text with button 1, keeping button 1 held down, then clicking button 2 and button 3 in succession, will save the selected text in the snarf buffer while leaving the original intact. The following table summarises the mouse commands in Acme:

B1	Select text.
B2	Execute text.
B3	Open file or search for text.
B1- B2	Cut text.
B1- B3	Paste text.
B2- B3	Cancel the pending B2 action.
B3- B2	Cancel the pending B3 action.

Acme mouse command summary

Scrolling and resizing Acme windows

The scroll bars in Acme are somewhat different from conventional scroll bars (including the scroll bars found in other parts of Inferno). Clicking, or dragging, with button- 2on the scrollbar acts the most like the conventional behaviour, namely that the further down the scroll bar you click, the further down the file you are shown.

True to form, however, Acme doesn't omit to make the other buttons useful: button- 1and button- 3 move backwards and forwards through the file respectively. The nearer the top of the scrollbar the mouse, the slower the movement. Holding one of these buttons down on the scrollbar will cause the scrolling motion to auto- repeat,so it is easy to scroll gently through the entire file, for instance.

The small square at the top left of each Acme window is the handle for resizing the window. Dragging this square from one place to another (within Acme) will move the window to the new place. A single button click in this square will grow the window: button 1 grows it a little bit; button 2 grows it as much as possible without obscuring the other window titles in the column; button 3 grows it so it covers the whole column (all other windows in the column are obscured).

Creating a new file

All Limbo programs are composed of *modules* and each module is stored in its own file. To write a Limbo program, you need to write at least one module, the Limbo *source file*, which will then be compiled into Dis code which can then be run by the Inferno Virtual Machine (VM). The first step is to decide where to store the file. When Acme starts up, it creates a new window containing a list of all the files in the directory in which it was started (usually your home directory). As a consequence of the mouse rules above, a click of button- 3on any of those filenames in that window will open a new window showing that file or, if it is a directory, a list of the files and directories it contains.

An important aspect in Acme's mouse commands, is that the command is interpreted *relative to the window's current directory*, where the current directory is determined from the filename in the window's tag. For instance, Acme commands executed in the tag or body of a window on the file `/usr/joebloggs/myfile.txt` would run in the directory `/usr/joebloggs`.

So, to create a new file in Acme, first open the directory in which to create the file. (If this is your home directory, then it's probably already on the screen; otherwise, you can just type (anywhere) the name of the directory, and button- 3click on it. If the directory does not exist, then no window will be created. Then, within the directory's window or its tag, choose a name, *filename*, for your file (I'll use `myprog` from here on, for explanatory convenience) , type the text:

`New` *filename*`.b`

select this text (the Escape key can also be used to highlight text that you have just typed), and button- 2 click on it. This should create a new empty window in which you can edit your Limbo source file. It will

also create a window giving a warning that the file does not currently exist - you can get rid of this by clicking with button- 2on the text `Del` in the tag of that window.

Editing the source file

You can now edit text in the new window. Type in the following program:

```
implement Myprog;
include "sys.m";
    sys: Sys;
include "draw.m";

Myprog: module {
    init: fn(nil: ref Draw->Context, argv: list of string);
};

init(nil: ref Draw->Context, argv: list of string)
{
    sys = load Sys Sys->PATH;
    sys->print("Hello, world\n");
}
```

When typing it in, note that two new commands have appeared in the tag of the new window: `Put` and `Undo`. `Put` saves the file; `Undo` undoes the last change to the file, and successive executions of `Undo` will move further back in time. In case you move too far back accidentally, there is also `Redo`, which redoes a change that you have just undone. Changes in the body of any window in Acme can be undone this way.

Click with button- 2on the `Put` command, and the file is now saved and ready to be compiled. If you have problems at this point (say Acme complains about not being able to write the file), you have probably chosen an inappropriate directory, one in which you do not have write permission, in which to put the file. In this case you can change the name of the file simply by editing its name in the window's tag, and clicking on `Put` again.

Compiling the source file

Now, you are in a position to compile the Limbo program. Although you can execute the Limbo compiler directly from the tag of the new file's window, it is usually more convenient to do it from a shell window. To start a shell window, type ''`win`'' at the right of the tag of the new file's window, select it, and click with button- 2on it. A new window should appear showing a shell prompt (usually ''`; `'' or ''`% `''). At this, you can type any of the commands mentioned in Section 1 of the Programmer's Manual. Note that, following Acme's usual rule, the shell has started up in the same directory as the new file; typing

```
lc
```

at the prompt will show all the files in the directory, including hopefully the newly written Limbo file.

