Contents

1 The Lua-ML application program interface

1.1 Values

As in Lua, we keep the value space simple and small. Unlike Lua, we have only one kind of function. The type `userdata'` is not specified here; it is intended to be supplied by a combination of user-defined libraries.

The `VALUE` interface is a key for clients because it not only specifies what a Lua value is, but also provides higher-order embedding/projection pairs so that Caml values can be mapped to Lua values and back again.

1 ⟨*lua.mli* 1⟩≡ 2a ▷

```
module type VALUE = sig
  type 'a userdata'
  type srcloc
  type initstate
  type value =
    | Nil
    | Number   of float
    | String   of string
    | Function of srcloc * func
    | Userdata of userdata
    | Table    of table
  and func  = value list -> value list
  and table = (value, value) Luahash.t
  and userdata  = value userdata'
  and state = { globals : table
              ; fallbacks : (string, value) Hashtbl.t
              ; mutable callstack : activation list
              ; mutable currentloc : Luasrcmap.location option (* supersedes top of stack *)
              ; startup : initstate
              }
  and activation = srcloc * Luasrcmap.location option

  val caml_func : func -> value (* each result unique *)
  val lua_func  : file:string -> linedefined:int -> func -> value
  val srcloc    : file:string -> linedefined:int -> srcloc (* must NOT be reused *)
  val eq        : value -> value -> bool
  val to_string : value -> string
  val activation_strings : state -> activation -> string list
  type objname = Fallback of string | Global of string | Element of string * value
  val objname : state -> value -> objname option
     (* 'fallback', 'global', or 'element', name *)

  val state : unit -> state (* empty state, without even fallbacks *)
  val at_init : state -> string list -> unit  (* run code at startup time *)
  val initcode : state -> (string -> unit) -> unit (* for the implementation only *)
```

Defines:
  `activation_strings`, used in chunks 44b and 62b.
  `at_init`, never used.

caml_func, used in chunks 43b, 88, 90, 91a, 95, and 102.
eq, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
initcode, used in chunk 41.
initstate, never used.
lua_func, never used.
objname, used in chunks 47c, 82b, and 95.
srcloc, used in chunks 13, 39, 63, 82b, and 97.
state, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
to_string, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
value, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

Uses activation 80a, bool 3a 76a 85a 99, file 99 113, float 3a 76a 84a 111b, func 5a 64a 78a 80a 90, int 3a 76a 85a 99, list 3d 76d 86b 99, location 127, name 127, option 3b 76b 86a, result 5a 78a 89, string 3a 76a 84a 99, table 4a 77a 80a 87a, unit 3a 76a 85a, userdata 3a 76a 80a 84a, and userdata' 80a.

If a library wants to register Lua code to be executed at startup time, it can call at_init. No library should ever call initcode; that function is reserved for the implementation, which uses it to run the registered code.

Lua tables are not quite like Caml tables, but they are close.

2a ⟨*lua.mli* 1⟩+≡ ◁1 2b▷
```
module Table : sig
  val create : int -> table
  val find   : table -> key:value -> value
  val bind   : table -> key:value -> data:value -> unit
  val of_list : (string * value) list -> table
end
```
Defines:
bind, used in chunks 50, 53, 66, 86b, 92, 95, and 113.
create, used in chunks 5b, 53, 66, 78b, 86b, 92, 93, and 104.
find, used in chunks 44, 50, 66, 85b, 86b, 91, 92, and 102.
of_list, used in chunks 87b, 92, 93, and 99.

Uses int 3a 76a 85a 99, list 3d 76d 86b 99, string 3a 76a 84a 99, table 4a 77a 80a 87a, unit 3a 76a 85a, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

Now, for embedding and projection. This is a key, as it completely replaces the stupid stack-based interface found in C. Instead, we use higher-order functions to whip up functionality as needed. If a projection fails, it raises Projection giving the value it tried to project and a string indicating what it tried to project to. Library clients may use the function projection to achieve this effect.

