A Descent into Limbo

Brian W. Kernighan
bwk@bell-labs.com

ABSTRACT

If, reader, you are now slow to believe
What I shall tell, that is no cause for wonder,
For I who saw it hardly can accept it.
Dante Alighieri, *Inferno*, Canto XXV.

Limbo is a new programming language, designed by Sean Dorward, Phil Winterbottom, and Rob Pike. Limbo borrows from, among other things, C (expression syntax and control flow), Pascal (declarations), Winterbottom's Alef (abstract data types and channels), and Hoare's CSP and Pike's Newsqueak (processes). Limbo is strongly typed, provides automatic garbage collection, supports only very restricted pointers, and compiles into machine-independent byte code for execution on a virtual machine.

This paper is an introduction to Limbo. Since Limbo is an integral part of the Inferno system, the examples here illustrate not only the language but also a certain amount about how to write programs to run within Inferno.

1. Introduction

This document is a quick look at the basics of Limbo; it is not a replacement for the reference manual. The first section is a short overview of concepts and constructs; subsequent sections illustrate the language with examples. Although Limbo is intended to be used in Inferno, which emphasizes networking and graphical interfaces, the discussion here begins with standard text-manipulation examples, since they require less background to understand.

Modules:

A Limbo program is a set of modules that cooperate to perform a task. In source form, a module consists of a `module` declaration that specifies the public interface—the functions, abstract data types, and constants that the module makes visible to other modules—and an implementation that provides the actual code. By convention, the module declaration is placed in a separate `.m` file so it can be included by other modules, and the implementation is stored in a `.b` file. Modules may have multiple implementations, each in a separate implementation file.

At run time, modules are loaded dynamically; the `load` statement fetches the code and performs run-time type checking. Once a module has been loaded, its functions can be called.

Limbo is strongly typed; programs are checked at compile time, and further when modules are loaded. The Limbo compiler compiles each source file into a machine-independent byte-coded `.dis` file that can be loaded at run time.

Functions and variables:

Functions are associated with specific modules, either directly or as members of abstract data types within a module. Functions are visible outside their module only if they are part of the module interface. If the target module is loaded, specific names can be used in a qualified form like sys->print or without the qualifier if imported with an explicit import statement.

Besides normal block structure within functions, variables may have global scope within a module; module data can be accessed via the module pointer.

Data:

The numeric types are:

```
byte    unsigned, 8 bits
int     signed, 32 bits
big     signed, 64 bits
real    IEEE long float, 64 bits
```

The size and signedness of integral types are as specified above, and will be the same everywhere. Character constants are enclosed in single quotes and may use escapes like '\n' or '\udddd', but the characters themselves are in Unicode and have type int. There is no enumeration type, but there is a con declaration that creates a named constant.

Limbo also provides Unicode strings, arrays of arbitrary types, lists of arbitrary types, tuples (in effect, unnamed structures with unnamed members of arbitrary types), abstract data types or adts (in effect, named structures with function members as well as data members), reference types (in effect, restricted pointers that can point only to adt objects), and typed channels (for passing objects between processes).

A channel is a mechanism for synchronized communication. It provides a place for one process to send or receive an object of a specific type; the attempt to send or receive blocks until a matching receive or send is attempted by another process. The alt statement selects randomly but fairly among channels that are ready to read or write. The spawn statement creates a new process that, except for its stack, shares memory with other processes. Processes are pre-emptively scheduled by the Inferno kernel. (Inferno processes have much in common with threads in other operating systems.)

Limbo performs automatic garbage collection, so there is no need to free dynamically created objects. Objects are deleted and their resources freed when the last reference to them goes away. In general this release of resources happens immediately (instant free); release of cyclic data structures may be delayed.

Operators and expressions:

Limbo provides many of C s operators, but not the ?: or comma (sequential execution) operators. Pointers, created with ref, are very restricted and there is no & (address of) operator; there is no address arithmetic and pointers can only point to adt objects. Array slicing is supported, however, and replaces many pointer constructions.

There are no implicit coercions between types, and only a handful of explicit casts. The numeric types byte, int, etc., can be used to convert a numeric expression, as in

```
nl := byte 10;
```

and string can be used as a unary operator to convert any numeric expression or array of bytes to a string (in %g format).

Statements:

Statements and control flow in Limbo are similar to those in C. A statement is an expression followed by a semicolon, or a sequence of statements enclosed in braces. The similar control flow statements are

> `if (` *expr* `)` *stat*
> `if (` *expr* `)` *stat* `else` *stat*
> `while (` *expr* `)` *stat*
> `for (` *expr* `;` *expr* `;` *expr* `)` *stat*
> `do` *stat* `while (` *expr* `) ;`
> `return` *expr* `;`
> `exit ;`

The `exit` statement terminates a process and frees its resources. There is also a case statement analogous to C's `switch`; it also supports string and range tests. A `break` or `continue` followed by a label causes a break out of, or the next iteration of, the enclosing construct that is labeled with the same label.

Comments begin with `#` and extend to the end of the line. There is no preprocessor, but an `include` statement can be used to include source code, usually module declaration files.

Libraries:

Limbo has a small but growing set of standard libraries, each implemented as a module. A handful of these (notably `Sys`, `Draw`, and `Tk`) are included in the Inferno kernel because they will be needed to support almost any Limbo program. Among the others are `Bufio`, a buffered I/O package based on Plan 9's Bio; `Regex`, for regular expressions; and `Math`, for mathematical functions. Some of the examples that follow provide the sort of functionality that might be a suitable module.

2. Examples

The examples in this section are each complete, in the sense that they will run as presented; I have tried to avoid code fragments that merely illustrate syntax.

