A Descent into Limbo

Brian W. Kernighan

bwk@bell-labs.com

Revised April 2005 by Vita Nuova

ABSTRACT

''If, reader, you are slow now 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 a brief 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 usually 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.

Modules are always loaded dynamically, at run time: the Limbo `load` operator fetches the code and performs run-time type checking. Once a module has been loaded, its functions can be called. Several instances of the same module type can be in use at once, with possibly different implementations.

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, and a special `iota` operation that can be used to generate unique values.

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 adt's (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 are sometimes called ''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. This release of resources happens immediately (''instant free'') for non- cyclicstructures; release of cyclic data structures might be delayed but will happen eventually. (The language allows the programmer to ensure a given structure is non- cyclic when required.)

Operators and expressions:

Limbo provides many of C's operators, but not the `?:` or 'comma' (sequential execution) operators. Pointers, or 'references', created with `ref`, are restricted compared to C: they can only refer to adt values on the heap. There is no `&` (address of) operator, nor is address arithmetic possible. Arrays are also reference types, however, and since array slicing is supported, that replaces many of C's 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 to a string (in `%g` format) and to convert an array of bytes in UTF- 8format to a Limbo `string` value. In the other direction, the cast `array of byte` converts a string to its UTF- 8representation in an array of bytes.

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`, but it differs in that it also supports string and range tests, and more critically, control flow does not ''flow through'' one arm of the case to another but stops without requiring an explicit `break` (in that respect it is closer to Pascal's `case` statement, hence the change of name). 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 an extensive and 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, args: list of string);
};

init(ctxt: ref Draw->Context, args: 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. 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- linearguments, though neither is used here. This is like main in a C program. Essentially all of the other examples begin with this standard code. Commands are unusual, though, in that a command's module declaration appears in the same file as its implementation.

Most modules have a more extensive set of declarations; for example, draw.m is 298 lines of constants, function prototypes, and type declarations for graphics types like Point and Rect, and sys.m is 160 lines of declarations for functions like open, read, and print. Most module declarations are therefore stored in separate files, conventionally suffixed with .m, so they can be included in other modules. The system library module declaration files are collected in the module directory at the root of the Inferno source tree. Modules that are components of a single program are typically stored in that program's source directory.

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. Note that the Draw module is not loaded because none of its functions is used, but it is included to define the type Draw->Context.

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- codedversion of the program for the Dis virtual machine. The -g argument adds a symbol table, useful for subsequent debugging. (Another common option is -w, which causes the compiler to produce helpful warnings about possible errors.) 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
; /usr/bwk/hello
hello, world
;
```

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

2.2. A Graphical "Hello World"

The following module creates and displays a window containing only a button with the label ''hello, world'' as shown in the screen shot in Figure 1.

```
implement Hello2;

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

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

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

        tkclient->init();

        (t, nil) := tkclient->toplevel(ctxt, "", "Hello", Tkclient->Plain);

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

        tkclient->onscreen(t, nil);

        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 similar in functional-

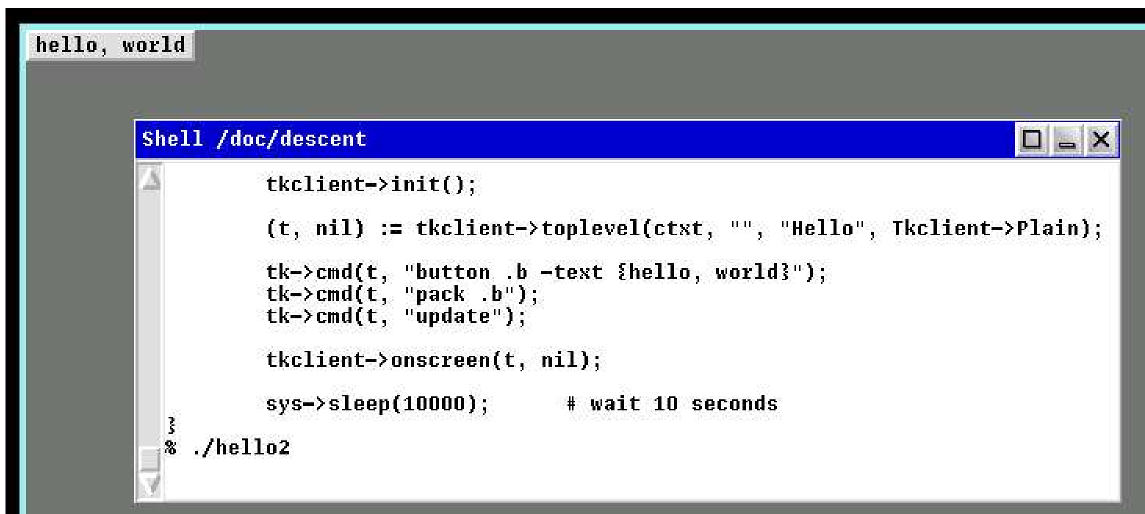

Figure 1. 'Hello, world' button.

ity to the original Tk but it does not support any Tcl constructs, such as variables, procedures, or expression evaluation, since all processing is done using Limbo. There are ten functions in the `Tk` interface, only one of which is used here: `cmd`, which executes a command string. (It is the most commonly used `Tk` function.)