2b ⟨*lua.mli* 1⟩+≡ ◁2a 3a▷
```
exception Projection of value * string
val projection : value -> string -> 'a
type ('a, 'b, 'c) ep = ('a, 'b, 'c) Luavalue.ep
  = { embed : 'a -> 'b; project : 'b -> 'a; is : 'c -> bool }
type 'a map  = ('a, value, value) ep
type 'a mapf  (* used to build function maps that curry/uncurry *)
```
Defines:
projection, used in chunks 6b, 8a, 19, 79b, 99, 104, and 113.

Uses bool 3a 76a 85a 99, map 83, mapf 83, string 3a 76a 84a 99, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

These functions enable us to convert the basic types.

3a ⟨*lua.mli* 1⟩+≡ ◁2b 3b▷
```
    val float    : float  map
    val int      : int    map
    val bool     : bool   map
    val string   : string map
    val userdata : userdata map
    val unit     : unit   map
```
Defines:
- `bool`, used in chunks 1, 2b, 6b, 8a, 13, 19, 39, 41, 48c, 49b, 73–75, 79b, 80a, 83, 91a, 117, 120, and 127.
- `float`, used in chunks 1, 48, 65, 74, 80a, 95, 104, and 120.
- `int`, used in chunks 1, 2a, 67b, 69, 74, 75a, 80a, 95, 102, 104, 111b, 113, 117, 120, and 127.
- `string`, used in chunks 1, 2, 4, 6b, 8, 10a, 13, 14b, 16, 19–21, 39, 41, 43a, 44b, 48c, 49a, 69–71, 74, 75, 77, 79b, 80a, 82a, 83, 87b, 93–95, 97, 102, 104, 112a, 113, and 127.
- `unit`, used in chunks 1, 2a, 10, 11a, 13, 14b, 16, 21, 22a, 24a, 39, 41, 70, 71, 74, 75a, 80a, 89, 93–95, 99, 102, 111b, 113, and 117.
- `userdata`, used in chunks 1, 74, 95, 99, and 113.

Uses `map` 83.

To convert a value of option type, we represent `None` as `Nil`. Woe betide you if `Nil` is a valid value of your type! We won't see it.

3b ⟨*lua.mli* 1⟩+≡ ◁3a 3c▷
```
    val option : 'a map -> 'a option map
```
Defines:
- `option`, used in chunks 1, 74, 80a, 102, 113, and 127.

Uses `map` 83.

To project with a default value, we provide `default` v `t`, which behaves just as `t` except it projects `Nil` to v.

3c ⟨*lua.mli* 1⟩+≡ ◁3b 3d▷
```
    val default : 'a -> 'a map -> 'a map
```
Defines:
- `default`, used in chunks 86b, 95, and 102.

Uses `map` 83.

To embed a list of values, we produce a table with a binding of the length to the name `n` and bindings of the values to the numbers $1..n$. To project a Lua table down to a list, we first look to see if the table binds the name `n`. If so, we take that to be the number of elements; otherwise we use the table's population. (In the latter case, lists cannot contain `nil`.) This way, users are free to include `n` or not as they choose.

3d ⟨*lua.mli* 1⟩+≡ ◁3c 4a▷
```
    val list    : 'a map -> 'a list map   (* does not project nil *)
    val optlist : 'a map -> 'a list map   (* projects nil to empty list *)
```
Defines:
- `list`, used in chunks 1, 2a, 4, 5, 7a, 10a, 13, 14b, 16, 21a, 39, 41, 42a, 47c, 51, 58, 60d, 63, 70, 71, 74, 75a, 77, 78, 80a, 83, 88–91, 94, 102, 117, and 127.
- `optlist`, never used.

Uses `map` 83 and `nil` 65.

If for some reason a Caml function operates on Lua values, we need an identity pair. We also enable functions that expect tables.

4a ⟨*lua.mli* 1⟩+≡ ◁3d 4b▷

```
    val value  : value map
    val table  : table map
```

Defines:
`table`, used in chunks 1, 2a, 5a, 10a, 13, 39, 43a, 53, 66, 74, 75a, 78a, 81b, 86b, 87b, 91c, 95, 117, and 120.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.
Uses `map` 83.