2.1. Hello, World

The first example is the traditional `hello, world`, in the file `hello.b`:

```
implement Hello;

include "sys.m";
        sys: Sys;
include "draw.m";

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

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

An implementation file implements a single module, named in the `implement` declaration at the top of the file. The two `include` lines copy interface definitions from two other modules, Sys (which describes a variety of system functions like print), and Draw (which describes a variety of graphics types and functions, only one of which, Context , is used here).

The module declaration defines the external interface that this module presents to the rest of the world. (This declaration is what would go into a hello.m in a larger example.) In this case, it's a single function named init . Since this module is to be called from a command interpreter (shell), by convention its init function takes two arguments, the graphical context and a list of strings, the command-line arguments, though neither is used here. This is like main in a C program. Essentially all of the other examples begin with this standard code.

Most modules have a more extensive set of declarations; for example, draw.m is 170 lines of constants, function prototypes, and type declarations for graphics types like Point and Rect , and sys.m is 120 lines of declarations for functions like open , read , and print . Most module declarations will also be stored in separate files, conventionally suffixed with .m , so they can be included in other modules.

The last few lines of hello.b are the implementation of the init function, which loads the Sys module, then calls its print function. By convention, each module declaration includes a pathname constant that points to the code for the module; this is the second parameter Sys->PATH of the load statement.

Compiling and Running Limbo Programs

With this much of the language described, we can compile and run this program. On Unix or Windows, the command

```
$ limbo -g hello.b
```

creates hello.dis , a byte-coded version of the program for the Dis virtual machine. The -g argument adds a symbol table, useful for subsequent debugging. The program can then be run as hello in Inferno; this shows execution under the Inferno emulator on a Unix system:

```
$ limbo -g hello.b
$ emu
Inferno main (pid=6559) interp
Initialize Dis: /dis/sh.dis
slocum.cs.bell-labs.com$ /usr/bwk/hello
hello, world
slocum.cs.bell-labs.com$
```

From within Inferno, it's also possible to run a program by selecting it from a menu. In any case, as the program runs, other modules that it uses will be loaded as necessary.

2.2. A Graphical "Hello World"

The following module creates and displays a button with the label hello, world as shown in the screenshot in Figure 1.

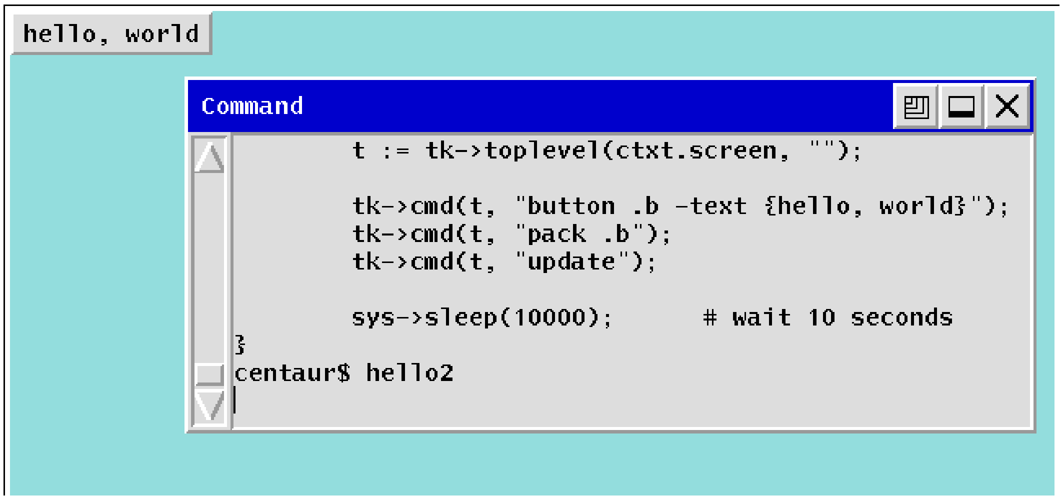

Figure 1. 'Hello, world' button.

```
implement Hello2;

include "sys.m";
        sys: Sys;
include "draw.m";
        draw: Draw;
include "tk.m";
        tk: Tk;

Hello2: module
{
        init: fn(ctxt: ref Draw->Context, argv: list of string);
};

init(ctxt: ref Draw->Context, argv: list of string)
{
        sys = load Sys Sys->PATH;
        draw = load Draw Draw->PATH;
        tk = load Tk Tk->PATH;

        t := tk->toplevel(ctxt.screen, "");

        tk->cmd(t, "button .b -text {hello, world}");
        tk->cmd(t, "pack .b");
        tk->cmd(t, "update");

        sys->sleep(10000);      # wait 10 seconds
}
```

This is not very exciting, but it illustrates the absolute minimum required to get a picture on the screen. The Tk module is modeled closely after John Ousterhout's Tk interface toolkit, but Limbo is used as the programming language instead of Tcl. The Inferno version is quite similar in functionality to the original, except that it does not support any Tcl constructs, e.g., variables, procedures, or expression evaluation. There are only six functions in the Tk interface, two of which are used here: toplevel, which makes a top-level window and returns a handle to it, and cmd, which executes a command string.

The sleep delays exit for 10 seconds so the button can be seen and pressed a few times. In a real application, some action would be bound to pressing the button.

Such actions are handled by setting up a channel from the Tk module to one s own code, and processing the events that appear on this channel. The function tk->namechan establishes a correspondence between a Limbo channel variable and a channel named as a string in the Tk module. When an event occurs in a Tk widget with a -command option, send causes the string to be sent on the channel and the Limbo code can act on it. In this example, the Limbo code is trivial: it waits for a message, discards the value, and exits. A more realistic example would have a loop that contains a case to process the strings that might appear on the channel.