Tk itself displays graphics and handles mouse and keyboard interaction within a window. There can however be many different windows on a display. A separate window manager, `wm`, multiplexes control of input and output among those windows. The module `Tkclient` provides the interface between the window manager and Tk. Its function `toplevel`, used above, makes a top- level window and returns a reference to it, for subsequent use by Tk. The contents of the window are prepared by calls to `tk->cmd` before the window is finally displayed by the call to `onscreen`. (The second parameter to `onscreen`, a string, controls the position and style of window; here we take the default by making that `nil`.)

Note that `Tkclient` must also be explicitly initialized by calling its `init` function after loading. This is a common convention, although some modules do not require it (typically those built in to the system, such as `Sys` or `Tk`).

The `sleep` delays exit for 10 seconds so the button can be seen. If you try to interact with the window, for instance by pressing the button, you will see no response. That is because the program has not done what is required to receive mouse or keyboard input in the window. In a real application, some action would also be bound to pressing the button. Such actions are handled by setting up a connection (a 'channel') from the Tk module to one's own code, and processing the messages ('events') that appear on this channel. The Tk module and its interface to the window manager is explained in more detail later, as are a couple of other constructions, after we have introduced processes and channels.

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, args: list of string)
{
        sys = load Sys Sys->PATH;

        args = tl args;         # skip over program name
        for (s := ""; args != nil; args = tl args)
                s += " " + hd args;
        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; `args` 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 `args` list until the end, printing the head element (`hd args`), then advancing (`args = tl args`).

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, args: 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. When the values are arbitrary, a different form is normally used:

```
OUT, IN: con iota;
```

The operator `iota`, when used in `con` declarations will produce the sequence of values 0, 1, ..., one value in turn for each name declared in the same declaration. It can appear in more complex expressions:

```
M1, M2, M4, M8: con 1 << iota;
N1, N3, N5, N7: con (2*iota)+1;
```

The first example generates a set of bitmask values; the second generates a sequence of odd numbers.

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 coercions 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 `int`'s; this is done by the line

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

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

2.5. Word Count Version 2

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- byteUnicode characters, this code is plain wrong. The assignment to `c` is a specific example: the integer value of the first byte of a multi- byteUnicode character is not the character.

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- bytecharacter. The second version of word counting uses `Bufio`. This example will also illustrate rules for accessing objects within modules.

```
# declarations omitted...

include "bufio.m";
        bufio: Bufio;
        Iobuf: import bufio;

init(nil: ref Draw->Context, nil: list of string)
{
        sys = load Sys Sys->PATH;
        bufio = load Bufio Bufio->PATH;
        if (bufio == nil) {
                sys->fprint(sys->fildes(2), "wc: can't load %s: %r\n", Bufio->PATH);
                raise "fail:load";
        }

        stdin := sys->fildes(0);
        iob := bufio->fopen(stdin, bufio->OREAD);
        if (iob == nil) {
                sys->fprint(sys->fildes(2), "wc: can't open stdin: %r\n");
                raise "fail:open";
        }

        OUT, IN: con iota;

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

The lines

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

include the declarations from `bufio.m` and declare a variable `bufio` that will serve as a handle when we load an implementation of the `Bufio` module. (The use of a module's type in lower case as the name of a loaded instance is a common convention in Limbo programs.) With this handle, we can refer to the functions and types the module defines, which are in the file `/usr/inferno/module/bufio.m` (the full name might be different on your system). 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- qualifiednames, 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 `bufio`.

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 value 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

moduleval`->`*functionname*
moduleval`->`*variablename*
moduleval`->`*adtname*`.`*membername*

where adt members can be variables or functions. Thus:

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

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

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

