The Limbo Programming Language

Dennis M. Ritchie

Limbo is a programming language intended for applications running distributed systems on small computers. It supports modular programming, strong type checking at compile- and run-time, interprocess communication over typed channels, automatic garbage collection, and simple abstract data types. It is designed for safe execution even on small machines without hardware memory protection.

In its initial implementation for the Inferno operating system, object programs generated by the Limbo compiler run using an interpreter for a fixed virtual machine. Inferno and its accompanying virtual machine run either stand-alone on bare hardware or as an application under conventional operating systems like Unix, Windows 95, Windows NT, and Plan 9. For several architectures, including Intel x86 and MIPS, Limbo object programs are transformed on-the-fly into instructions for the underlying hardware.

1. Overview and introduction

A Limbo application consists of one or more *modules*, each of which supplies an interface declaration and an implementation part. A module that uses another module includes its declaration part. During execution, a module dynamically attaches another module by stating the other module's type identifier and a place from which to load the object code for its implementation.

A module declaration specifies the functions and data it will make visible, its data types, and constants. Its implementation part defines the functions and data visible at its interface and any functions associated with its data types; it may also contain definitions for functions used only internally and for data local to the module.

Here is a simple module to illustrate the flavour of the language.

```
implement Command;

include "sys.m";
include "draw.m";

sys:    Sys;

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

```
8     # The canonical "Hello world" program, enhanced
9     init(ctxt: ref Draw->Context, argv: list of string)
10    {
11            sys = load Sys Sys->PATH;
12            sys->print("hello world\n");
13            for (; argv!=nil; argv = tl argv)
14                    sys->print("%s ", hd argv);
15            sys->print("\n");
16    }
```

A quick glance at the program reveals that the syntax of Limbo is influenced by C in its expressions, statements, and some conventions (for example, look at lines 13-14), and also by Pascal and its successors (the declarations on lines 4, 6, 9). When executed in the Inferno environment, the program writes hello world somewhere, then echoes its arguments.

Lets look at the program line-by-line. It begins (line 1) by saying that this is the implementation of module Command . Line 2 includes a file (found in a way analogous to C s #include mechanism) named sys.m . This file defines the interface to module Sys ; it says, in part,

```
Sys: module {
        PATH: con "$Sys";
        . . .
        print: fn (s: string, *): int;
        . . .
};
```

This declares Sys to be the type name for a module containing among other things a function named print ; the first argument of print is a string. The * in the argument list specifies that further arguments, of unspecified type, may be given.

Line 3 includes draw.m ; only one piece of information, mentioned below, is used from it. Line 4 declares the variable sys to be of type Sys ; its name will be visible throughout the remainder of the file describing this module. It will be used later to refer to an instance of the Sys module. This declaration initializes it to nil ; it still needs to be set to a useful value.

Lines 4-7 constitute the declaration of Command , the module being implemented. It contains only a function named init , with two arguments, a ref Draw->Context and a list of strings, and it doesn t return any value. The ref Draw->Context argument would be used if the program did any graphics; it is a data type defined in draw.m and refers to the display. Since the program just writes text, it won t be used. The init function isn t special to the Limbo language, but it is conventional in the environment, like main in C.

In a module designed to be useful to other modules in an application, it would be wise to take the module declaration for Command out, put it in a separate file called command.m and use include command.m to allow this module and others to refer to it. It is called, for example, by the program loader in the Inferno system to start the execution of applications.

Line 8 is a comment; everything from the # to the end of line is ignored.

Line 9 begins the definition for the init function that was promised in the module s declaration (line 6). The argument that is a list of strings is named argv .

Line 11 connects the program being written to the Sys module. The first token after load is the target module s name as defined by its interface (here found in the include on line 2) The next token is the place where the code for the module can be found; it is a string that usually names a file. Conventionally, in the Inferno system, each module contains a constant declaration for the name PATH as a string that names the file where the object module can be found. Loading the file is performed dynamically during execution except for a few modules built into the execution environment. (These include Sys; this accounts for the peculiar file name $Sys as the value of PATH.)

The value of load is a reference to the named module; line 11 assigns it to the variable sys for later use. The load operator dynamically loads the code for the named module if it is not already present and instantiates a new instance of it.

Line 12 starts the work by printing a familiar message, using the facilities provided by module Sys through its handle sys; the notation sys->print(...) means to call the print function of the module referred to by sys. The interface of Sys resembles a binding to some of the mechanisms of Unix and the ISO/ANSI C library.

The loop at lines 13-14 takes the list of string argument to init and iterates over it using the hd (head) and tl (tail) operators. When executed, this module combines the traditional Hello world and echo.

2. Lexical conventions

There are several kinds of tokens: keywords, identifiers, constants, strings, expression operators, and other separators. White space (blanks, tabs, new-lines) is ignored except that it serves to separate tokens; sometimes it is required to separate tokens. If the input has been parsed into tokens up to a particular character, the next token is taken to include the longest string of characters that could constitute a token.

The native character set of Limbo is Unicode, which is identical with the first 16-bit plane of the ISO 10646 standard. Any Unicode character may be used in comments, or in strings and character constants. The implementation assumes that source files use the UTF-8 representation, in which 16-bit Unicode characters are represented as sequences of one, two, or three bytes.

2.1. Comments

Comments begin with the # character and extend to the end of the line. Comments are ignored.

2.2. Identifiers

An identifier is a sequence of letters and digits of which the first is a letter. Letters are the Unicode characters a through z and A through Z, together with the underscore character, and all Unicode characters with encoded values greater than 160 (A0 hexadecimal, the beginning of the range corresponding to Latin-1).

Only the first 256 characters in an identifier are significant.

2.3. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

adt	alt	array	big
break	byte	case	chan
con	continue	cyclic	do
else	exit	fn	for
hd	if	implement	import
include	int	len	list
load	module	nil	of
or	pick	real	ref
return	self	spawn	string
tagof	tl	to	type
while			

The word union is not currently used by the language.

2.4. Constants

There are several kinds of constants for denoting values of the basic types.

2.4.1. Integer constants

Integer constants have type int or big . They can be represented in several ways.

Decimal integer constants consist of a sequence of decimal digits. A constant with an explicit radix consists of a decimal radix followed by R or r followed by the digits of the number. The radix is between 2 and 36 inclusive; digits above 10 in the number are expressed using letters A to Z or a to z . For example, 16r20 has value 32.

The type of a decimal or explicit-radix number is big if its value exceeds $2^{31}-1$, otherwise it is int .

Character constants consist of a single Unicode character enclosed within single-quote characters ' . Inside the quotes the following escape sequences represent special characters:

\'	single quote
\"	double quote
\\	backslash
\t	tab
\n	newline
\r	carriage return
\b	backspace
\a	alert character (bell)
\v	vertical tab
\u *dddd*	Unicode character named by 4 hexadecimal digits
\0	NUL

Character constants have type int .

2.4.2. Real constants

Real constants consist of a sequence of decimal digits containing one period. and optionally followed by e or E and then by a possibly signed integer. If there is an explicit exponent, the period is not required. Real constants have type real .

2.4.3. Strings

String constants are sequences of Unicode characters contained in double quotes. They cannot extend across source lines. The same escape sequences listed above for character constants are usable within string constants. Strings have type string .

2.4.4. The nil constant

The constant `nil` denotes a reference to nothing. It may be used where an object of a reference type is expected; otherwise uninitialized values of reference type start off with this value, it can be assigned to reference objects, and reference types can be tested for equality with it. (The keyword has other uses as well.)

2.5. Operators and other separators

The operators are

```
+       -       *       /       %       &       |       ^
==      <       >       <=      >=      !=      <<      >>
&&      ||      <-      ::
=       +=      -=      *=      /=      %=      &=      |=      ^=      <<=     >>=
:=
~       ++      --      !
```

The other separators are

```
:       ;       (       )       {       }       [       ]
,       .       ->      =>
```

3. Syntax notation

In this manual, Limbo syntax is described by a modified BNF in which syntactic categories are named in an *italic* font, and literals in `typewriter` font. Alternative productions are listed on separate lines, and an optional symbol is indicated with the subscript opt.

4. Types and objects

Limbo has three kinds of objects. *Data* objects exist in the storage associated with a module; they can be manipulated by arithmetic operations, assignment, selection of component entities, and other concrete operations. Each data object has a type that determines what can be stored in it and what operations are applicable.

The second kind of object is the *function*. Functions are characterized by the types of the arguments they accept and the values they return, and are associated with the modules in which they are defined. Their names can be made visible in their module's declaration, or they can be encapsulated within the `adt` (abstract data types) of their modules, or they can exist privately within their module.

Finally, Limbo programs are organized into *modules*: a named collection of constants, abstract data types, data, and functions made available by that module. A module declaration displays the members visible to other modules; the module's implementation defines both the publicly visible members and its private parts, including the data objects it uses. A module that wishes to use the facilities of another includes its declaration in order to understand what it exports, but before using them it explicitly loads the new module.

4.1. Types

Limbo has several basic types, some built-in higher abstractions, and other ways of composing new types. In declarations and some other places, constructions naming a type are used. The syntax is:

type:
    data-type
    function-type

Functions will be discussed in §7 below. First, data types will be explored.

4.2. Data types

The syntax of data types is

data-type:
  byte
  int
  big
  real
  string
  tuple-type
  array of *data-type*
  list of *data-type*
  chan of *data-type*
  adt-type
  ref *adt-type*
  module-type
  module-qualified-type
  type-name

data-type-list:
  data-type
  data-type-list , *data-type*

Objects of most data types have *value* semantics; when they are assigned or passed to functions, the destination receives a copy of the object. Subsequent changes to the assigned object itself have no effect on the original object. The value types are byte, int, big, real, string, the tuple types, and abstract data types or adt. The rest have *reference* semantics. When they are assigned, the quantity actually assigned is a reference to (a pointer to) an underlying object that is not copied; thus changes or operations on the assigned value affect the original object. Reference types include lists, arrays, channels, modules, and ref adt types.

4.2.1. Basic types

The five basic data types are denoted by byte, int, big, real, and string.

Bytes are unsigned 8-bit quantities.

Integers (int) are 32-bit signed quantities represented in two's complement notation. Large integers (big) are 64-bit signed quantities represented in two's complement notation.

Real numbers (real) are 64-bit quantities represented in the IEEE long floating notation.

The byte, int, big, and real types are collectively called arithmetic types.

Strings are rows of Unicode characters. They may be concatenated and extended character-by-character. When a string is indexed with a single subscript, it yields an integer with the Unicode encoding of the character; when it is indexed by a range, it yields another string.

4.2.2. Tuple type

The *tuple* type, denoted

tuple–type:
  (*data–type–list*)

is a type consisting of an ordered collection of two or more objects, each having its own data type. For each tuple type, the types of the members are fixed, but need not be identical; for example, a function might return a tuple containing an integer and a string. Each tuple type is characterized solely by the the order and identity of the types it contains. Objects of tuple type may be assigned to a list of identifiers (to pick out the components), and a parenthesized, comma-separated list of expressions denotes a tuple.

4.2.3. Array types

The *array* type describes a dynamically-sized row of objects, all of the same type; it is indexed starting from 0. An array type is denoted by

```
array of data–type
```

The size of an array is not part of its type; instead it is part of the value. The *data–type* may itself be an array, to achieve a multidimensional array.

4.2.4. List types

A *list* is a sequence of like-typed objects; its denotation is

```
list of data–type
```

A list is a stack-like object, optimized for a few operations: get the head (the first object), get the tail (the rest of the list), place an object at the beginning.

4.2.5. Channel types

A *channel*, whose type is written

```
chan of data–type
```

is a communication mechanism capable of sending and receiving objects of the specified type to another agent in the system. Channels may be used to communicate between local processes; using library procedures, they may be connected to named destinations. In either case *send* and *receive* operations may be directed to them. For example,