A common case is to use a table as a record, with string keys and homogenous values.

4b ⟨*lua.mli* 1⟩+≡ ◁4a 4c▷

```
    val record : 'a map -> (string * 'a) list map
```

Defines:
`record`, never used.
Uses `list` 3d 76d 86b 99, `map` 83, and `string` 3a 76a 84a 99.

Another common case is to represent an enumeration type using strings. The string passed to `enum` is the name of the type, which is used in projection errors. The list passed to `enum` must contain *every* value of type `'a`, which must be comparable using `=`. To do otherwise is to risk an assertion failure during embedding.

4c ⟨*lua.mli* 1⟩+≡ ◁4b 5a▷

```
    val enum   : string -> (string * 'a) list -> 'a map
```

Defines:
`enum`, never used.
Uses `list` 3d 76d 86b 99, `map` 83, and `string` 3a 76a 84a 99.

Here is the support for converting functions. First, if one wants a Lua function to be curried (as the Caml functions are), one can simply use `-->`. There's a small gotcha, in that we can't make `-->` right associative. That's OK, as it probably shouldn't be used for curried functions.

For curried functions that should take lists of arguments in Lua, we use `**->`, `func`, and `result`. The idea is this: if we have a Caml function type `t -> u -> v -> w`, we can turn this into a Lua function of three arguments by using the embedding/projection pair produced by

```
pfunc (t **-> u **-> v **-> result w)
```

We recommend defining the abbreviation `v **->> w` ≡ `v **-> result w`.

5a ⟨*lua.mli* 1⟩+≡ ◁4c 5b▷

```
val ( -->  ) : 'a map  -> 'b map  -> ('a -> 'b) map
val ( **-> ) : 'a map  -> 'b mapf -> ('a -> 'b) mapf
val result   : 'a map  -> 'a mapf
val resultvs : value list mapf                  (* functions returning value lists*)
val resultpair:'a map  -> 'b map  -> ('a * 'b)       mapf
val dots_arrow:'a map  -> 'b map  -> ('a list -> 'b) mapf     (* varargs functions *)
val results  : ('a -> value list) -> (value list -> 'a) -> 'a mapf
                                 (* 'a represents multiple results (general case) *)
val func     : 'a mapf -> 'a map                (* function *)
val closure  : 'a mapf -> 'a map                (* function or table+apply method *)
val efunc    : 'a mapf -> 'a -> value           (* efunc f = (closure f).embed *)
```

Defines:
`**->`, used in chunks 91a, 95, 99, 100, 102, 103, 111, 113, and 116b.
`-->`, used in chunks 99 and 100.
`closure`, never used.
`dots_arrow`, used in chunks 103 and 116b.
`efunc`, used in chunks 91a, 95, 99, 102, 111b, and 113.
`func`, used in chunks 1, 74, and 99.
`result`, used in chunks 1, 65, 74, 88, 90, 91a, 95, 99, 103, 111c, and 116b.
`resultpair`, never used.
`results`, used in chunks 63, 64b, and 117.
`resultvs`, used in chunks 95 and 102.
Uses `apply` 10a 21a 46, `list` 3d 76d 86b 99, `map` 83, `mapf` 83, `table` 4a 77a 80a 87a, `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c, and `varargs` 127.

Support for dynamically typed Lua functions Type-based dispatch defines several alternatives for a function and at each call, chooses the right function based on the types of the arguments.

5b ⟨*lua.mli* 1⟩+≡ ◁5a 6a▷

```
type alt                             (* an alternative *)
val alt    : 'a mapf -> 'a -> alt    (* create an alternative *)
val choose : alt list -> value       (* dispatch on type/number of args *)
```

Defines:
`alt`, used in chunk 113.
`choose`, used in chunk 113.
Uses `create` 2a 75a 117 120, `list` 3d 76d 86b 99, `mapf` 83, `number` 84b, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

It's also possible to combine multiple types for a single argument. The idea is just like parsing combinators, and we use notation from Jeroen Fokker's paper *Functional Parsers*. We can use the choice operator `<|>` to combine two maps. To project, map `t <|> t'` projects using `t` if `t` claims to recognize the argument. If `t` does not recognize the argument, the map projects using `t'`. To embed, map `t <|> t'` always embeds using `t'`.