```
Iobuf: import bufio;
```

imports the adt name `Iobuf`, which means that functions within that adt (like `getc`) can be used without module qualification, i.e., without `bufio->`. (It is still necessary to say `iob.getc()` for reasons given below.) In all cases, imported names must be unique.

The second parameter of `load` is a string giving the location of the module implementation, typically a `.dis` file. (The string need not be static.) 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 `bufio->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 `bufio->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.

Several other small changes were made as realistic examples: it keeps the counts as `big` to cope with larger files (hence the use of `%bd` as the output format); it prints diagnostics on the standard error stream, `sys->fildes(2)`, using `sys->fprint`, a variant of `sys->print` that takes an explicit file descriptor; and it returns an error status to its caller (typically the shell) by raising an exception.

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- valuestrings). 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- valuepair 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- downversion 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. It is different from C or C++, however: when an adt is declared by a `module` declaration, the adt's implementation (the bodies of its functions) will be defined by the module's implementation, and there can be more than one. 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 a string, array or list. 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 value itself; this argument does not appear in the actual parameter list 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's 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. Remember that a module using `Table` must `import` it from some instance of `Hashtab`, or qualify all references to it by a module value.

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(args: 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;
args:       list of string;

init(av: list of string)
{
    args = tl av;
    if (len args == 0)  # no args => stdin
        args = "-" :: 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 (args != nil) {
        if (_FILENAME == nil) { # advance to next file
            _FILENAME = hd args;
            if (_FILENAME == "-")
                iobuf = bufio->fopen(sys->fildes(0), bufio->OREAD);
            else
                iobuf = bufio->open(_FILENAME, bufio->OREAD);
            if (iobuf == nil) {
                sys->fprint(sys->fildes(2), "can't open %s: %r\n", _FILENAME);
                args = nil;
                return nil;
            }
        }

        s := iobuf.gets('\n');
        if (s == nil) {
            iobuf.close();
            _FILENAME = nil;
            args = tl args;
            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 value (i.e., awk->NR).

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 afterfmt, that tests the Awk module:

```
implement Fmt;

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

Fmt: module
{
        init:   fn(nil: ref Draw->Context, args: 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, args: 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->fprint(sys->fildes(2), "fmt: can't load %s: %r\n",
                        Awk->PATH);
                raise "fail:load";
        }
        awk->init(args);

        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. It is more usual Limbo style to use explicit references such as `sys->read` or `Bufio->EOF` for clarity, and import only adts (and perhaps commonly used constants).

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;
        bufio = load Bufio Bufio->PATH;

        stdin := sys->fildes(0);
        iob = bufio->fopen(stdin, bufio->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))
{
        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 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 `<-=` sends an expression to a channel; the operator `<-` receives from a channel. (Receive is combined here with `:=` to receive a tuple, and assign its elements to newly- declared variables.) 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' =>
```

Remember that control does not flow from one case arm to the next, unlike C, thus no break statements appear.

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. As we saw earlier, 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. It also provides a grid command to simplify the common case of objects arranged in a matrix or grid. 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 channels to send data and 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 function tk->namechan establishes a correspondence between a Limbo channel value 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. The program will often use a case to process the strings that might appear on the channel, particularly when the same channel is used for several widgets.

We observed earlier that Tk provides a user interface for an application's window, but there might be many windows on the screen. Normally, a graphical application is meant to run under the window manager wm as a window that can be managed, reshaped, etc. This is done by calling functions in the module Tkclient, which provides the interface between Tk and wm.

Several functions must be called to create a window, put it on the screen, and start giving it input. We have already seen Tkclient's toplevel for window creation and onscreen to give a window space on the screen. Input arrives from several sources: from the mouse and keyboard, from the higher- level Tk widgets such as buttons, and from the window manager itself. In Limbo, each input source is represented by a channel, either given to the program by the window manager, or associated with one by namechan, as above.

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

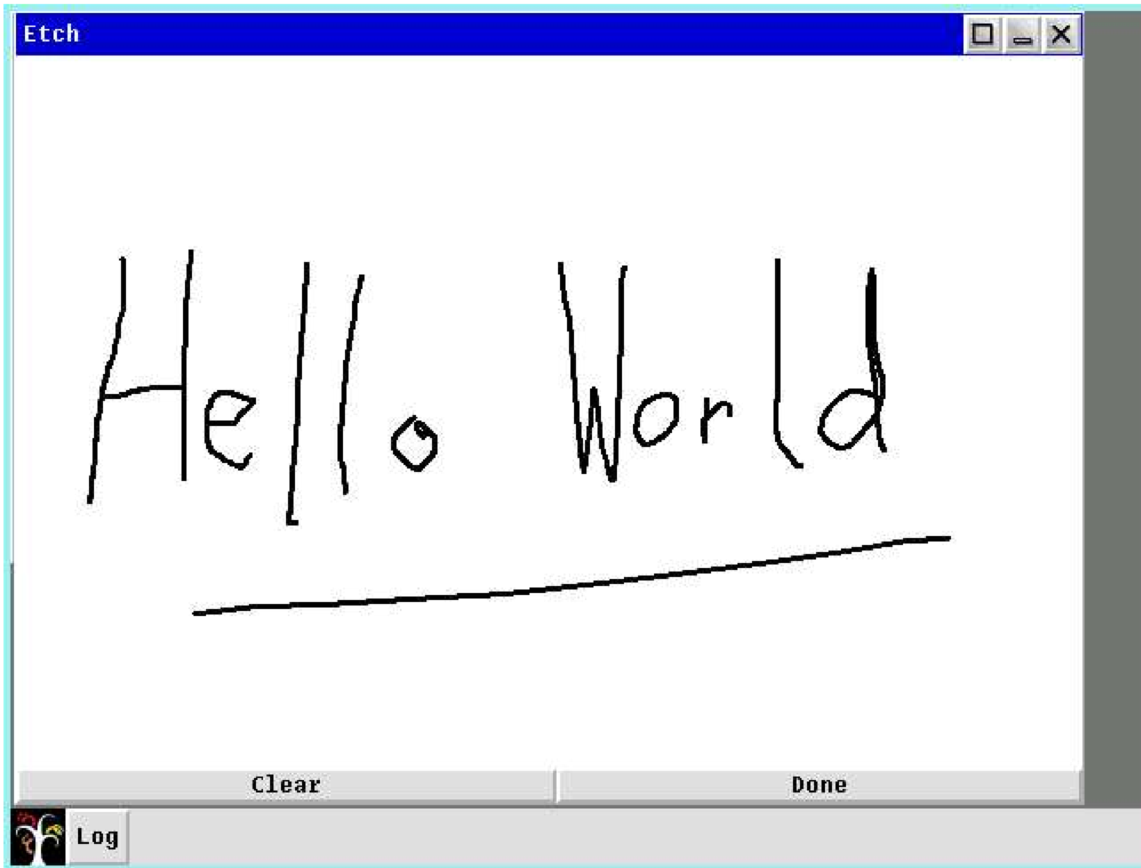

Figure 2. Etch-a-Sketch display.