```
t := tk->toplevel(ctxt.screen, "");

cmd := chan of string;
tk->namechan(t, cmd, "cmd"); # associate Limbo chan with Tk string
tk->cmd(t, "button .b -text {Delete me} -command {send cmd bye}");
tk->cmd(t, "pack .b");
tk->cmd(t, "update");

<- cmd; # wait for something to arrive on channel
```

The Tk module is explained in more detail later, as are a couple of other constructions.

2.3. Echo

The next example, echo, prints its command-line arguments. Declarations are the same as in the first example, and have been omitted.

```
# declarations omitted...

init(ctxt: ref Draw->Context, argv: list of string)
{
        sys = load Sys Sys->PATH;

        argv = tl argv;            # skip over program name
        for (s := ""; argv != nil; argv = tl argv)
                s += " " + hd argv;
        if (s != "")               # something was stored in s
                sys->print("%s\n", s[1:]);
}
```

The arguments are stored in a list. Lists may be of any type; argv is a list of string. There are three list operators: hd and tl return the head and tail of a list, and :: adds a new element to the head. In this example, the for loop walks along the argv list until the end, printing the head element (hd argv), then advancing (argv = tl argv).

The value nil is the undefined or explicitly empty value for non-numeric types.

The operator := combines the declaration of a variable and assignment of a value to it. The type of the variable on the left of := is the type of the expression on the right. Thus, the expression

```
s := ""
```

in the `for` statement declares a string `s` and initializes it to empty; if after the loop, `s` is not empty, something has been written in it. By the way, there is no distinction between the values `nil` and `""` for strings.

The `+` and `+=` operators concatenate strings. The expression `s[1:]` is a *slice* of the string `s` that starts at index 1 (the second character of the string) and goes to the end; this excludes the unwanted blank at the beginning of `s`.

2.4. Word Count

The word count program `wc` reads its standard input and counts the number of lines, words, and characters. Declarations have again been omitted.

```
# declarations omitted...

init(nil: ref Draw->Context, argv: list of string)
{
	sys = load Sys Sys->PATH;
	buf := array[1] of byte;

	stdin := sys->fildes(0);

	OUT: con 0;
	IN: con 1;

	state := OUT;
	nl := 0; nw := 0; nc := 0;
	for (;;) {
		n := sys->read(stdin, buf, 1);
		if (n <= 0)
			break;
		c := int buf[0];
		nc++;
		if (c == '\n')
			nl++;
		if (c == ' ' || c == '\t' || c == '\n')
			state = OUT;
		else if (state == OUT) {
			state = IN;
			nw++;
		}
	}
	sys->print("%d %d %d\n", nl, nw, nc);
}
```

This program contains several instances of the `:=` operator. For example, the line

```
	nl := 0; nw := 0; nc := 0;
```

declares three integer variables and assigns zero to each.

A Limbo program starts with three open files for standard input, standard output, and standard error, as in Unix. The line

```
	stdin := sys->fildes(0);
```

declares a variable `stdin` and assigns the corresponding file descriptor to it. The type of `stdin` is whatever the type of `sys->fildes(0)` is, and it's possible to get by without ever knowing the name of that type. (We will return to this shortly.)

The lines

```
OUT: con 0;
IN: con 1;
```

declare two integer constants with values zero and one. There is no enum type in Limbo; the con declaration is the closest equivalent.

Given the declarations of IN and OUT, the line

```
state := OUT;
```

declares state to be an integer with initial value zero.

The line

```
buf := array[1] of byte;
```

declares buf to be a one-element array of bytes. Arrays are indexed from zero, so buf[0] is the only element. Arrays in Limbo are dynamic, so this array is created at the point of the declaration. An alternative would be to declare the array and create it in separate statements:

```
buf : array of byte;      # no size at declaration

buf = array[1] of byte;   # size needed at creation
```

Limbo does no automatic coercion between types, so an explicit coercion is required to convert the single byte read from stdin into an int that can be used in subsequent comparisons with ints; this is done by the line

```
c := int buf[0];
```

which declares c and assigns the integer value of the input byte to it.

Warning: The word count program above tacitly assumes that its input is in the ASCII subset of Unicode, since it reads input one byte at a time instead of one Unicode character at a time. If the input contains any multi-byte Unicode characters, this code is plain wrong. The assignment to c is a specific example: the integer value of the first byte of a multi-byte Unicode character is not the character.

2.5. Word Count Version 2

There are several ways to address this shortcoming. Among the possibilities are rewriting to use the bufio module, which does string I/O, or checking each input byte sequence to see if it is a multi-byte character. The second version of word counting uses bufio. This example will also illustrate rules for accessing objects within modules.