We can use the continuation operator `>>=` to apply a function to a value after projection. To project, the map `t <@ f` applies `f` to the result of projecting with `t`. Because function `f` cannot be inverted, the map `t <@ f` is not capable of embedding. It is therefore useful primarily on the left-hand side of the `<|>` operator.

6a ⟨*lua.mli* 1⟩+≡ ◁5b 6b▷

```
    val ( <|> ) : 'a map -> 'a map -> 'a map
    val ( <@  ) : 'a map -> ('a -> 'b) -> 'b map   (* apply continuation after project *)
  end
```

Defines:
`<@`, never used.
`<|>`, never used.
Uses `apply` 10a 21a 46 and `map` 83.

1.2 User-defined types and state

For the rest of the interface, we can make Lua values by supplying an appropriate `userdata` type.

6b ⟨*lua.mli* 1⟩+≡ ◁6a 7a▷

```
  module type USERDATA = sig
    type 'a t                            (* type parameter will be Lua value *)
    val tname : string  (* name of this type, for projection errors *)
    val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
    val to_string : ('a -> string) -> 'a t -> string
  end
```

Defines:
`eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
`tname`, used in chunks 24a, 25, 28, 84a, and 115.
`to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
Uses `bool` 3a 76a 85a 99, `name` 127, `projection` 2b 75b 83, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

1.3 Parsers

It is possible for users to replace the Lua parser with a parser of their own, but it must read the same kinds of tokens and produce the same kinds of ASTs as the existing parser. Because most users won't do this, we don't document the token and AST types here—you must look in the appropriate source modules.

7a ⟨*lua.mli* 1⟩+≡ ◁6b 7b▷

```
module type AST = Luaast.S
module Parser : sig
  type token = Luaparser.token
  module type S =
    sig
      type chunk
      val chunks : (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> chunk list
    end
  module type MAKER = functor (Ast : AST) -> S with type chunk = Ast.chunk
  module MakeStandard : MAKER
end
```

Defines:
`chunk`, used in chunks 13, 14b, 16, 39, 64b, and 94.
`chunks`, used in chunks 64b and 97.
`token`, used in chunks 16 and 97.
Uses `list` 3d 76d 86b 99.

You should never have to worry about a scanner for Lua—that should be taken care of for you by `dofile` and `dostring` in the basic library.

1.4 Libraries

The module `Lua.Lib` provides library support.

7b ⟨*lua.mli* 1⟩+≡ ◁7a 13▷

```
module Lib : sig
  ⟨library support 8a⟩
end
```

To build a Lua interpreter, one must specify the type of userdata, and one must specify what libraries are to be included. Complexities arise when the library code depends on the type of userdata. For example, the I/O library must be able to project userdata to values of type `in_channel` and `out_channel`, representing open files.

The approved technique occurs in three stages:

1. Combine all the types using `Lualib.Combine.T`n, where n is the number of different types of userdata supported.
2. Use the `COMBINED_TYPE` module returned to inform all the libraries how to get "views" of the types they depend on. This may mean passing one or more views to each library.
3. Combine all the libraries into a single library using `Lualib.Combine.T`m, where m is the number of libraries.

The combined types and libraries can then be used to build an interpreter.

While this scheme is a bit more elaborate than a scheme in which types and libraries are bundled together, it makes it possible for each library to depend on any set of types—essential for complex interpreters.

The types that are combined all match `USERTYPE`.