```
implement Etch;

include "sys.m";
    sys: Sys;
include "draw.m";
include "tk.m";
    tk: Tk;
include "tkclient.m";
    tkclient: Tkclient;

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

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

    tkclient->init();

    (t, winctl) := tkclient->toplevel(ctxt, nil, "Etch", Tkclient->Appl);

    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");

    tkclient->startinput(t, "ptr" :: "kbd" :: nil);
    tkclient->onscreen(t, nil);

    lastx, lasty: int;
    for (;;) {
        alt {
            s := <-cmd =>
            (nil, 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;
            }

        p := <-t.ctxt.ptr =>
            tk->pointer(t, *p);

        c := <-t.ctxt.kbd =>
            tk->keyboard(t, c);

        ctl := <-winctl or
        ctl = <-t.ctxt.ctl or
        ctl = <-t.wreq =>
            tkclient->wmctl(t, ctl);
        }
        tk->cmd(t, "update");
    }
}
```

The function `toplevel` returns a tuple containing the `Tk->Toplevel` for the new window and a channel upon which the window manager will send messages for events such as hitting the exit button. An earlier example assigned the channel value to `nil`, discarding it; here it is assigned the name `winctl`. The parameters to `toplevel` includes a graphics context `ctxt` where the window will be created, a configuration string (simply `nil` here), the program name (which appears in the window's "title bar" if it has one), and a value `Tkclient->Appl` that denotes a style of window suitable for most applications. Note that `ctxt` was one of the arguments to `init`. (We do not use the argument list for `init`, and so declare it as `nil`).

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 buttons are created with a `-command` to send a suitable string on channel `cmd`, and two `bind` commands make the same channel the target for messages about mouse button presses and movement in the canvas. Note the `%x` and `%y` parameters in the latter case to include the mouse's coordinates in the string.

The window manager sends keyboard and mouse input to the currently selected window using two more channels `t.ctxt.kbd` and `t.ctxt.ptr`. A further channel `t.wreq` is used by the `Tk` module itself to request changes to the window displaying `Toplevel t`.

Now there are many channels watching events: one for the buttons and canvas created by the drawing program itself, one for the mouse, and three for window management. We use an `alt` statement to select from events on any of those channels. The expression

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

declares a variable `s` of the type carried by the channel `cmd`, i.e., a `string`; when a string is received on the channel, the assignment is executed, and the subsequent `case` decodes the message. The channel `t.ctxt.ptr` carries references to `Draw->Pointer` values, which give the state and position of the pointing device (mouse or stylus). They are handed as received to `tk->pointer` for processing by Tk. Similarly, Unicode characters from the keyboard are given to Tk using `tk->keyboard`. Internally, Tk hands those values on to the various widgets for processing, possibly resulting in messages being sent on one of the other channels. Finally, a value received from any of the `winctl`, `t.ctxt.ctl` or `t.wreq` channels is passed back to `Tkclient`'s `wmctl` function to be handled there.

As another example, 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, args: list of string)
{
    sys = load Sys Sys->PATH;
    tk = load Tk Tk->PATH;
    tkclient = load Tkclient Tkclient->PATH;

    sys->pctl(Sys->NEWPGRP, nil);

    tkclient->init();
    (t, winctl) := tkclient->toplevel(ctxt, nil, "Othello", Tkclient->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 .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");
    tkclient->startinput(t, "ptr" :: "kbd" :: nil);
    tkclient->onscreen(t, nil);

    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);
            }

        p := <-t.ctxt.ptr =>
            tk->pointer(t, *p);

        c := <-t.ctxt.kbd =>
            tk->keyboard(t, c);

        ctl := <-winctl or
        ctl = <-t.ctxt.ctl or
        ctl = <-t.wreq =>
            tkclient->wmctl(t, ctl);
        }
    }
}
```