```
# declarations omitted...

include "bufio.m";
        bufmod: Bufio;
Iobuf:  import bufmod;
        iob:    ref Iobuf;

init(nil: ref Draw->Context, nil: list of string)
{
        sys = load Sys Sys->PATH;
        bufmod = load Bufio Bufio->PATH;
        if (bufmod == nil) {
                sys->print("bufmod load: %r\n");
                exit;
        }

        stdin := sys->fildes(0);
        iob = bufmod->fopen(stdin, bufmod->OREAD);
        if (iob == nil) {
                sys->print("iob open: %r\n");
                exit;
        }

        OUT: con 0;
        IN: con 1;

        state := OUT;
        nl := 0; nw := 0; nc := 0;
        for (;;) {
                c := iob.getc();
                if (c == bufmod->EOF)
                        break;
                nc++;
                if (c == '\n')
                        nl++;
                if (c == ' ' || c == '\t' || c == '\n')
                        state = OUT;
                else if (state == OUT) {
                        state = IN;
                        nw++;
                }
        }
        sys->print("%d %d %d\n", nl, nw, nc);
}
```

The lines

```
include "bufio.m";
        bufmod: Bufio;
```

include the declarations from `bufio.m` and declare a variable `bufmod` that will serve as a handle when we load an implementation of the `Bufio` module. With this handle, we can refer to the functions and types the module defines, which are in the file `/usr/inferno/module/bufio.m`. Parts of this declaration are shown here:

```
Bufio: module     # edited to fit your screen
{
        PATH:    con "/dis/bufio.dis";
        EOF:     con -1;
        Iobuf:  adt {
                fd:       ref Sys->FD;        # the file
                buffer:   array of byte;      # the buffer
                                              # other variables omitted
                getc:     fn(b: self ref Iobuf) : int;
                gets:     fn(b: self ref Iobuf, sep: int) : string;
                close:    fn(b: self ref Iobuf);
        };
        open:    fn(name: string, mode: int) : ref Iobuf;
        fopen:   fn(fd: ref Sys->FD, mode: int) : ref Iobuf;
};
```

The bufio module defines open and fopen functions that return references to an Iobuf; this is much like a FILE* in the C standard I/O library. A reference is necessary so that all uses refer to the same entity, the object maintained by the module.

Given the name of a module (e.g., Bufio), how do we refer to its contents? It is always possible to use fully-qualified names, and the import statement permits certain abbreviations. We must also distinguish between the name of the module itself and a specific implementation returned by load, such as bufmod.

The fully-qualified name of a type or constant from a module is

Modulename -> *name*

as in Bufio->Iobuf or Bufio->EOF. To refer to members of an adt or functions or variables from a module, however, it is necessary to use a module handle instead of a module name; although the interface is always the same, the implementations of different instances of a module will be different, and we must refer to a specific implementation. A fully-qualified name is

modulehandle -> *functionname*
modulehandle -> *variablename*
modulehandle -> *adtname* . *membername*

where adt members can be variables or functions. Thus:

```
iob: ref bufmod->Iobuf;
bufmod->open(...)
bufmod->iob.getc()
bufmod->iob.fd
```

It is also legal to refer to module types, constants, and variables with a module handle, as in bufmod->EOF.

An import statement makes a specific list of names from a module accessible without need for a fully-qualified name. Each name must be imported explicitly, and adt member names can not be imported. Thus, the line

```
Iobuf: import bufmod;
```

imports the adt name Iobuf, which means that functions within that adt (like getc) can be used without module qualification, i.e., without bufmod->, but it is still necessary to say iob.getc(). In all cases, imported names must be unique.

The second parameter of load is the location of the module implementation, typically a .dis file. Some modules are part of the system; these have location names that begin with `$` but are otherwise the same for users. By convention, modules include a constant called `PATH` that points to their default location.

The call to `bufmod->fopen` attaches the I/O buffer to the already open file `stdin`; this is rather like `freopen` in `stdio`.

The function `iob.getc` returns the next Unicode character, or `bufmod->EOF` if end of file was encountered.

A close look at the calls to `sys->print` shows a new format conversion character, `%r`, for which there is no corresponding argument in the expression list. The value of `%r` is the text of the most recent system error message.

2.6. An Associative Array Module

This section describes a module that implements a conventional associative array (a hash table pointing to chained lists of name-value strings). This module is meant to be part of a larger program, not a standalone program like the previous examples.

The `Hashtab` module stores a name-value pair as a tuple of `(string, string)`. A tuple is a type consisting of an ordered collection of objects, each with its own type. The hash table implementation uses several different tuples.

The hash table module defines a type to hold the data, using an `adt` declaration. An `adt` defines a type and optionally a set of functions that manipulate an object of that type. Since it provides only the ability to group variables and functions, it is like a really slimmed-down version of a C++ class, or a slightly fancier C `struct`. In particular, an `adt` does not provide information hiding (all member names are visible if the `adt` itself is visible), does not support inheritance, and has no constructors, destructors or overloaded method names. To create an instance of an `adt`,

> *adtvar* `:=` *adtname* `(` *list of values for all members, in order* `);`
> *adtvar* `:=` `ref` *adtname* `(` *list of values for all members, in order* `);`

Technically these are casts, from tuple to `adt`; that is, the `adt` is created from a tuple that specifies all of its members in order.

The `Hashtab` module contains an `adt` declaration for a type `Table`; the operations are a function `alloc` for initial allocation (in effect a constructor), a hash function, and methods to add and look up elements by name. Here is the module declaration, which is contained in file `hashtab.m`:

```
Hashtab: module
{
    PATH:    con "/usr/bwk/hashtab.dis";       # temporary name

    Table: adt {
        tab: array of list of (string, string);

        alloc: fn(n: int) : ref Table;

        hash: fn(ht: self ref Table, name: string) : int;
        add: fn(ht: self ref Table, name: string, val: string);
        lookup: fn(ht: self ref Table, name: string) : (int, string);
    };
};
```

The implementation is in file `hashtab.b`:

```
implement Hashtab;

include "hashtab.m";

Table.alloc(n: int) : ref Table
{
	return ref Table(array[n] of list of (string,string));
}

Table.hash(ht: self ref Table, s: string) : int
{
	h := 0;
	for (i := 0; i < len s; i++)
		h = (h << 1) ^ int s[i];
	h %= len ht.tab;
	if (h < 0)
		h += len ht.tab;
	return h;
}