8a ⟨*library support* 8a⟩≡ (7b) 8b▷

```
module type USERTYPE = sig
  type 'a t                             (* type parameter will be Lua value *)
  val tname : string  (* name of this type, for projection errors *)
  val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  val to_string : ('a -> string) -> 'a t -> string
end
```

Defines:
`eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
`tname`, used in chunks 24a, 25, 28, 84a, and 115.
`to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
Uses `bool` 3a 76a 85a 99, `name` 127, `projection` 2b 75b 83, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

A "combined type" is composed of up to 10 individual types. We provide a "view" of each type, which is essentially the ability to convert between values of the combined type and values of the individual type. We could provide this ability as a pair of type `('a t, 'a combined) Luavalue.ep`, but it is more convenient for the clients to provide it as a function. Given a view `T : TYPEVIEW` and a suitable module `V : Luavalue.S`, clients will have code such as

```
let map = T.makemap V.userdata V.projection
```

8b ⟨*library support* 8a⟩+≡ (7b) ◁8a 9▷

```
module type TYPEVIEW = sig
  type 'a combined
  type 'a t  (* the individual type of which this is a view *)
  val makemap : ('a combined, 'b, 'b) Luavalue.ep -> ('b -> string -> 'a t)
              -> ('a t, 'b, 'b) Luavalue.ep
end
```

Defines:
`makemap`, used in chunks 25, 28, 31, 99, and 113.
Uses `string` 3a 76a 84a 99.

When up to ten types are combined into a single type, the result provides views of all ten. This code would be a lot cleaner with a better calculus of signatures.

9 ⟨*library support* 8a⟩+≡ (7b) ◁8b 10a▷

```
module type COMBINED_CORE = sig
  type 'a also_t
  module type VIEW = TYPEVIEW with type 'a combined = 'a also_t
  module TV1  : VIEW
  module TV2  : VIEW
  module TV3  : VIEW
  module TV4  : VIEW
  module TV5  : VIEW
  module TV6  : VIEW
  module TV7  : VIEW
  module TV8  : VIEW
  module TV9  : VIEW
  module TV10 : VIEW
end
module type COMBINED_VIEWS = sig
  type 'a t
  include COMBINED_CORE with type 'a also_t = 'a t
end
module type COMBINED_TYPE = sig
  include USERTYPE
  include COMBINED_CORE with type 'a also_t = 'a t
end
```

A library module needs zero or more views, plus an *interpreter core*, which enables libraries to use some of the capabilities of the interpreter.

10a ⟨*library support* 8a⟩+≡ (7b) ◁9 10b▷

```
module type CORE = sig
  module V : Luavalue.S
  val error : string -> 'a  (* error fallback *)
  val getglobal : V.state -> V.value -> V.value
  val fallback : string -> V.state -> V.value list -> V.value list
    (* invoke named fallback on given state and arguments *)
  val setfallback : V.state -> string -> V.value -> V.value
    (* sets fallback, returns previous one *)
  val apply : V.value -> V.state -> V.value list -> V.value list

  val register_globals :           (string * V.value) list -> V.state -> unit
    (* registers values as named global variables *)
  val register_module  : string -> (string * V.value) list -> V.state -> unit
    (* register_module t l inserts members of l into global table t,
       creating t if needed *)
end
```

Defines:
`apply`, used in chunks 5a, 6a, 55c, 56a, 78a, 79a, and 89–91.
`error`, used in chunks 42–44, 47, 93, 95, 97, 102, 104, 107, 109, 110, and 113.
`fallback`, used in chunks 42b, 43a, 45–47, 50a, 82b, and 95.
`getglobal`, used in chunks 43a, 52a, 66, and 95.
`register_globals`, used in chunks 95, 103, 111c, and 116b.
`register_module`, used in chunk 100.
`setfallback`, used in chunk 95.
Uses `list` 3d 76d 86b 99, `state` 1 13 39 41 70 71 74 80a 93 112b, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

A library does two things: add values to the initial state of an interpreter and provide code to be executed at startup time. The values are the functions, etc, provided by the library. If the library needs to attach private data to the state of the interpreter, it can either use closures to capture this state (see the I/O library for an example) or it can keep the state in the interpreter's global variables. If per-interpreter state must be shared among two or more libraries, global variables are the only option. The code is a list of strings, and it should be registered using the `at_init` function in the `VALUE` structure. The *only* safe time to call `at_init` is from within an `init` function.

A *bare* library does not use any user types—only the basic ones built into every Lua interpreter. That means it works with any core at all.