```
chan of (int, string)
```

is the type of a channel that transmits tuples consisting of an integer and an string. Once an instance of such a channel (say c) has been declared and initialized, the statement

```
c <-= (123, "Hello");
```

sends such a tuple across it.

4.2.6. Abstract data types

An abstract data type or *adt* is an object that can contain data objects of several different types and declare functions that operate on them. The syntax for declaring an adt is given later. Once an adt has been declared, the identifier associated with it becomes a data-type name.

adt–type:
  identifier
  module–qualified–type

There is also a ref adt type representing a reference (pointer) to an adt. It is denoted `ref` *adt-type*

where the identifier is the name of an adt type.

4.2.7. Module types

A module typename is an identifier:

module-type:
identifier

The identifier is declared as a module identifier by a *module-declaration*, as described in §6.5 below. An object of module type serves as a handle for the module, and is used to access its functions.

4.2.8. Module-qualified type

When an adt is declared within a module declaration, the typename of that adt is not generally visible to the rest of the program unless a specific `import` request is given (see §6.6, §10 below). Without such a request, when adt objects implemented by a module are declared by a client of that module, the adt typename is qualified:

module-qualified-type:
identifier `->` *identifier*

Here the first identifier is either the name of a module or a variable of the module type; the second is the name of a type mentioned in the module declaration.

4.2.9. Named types

Finally, data types may be named, using a `type` declaration; this is discussed in §6.4 below.

type-name:
identifier

4.3. Function types

A function type characterizes the arguments and return value of a function. The syntax is

function−type:
  `fn` *function−arg−ret*

function−arg−ret:
  `(` *formal−arg−list* $_{opt}$ `)`
  `(` *formal−arg−list* $_{opt}$ `)` `:` *data−type*

formal−arg−list:
  formal−arg
  formal−arg−list `,` *formal−arg*

formal−arg:
  nil−or−D−list `:` *type*
  nil−or−D `:` `self` `ref` $_{opt}$ *identifier*
  nil−or−D `:` `self` *identifier*
  `*`

nil−or−D−list:
  nil−or−D
  nil−or−D−list `,` *nil−or−D*

nil−or−D:
  identifier
  `nil`

That is, the denotation of a function type has the keyword `fn` followed by a comma-separated list of its arguments enclosed in parentheses, and perhaps followed by the type the function returns. Absence of a return value means that the function returns no value: it is a procedure. The names and types of arguments are specified. However, the name of an argument may be replaced by `nil`; in this case it is nameless. For example,

```
fn (nil: int, nil: int): int
fn (radius: int, angle: int): int
fn (radius, angle: int): int
```

all denote exactly the same type, namely a function of two integers that returns an integer. As another example,

```
fn (nil: string)
```

is the type of a function that takes a string argument and returns no value.

The `self` keyword has a specialized use within `adt` declarations. It may be used only for the first argument of a function declared within an `adt`; its meaning is discussed in §6.3 below.

The star character `*` may be given as the last argument in a function type. It declares that the function is variadic; during a call, actual arguments at its position and following are passed in a manner unspecified by the language. For example, the type of the `print` function of the `Sys` module is

```
fn (s: string, *): int
```

This means that the first argument of `print` is a string and that other arguments may be given when the function is called. The Limbo language itself has no way of accessing these arguments; the notation is an artifice for describing facilities built into the runtime system, such as the `Sys` module.

5. Limbo programs

Limbo source programs that implement modules are stored in files, conventionally named with the suffix `.b`. Each such file begins with a single `implement` directive naming the type of the module being implemented, followed by a sequence of declarations. Other files, conventionally named with the suffix `.m`, contain declarations for things obtainable from other modules. These files are incorporated by an `include` declaration in the implementation modules that need them. At the top level, a program consists of a sequence of declarations. The syntax is

program:
  `implement` *identifier* `;` *top-declaration-sequence*

top-declaration-sequence:
  top-declaration
  top-declaration-sequence *top-declaration*

top-declaration:
  declaration
  identifier-list `:=` *expression* `;`
  identifier-list `=` *expression* `;`
  `(` *identifier-list* `)` `:=` *expression* `;`
  module-declaration
  function-definition
  adt-declaration

The `implement` declaration at the start identifies the type of the module that is being implemented. The rest of the program consists of a sequence of various kinds of declarations and definitions that announce the names of data objects, types, and functions, and also create and initialize them. It must include a module declaration for the module being implemented and the objects it announces, and may also include declarations for the functions, data objects, types, and constants used privately within the module as well as declarations for modules used by it.

Declarations are used both at the top level (outside of functions) and also inside functions and module declarations. Some styles of declaration are allowed only in certain of these places, but all will be discussed together.

6. Declarations

Declarations take several forms:

declaration:
  identifier-list `:` *type* `;`
  identifier-list `:` *type* `=` *expression* `;`
  identifier-list `:` `con` *expression* `;`
  identifier-list `:` `import` *identifier* `;`
  identifier-list `:` `type` *type* `;`
  `include` *string-constant* `;`

identifier-list:
  identifier
  identifier-list `,` *identifier*

expression-list:
  expression
  expression-list `,` *expression*

6.1. Data declarations

These forms constitute the basic way to declare and initialize data:

identifier-list : *type* ;
identifier-list : *type* = *expression* ;

A comma-separated sequence of identifiers is followed by a colon and then the name of a type. Each identifier is declared as having that type and denotes a particular object for rest of its scope (see §11 below). If the declaration contains = and an expression, the type must be a data type, and all the objects are initialized from the value of the expression. In a declaration at the top level (outside of a function), the expression must be constant (see §8.5) or an array initialized with constant expressions; the bound of any array must be a constant expression. Lists and ref adt types may not be initialized at the top level. If an object is not explicitly initialized, then it is always set to nil if it has a reference type; if it has arithmetic type, then it is set to 0 at the top level and is undefined if it occurs within a function.

For example,

```
i, j: int = 1;
r, s: real = 1.0;
```

declares i and j as integers, r and s as real. It sets i and j to 1, and r and s to 1.0.

Another kind of declaration is a shorthand. In either of

identifier := *expression* ;
(*identifier-list*) := *expression* ;

identifiers on the left are declared using the type of the expression, and are initialized with the value of the expression. In the second case, the expression must be a tuple or an adt, and the types and values attributed to the identifiers in the list are taken from the members of the tuple, or the data members of the adt respectively. For example,

```
x: int = 1;
```

and

```
x := 1;
```

are the same. Similarly,

```
(p, q) := (1, 2.1);
```

declares the identifiers on the left as int and real and initializes them to 1 and 2.1 respectively. Declarations with := can also be expressions, and are discussed again in §8.4.4 below.

6.2. Constant declarations

The con declaration

identifier-list : con *expression* ;

declares a name (or names) for constants. The *expression* must be constant (see §8.5). After the declaration, each identifier in the list may be used anywhere a constant of the appropriate type is needed; the type is taken from the type of the constant. For example, after

```
Seven: con 3+4;
```

the name Seven is exactly the same as the constant 7.

The identifier `iota` has a special meaning in the expression in a `con` declaration. It is equivalent to the integer constant 0 when evaluating the expression for the first (leftmost) identifier declared, 1 for the second, and so on numerically. For example, the declaration

```
M0, M1, M2, M3, M4: con (1<<iota);
```

declares several constants `M0` through `M4` with the values 1, 2, 4, 8, 16 respectively.

The identifier `iota` is not reserved except inside the expression of the `con` declaration.

6.3. adt declarations

An `adt` or abstract data type contains data objects and functions that operate on them. The syntax is

adt-declaration:
  identifier : `adt` { *adt-member-list* $_{opt}$ } ;

adt-member-list:
  adt-member
  adt-member-list adt-member

adt-member:
  identifier-list : `cyclic` $_{opt}$ *data-type* ;
  identifier-list : `con` *expression* ;
  identifier-list : *function-type* ;
  `pick` { *pick-member-list* }

After an *adt-declaration*, the identifier becomes the name of the type of that `adt`. For example, after

```
Point: adt {
        x, y: int;
        add: fn (p: Point, q: Point): Point;
        eq: fn (p: Point, q: Point): int;
};
```

the name `Point` is a type name for an `adt` of two integers and two functions; the fragment

```
r, s: Point;
xcoord: int;
...
xcoord = s.x;
r = r.add(r, s);
```

makes sense. The first assignment selects one of the data members of `s`; the second calls one of the function members of `r`.

As this example indicates, `adt` members are accessed by mentioning an object with the `adt` type, a dot, and then the name of the member; the details will be discussed in §8.13 below. A special syntactic indulgence is available for functions declared within an `adt`: frequently such a function receives as an argument the same object used to access it (that is, the object before the dot). In the example just above, `r` was both the object being operated on and the first argument to the `add` function. If the first formal argument of a function declared in an `adt` is marked with the `self` keyword, then in any calls to the function, the `adt` object is implicitly passed to the function, and is not mentioned explicitly in the actual argument list at the call site. For example, in

```
Rect: adt {
        min, max: Point;
        contains: fn(r: self Rect, p: Point): int;
};

r1: Rect;
p1: Point;
...
if (r1.contains(p1)) ...
```

because the first argument of the `contains` function is declared with `self`, the subsequent call to it automatically passes `r1` as its first argument. The `contains` function itself is defined elsewhere with this first argument explicit. (This mechanism is analogous to the *this* construct in C++ and other languages, but puts the special-casing at the declaration site and makes it explicit.)

If `self` is specified in the declaration of a function, it must also be specified in the definition as well. For example, `contains` would be defined

```
Rect.contains(r: self Rect, p: Point)
{
        . . .
}
```

The `adt` type in Limbo does not provide control over the visibility of its individual members; if any are accessible, all are.

Constant `adt` members follow the same rules as ordinary constants (§6.2).

The `cyclic` modifier will be discussed in §11.1.

6.3.1. pick adts

An `adt` which contains a `pick` member is known as a *pick adt*. A `pick adt` is Limbo's version of a *discriminated union*. An `adt` can only contain one `pick` member and it must be the last component of the `adt`. Each *identifier* enumerated in the *pick-tag-list* names a variant type of the `pick adt`. The syntax is

pick-member-list:
    pick-tag-list `=>`
    pick-member-list pick-tag-list `=>`
    pick-member-list identifier-list `:` `cyclic` $_{opt}$ *data-type* `;`

pick-tag-list:
    identifier
    pick-tag-list `or` *identifier*

The *pick-member-list* contains a set of data members for each *pick-tag-list*. These data members are specific to those variants of the `pick adt` enumerated in the *pick-tag-list*. The `adt` data members found outside of the `pick` are common to all variants of the `adt`. A `pick adt` can only be used as a `ref adt` and can only be initialized from a value of one of its variants. For example, if `Constant` is a `pick adt` and `Constant.Real` is one of its variant types then

```
c : ref Constant = ref Constant.Real("pi", 3.1);
```

will declare `c` to have type `ref Constant` and initialize it with a value of the variant type `ref Constant.Real`.

6.4. Type declarations

The type declaration

> *identifier-list* : type *data-type* ;

introduces the identifiers as synonyms for the given type. Type declarations are transparent; that is, an object declared with the newly-named type has the same type as the one it abbreviates.

6.5. Module declarations

A module declaration collects and packages declarations of adt, functions, constants and simple types, and creates an interface with a name that serves to identify the type of the module. The syntax is

module-declaration:
  identifier : module { *mod-member-list*$_{opt}$ } ;

mod-member-list:
  mod-member
  mod-member-list mod-member

mod-member:
  identifier-list : *function-type* ;
  identifier-list : *data-type* ;
  adt-declaration ;
  identifier-list : con *expression* ;
  identifier-list : type *type* ;

After a module declaration, the named *identifier* becomes the name of the type of that module. For example, the declaration