Table.add(ht: self ref Table, name: string, val: string)
{
	h := ht.hash(name);
	for (p := ht.tab[h]; p != nil; p = tl p) {
		(tname, nil) := hd p;
		if (tname == name) {
			# illegal: hd p = (tname, val);
			return;
		}
	}
	ht.tab[h] = (name, val) :: ht.tab[h];
}

Table.lookup(ht: self ref Table, name: string) : (int, string)
{
	h := ht.hash(name);
	for (p := ht.tab[h]; p != nil; p = tl p) {
		(tname, tval) := hd p;
		if (tname == name)
			return (1, tval);
	}
	return (0, "");
}
```

This is intentionally simple-minded, to focus on the language rather than efficiency or flexibility. The function `Table.alloc` creates and returns a `Table` with a specified size and an array of elements, each of which is a list of `(string, string)`.

The `hash` function is trivial; the only interesting point is the `len` operator, which returns the number of items in an object. For a string, `len s` is the number of Unicode characters.

The `self` declaration says that the first argument of every call of this function is implicit, and refers to the object itself; this argument does not appear at any call site. `Self` is similar to `this` in C++.

The `lookup` function searches down the appropriate list for an instance of the `name` argument. If a match is found, `lookup` returns a tuple consisting of 1 and the value field; if no match is found, it returns a tuple of 0 and an empty string. These return types match the function return type, (int, string).

The line

```
(tname, tval) := hd p;
```

shows a tuple on the left side of a declaration-assignment. This splits the pair of strings referred to by hd p into components and assigns them to the newly declared variables tname and tval.

The add function is similar; it searches the right list for an instance of the name. If none is found,

```
ht.tab[h] = (name, val) :: ht.tab[h];
```

combines the name and value into a tuple, then uses :: to stick it on the front of the proper list.

The line

```
(tname, nil) := hd p;
```

in the loop body is a less obvious use of a tuple. In this case, only the first component, the name, is assigned, to a variable tname that is declared here. The other component is assigned to nil, which causes it to be ignored.

The line

```
# illegal: hd p = (tname, val);
```

is commented out because it is illegal: Limbo does not permit the assignment of a new name-value to a list element; list elements are immutable.

To create a new Table, add some values, then retrieve one, we can write:

```
nvtab = Table.alloc(101);       # make a Table

nvtab.add("Rob", "Pike");
nvtab.add("Howard", "Trickey");
(p, phil) := nvtab.lookup("Phil");
(q, sean) := nvtab.lookup("Sean");
```

Note that the ref Table argument does not appear in these calls; the self mechanism renders it unnecessary.

2.7. An AWK-like Input Module

This example presents a simple module based on Awk's input mechanism: it reads input a line at a time from a list of of files, splits each line into an array of NF+1 strings (the original input line and the individual fields), and sets NF, NR, and FILENAME. It comes in the usual two parts, a module:

```
Awk: module
{
        PATH:           con "/usr/bwk/awk.dis";

        init:           fn(argv: list of string);
        getline:        fn() : array of string;
        NR:             fn() : int;
        NF:             fn() : int;
        FILENAME:       fn() : string;
};
```

and an implementation:

```
implement Awk;

include "sys.m";
	sys:	Sys;
include "bufio.m";
	bufio: Bufio;
Iobuf: import bufio;
	iobuf:	ref Iobuf;

include "awk.m";

_NR:		int;
_NF:		int;
_FILENAME:	string;
argv:		list of string;

init(av: list of string)
{
	argv = tl av;
	if (len argv == 0)	# no args => stdin
		argv = "-" :: nil;

	sys = load Sys Sys->PATH;
	bufio = load Bufio Bufio->PATH;
}

getline() : array of string
{
	t := array[100] of string;
	fl : list of string;

  top:
	while (argv != nil) {
		if (_FILENAME == nil) { # advance to next file
			_FILENAME = hd argv;
			if (_FILENAME == "-")
				iobuf = bufio->fopen(sys->fildes(0), bufio->OREAD);
			else
				iobuf = bufio->open(_FILENAME, bufio->OREAD);
			if (iobuf == nil) {
				sys->print("getline %s: %r\n", _FILENAME);
				argv = nil;
				return nil;
			}
		}

		s := iobuf.gets('\n');
		if (s == nil) {
			iobuf.close();
			_FILENAME = nil;
			argv = tl argv;
			continue top;
		}
```

```
            t[0] = s[0:len s - 1];
            _NR++;
            (_NF, fl) = sys->tokenize(t[0], " \t\n\r");
            for (i := 1; fl != nil; fl = tl fl)
                t[i++] = hd fl;
            return t[0:i];
        }
        return nil;
    }

    NR() : int     { return _NR; }
    NF() : int     { return _NF; }
    FILENAME() : string { return _FILENAME; }
```

Since `NR`, `NF` and `FILENAME` should not be modified by users, they are accessed as functions; the actual variables have related names like `_NF`. It would also be possible to make them ordinary variables in the `Awk` module, and refer to them via a module handle.

The `tokenize` function in the line

```
        (_NF, fl) = sys->tokenize(t[0], " \t\n\r");
```

breaks the argument string `t[0]` into tokens, as separated by the characters of the second argument. It returns a tuple consisting of a length and a list of tokens. Note that this module has an `init` function that must be called explicitly before any of its other functions are called.

2.8. A Simple Formatter

This program is a simple-minded text formatter, modeled after `fmt`, that tests the Awk module:

```
implement Fmt;

include "sys.m";
        sys: Sys;
include "draw.m";