Type the following command to the shell:

```
limbo -g myprog.b
```

If you typed in the example program correctly, then you'll get a short pause, and then another shell prompt. This indicates a successful compilation (no news is good news), in which case you will now have two new files in the current directory, `myprog.sbl` and `myprog.dis`. The `-g` option to the `limbo` command directed it to produce the `myprog.sbl` file, which contains symbolic information relating the source code to the Dis executable file. The `myprog.dis` file contains the actual executable file. At this point, if you type `lc`, to get a listing of the files in the current directory, and then click with button- 2on the `myprog.dis` file, and you should see the output ''Hello, world''. You could also just type `myprog` at the shell prompt.

If you are normal, however, the above compilation probably failed because of some mistyped characters in the source code; and for larger newly created programs, in my experience, this is almost invariably the case. If you got no errors in the above compilation, try changing `sys->print` to `print`, saving the file again, and continue with the next section.

Finding compilation errors

When the Limbo compiler finds errors, it prints the errors, one per line, each one looking something like the following:

```
myprog.b:13: print is not declared
```

This shows the filename where the error has occurred, its line number in the file, and a description of the error. Acme's button-3 mouse clicking makes it extremely easy to see where in the source code the error has occurred. Click with button-3 anywhere in the filename on the line of the compilation error, and Acme will automatically take the cursor to the file of that name and highlight the correct line.

If there had been no currently appropriate open Acme window representing the file, then a new one would be created, and the appropriate line selected.

Edit `myprog.b` until you have a program that compiles successfully and produces the ''Hello, world'' output. For a program as simple as this, that's all there is to it - you now know the essential stages involved in writing a Limbo program; there's just the small matter of absorbing the Limbo language and familiarising yourself with the libraries (''The Limbo Programming Language'' elsewhere in this volume, and *intro*(2) are the two essential starting points here).

Finding run-time errors

For larger programs, there is the problem of programs that die unexpectedly with a run-time error. This will happen when, for instance, a Limbo program uses a reference that has not been initialised, or refers to an out-of-bounds array element.

When a Limbo program dies with a run-time exception, it does not go away completely, but remains hanging around, dormant, in a *broken* state; the state that it was in when it died may now be examined at leisure. To experiment with this, edit the Myprog module above to delete the line that loads the `Sys` module (`sys = load Sys`...), and recompile the program.

This time when you come to run `myprog`, it will die, printing a message like:

```
sh: 319 "Myprog":module not loaded
```

The number `319` is the *process id* (or just *pid*) of the broken process. The command `ps`, which shows all currently running processes, can be used at this point - you will see a line like this:

```
319      245        rog     broken    64K Myprog
```

The first number is the pid of the process; the second is the *process group* id of the process; the third field gives the owner of the process; the fourth gives its state (broken, in this case); the fifth shows the current size of the process, and the last gives the name of the module that the process is currently running.

The `stack` command can be used to quickly find the line at which the process has broken; type:

```
stack pid
```

where *pid* is the number mentioned in the ''module not loaded'' message (319 in this case). It produces something like the following output:

```
init() myprog.b:12.1, 29
unknown fn() Module /dis/sh.dis PC 1706
```

As usual, a quick button-3 click on the `myprog.b` part of the first line takes you to the appropriate part of the source file. The reason that the program has died here is that, in Limbo, all external modules must be explicitly loaded before they can be used; to try to call an uninitialised module is an error and causes an exception.

More sophisticated debugging

`Stack` is fine for getting a quick summary of the state in which a program has died, but there are times when such a simple post-mortem analysis is inadequate. The `wm/deb` (see *wm-deb*(1)) command provides an interactive windowing debugger for such occasions. It runs outside Acme, in the default window system. A convenient way to start debugging an existing process is to raise `wm/task` (''Task Manager'' on the main menu), select with the mouse the process to debug, and click ''Debug''. This will start `wm/deb` on that process. Before it can start, the debugger will ask for the names of any source files that it has not been able to find (usually this includes the source for the shell, as the module being debugged is often started by the shell, and so the top-level function will be in the shell's module).

`Wm/deb` can be used to debug multiple threads, to inspect the data structures in a thread, and to interactively step through the running of a thread (single stepping). See *wm-deb*(1) for details.