```
Linear: module {
        setflags: fn (flag: int);
        TRUNCATE: con 1;
        Vector: adt {
                v: array of real;
                add: fn (v1: self Vector, v2: Vector): Vector;
                cross: fn (v1: self Vector, v2: Vector): Vector;
                dot: fn (v1: self Vector, v2: Vector);
                make: fn (a: array of real): Vector;
        };
        Matrix: adt {
                m: array of array of real;
                add: fn (m1: self Matrix, m2: Matrix): Matrix;
                mul: fn (m1: self Matrix, m2: Matrix): Matrix;
                make: fn (a: array of array of real): Matrix;
        };
};
```

is a module declaration for a linear algebra package that implements two adt, namely Vector and Matrix, a constant, and a function setflags. The name Linear is the type name for the module, and it may be used to declare an object referring to an instance of the module:

```
        linearmodule: Linear;
```

Before the module can be used, it must be loaded, for example in the style:

```
linearmodule = load Linear "/usr/dmr/limbo/linear.dis";
if (linearmodule == nil) {
        sys->print("Can't load Linear\n");
        exit;
}
```

The `load` operator is discussed more fully in §8.4.5 below.

To initialize data declared as part of a module declaration, an assignment expression may be used at the top level. For example:

```
implement testmod;
testmod: module {
        num: int;
};
. . .
num = 5;
```

The right side of the assignment must be a constant expression (§8.5).

6.6. Declarations with `import`

These declarations take the form

identifier-list `:` `import` *identifier* `;`

Identifiers for entities declared within a module declaration are normally meaningful only in a context that identifies the module. The `import` declaration lifts the names of specified members of a module directly into the current scope. The use of `import` will be discussed more fully in §8.1.4 below, after the syntax for expressions involving modules has been presented.

6.7. Declarations with `include`

The string following the `include` keyword names a file, which is inserted into the program's text at that point. The included text is treated like text literally present. Conventionally, included files declare module interfaces and are named with the suffix `.m`. The directories to be searched for included files may be specified to the Limbo compiler command. Include files may be nested.

7. Function definitions

All executable code is supplied as part of a function definition. The syntax is

function-definition:
  function-name-part *function-arg-ret* `{` *statements* `}`

function-name-part:
  identifier
  function-name-part `.` *identifier*

The syntax of the statements in a function will be discussed in §9 below. As a brief example,

```
add_one(a: int): int
{
        return a+1;
}
```

is a simple function that might be part of the top level of a module.

Functions that are declared within an adt use the qualified form of definition:

```
Point: adt {
        x, y: int;
        add: fn (p: Point, q: Point): Point;
        eq: fn (p: Point, q: Point): int;
}
. . .
Point.add(p: Point, q: Point): Point
{
        return Point(p.x+q.x, p.y+q.y);
}
```

Because an adt may contain an adt, more than one qualification is possible.

8. Expressions

Expressions in Limbo resemble those of C, although some of the operators are different. The most salient difference between Limbo's expression semantics and those of C is that Limbo has no automatic coercions between types; in Limbo every type conversion is explicit.

8.1. Terms

The basic elements of expressions are terms:

term:
- *identifier*
- *constant*
- *real-constant*
- *string-constant*
- nil
- (*expression-list*)
- *term* . *identifier*
- *term* -> *term*
- *term* (*expression-list* $_{opt}$)
- *term* [*expression*]
- *term* [*expression* : *expression*]
- *term* [*expression* :]
- *term* ++
- *term* --

The operators on terms all associate to the left, and their order of precedence, with tightest listed first, is as follows:

```
.
->
() [] ++ --
```

8.1.1. Simple terms

The first five kinds of term are constants and identifiers. Constants have a type indicated by their syntax. An identifier used in an expression is often a previously declared data object with a particular data type; when used as a term in an expression it denotes the value stored in the object, and the term has the declared object's type. Sometimes, as discussed below, identifiers used in expressions are type names, function names, or module identifiers.

8.1.2. Parenthesized terms

A comma-separated list of expressions enclosed in parentheses is a term. If a single expression is present in the list, the type and value are those of the expression; the parentheses affect only the binding of operators in the expression of which the term is a part. If there is more than one expression in the list, the value is a tuple. The member types and values are taken from those of the expressions.

8.1.3. Selection

A term of the form

term . *identifier*

denotes selection of a member of an `adt`. The term must be a type name or yield an object; its type must be `adt` or `ref adt`; the identifier must be a member of the `adt`. The result denotes the named member (either a data object or a function).

8.1.4. Module qualification

A term of the form

term -> *term*

denotes module qualification. The first term identifies a module: either it is a module type name, or it is an expression of module type. The second term is a constant name, type, or function specified within that module's declaration. Either the module type name or an object of the module's type suffices to qualify constants and types; functions directly exported by the module or contained within its `adt` must be qualified by an object of the module's type, initialized with `load`.

An example using an abridged version of an example above: given

```
Linear: module {
        setflags: fn(flag: int);
        TRUNCATE: con 1;
        Vector: adt {
                make: fn(v: array of real): Vector;
                v: array of real;
        };
};
```

one might say

```
lin := load Linear "/dis/linear.dis";
a: array of real;

v1: lin->Vector;
v2: Linear->Vector;
lin->setflags(Linear->TRUNCATE);
v1 = lin->(Linear->Vector).make(a);
v1 = lin->v1.make(a);
v1 = lin->v1.add(v1);
v1.v = nil;
```

Here, the declarations for `v1` and `v2` are equivalent; either a module type name (here, `Linear`) or a handle (here, `lin`) suffices to identify the module. In the call to setflags, a handle is required for the call itself; the type name is sufficient for the constant.

When calling a function associated with an adt of another module, it is necessary to identify both the module and the adt as well as the function. The two calls to the make function illustrate two ways of doing this. In the first,

```
v1 = lin->(Linear->Vector).make(a);
```

the module handle lin is specified first, then the type name of the Vector adt within it, and then the function. In the second call

```
v1 = lin->v1.make(a);
```

instead of using a type name to specify the adt, an instance of an object of the appropriate type is used instead. In the first example, the parentheses are required because the qualification operators associate to the left.

```
v1 = lin->Vector.make(a);         # Wrong
v1 = lin->Linear->Vector.make(a);          # Wrong
```

The first is wrong because the same lin can't serve as a qualifier for both the type and the call; the second is wrong because lin->Linear is meaningless.

Using import makes the code less verbose:

```
lin := load Linear "/usr/dmr/limbo/linear.dis";
Vector, TRUNCATE, setflags: import lin;
a: array of real;

v1: Vector;
v2: Vector;
setflags(TRUNCATE);
v1 = Vector.make(a);
v1 = v1.make(a);
v1 = v1.add(v1);
v1.v = nil;
```

8.1.5. Function calls

The interpretation of an expression in the form

term (*expression-list* $_{opt}$)

depends on the declaration of the term. If it is the (perhaps qualified) name of an adt, then the expression is a cast; this is discussed in §8.2.11 below. If the term is the (perhaps qualified) name of a function, the expression means a function call; this is discussed here.

A plain identifier as the *term* names a function defined in the current module or imported into it. A term qualified by using the selection operator . specifies a function member of an adt; a term using -> specifies a function defined in another module.

Function calls in Limbo create a copy of each argument of value type, and the execution of a function cannot affect the value of the corresponding actual argument. For arguments of reference type, execution of the function may affect the value of the object to which the reference refers, although it cannot change the argument itself. The actual arguments to a function are evaluated in an unspecified order, although any side effects caused by argument evaluation occur before the function is called.

Function calls may be directly or indirectly recursive; objects declared within each function are distinct from those in their dynamic predecessors.

Functions (§4.3, §7) may either return a value of a specified type, or return no value. If a function returns a value, it has the specified type. A call to a function that returns no value may appear only as the sole expression in a statement (§9.1).

8.1.6. Subscripting and slicing

In a term of the form

> *term* [*expression*]

the first term must be an array or a string, and the bracketed expression must have `int` type. The whole term designates a member of the array or string, indexed by the bracketed expression; the index origin is 0. For an array, the type of the whole term is the type from which the array is constructed; for a string, the type is an `int` whose value is the Unicode character at that position in the string.

It is erroneous to refer to a nonexisting part of an array or string. (A single exception to this rule, discussed in §8.4.1 below, allows extending a string by assigning a character at its end.)

In a term of the form

> *term* [*expression* : *expression*]

the first term must be an array or a string, and the whole term denotes a slice of it. The first expression is the lower bound, and the second is the upper. If `e1` is the first expression and `e2` is the second, then in `a[e1:e2]` it must be the case that `0<=e1`, `e1<=e2`, `e2<=len a`, where `len` gives the number of elements in the array or string. When the term is an array, the value is an array of the same type beginning at the indicated lower bound and extending to the element just before the upper bound. When the term is a string, the value is similarly the substring whose first character is indexed by the lower bound and whose last character lies just before the upper bound.

Thus, for both arrays and strings, the number of elements in `a[e1:e2]` is equal to `e2-e1`.

A slice of the form `a[e:]` means `a[e:len a]`.

When a string slice is assigned to another string or passed as an argument, a copy of its value is made.

A slice of an array produces a reference to the designated subarray; a change to an element of either the original array or the slice is reflected in the other.

In general, slice expressions cannot be the subject of assignments. However, as a special case, an array slice expression of the form `a[e1:]` may be assigned to. This is discussed in §8.4.1.

The following example shows how slices can be used to accomplish what would need to be done with pointer arithmetic in C:

```
fd := sys->open( ... );
want := 1024;
buf := array[want] of byte;
b := buf[0:];
while (want>0) {
        got := sys->read(fd, b, want);
        if (got<=0)
                break;
        b = b[got:];
        want -= got;
}
```

Here the array buf is filled by successive calls to sys->read that may supply fewer bytes than requested; each call stores up to want bytes starting at b[0], and returns the number of bytes stored. The invariant is that the slice b always refers to the part of the array still to be stored into.

8.1.7. Postfix increment and decrement

A term of the form

term ++

is called a *post-increment*. The term must be an lvalue (see §8.4 below) and must have an arithmetic type. The type and value of the whole term is that of the incremented term. After the value is taken, 1 of the appropriate type is added to the lvalue. The result is undefined if the same object is changed more than once in the same expression.

The term

term --

behaves analogously to the increment case except that 1 is subtracted from the lvalue.

8.2. Monadic expressions

Monadic expressions are expressions with monadic operators, together with a few more specialized notations:

monadic-expression:
    term
    monadic-operator monadic-expression
    array [*expression*] of *data-type*
    array [*expression* $_{opt}$] of { *init-list* }
    list of { *expression-list* }
    chan of *data-type*
    data-type monadic-expression

monadic-operator: *one of*
    + - ! ~ ref * ++ -- <- hd tl len

8.2.1. Monadic additive operators

The - operator produces the negative of its operand, which must have an arithmetic type. The type of the result is the same as the type of its operand.

The + operator has no effect; it is supplied only for symmetry. However, its argument must have an arithmetic type and the type of the result is the same.

8.2.2. Logical negation

The `!` operator yields the `int` value 1 if its operand has the value 0, and yields 0 otherwise. The operand must have type `int`.

8.2.3. One's complement

The `~` operator yields the 1's complement of its operand, which must have type `int` or `byte`. The type of the result is the same as that of its operand.

8.2.4. Reference and indirection operators

If *e* is an expression of an `adt` type, then `ref` *e* is an expression of `ref adt` type whose value refers to (points to) an anonymous object with value *e*. The `ref` operator differs from the unary `&` operator of C; it makes a new object and returns a reference to it, rather than generating a reference to an existing object.

If *e* is an expression of type `ref adt`, then `*` *e* is the value of the `adt` itself. The value of *e* must not be `nil`.

For example, in