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);
}
```

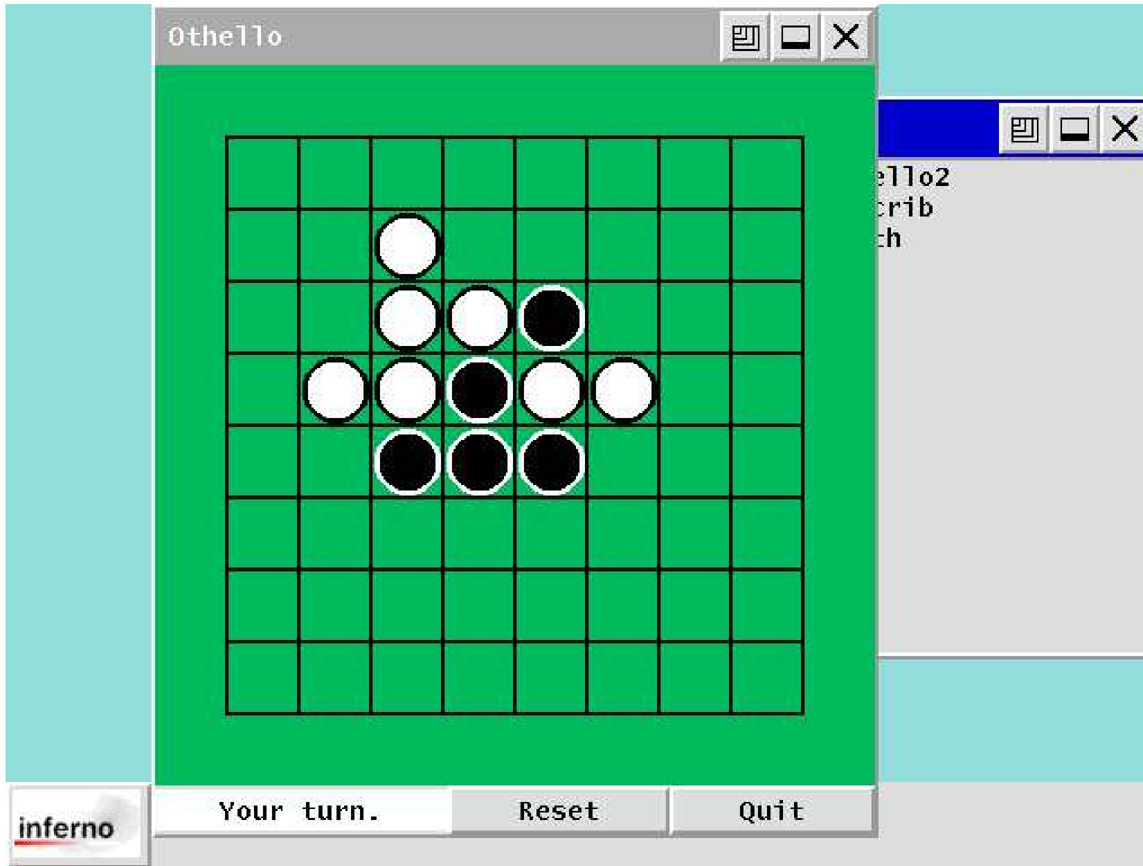

Figure 3. Screen shot of Inferno display showing Othello window.

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.

2.11. 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.