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

include "awk.m";
        awk: Awk;
        getline, NF: import awk;

out:    array of string;
nout:   int;
length: int;
linelen := 65;
```

```
init(nil: ref Draw->Context, argv: list of string)
{
	t: array of string;
	out = array[100] of string;

	sys = load Sys Sys->PATH;
	awk = load Awk Awk->PATH;
	if (awk == nil) {
		sys->print("load awk: %r\n");
		return;
	}
	awk->init(argv);

	nout = 0;
	length = 0;
	while ((t = getline()) != nil) {
		nf := NF();
		if (nf == 0) {
			printline();
			sys->print("\n");
		} else for (i := 1; i <= nf; i++) {
			if (length + len t[i] > linelen)
				printline();
			out[nout++] = t[i];
			length += len t[i] + 1;
		}
	}
	printline();
}
printline()
{
	if (nout == 0)
		return;
	for (i := 0; i < nout-1; i++)
		sys->print("%s ", out[i]);
	sys->print("%s\n", out[i]);
	nout = 0;
	length = 0;
}
```

The functions getline and NF have been imported so their names need no qualification.

2.9. Channels and Communications

Another approach to a formatter is to use one process to fetch words and pass them to another process that formats and prints them. This is easily done with a channel, as in this alternative version:

```
# declarations omitted...

WORD, BREAK, EOF : con iota;
wds: chan of (int, string);

init(nil: ref Draw->Context, nil: list of string)
{
        sys = load Sys Sys->PATH;
        bufmod = load Bufio Bufio->PATH;

        stdin := sys->fildes(0);
        iob = bufmod->fopen(stdin, bufmod->OREAD);

        wds = chan of (int, string);
        spawn getword(wds);
        putword(wds);
}

getword(wds: chan of (int, string))
{
        while ((s := iob.gets('\n')) != nil) {
                (n, fl) := sys->tokenize(s, " \t\n");
                if (n == 0)
                        wds <-= (BREAK, "");
                else for ( ; fl != nil; fl = tl fl)
                        wds <-= (WORD, hd fl);
        }
        wds <-= (EOF, "");
}

putword(wds: chan of (int, string))
{
        wd: int;
        s: string;
        for (length := 0;;) {
                (wd, s) =<- wds;
                case wd {
                BREAK =>
                        sys->print("\n\n");
                        length = 0;
                WORD =>
                        if (length + len s > 65) {
                                sys->print("\n");
                                length = 0;
                        }
                        sys->print("%s ", s);
                        length += len s + 1;
                EOF =>
                        sys->print("\n");
                        exit;
                }
        }
}
```

This omits declarations and error checking in the interest of brevity.

The operator `iota` is used in `con` declarations to produce the sequence of values 0, 1, ...

The channel passes a tuple of (int, string); the int indicates what kind of string is present a real word, a break caused by an empty input line, or EOF.

The spawn statement creates a separate process by calling the specified function; except for its own stack, this process shares memory with the process that spawned it. Any synchronization between processes is handled by channels.

The operator <-= writes an expression to a channel; the operator =<- reads from a channel and assigns to a variable. In this example, getword and putword alternate, because each input word is sent immediately on the shared channel, and no subsequent word is processed until the previous one has been received and printed.

The case statement consists of a list of case values, which must be string or numeric constants, followed by => and associated code. The value * (not used here) labels the default. Multiple labels can be used, separated by the or operator, and ranges of values can appear delimited by to, as in

```
'a' to 'z' or 'A' to 'Z' =>
```

2.10. Tk and Interface Construction

Inferno supports a rather complete implementation of the Tk interface toolkit developed by John Ousterhout. In other environments, Tk is normally accessed from Tcl programs, although there are also versions for Perl, Scheme and other languages that call Ousterhout's C code. The Inferno Tk was implemented from scratch, and is meant to be called from Limbo programs. There is a module declaration tk.m and a kernel module Tk.

The Tk module provides all the widgets of the original Tk with almost all their options, the pack command for geometry management, and the bind command for attaching code to user actions. In this implementation Tk commands are written as strings and presented to one function, tk->cmd; Limbo calls this function and captures its return value, which is the string that the Tk command produces. For example, widget creation commands like button return the widget name, so this will be the string returned by tk->cmd.

There is one unconventional aspect: the use of a channel to send events from the interface into the Limbo program. To create a widget, as we saw earlier, one writes