```
Point: adt { ... };
p: Point;
pp: ref Point;
p = Point(1, 2);
pp = ref p;          # pp is a new Point; *pp has value (1, 2)
p = Point(3, 4);     # This makes *pp differ from p
*pp = Point(4, 5); # This does not affect p
```

the expression `*pp` at first refers to a copy of the value stored in `p`, so `*pp == p` is true; however, when `p` is changed later, `*pp` does not change.

8.2.5. Prefix increment and decrement

A monadic expression of the form

`++` *monadic-expression*

is called a *pre-increment*. The monadic expression must be an lvalue (see §8.4 below) and must have an arithmetic type. Before the value is taken, 1 of the appropriate type is added to the lvalue. The type and value of the whole expression is that of the now incremented term. The result is undefined if the same object is changed more than once in the same expression.

The term

`--` *monadic-expression*

behaves analogously to the increment case except that 1 is subtracted from the lvalue.

8.2.6. Head and tail

The operand of the `hd` operator must be a non-empty list. The value is the first member of the list and has that member's type.

The operand of the `tl` operator must be a non-empty list. The value is the tail of the list, that is, the part of the list after its first member. The tail of a list with one member is `nil`.

8.2.7. Length

The operand of the `len` operator is a string, an array, or a list. The value is an `int` giving the number of elements currently in the item.

8.2.8. Tagof

The operand of the `tagof` operator is a monadic expression of type `ref adt` that refers to a `pick adt`. or the type name of a `pick adt` or one of its variants. The value is an `int` giving a unique value for each of the variants and for the `pick adt` type itself.

8.2.9. Channel communication

The operand of the communication operator `<-` has type `chan of` *sometype*. The value of the expression is the first unread object previously sent over that channel, and has the type associated with the channel. If the channel is empty, the program delays until something is sent.

As a special case, the operand of `<-` may have type `array of chan of` *sometype*. In this case, all of the channels in the array are tested; one is fairly selected from those that have data. The expression yields a tuple of type `(int,` *sometype* `)`; its first member gives the index of the channel from which data was read, and its second member is the value read from the channel. If no member of the array has data ready, the expression delays.

Communication channels are treated more fully in §9.8 and §9.13 below with the discussion of the `alt` and `spawn` statements.

8.2.10. Creation of arrays

In the expressions

> `array [` *expression* `] of` *data-type*
> `array [` *expression* $_{opt}$ `] of {` *init-list* `,` $_{opt}$ `}`

the value is a new array of the specified type. In both forms, the *expression* must be of type `int`, and it supplies the size of the array. In the first form, the type is given, and the values in the array are initialized as appropriate to the underlying type. In the second form, a comma-separated list of values to initialize the array is given, optionally followed by a trailing comma. The type of the array is taken from the types of the initializers, which must all be the same. The list of initializers has the syntax

init-list:
  element
  init-list `,` *element*

element:
  expression
  expression `=>` *expression*
  `* =>` *expression*

In an *init-list* of plain expressions (without `=>`), the members of the array are successively initialized with the corresponding elements of the init-list. An element of the form `e1=>e2` initializes the member of the array at subscript `e1` with the expression `e2`. After such an element has been given, subsequent simple elements (without `=>`) begin initializing at position `e1+1` and so on. Each of the first expressions must be of type `int` and must evaluate to a constant (§8.5).

If an element of the form `* =>e2` is present, all members of the array not otherwise initialized are set to the value `e2`. The expression `e2` is evaluated for each subscript position, but in an undefined order. For example,

```
arr := array[3] of { * => array[3] of { * => 1 } };
```

yields a 2-dimensional array (actually an array of arrays) filled with `1`s.

If the expression giving the size of the array is omitted, its size is taken from the largest subscript of a member explicitly initialized. It is erroneous to initialize a member twice.

8.2.11. Creation of lists

The value of an expression

`list of {` *expression-list* `}`

is a list consisting of the expressions given. The types of the expressions must be identical, and this type is the underlying type of the list. The first expression is the head of the list, and the remaining expressions are a list constituting its tail. Where a list is expected, `nil` specifies an empty list.

8.2.12. Creation of channels

The value of

`chan of` *data-type*

is an initialized channel of the specified type. Just a declaration of a channel leaves it initialized only to `nil`; before it can be used it must be created. For example,

```
ch: chan of int;              # just declares, sets ch to nil
. . .
ch = chan of int;  # creates the channel and assigns it
```

8.2.13. Casts

An expression of the form

data-type monadic-expression

in which a type name is followed by an expression is called a *cast*, and converts the monadic expression to the named type. Only certain specialized forms are provided for.

8.2.13.1. Arithmetic casts

In arithmetic casts, the named type must be one of `byte`, `int`, `big`, or `real`, and the monadic-expression must have arithmetic type. For example,

```
byte 10
```

is an expression of `byte` type and value `10`. When real values are converted to integral ones, they are rounded to the nearest integer, and away from 0 if there is a tie. The effect of overflow during conversion is undefined.

8.2.13.2. Casts to strings

Here the named data type is `string`. In a first form, the monadic expression has arithmetic type (`byte`, `int`, `big`, or `real`) and the value is a string containing the decimal representation of the value, which may be either positive or negative. A `real` operand is converted as if by format %g, and if the result is converted back to real, the original value will be recovered exactly.

In a second form, the monadic expression has type array of byte. The value is a new string containing the Unicode characters obtained by interpreting the bytes in the array as a UTF-8 representation of that string. (UTF-8 is a representation of 16-bit Unicode characters as one, two, or three bytes.) The result of the conversion is undefined if the byte array ends within a multi-byte UTF-8 sequence.

8.2.13.3. Casts from strings

In a first form, the monadic expression is a string, and the named type is an arithmetic type. The value is obtained by converting the string to that type. Initial white space is ignored; after a possible sign, conversion ceases at the first character not part of a number.

In a second form, the named type is array of byte and the monadic-expression is a string. The value is a new array of bytes containing the UTF-8 representation of the Unicode characters in the string. For example,

```
s := "Ångström";
a := array of byte s;
s = string a;
```

takes the string s apart into bytes in the second line, and puts it back in the third. The length of s is 8, because it contains that many characters; the length of a is larger, because some of its characters require more than one byte in the UTF-8 representation.

8.2.13.4. Casts to adt and ref adt

Here the named type is that of an adt or ref adt, and the monadic expression is a comma-separated list of expressions within parentheses. The value of the expression is an instance of an adt of the named type whose data members are initialized with the members of the list, or whose single data member is initialized with the parenthesized expression. In case the type is ref adt, the value is a reference to the new instance of the adt.

The expressions in the list, read in order, correspond with the data members of the adt read in order; their types and number must agree. Placement of any function members of the adt is ignored. For example,

```
Point: adt {
         x: int;
         eq: fn (p: Point): int;
         y: int;
};
. . .
p: Point;
p = Point(1, 2);
```

puts in p a Point whose x value is 1 and whose y value is 2. The declaration and assignment could also be written

```
p := Point(1, 2);
```

8.3. Binary expressions

Binary expressions are either monadic expressions, or have two operands and an infix operator; the syntax is

binary-expression:
  monadic-expression
  binary-expression binary-operator binary-expression

binary-operator: one of
  `* / % + - << >> < > <= >= == != & ^ | :: && ||`

All these binary operators are left-associative except for `::`, which associates to the right. Their precedence is as listed here, with tightest first:

```
* / %
+ -
<< >>
< > <= >=
== !=
&
^
|
::
&&
||
```

8.3.1. Multiplicative operators

The `*`, `/`, and `%` operators respectively accomplish multiplication, division, and remainder. The operands must be of identical arithmetic type, and the result has that same type. The remainder operator does not apply to type `real`. If overflow or division by 0 occurs, the result is undefined. The absolute value of `a%b` is less than the absolute value of `b`; `(a/b)*b + a%b` is always equal to `a`; and `a%b` is non-negative if `a` and `b` are.

8.3.2. Additive operators

The `+` and `-` operators respectively accomplish addition and subtraction of arithmetic operands of identical type; the result has the same type. The behavior on overflow or underflow is undefined. The `+` operator may also be applied to strings; the result is a string that is the concatenation of the operands.

8.3.3. Shift operators

The shift operators are `<<` and `>>`. The left operand may be `big`, `int`, or `byte`; the right operand is `int`. The type of the value is the same as its left operand. The value of the right operand must be non-negative and smaller than the number of bits in the left operand. For the left-shift operator `<<`, the fill bits are 0; for the right-shift operator `>>`, the fill bits are a copy of the sign for the `int` case, and 0 for the `byte` case.

8.3.4. Relational operators

The relational operators are `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `==` (equal to), `!=` (not equal to). The first four operators, which generate orderings, apply only to arithmetic types and to strings; the types of their operands must be identical, except that a string may be compared to `nil`. Comparison on strings is lexicographic over the Unicode character set.

The equality operators `==` and `!=` accept operands of arithmetic, string, and reference types. In general, the operands must have identical type, but reference types and strings may be compared for identity with `nil`. Equality for reference types occurs when the operands refer to the same object, or when both are `nil`. An uninitialized string, or one set to `nil`, is identical to the empty string denoted `""` for all the relational operators.

The value of any comparison is the int value 1 if the stated relation is true, 0 if it is false.

8.3.5. Bitwise logical operators

The logical operators & (and), ^ (exclusive or) and | (inclusive or) require operands of the same type, which must be byte, int, or big. The result has the same type and its value is obtained by applying the operation bitwise.

8.3.6. List concatenation

The concatenation operator :: takes a object of any data type as its left operand and a list as its right operand. The list's underlying type must be the same as the type of the left operand. The result is a new list with the left operand tacked onto the front:

```
hd (a :: l)
```

is the same as a.

8.3.7. Logical operators

The logical *and* operator && first evaluates its left operand. If the result is zero, then the value of the whole expression is the int value 0. Otherwise the right operand is evaluated; if the result is zero, the value of the whole expression is again 0; otherwise it is 1. The operands must have the same arithmetic type.

The logical *or* operator || first evaluates its left operand. If the result is non-zero, then the value of the whole expression is the int value 1. Otherwise the right operand is evaluated; if the result is non-zero, the value of the whole expression is again 1; otherwise it is 0. The operands must have the same arithmetic type.

8.4. General Expressions

The remaining syntax for expressions is

expression:
  binary-expression
  lvalue-expression assignment-operator expression
  (*lvalue-expression-list*) = *expression*
  send-expression
  declare-expression
  load-expression

assignment-operator: one of
  = &= |= ^= <<= >>= += -= *= /= %=

The left operand of an assignment can take only certain forms, called lvalues.

lvalue-expression:
  identifier
  `nil`
  term `[` *expression* `]`
  term `[` *expression* `:` `]`
  term `.` *identifier*
  `(` *lvalue-expression-list* `)`
  `*` *monadic-expression*

lvalue-expression-list:
  lvalue
  lvalue-expression-list `,` *lvalue*

8.4.1. Simple assignments with `=`

In general, the types of the left and right operands must be the same; this type must be a data type. The value of an assignment is its new left operand. All the assignment operators associate right-to-left.

In the ordinary assignment with `=`, the value of the right side is assigned to the object on the left. For simple assignment only, the left operand may be a parenthesized list of lvalues and the right operand either a tuple or an `adt` whose data members correspond in number and type to the lvalues in the list. The members of the tuple, or the data members of the `adt`, are assigned in sequence to lvalues in the list. For example,

```
p: Point;
x, y: int;
(x, y) = p;
```

splits out the coordinates of the point into `x` and `y`. These rules apply recursively, so that if one of the components of the left side is a parenthesized list of lvalues, it is assigned from a corresponding `adt` or tuple on the right.

If the left operand of a simple assignment is an `adt` and the right side is a tuple, then the assignment assigns the members of the tuple to the `adt` data members; these must correspond in number and type with the members of the tuple.

The constant `nil` may be assigned to an lvalue of any reference type. This lvalue will compare equal to `nil` until it is subsequently reassigned. In the Inferno implementation of Limbo, such an assignment also triggers the removal of the object referred to unless other references to it remain.

The left operand of an assignment may be the constant `nil` to indicate that a value is discarded. This applies in particular to any of the lvalues in a tuple appearing on the left; to extend the examples above,

```
(x, nil) = p;
```

assigns the `x` member of the Point `p` to the variable `x`.

A special consideration applies to strings. If an `int` containing a Unicode character is assigned to a subscripted string, the subscript is normally required to lie within the string. As a special case, the subscript's value may be equal to the length of the string (that is, just beyond its end); in this case, the character is appended to the string, and the string's length increases by 1.

A final special case applies to array slices in the form `e1[e2:]`. Such expressions may lie on the left of `=`. The right side must be an array of the same type as `e1`, and its length must be less than or equal to `(len e1)-e2`. In this case, the elements in the array on the right replace the elements of e1 starting at position e2 . The length of the array is unchanged.

8.4.2. Compound assignments

A compound assignment with *op*= is interpreted in terms of the plain assignment;

```
e1 op= e2;
```

is equivalent to

```
e1 = (e1) op (e2);
```

except that e1 is evaluated only once.

8.4.3. Send expressions

A *send-expression* takes the form

send-expression:
  lvalue-expression <- = *expression*

In the expression

```
e1 <- = e2
```

the lvalue e1 must have type chan of *type* , and e2 must be of that type. The value of e2 is sent over the channel. If no task is executing a channel receive operation on the specified channel, the sender blocks. Task synchronization is discussed in §9.8 and §9.13 below.

8.4.4. Declare-expressions

A *declare-expression* is an assignment that also declares identifiers on its left:

declare-expression:
  lvalue-expression := *expression*

Each of the constituent terms in the *lvalue-expression* must be an identifier or nil . A plain identifier on the left is declared as having the type of the expression, and it is initialized with the expression's value. When a parenthesized list of identifiers is given, the expression must be a tuple or an adt , and the individual identifiers in the list are declared and initialized with the members of the tuple, or the data members of the adt . As with ordinary assignments, the keyword nil may stand for an identifier whose declaration and assignment are skipped.

The value and type of a declare-expression are the same as those of the expression.

8.4.5. Load expressions

A *load-expression* has the form

load-expression:
  load *identifier expression*

The identifier is the identifier of a module, that is, the type name declared in a module declaration. The expression following load has type string and names a file containing the compiled form of the module. The load expression yields a handle for referring to the functions provided by a module and its adt .

Execution of load brings the file containing the module into local memory and dynamically type-checks its interface: the run-time system ascertains that the declarations exported by the module are compatible with the module declaration visible in the scope of the `load` operator (see §11.2). In the scope of a module declaration, the types and constants exported by the module may be referred to without a handle, but the functions and data exported by the module (directly at its top level, or within its `adt`) may be called only using a valid handle acquired by the `load` operator.

The value of `load` is `nil` if the attempt to load fails, either because the file containing the module can not be found, or because the found module does not export the specified interface.

Each evaluation of `load` creates a separate instance of the specified module; it does not share data with any other instance.

8.5. Constant expressions

In several places a constant expression is required. Such an expression contains operands that are identifiers previously declared with `con`, or `int`, `big`, `real`, or `string` constants. These may be connected by any of the following operators:

```
+    -    *    /    %    &    |    ^
==   <    >    <=   >=   !=   <<   >>
&&   ||
~    !
```

together with arithmetic and string casts, and parentheses for grouping.

8.6. Expression evaluation

Expressions in Limbo are not reordered by the compiler; values are computed in accordance with the parse of the expression. However there is no guarantee of temporal evaluation order for expressions with side effects, except in the following circumstances: function arguments are fully evaluated before the function is called; the logical operators `&&` and `||` have fully defined order of evaluation, as explained above. All side effects from an expression in one statement are completed before the next statement is begun.

In an expression containing a constant subexpression (in the sense of §8.5), the constant subexpression is evaluated at compile-time with all exceptions ignored.

Underflow, overflow, and zero-divide conditions during integer arithmetic produce undefined results.

The `real` arithmetic of Limbo is all performed in IEEE double precision, although denormalized numbers may not be supported. By default, invalid operations, zero-divide, overflow, and underflow during real arithmetic are fatal; inexact-result is quiet. The default rounding mode is round-to-nearest-even. A set of routines in the `Math` library module permits independent control of these modes within each thread.

9. Statements

The executable code within a function definition consists of a sequence of statements and declarations. As discussed in the Scope section §11 below, declarations become effective at the place they appear. Statements are executed in sequence except as discussed below. In particular, the optional labels on some of the statements are used with `break` and `continue` to exit from or re-execute the labeled statement.

statements:
  (empty)
  statements declaration
  statements statement

statement:
  expression **;**
  ;
  { *statements* }
  if (*expression*) *statement*
  if (*expression*) *statement* else *statement*
  label$_{opt}$ while (*expression*$_{opt}$) *statement*
  label$_{opt}$ do *statement* while (*expression*$_{opt}$) **;**
  label$_{opt}$ for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$) *statement*
  label$_{opt}$ case *expression* { *qual–statement–sequence* }
  label$_{opt}$ alt { *qual–statement–sequence* }
  label$_{opt}$ pick *identifier* := *expression* { *pqual–statement–sequence* }
  break *identifier*$_{opt}$ **;**
  continue *identifier*$_{opt}$ **;**
  return *expression*$_{opt}$ **;**
  spawn *term* (*expression–list*$_{opt}$) **;**
  exit **;**

label:
  identifier **:**

9.1. Expression statements

Expression statements consist of an expression followed by a semicolon:

  expression **;**

Most often expression statements are assignments, but other expressions that cause effects are often useful, for example calling a function or sending or receiving on a channel.

9.2. Null statement

The null statement consists of a lone semicolon. It is most useful for supplying an empty body to a looping statement with internal side effects.

9.3. Blocks

Blocks are *statements* enclosed in {} characters.

  { *statements* }

A block starts a new scope. The effect of any declarations within a block disappears at the end of the block.

9.4. Conditional statements

The conditional statement takes two forms:

  if (*expression*) *statement*
  if (*expression*) *statement* else *statement*

The *expression* is evaluated; it must have type `int`. If it is non-zero, then the first *statement* is executed. In the second form, the second *statement* is executed if the *expression* is 0. The statement after `else` is connected to the nearest `else`-less `if`.

9.5. Simple looping statements

The simple looping statements are

> *label*$_{opt}$ `while ( `*expression*$_{opt}$` )` *statement*
> *label*$_{opt}$ `do` *statement* `while ( `*expression*$_{opt}$` ) ;`

In both cases the expression must be of type `int`. In the first form, the *expression* is first tested against 0; while it is not equal, the *statement* is repeatedly executed. In the second form, the *statement* is executed, and then, while the *expression* is not 0, the statement is repeatedly executed. If the *expression* is missing, it is understood to be non-zero.

9.6. `for` statement

The `for` statement has the form

> *label*$_{opt}$ `for (` *expression-1*$_{opt}$ `;` *expression-2*$_{opt}$ `;` *expression-3*$_{opt}$ `)` *statement*

It is equivalent to

```
expression-1 ;
while ( expression-2 ) {
     statement
     expression-3 ;
}
```

in the absence of `continue` or `break` statements. Thus (just as in C), the first expression is an initialization, the second a test for starting and continuing the loop, and the third a re-initialization for subsequent travels around the loop.

9.7. `case` statement

The `case` statement transfers control to one of several places depending on the value of an expression:

> *label*$_{opt}$ `case` *expression* `{` *qual-statement-sequence* `}`

The expression must have type `int` or `string`. The `case` statement is followed by sequence of qualified statements, which are statements labeled by expressions or expression ranges:

qual–statement–sequence:
  qual–list `=>`
  qual–statement–sequence *qual–list* `=>`
  qual–statement–sequence *statement*
  qual–statement–sequence *declaration*

qual–list:
  qualifier
  qual–list `or` *qualifier*

qualifier:
  expression
  expression `to` *expression*
  `*`

A *qual–statement–sequence* is a sequence of statements and declarations, each of which is preceded by one or more qualifiers. Syntactically, the qualifiers are expressions, expression ranges with `to`, or `*`. If the expression mentioned after `case` has `int` type, all the expressions appearing in the qualifiers must evaluate to integer constants (§8.5). If the expression has `string` type, all the qualifiers must be string constants.

The `case` statement is executed by comparing the expression at its head with the constants in the qualifiers. The test is for equality in the case of simple constant qualifiers; in range qualifiers, the test determines whether the expression is greater than or equal to the first constant and less than or equal to the second.

None of the ranges or constants may overlap. If no qualifier is selected and there is a `*` qualifier, then that qualifier is selected.

Once a qualifier is selected, control passes to the set of statements headed by that qualifier. When control reaches the end of that set of statements, control passes to the end of the `case` statement. If no qualifier is selected, the `case` statement is skipped.

Each qualifier and the statements following it up to the next qualifier together form a separate scope, like a block; declarations within this scope disappear at the next qualifier (or at the end of the statement.)

As an example, this fragment separates small numbers by the initial letter of their spelling:

```
case i {
1 or 8 =>
        sys->print("Begins with a vowel\n)";
0 or 2 to 7 or 9 =>
        sys->print("Begins with a consonant\n");
* =>
        sys->print("Sorry, didn't understand\n");
}
```

9.8. `alt` statement

The `alt` statement transfers control to one of several groups of statements depending on the readiness of communication channels. Its syntax resembles that of `case`:

label$_{opt}$ `alt {` *qual–statement–sequence* `}`

However, the qualifiers take a form different from those of `case`. In `alt`, each qualifier must be a `*`, or an expression containing a communication operator `<-` on a channel; the operator may specify either sending or receiving. For example,

```
outchan := chan of string;
inchan := chan of int;
alt {
        i := <-inchan =>
                sys->print("Received %d\n", i);
        outchan <- = "message" =>
                sys->print("Sent the message\n");
}
```

The alt statement is executed by testing each of the channels mentioned in the *qual-list* expressions for ability to send or receive, depending on the operator; if none is ready, the program blocks until at least one is ready. Then a random choice from the ready channels is selected and control passes to the associated set of statements.

If a qualifier of the form * is present, then the statement does not block; if no channel is ready the statements associated with * are executed.

If two communication operators are present in the same qualifier expression, only the leftmost one is tested by alt . If two or more alt statements referring to the same receive (or send) channel are executed in different threads, the requests are queued; when the channel becomes unblocked, the thread that executed alt first is activated.

As with case , each qualifier and the statements following it up to the next qualifier together form a separate scope, like a block; declarations within this scope disappear at the next qualifier (or at the end of the statement.) Thus, in the example above, the scope of i in the arm

```
        i := <-inchan =>
                sys->print("Received %d\n", i);
```

is restricted to these two lines.

As mentioned in the specification of the channel receive operator <- in §8.2.8, that operator can take an array of channels as an argument. This notation serves as a kind of simplified alt in which all the channels have the same type and are treated similarly. In this variant, the value of the communication expression is a tuple containing the index of the channel over which a communication was received and the value received. For example, in

```
a: array [2] of chan of string;
a[0] = chan of string;
a[1] = chan of string;
. . .
(i, s) := <- a;
# s has now has the string from channel a[i]
```

the <- operator waits until at least one of the members of a is ready, selects one of them at random, and returns the index and the transmitted string as a tuple.

During execution of an alt , the expressions in the qualifiers are evaluated in an undefined order, and in particular subexpressions may be evaluated before the channels are tested for readiness. Therefore qualifying expressions should not invoke side effects, and should avoid subparts that might delay execution. For example, in the qualifiers