```
tk->cmd("button .b -text {Push me} -command {send cmd .bpush}");
```

to create a button .b and attach a command to be executed when the button is pushed. That command sends the (arbitrary) string .bpush on the channel named cmd. The Limbo code that reads from this channel will look for the string .bpush and act accordingly. The link between a channel variable in Limbo and the string sent from the Tk library is established by the function Tk->namechan.

This is all illustrated in the program below, which implements a trivial version of Etch-a-Sketch, shown in action in Figure 2.

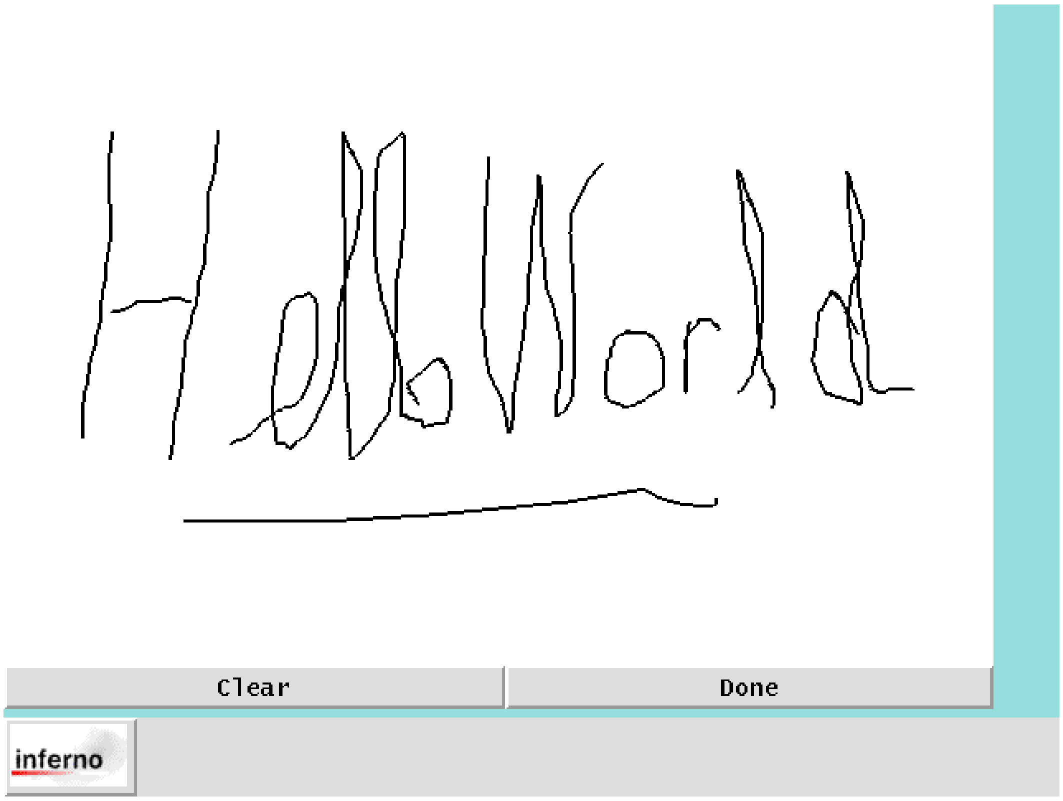

Figure 2. Etch−a−Sketch display.

```
implement Etch;

include "sys.m";
     sys: Sys;
include "draw.m";
     draw: Draw;
include "tk.m";
     tk: Tk;

Etch: module
{
          init:     fn(ctxt: ref Draw->Context, argv: list of string);
};
```

```
init(ctxt: ref Draw->Context, argv: list of string)
{
	sys = load Sys Sys->PATH;
	draw = load Draw Draw->PATH;
	tk = load Tk Tk->PATH;

	x, y, lastx, lasty: int;

	t := tk->toplevel(ctxt.screen, "");

	cmd := chan of string;
	tk->namechan(t, cmd, "cmd");
	tk->cmd(t, "canvas .c -height 400 -width 600 -background white");
	tk->cmd(t, "frame .f");
	tk->cmd(t, "button .f.c -text {Clear} -command {send cmd clear}");
	tk->cmd(t, "button .f.d -text {Done} -command {send cmd quit}");
	tk->cmd(t, "pack .f.c .f.d -side left -fill x -expand 1");
	tk->cmd(t, "pack .c .f -side top -fill x");
	tk->cmd(t, "bind .c <ButtonPress-1> {send cmd b1down %x %y}");
	tk->cmd(t, "bind .c <Button-1-Motion> {send cmd b1motion %x %y}");
	tk->cmd(t, "update");
	for (;;) {
			s := <-cmd;
		(n, cmdstr) := sys->tokenize(s, " \t\n");
		case hd cmdstr {
		"quit" =>
			exit;
		"clear" =>
			tk->cmd(t, ".c delete all; update");
		"b1down" =>
			lastx = int hd tl cmdstr;
			lasty = int hd tl tl cmdstr;
			cstr := sys->sprint(".c create line %d %d %d %d -width 2",
				lastx, lasty, lastx, lasty);
			tk->cmd(t, cstr);
		"b1motion" =>
			x = int hd tl cmdstr;
			y = int hd tl tl cmdstr;
			cstr := sys->sprint(".c create line %d %d %d %d -width 2",
				lastx, lasty, x, y);
			tk->cmd(t, cstr);
			lastx = x; lasty = y;
		}
		tk->cmd(t, "update");
	}
}
```

The program creates a canvas for drawing, a button to clear the canvas, and a button to quit. The sequence of calls to `tk->cmd` creates the picture and sets up the bindings. The expression