```
ch <- = getchar() =>        # Bad idea
ich <- = next++ => # Bad idea
```

getchar() may be called early in the elaboration of the alt statement; if it delays, the entire alt may wait. Similarly, the next++ expression may be evaluated before testing the readiness of ich .

9.9. `pick` statement

The `pick` statement transfers control to one of several groups of statements depending upon the resulting variant type of a `pick adt` expression. The syntax resembles that of `case`:

> *label*$_{opt}$ `pick` *identifier* `:=` *expression* `{` *pqual-statement-sequence* `}`

The expression must have type `ref adt` and the `adt` must be a `pick adt`. The `pick` statement is followed by a sequence of qualified statements, which are statements labeled by the `pick` variant names:

pqual-statement-sequence:
  pqual-list `=>`
  pqual-statement-sequence *pqual-list* `=>`
  pqual-statement-sequence *statement*
  pqual-statement-sequence *declaration*

pqual-list:
  pqualifier
  pqual-list `or` *pqualifier*

pqualifier:
  identifier
  `*`

A *pqual-statement-sequence* is a sequence of statements and declarations, each of which is preceded by one or more qualifiers. Syntactically, the qualifiers are identifiers, identifier lists (constructed with `or`), or `*`. The identifiers must be names of the variant types of the `pick adt`. The `pick` statement is executed by comparing the variant type of the `pick adt` referenced by the expression at its head with the variant type names in the qualifiers. The matching qualifier is selected. None of the variant type names may appear more than once. If no qualifier is selected and there is a `*` qualifier, then that qualifier is selected.

Once a qualifier is selected, control passes to the set of statements headed by that qualifier. When control reaches the end of that set of statements, control passes to the end of the `pick` statement. If no qualifier is selected, the `pick` statement is skipped.

Each qualifier and the statements following it up to the next qualifier together form a separate scope, like a block; declarations within this scope disappear at the next qualifier (or at the end of the statement.)

The *identifier* and *expression* given in the `pick` statement are used to bind a new variable to a `pick adt` reference expression, and within the statements associated with the selected qualifier the variable can be used as if it were of the corresponding variant type.

As an example, given a `pick adt` of the following form:

```
Constant: adt {
        name: string;
        pick {
                Str or Pstring =>
                        s: string;
                Real =>
                        r: real;
        }
};
```

the following function could be used to print out the value of an expression of type `ref Constant`:

```
printconst(c: ref Constant)
{
        sys->print("%s: ", c.name);
        pick x := c {
        Str =>
                sys->print("%s\n", x.s);
        Pstring =>
                sys->print("[%s]\n", x.s);
        Real =>
                sys->print("%f\n", x.r);
        };
}
```

9.10. `break` statement

The `break` statement

`break` *identifier* $_{opt}$ `;`

terminates execution of `while`, `do`, `for`, `case`, `alt`, and `pick` statements. Execution of `break` with no identifier transfers control to the statement after the innermost `while`, `do`, `for`, `case`, `alt`, or `pick` statement in which it appears as a substatement. Execution of `break` with an identifier transfers control to the next statement after the unique enclosing `while`, `do`, `for`, `case`, `alt`, or `pick` labeled with that identifier.

9.11. `continue` statement

The `continue` statement

`continue` *identifier* $_{opt}$ `;`

restarts execution of `while`, `do`, and `for` statements. Execution of `continue` with no identifier transfers control to the end of the innermost `while`, `do`, or `for` statement in which the `continue` appears as a substatement. The expression that controls the loop is tested and if it succeeds, execution continues in the loop. The initialization portion of `for` is not redone.

Similarly, execution of `continue` with an identifier transfers control to the end of the enclosing `while`, `do`, or `for` labeled with the same identifier.

9.12. `return` statement

The `return` statement,

`return` *expression* $_{opt}$ `;`

returns control to the caller of a function. If the function returns a value (that is, if its definition and declaration mention a return type), the expression must be given and it must have the same type that the function returns. If the function returns no value, the expression must generally be omitted. However, if a function returns no value, and its last action before returning is to call another function with no value, then it may use a special form of `return` that names the function being called. For example,

```
f, g: fn(a: int);
f(a: int) {
        . . .
        return g(a+1);
}
```

is permitted. Its effect is the same as

```
f(a: int) {
        . . .
        g(a+1);
        return;
}
```

This *ad hoc* syntax offers the compiler a cheap opportunity to recognize tail-recursion.

Running off the end of a function is equivalent to return with no expression.

9.13. spawn **statement**

The spawn statement creates a new thread of control. It has the form

spawn *term* (*expression-list* $_{opt}$) ;

The term and expression-list are taken to be a function call. Execution of spawn creates an asynchronous, independent thread of control, which calls the function in the new thread context. This function may access the accessible objects in the spawning thread; the two threads share a common memory space. These accessible objects include the data global to the current module and reference data passed to the spawned function. Threads are preemptively scheduled, so that changes to objects used in common between threads may occur at any time. The Limbo language provides no explicit synchronization primitives; §12.3 shows examples of how to use channel communication to control concurrency.

9.14. exit **statement**

The exit statement

exit ;

terminates a thread and frees any resources belonging exclusively to it.

10. Referring to modules; import

As discussed above, modules present constants, functions, and types in their interface. Their names may be the same as names in other modules or of local objects or types within a module that uses another. Name clashes are avoided because references to the entities presented by a module are qualified by the module type name or an object of that module type.

For example, after the module and variable declarations

```
M: module {
        One: con 1;
        Thing: adt {
                t: int;
                f: fn();
        };
        g: fn();
};
m: M;
```

the name `One` refers to the constant defined in module `M` only in the contexts `M->One` or `m->One`; the name `Thing` as the particular data type associated with the `M` module can be referred to only in contexts like

```
th1: M->Thing;
th2: m->Thing;
```

Finally, to call a function defined either as a top-level member of the module, or as a member of one of its `adt`, it is necessary to declare, and also dynamically initialize using `load`, a handle for the module. Then calls of the form

```
m->g();
m->th1.f();
```

become appropriate. It is possible to use just the type name of a module to qualify its constants and types because constants and types can be understood without having the code and data present. Calling a function declared by a module or one of its `adt` requires loading the module.

The `import` declaration

identifier-list `: import` *identifier* `;`

lifts the identifiers in the *identifier-list* into the scope in which `import` appears, so that they are usable without a qualifier. The identifier after the `import` keyword is either a module identifier, or an identifier declared as having that type. The initial list of identifiers specifies those constants, types, and functions of the module whose names are promoted. In the case of constants and types, `import` merely makes their names accessible without using a qualifier. In the example above, if the `module` declaration above had been followed by

```
One, Thing: import M;
```

then one could refer to just `One` instead of `M->One`; similarly an object could be declared like

```
th: Thing;
```

For functions, and also `adt` with functions as members, `import` must specify a module variable (as opposed to a module identifier). Each imported name is associated with the specified module variable, and the current value of this module variable controls which instance of the module will be called. For example, after

```
g, Thing: import m;
```

then

```
g();
```

is equivalent to

```
m->g();
```

and

```
th: Thing;
th.f();
```

is equivalent to

```
th: M.Thing;
m->th.f();
```

When the module declaration for the module being implemented is encountered, an implicit import of all the names of the module is executed. That is, given

```
implement Mod;
. . .
Mod: module {
        . . .
};
```

the constants and types of Mod are accessed as if they had been imported; the functions declared in Mod are imported as well, and refer dynamically to the current instance of the module being implemented.

11. Scope

The scope of an identifier is the lexical range of a program throughout which the identifier means a particular type of, or instance of, an object. The same identifier may be associated with several different objects in different parts of the same program.

The names of members of an adt occupy a separate, nonconflicting space from other identifiers; they are declared in a syntactically distinct position, and are always used in a distinguishable way, namely after the . selection operator. Although the same scope rules apply to adt members as to other identifiers, their names may coincide with other entities in the same scope.

Similarly, the names of constants, functions, and adt appearing within a module declaration are ordinarily qualified either with the name of the module or with a module variable using the -> notation. As discussed above, the import declaration lifts these names into the current scope.

Identifiers declared in a top-declaration (§5) have scope that lasts from the declaration throughout the remainder of the file in which it occurs, unless it is overridden by a redeclaration of that name within an inner scope. Each function definition, and each block within a function, introduces a new scope. A name declared within the block or function (including a formal argument name of a function) has a scope that begins at the completion of its declaration and lasts until the end of the block or function. If an already-declared identifier is redeclared within such an inner scope, the declaration previously in force is used in any initialization expression that is part of the new declaration.

As discussed above, within case alt and pick, each qualifier and the statements following it form an inner scope just like a block.

The scope of a label is restricted to the labeled statement, and label names may coincide with those of other entities in the same scope.

11.1. Forward referencing

In general, names must be declared before they are used.

The first exception to this rule is that a function local to a module need not have a declaration at all; it is sufficient to give its definition, and that definition may appear anywhere in the module.

The general rule implies that no adt may contain, as a member, an adt not previously declared (including an instance of itself). A second exception to this rule applies to ref adt types. An adt may contain a member whose type is a ref to itself, or to another adt even if the second adt has not yet been declared. Unless a special notation is used, such references are restricted: all mutual or self references among adt are checked statically throughout all the adt visible in a module to determine which members refer to other adt. Any member of an adt of ref adt type that refers directly, or indirectly through a chain of references, back to its own underlying type may not be assigned to individually; it can gain a value only by an assignment to the adt as a whole. For example, in

```
Tree: adt {
        l: ref Tree;
        r: ref Tree;
        t: ref Ntree;
};
Ntree: adt {
        t: ref Tree;
};

t1 := Tree(nil, nil, nil);  # OK
t2 := Tree(ref t1, ref t1, nil);      # OK
t1 = Tree(ref t1, ref t2, nil);       # OK
t1.l = ... ;        # not OK

nt := ref Ntree(nil);         # OK
nt.t = ...          # not OK
```

the first three assignments are correct, but any assignment to t1.l is forbidden, because it is self-referential. The situation is the same with the mutually referential fields of the Tree and Ntree adt.

These restrictions suffice to prevent the creation of circular data structures. Limbo implementations guarantee to destroy all data objects not involved in such circularity immediately after they become non-referenced by active tasks, whether because their names go out of scope or because they are assigned new values. This property has visible effect because certain system resources, like windows and file descriptors, can be seen outside the program. In particular, if a reference to such a resource is held only within an adt, then that resource too is destroyed when the adt is.

The default rules are burdensome because they impede the construction even of harmless structures like trees. Therefore an escape is provided: using the word cyclic before the type in an adt member removes the circular-reference restriction for that member. For example,

```
Tree: adt {
        l: cyclic ref Tree;
        r: cyclic ref Tree;
        t: ref Ntree;
};
Ntree: adt {
        t: cyclic ref Tree;
};

t1 := Tree(nil, nil, nil);    # OK
t2 := Tree(ref t1, ref t1, nil);     # OK
t1 = Tree(ref t1, ref t2, nil);      # OK
t1.l = ... ;      # OK now

nt := ref Ntree(nil);        # OK
nt.t = ...        # OK now
```

With the use of cyclic, circular data structures can be created. When they become unreferenced except by themselves, they will be garbage-collected eventually, but not instantly.

11.2. Type equality and compatibility

In an assignment and in passing an actual argument to a function, the types of the target and the expression being assigned or passed must be equal (with certain exceptions, e.g. assignment of nil to a reference type). When a function is defined, its type must be equal to the type of a function with the same name if one is in scope. Type equality is determined as follows.

Two basic types are equal if and only if they are identical.

Two tuple types are equal if and only if they are composed of equal types in the same order.

Two array types are equal if and only if they are arrays of equal types. The size of an array is not part of its type.

Two list types are equal if and only if they are composed of equal types.

Two channel types are equal if and only if they transmit equal types.

Two adt types are equal if and only if their data members have the same names and correspondingly equal types, including any cyclic attribute. The order of member declaration is insignificant, and constant and function members of an adt do not enter into the comparison, nor does the name of the adt type itself. In particular, with the declarations

```
A: adt { x: ref B; };
B: adt { x: ref A; };
```

the types A and B are equal.

Two ref adt types are equal if and only if they are references to equal adt types.

Two module types are equal if and only if their data and function members have the same names and correspondingly equal types; the order of their mention is insignificant. Constant members and type members do not enter into the comparison.

Two function types are equal if and only if their return values have the same type and their argument lists have correspondingly equal types. Any self attributes given to arguments much match. Names given to arguments do not enter into the comparison.

A typename has the same type as the type from which it was constructed.

When a module is loaded, the module stored in the file system must have a type that is *compatible* with the type mentioned in the load expression. The type of the stored module type is compatible with the mentioned type if and only if all data members of the two types are equal in name and type, and all adt or functions actually mentioned by the program executing load have names and types equal to corresponding members of the stored module.

12. Examples

Because Limbo was designed for the Inferno environment, several of these examples consist of simplified versions of already simple Inferno applications in a prototype Inferno implementation. Some appreciation for the resources available in this environment should become evident, but its full description is available elsewhere; the discussion here will focus on language features. However, several of the programs use facilities from the module Sys , which provides an interface to a file system and its methods resembling those of Unix or Plan 9, as well as other useful library facilities.

Some of the programs are annotated with line numbers; they are there only for descriptive purposes.

12.1. A simple command interpreter module

This version of a shell program reads from a keyboard and executes commands typed by the user. Its own interface has the type of a Command module, and that is the type of the things it executes. In particular, it can call modules like the hello example at the beginning of the paper.

```
1	implement Command;

2	include "sys.m";
3	include "draw.m";

4	sys: Sys;
5	stdin: ref Sys->FD;

6	Command: module
7	{
8		init: fn(nil: ref Draw->Context, nil: list of string);
9	};
```

After the boilerplate on lines 1–3, the variables sys and stdin are declared on lines 4 and 5. The I/O operations of the Sys module use the ref FD type to refer to open files.

```
10        init(ctx: ref Draw->Context, nil: list of string)
11        {
12
13
14                buf := array[256] of byte;

15                sys = load Sys Sys->PATH;
16                stdin = sys->fildes(0);

17                for(;;) {
18                        sys->print("$ ");
19                        n := sys->read(stdin, buf, len buf);
20                        if(n <= 0)
21                                break;
22                        (nw, arg) :=
                            sys->tokenize(string buf[0:n], " \t\n");
23                        if(nw != 0)
24                                exec(ctx, arg);
25                }
26        }
```

Line 10: conventionally, stand-alone modules are started by calling their init functions. The Command module follows this convention. The arguments are presented as a list of strings. In this simple example, the command interpreter itself ignores its argument, so it need not be given a name.

Local variables are declared on lines 12-14; line 15 loads the Sys module and stores a handle for it in the variable sys . Line 16 creates an FD for the standard input by calling the fildes function of the Sys module using the -> operator; the notation modhandle->func(...) specifies a call to the function named func in the module currently referred to by modhandle . (In general there can be several modules of the same type and name active, and there can also be unrelated modules containing identically named functions. The import declaration, described in §6.6 above, can be used to abbreviate the references when names do not clash.)

The loop on lines 17-25 prints a prompt (line 18), reads a line from the standard input (line 19), parses it into tokens (line 22), and executes the command.

The function call sys->tokenize is worth discussing as an example of style. It takes two strings as arguments. The characters in the second string are interpreted as separators of tokens in the first string. It returns a tuple whose first member is the number of tokens found, and whose second is a list of strings containing the tokens found: its declaration is

```
tokenize: fn (s: string, sep: string): (int, list of string);
```

In the example, the second argument is " \t\n" , so that the routine returns the number of, and a list of, words separated by blanks, tabs, and new-lines. The free use of strings, lists, and tuple-returning functions is common in Limbo.

The sys->read routine gathers an array of bytes into buf . Thus the expression for the first argument of sys->tokenize converts this array to a string by slicing the array with [0:n] , using the actual number of bytes gathered by the read , and using a cast.

At lines 23-24, if there were any words found, exec is called:

```
27        exec(ctx: ref Draw->Context, args: list of string)
28        {
29                c: Command;
30                cmd, file: string;

31                cmd = hd args;

32                file = cmd + ".dis";
33                c = load Command file;
34                if(c == nil)
35                        c = load Command "/dis/"+file;

36                if(c == nil) {
37                        sys->print("%s: not found\n", cmd);
38                        return;
39                }
40                c->init(ctx, args);
41        }
```

On lines 31 and 32 of exec, cmd is set to the first of the words in the argument list, and the string .dis is concatenated to it (to account for the fact that Limbo object program files are conventionally named using this suffix). On line 33 an attempt is made to load the named module from the derived file name; it will fail if the file does not exist. The attempt will succeed, and a non-nil handle assigned to c, if the file is found, and if the module stored in that file does in fact implement the Command module type. In case this fails, lines 34-35 make another attempt, after prefixing /dis/ to the file name.

If either attempt to get a handle to the named module succeeds, c will contain a valid handle to it; line 40 calls its init function, passing it the whole argument list. When it returns, the exec function returns, and the main loop resumes.

12.2. Infrared remote control

This example shows two instances of a module for interfacing to a TV remote control; one is for the real remote, which in this case is connected to a serial port on a set-top box, and the other is simulated for testing programs running on a regular operating system. The techniques of special interest are the dynamic use of modules and the communication using a channel.

The module is used by creating a channel and passing it to the module s init function, which returns a success/error indicator and starts an asynchronous process to read the remote control. The user of the module executes a receive on the channel whenever it wishes to accept a button-push.

The (abridged) module declaration is

```
Ir: module
{
        # Codes buttons on IR remote control
        Zero:       con 0;
        One:        con 1;
        . . .
        Mute:       con 23;
        Error:      con 9999;

        init: fn(chan of int): int;
        PATH: con "/dis/ir.dis";
        SIMPATH: con "/dis/irsim.h";
};
```

The implementation for the real remote control is

```
implement Ir;

include "ir.m";
include "sys.m";
FD, Dir: import Sys;

sys: Sys;

init(keys: chan of int): int
{
        cfd, dfd: ref FD;

        sys = load Sys Sys->PATH;

        cfd = sys->open("/dev/eia1ctl", sys->OWRITE);
        if(cfd == nil)
                return -1;
        sys->fprint(cfd, "b9600");

        dfd = sys->open("/dev/eia1", sys->OREAD);
        cfd = nil;

        spawn reader(keys, dfd);
        return 0;
}
```

The `init` routine accepts a `chan` argument; it will be used by the module to send codes for the buttons pressed by the user. In this routine, the calls to `sys->open` and `sys->fprint` open and set up the device data and control files `/dev/eia1` and `/dev/eia1ctl` used to communicate with the device itself. The important step is at the end: the `spawn` statement creates a new, asynchronous task to read the device, using a routine that is passed the communication channel and the FD for the device:

```
reader(keys: chan of int, dfd: ref FD)
{
	n, ta, tb: int;
	dir: Dir;
	b1:= array[1] of byte;
	b2:= array[1] of byte;

	# find the number of bytes already
	# queued and flush that many
	(n, dir) = sys->fstat(dfd);
	if(n >= 0 && dir.length > 0) {
		while(dir.length) {
			n = sys->read(dfd,
			    array[dir.length] of byte,
			    dir.length);
			if(n < 0)
				break;
			dir.length -= n;
		}
	}

loop:	for(;;) {
		n = sys->read(dfd, b1, len b1);
		if(n <= 0)
			break;
		ta = sys->millisec();
		# Button pushes are pairs of characters
		# that arrive closer together than
		# 200 ms.  Longer than that is likely
		# to be noise.
		for(;;) {
			n = sys->read(dfd, b2, 1);
			if(n <= 0)
				break loop;
			tb = sys->millisec();
			if(tb - ta <= 200)
				break;
			ta = tb;
			b1[0] = b2[0];
		}
		# map the character pair; the significant
		# bits are the lowest 5.
		case ((int b1[0]&16r1f)<<5) | (int b2[0]&16r1f) {
		975 =>	n = Ir->Zero;
		479 =>	n = Ir->One;
		. . .
		791 =>	n = Ir->Mute;
		* =>	n = Ir->Error;
		}
		# found a button-push; send the value
		keys <-= n;
	}
	keys <-= Ir->Error;
}
```

The code in the middle is related to noise-filtering and is uninteresting in detail except as it illustrates some of the methods provided by the Sys module; the crucial actions are found at the bottom, where the routine sends either a true button-push or an error code over the channel to the module s client.

Here is another implementation of the same interface. Its init function performs the same kind of initialization as the other version, but using the operating system s keyboard files /dev/cons and /dev/consctl. In the Inferno environment, operations corresponding to the Unix stty primitive are accomplished by writing messages to a control file associated with the file that handles the data.

```
implement   Ir;

include   "ir.m";
include   "sys.m";
FD:  import   Sys;

sys:  Sys;
cctlfd:   ref  FD;

init(keys:    chan  of  int):  int
{
          dfd:  ref  FD;

          sys  =  load  Sys  Sys->PATH;

          cctlfd   =  sys->open("/dev/consctl",      sys->OWRITE);
          if(cctlfd   ==  nil)
                    return  -1;
          sys->write(cctlfd,     array  of  byte  "rawon",   5);

          dfd  =  sys->open("/dev/cons",      sys->OREAD);
          if(dfd  ==  nil)
                    return  -1;

          spawn  reader(keys,    dfd);
          return   0;
}
```

A fine point: the variable cctlfd that contains the FD for the control device is declared external to the init function, even though it appears to be used only inside it. Programming cleanliness suggests that its declaration be moved inside, but here that won t work; device control files in Inferno retain settings like raw mode only while they remain open. If cctlfd were declared inside init, then returning from init would destroy the last reference to the FD for the control file, and the device would be closed automatically.

The reader function for this module has the same structure as the first example, but doesn t have to worry about a noisy infrared detector:

```
reader(keys: chan of int, dfd: ref FD)
{
        n: int;
        b:= array[1] of byte;

        for(;;) {
                n = sys->read(dfd, b, 1);
                if(n != 1)
                        break;
                case int b[0] {
                '0' =>  n = Ir->Zero;
                '1' =>  n = Ir->One;
                . . .
                16r7f => n = Ir->Mute;
                * =>    n = Ir->Error;
                }
                keys <-= n;
        }
        keys <-= Ir->Error;
}
```

The following module can be used to test the above code. It simply prints the name of the button that was pressed.

```
implement Irtest;

include "sys.m";
include "draw.m";
FD: import Sys;
include "ir.m";

Irtest: module
{
        init: fn(nil: ref Draw->Context, nil: list of string);
};
ir: Ir;
sys: Sys;
```

```
init(nil: ref Draw->Context, nil: list of string)
{
	c: int;
	stderr: ref FD;
	irchan := chan of int;

	sys = load Sys Sys->PATH;
	stderr = sys->fildes(2);

	# If the real IR remote application can
	# be found, use it, otherwise use the simulator:
	ir = load Ir Ir->PATH;
	if(ir == nil)
		ir = load Ir Ir->SIMPATH;
	if(ir == nil) {
		# %r format code means the last system error string
		sys->fprint(stderr, "load ir: %r\n");
		return;
	}
	if(ir->init(irchan) != 0) {
		sys->fprint(stderr, "Ir.init: %r\n");
		return;
	}
	names := array[] of {
		"Zero",
		"One",
		. . .
		"Mute",
	};
	for(;;) {
		c = <-irchan;
		if(c == ir->Error)
			sys->print("Error %d\n", c);
		else
			sys->print("%s\n", names[c]);
	}
}
```

Finally, here is a snippet from a movie application that uses the IR module; it demonstrates how alt is useful for dealing with multiple events. This is only one of the functions of the movie module, so not everything is defined. It uses the Mpeg module, which actually copies the MPEG data stream to the screen asynchronously. Its play function takes, as one of its arguments, a channel; before starting to play it writes a string on the channel. An empty string indicates success at locating the movie; a non-empty string contains an error message. When it finishes it, writes another string.