```
s := <-cmd
```

declares a variable `s` of the type returned by the channel `cmd`, i.e. a `string`; when a string is received on the channel, the assignment is executed.

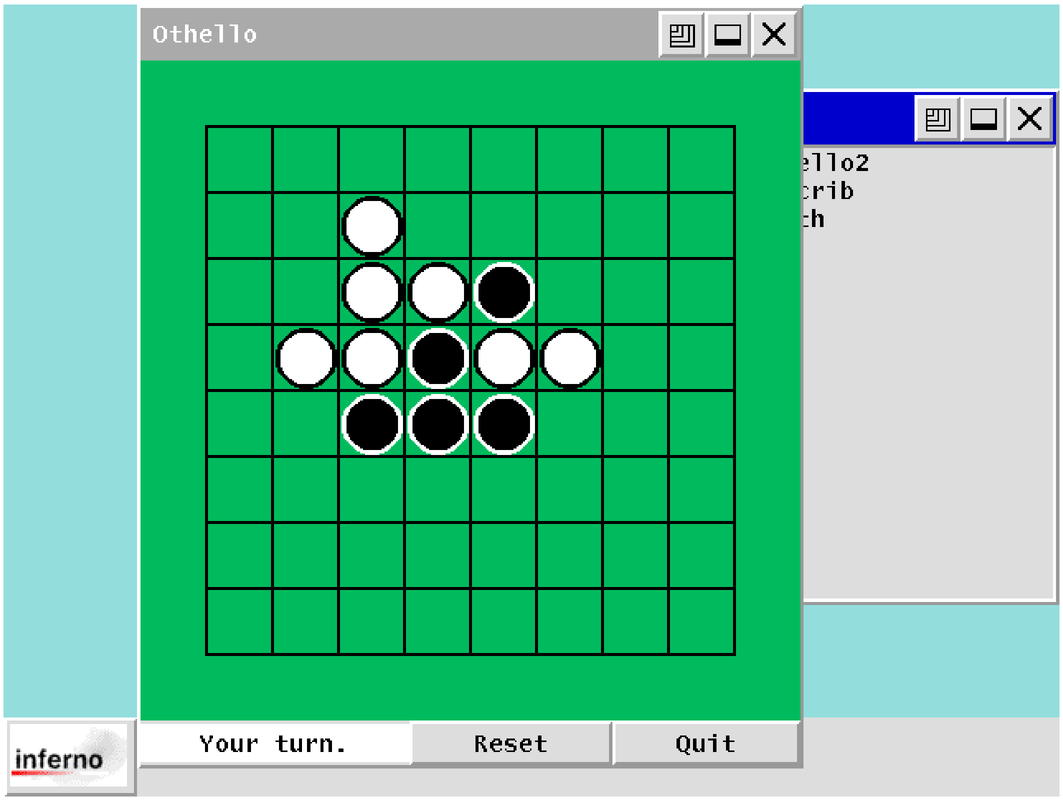

Figure 3. Screen shot of Inferno display showing Othello window.

2.11. Adding a Menubar

Normally, a graphical application is meant to run under the window manager wm as a window that can be managed, reshaped, etc. This is best done by calling upon the function titlebar in the window library Wmlib. Here is the startup code for an implementation of Othello, adapted from a Java version by Muffy Barkocy, Arthur van Hoff, and Ben Fry.

```
init(ctxt: ref Draw->Context, argv: list of string)
{
     sys = load Sys Sys->PATH;
     draw = load Draw Draw->PATH;
     tk = load Tk Tk->PATH;
     wmlib = load Wmlib Wmlib->PATH;

     wmlib->init();

     tkargs := "";
     argv = tl argv;         # remove program name
     if (argv != nil) {      # and extract any -geom arg
          tkargs = hd argv;
          argv = tl argv;
     }
     (t, menubut) := wmlib->titlebar(ctxt.screen, tkargs,
                                          "Othello", Wmlib->Appl);

     cmd := chan of string;
     tk->namechan(t, cmd, "cmd");
     tk->cmd(t, "canvas .c -height 400 -width 400 -background green");
     tk->cmd(t, "frame .f");
     tk->cmd(t, "label .f.l -text {Othello?} -background white");
     tk->cmd(t, "button .f.c -text {Reset} -command {send cmd Reset}");
     tk->cmd(t, "button .f.d -text {Quit} -command {send cmd Quit}");
     tk->cmd(t, "pack .f.l .f.c .f.d -side left -fill x -expand 1");
     tk->cmd(t, "pack .Wm_t .c .f -side top -fill x");
     tk->cmd(t, "bind .c <ButtonRelease-1> {send cmd B1up %x %y}");

     for (i := 1; i < 9; i++)
     for (j := 1; j < 9; j++) {
          coord := sys->sprint("%d %d %d %d",
               SQ*i, SQ*j, SQ*(i+1), SQ*(j+1));
          tk->cmd(t, ".c create rectangle " + coord +
               " -outline black -width 2");
     }
     tk->cmd(t, "update");
     lasterror(t, "init");

     board = array[10] of {* => array[10] of int};
     score = array[10] of {* => array[10] of int};
     reinit();

     for (;;) {
          alt {
          s := <- cmd =>
               (n, l) := sys->tokenize(s, " \t");
               case hd l {
               "Quit" =>
                    exit;
               "Reset" =>
                    reinit();
               "B1up" =>
                    x := int hd tl l;
                    y := int hd tl tl l;
                    mouseUp(int x, int y);
               }

          menu := <-menubut =>
```

```
                wmlib->titlectl(t, menu);
            }
        }
    }
```

The titlebar function returns a tuple containing the Tk->Toplevel for the new window and a channel upon which events like hitting the exit button will appear. It also gives the titlebar a conventional name, .Wm_t .

Note that now there are two channels watching events, one for the buttons and canvas within the Othello game itself, and one for the menubar. This time we need an alt statement to select from events on either channel. The value returned from the menubut channel indicates what the user did; everything is passed back to the titlectl function to be handled there.

If some call to the Tk module results in an error, an error string is made available in a pseudo-variable lasterror maintained by Tk . When this variable is read, it is reset. The function lasterror shows how to test and print this variable:

```
lasterror(t: ref Tk->Toplevel, where: string)
{
    s := tk->cmd(t, "variable lasterror");
    if (s != nil)
        sys->print("%s: tk error %s\n", where, s);
}
```

In general, the Inferno implementation of Tk does not provide variables except for a few special ones like this. The most common instance is a variable that links a set of radiobuttons.

2.12. Acknowledgements

I am very grateful to Steven Breitstein, Ken Clarkson, Sean Dorward, Eric Grosse, Doug McIlroy, Rob Pike, Jon Riecke, Dennis Ritchie, Howard Trickey, Phil Winterbottom, and Margaret Wright for explaining mysteries of Limbo and Inferno and for valuable suggestions on this paper.