```
movie(entry: ref Dbinfo, cc: chan of int)
{
	i: int;
	m: Mpeg;
	b: ref Image;

	m = load Mpeg Mpeg->PATH;
	if (m == nil)
		return;
	# make a place on the screen
	w := screen.window(screen.image.r);

	mr := chan of string;
	s := m->play(w, 1, w.r, entry.movie, mr);
	if(s != "")
		return;
	# wait for the end of the movie
	# while watching for button pushes
	for(;;) {
		alt {
		<-mr =>
			return;
		i = <-cc =>
			case i {
			Ir->Select =>
				m->ctl("stop");
			Ir->Up or Ir->Dn =>
				m->ctl("pause");
			}
		}
	}
}
```

12.3. Monitors

Statically allocated storage within a module is accessible to all the functions of that module, and there is no explicit mechanism in Limbo for synchronizing concurrent updates to this storage from several tasks. However, it is straightforward to build a variety of concurrency-control mechanisms by using channel communications.

An example is a module that implements a Monitor abstract data type. Each instance of Monitor has a lock and an unlock operation; calling lock delays if another task holds the lock; calling unlock releases the lock and enables any other task attempting to execute lock.

```
implement Mon;

Mon: module
{
	Monitor: adt {
		create: fn(): Monitor;
		lock: fn(m: self Monitor);
		unlock: fn(m: self Monitor);
		ch: chan of int;
	};
};

Monitor.create(): Monitor
{
	m := Monitor(chan of int);
	spawn lockproc(m.ch);
	return m;
}

Monitor.lock(m: self Monitor)
{
	m.ch <- = 0;
}

Monitor.unlock(m: self Monitor)
{
	<- m.ch;
}

lockproc(ch: chan of int)
{
	for (;;) {
		<- ch;	# wait for someone to lock
		ch <- = 0; # wait for someone to unlock
	}
}
```

It would be used like this:

```
mp: Mon;
Monitor: import mp;
mp = load Mon "...";
l := Monitor.create();
l.lock();
# region of code to be protected;
# only one thread can execute here at once.
l.unlock();
```

The `create` method of `Monitor` allocates an instance of a `Monitor` containing an initialized channel. It also creates a thread executed in the `lockproc` routine, which repeatedly reads from the channel, then writes on it. The values transmitted over the channel are of no interest; it is the pure fact of communication that is put to use. The `lock` routine sends a message; in the idle state, the `lockproc` thread reads it and the sender proceeds. Meanwhile, `lockproc` tries to send a message over the same channel. If another thread attempts to `lock`, there is no reader for the channel, and so its transmission will block. At some point, the thread that gained the lock calls `unlock`, which receives from the channel. Depending on timing, this reception enables execution of either lockproc or one of the threads attempting to send via lock.

12.4. Buffered channels

Limbo channels are unbuffered; a sender blocks until there is a receiver. This example shows a way to make a buffered channel of strings from an unbuffered channel. It is written as a module whose bufchan function takes a chan of string and a size as argument, and returns a new channel; it creates an asynchronous task that accepts input from the argument channel and saves up to size strings, meanwhile trying to send them to its user.

```
implement Bufchan;
Bufchan: module {
	bufchan: fn(c: chan of string, size: int): chan of string;
};

xfer(oldchan, newchan: chan of string, size: int)
{
	temp := array[size] of string;
	fp := 0;		# first string in buffer
	n := 0;		# number of strings in buffer
	dummy := chan of string;
	sendch, recvch: chan of string;
	s: string;

	for (;;) {
		sendch = recvch = dummy;
		if (n > 0)
			sendch = newchan;
		if (n < size)
			recvch = oldchan;
		alt {
		s = <-recvch =>
			temp[(fp+n)%size] = s;
			n++;

		sendch <- = temp[fp] =>
			temp[fp++] = nil;
			n--;
			if (fp>=size)
				fp -= size;
		}
	}
}

bufchan(oldchan: chan of string, size: int): chan of string
{
	newchan := chan of string;
	spawn xfer(oldchan, newchan, size);
	return newchan;
}
```

The module is somewhat specialized, but it illustrates useful programming techniques. The most interesting occurs in xfer, which does the work. The problem xfer faces is that it doesn't want to receive input when its buffer is full, nor to try to send when it has nothing to transmit. The solution here is to use a dummy channel on which nothing is ever sent or received; in the alt statement, that channel substitutes for the real input channel when the buffer is full, and for the output channel when the buffer is empty.

The module could be used in the following way:

```
Bufchan: module {
        PATH: con "/appl/lib/bufchan.dis";
        bufchan: fn(c: chan of string, size: int): chan of string;
};
bufc := load Bufchan Bufchan->PATH;
sourcech := chan of string;

# ... (here, hand off sourcech to a process that
#       reads strings from it and copies them to ch)
ch: chan of string = bufc->bufchan(sourcech, 10);
s := <- ch;
```

13. Syntax summary

This section summarizes the grammar of Limbo above the lexical level; constants and identifiers are left undefined.

program:
  `implement` *identifier* `;` *top-declaration-sequence*

top-declaration-sequence:
  top-declaration
  top-declaration-sequence *top-declaration*

top-declaration:
  declaration
  identifier-list `:=` *expression* `;`
  identifier-list `=` *expression* `;`
  `(` *identifier-list* `)` `:=` *expression* `;`
  module-declaration
  function-definition
  adt-declaration

declaration:
  identifier-list `:` *type* `;`
  identifier-list `:` *type* `=` *expression* `;`
  identifier-list `:` `con` *expression* `;`
  identifier-list `:` `import` *identifier* `;`
  identifier-list `:` `type` *type* `;`
  `include` *string-constant* `;`

identifier-list:
  identifier
  identifier-list `,` *identifier*

expression-list:
  expression
  expression-list `,` *expression*

type:
  data-type
  function-type

data−type:
    `byte`
    `int`
    `big`
    `real`
    `string`
    tuple−type
    `array of` *data−type*
    `list of` *data−type*
    `chan of` *data−type*
    adt−type
    `ref` *adt−type*
    module−type
    module−qualified−type
    type−name

tuple−type:
    `(` *data−type−list* `)`

data−type−list:
    data−type
    data−type−list `,` *data−type*

adt−type:
    identifier
    module−qualified−type

module−type:
    identifier

module−qualified−type:
    identifier `->` *identifier*

type−name:
    identifier

function−type:
    `fn` *function−arg−ret*

function−arg−ret:
    `(` *formal−arg−list* $_{opt}$ `)`
    `(` *formal−arg−list* $_{opt}$ `)` `:` *data−type*

formal−arg−list:
    formal−arg
    formal−arg−list `,` *formal−arg*

formal−arg:
    nil−or−D−list `:` *type*
    nil−or−D `:` `self` `ref` $_{opt}$ *identifier*
    nil−or−D `:` `self` *identifier*
    `*`

nil−or−D−list:
    nil−or−D
    nil−or−D−list `,` *nil−or−D*

nil−or−D:
  identifier
  `nil`

module−declaration:
  identifier `:` `module` `{` *mod−member−list* $_{opt}$ `}` `;`

mod−member−list:
  mod−member
  mod−member−list *mod−member*

mod−member:
  identifier−list `:` *function−type* `;`
  identifier−list `:` *data−type* `;`
  adt−declaration `;`
  identifier−list `:` `con` *expression* `;`
  identifier−list `:` `type` *type* `;`

adt−declaration:
  identifier `:` `adt` `{` *adt−member−list* $_{opt}$ `}` `;`

adt−member−list:
  adt−member
  adt−member−list *adt−member*

adt−member:
  identifier−list `:` `cyclic` $_{opt}$ *data−type* `;`
  identifier−list `:` `con` *expression* `;`
  identifier−list `:` *function−type* `;`
  `pick` `{` *pick−member−list* `}`

pick−member−list:
  pick−tag−list `=>`
  pick−member−list *pick−tag−list* `=>`
  pick−member−list *identifier−list* `:` `cyclic` $_{opt}$ *data−type* `;`

pick−tag−list:
  identifier
  pick−tag−list `or` *identifier*

function−definition:
  function−name−part *function−arg−ret* `{` *statements* `}`

function−name−part:
  identifier
  function−name−part `.` *identifier*

statements:
  (empty)
  statements *declaration*
  statements *statement*

statement:
expression `;`
`;`
`{` *statements* `}`
`if` `(` *expression* `)` *statement*
`if` `(` *expression* `)` *statement* `else` *statement*
label $_{opt}$ `while` `(` *expression* $_{opt}$ `)` *statement*
label $_{opt}$ `do` *statement* `while` `(` *expression* $_{opt}$ `)` `;`
label $_{opt}$ `for` `(` *expression* $_{opt}$ `;` *expression* $_{opt}$ `;` *expression* $_{opt}$ `)` *statement*
label $_{opt}$ `case` *expression* `{` *qual–statement–sequence* `}`
label $_{opt}$ `alt` `{` *qual–statement–sequence* `}`
label $_{opt}$ `pick` *identifier* `:=` *expression* `{` *pqual–statement–sequence* `}`
`break` *identifier* $_{opt}$ `;`
`continue` *identifier* $_{opt}$ `;`
`return` *expression* $_{opt}$ `;`
`spawn` *term* `(` *expression–list* $_{opt}$ `)` `;`
`exit` `;`

label:
identifier `:`

qual–statement–sequence:
qual–list `=>`
qual–statement–sequence *qual–list* `=>`
qual–statement–sequence *statement*
qual–statement–sequence *declaration*

qual–list:
qualifier
qual–list `or` *qualifier*

qualifier:
expression
expression `to` *expression*
`*`

pqual–statement–sequence:
pqual–list `=>`
pqual–statement–sequence *pqual–list* `=>`
pqual–statement–sequence *statement*
pqual–statement–sequence *declaration*

pqual–list:
pqualifier
pqual–list `or` *pqualifier*

pqualifier:
identifier
`*`

expression:
binary–expression
lvalue–expression *assignment–operator* *expression*
`(` *lvalue–expression–list* `)` `=` *expression*
send–expression
declare–expression
load–expression

binary−expression:
monadic−expression
binary−expression binary−operator binary−expression

binary−operator: one of
`* / % + − << >> < > <= >= == != & ^ | :: && ||`

assignment−operator: one of
`= &= |= ^= <<= >>= += −= *= /= %=`

lvalue−expression:
identifier
`nil`
term `[` *expression* `]`
term `[` *expression* `:` `]`
term `.` *identifier*
`(` *lvalue−expression−list* `)`
`*` *monadic−expression*

lvalue−expression−list:
lvalue
lvalue−expression−list `,` *lvalue*

expression:
term
monadic−operator monadic−expression
`array` `[` *expression* `]` `of` *data−type*
`array` `[` *expression* $_{opt}$ `]` `of` `{` *init−list* `}`
`list` `of` `{` *expression−list* `}`
`chan` `of` *data−type*
data−type monadic−expression

term:
identifier
constant
real−constant
string−constant
`nil`
`(` *expression−list* `)`
term `.` *identifier*
term `->` *term*
term `(` *expression−list* $_{opt}$ `)`
term `[` *expression* `]`
term `[` *expression* `:` *expression* `]`
term `[` *expression* `:` `]`
term `++`
term `−−`

monadic−operator: one of
`+ − ! ~ ref * ++ −− <- hd tl len tagof`

init−list:
element
init−list `,` *element*

element:
  expression
  expression `=>` *expression*
  `*` `=>` *expression*

send−expression:
  lvalue−expression `<-` `=` *expression*

declare−expression:
  lvalue−expression `:=` *expression*

load−expression:
  `load` *identifier* *expression*