10b ⟨*library support* 8a⟩+≡ (7b) ◁10a 11a▷

```
module type BARECODE =
  functor (C : CORE) -> sig
    val init : C.V.state -> unit
  end
```

Defines:
`init`, used in chunks 24, 26, 27, 41, 95, 100, 102, 103, 111, and 116b.
Uses `state` 1 13 39 41 70 71 74 80a 93 112b and `unit` 3a 76a 85a.

A *typeful* library won't work with just any core—it works only with particular cores. In practice, it will work for cores in which the userdata type is the `combined` type of the view on which the typeful library depends.

11a ⟨*library support* 8a⟩+≡ (7b) ◁10b 11b▷
```
module type USERCODE = sig
  type 'a userdata'  (* the userdata' tycon of the core on which lib depends *)
  module M : functor (C : CORE with type 'a V.userdata' = 'a userdata') -> sig
    val init : C.V.state -> unit
  end
end
```
Defines:
`init`, used in chunks 24, 26, 27, 41, 95, 100, 102, 103, 111, and 116b.
Uses `state` 1 13 39 41 70 71 74 80a 93 112b, `unit` 3a 76a 85a, and `userdata'` 80a.

For simplicity, we combine only typeful libraries. This means we occasionally need to extend a bare library to make it typeful. We put the type first, not the library, because the partial application is useful.

11b ⟨*library support* 8a⟩+≡ (7b) ◁11a 11c▷
```
module WithType
  (T : USERTYPE) (L : BARECODE) : USERCODE with type 'a userdata' = 'a T.t
```
Uses `userdata'` 80a.

Herewith the module `Combine`, which contains members for combining up to 10 types and up to 10 libraries.

11c ⟨*library support* 8a⟩+≡ (7b) ◁11b 12▷
```
module Combine : sig
  module T10 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)
             (T5 : USERTYPE) (T6 : USERTYPE) (T7 : USERTYPE) (T8 : USERTYPE)
             (T9 : USERTYPE) (T10 : USERTYPE)
    : COMBINED_TYPE with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t
                    with type 'a TV3.t = 'a T3.t with type 'a TV4.t = 'a T4.t
                    with type 'a TV5.t = 'a T5.t with type 'a TV6.t = 'a T6.t
                    with type 'a TV7.t = 'a T7.t with type 'a TV8.t = 'a T8.t
                    with type 'a TV9.t = 'a T9.t with type 'a TV10.t = 'a T10.t
  ⟨similar specifications for T1 to T9 18⟩
  module C10 (C1 : USERCODE)
    (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C10 : USERCODE with type 'a userdata' = 'a C1.userdata') :
    USERCODE with type 'a userdata' = 'a C1.userdata'
  ⟨similar specifications for C1 to C9 17⟩
end
```
Uses `userdata'` 80a.

If a user needs more than 10 types, it's necessary to lift a nested view. See `main.nw` in the source distribution for an example of lifting.

12 ⟨*library support* 8a⟩+≡ (7b) ◁11c

```
module Lift (T : COMBINED_TYPE) (View : TYPEVIEW with type 'a t = 'a T.t) :
  COMBINED_VIEWS with type 'a t = 'a View.combined
     with type 'a TV1.t = 'a T.TV1.t
     with type 'a TV2.t = 'a T.TV2.t
     with type 'a TV3.t = 'a T.TV3.t
     with type 'a TV4.t = 'a T.TV4.t
     with type 'a TV5.t = 'a T.TV5.t
     with type 'a TV6.t = 'a T.TV6.t
     with type 'a TV7.t = 'a T.TV7.t
     with type 'a TV8.t = 'a T.TV8.t
     with type 'a TV9.t = 'a T.TV9.t
     with type 'a TV10.t = 'a T.TV10.t
```

1.5 Building an interpreter

To build an interpreter, a user must provide a user library to make an interpreter core, then add a basic library to this core. Because the basic library includes `dofile` and `dostring`, one must provide a parser to make this library.

The core interpreter contains a compiler for ASTs and some other stuff that I might feel like documenting one day. The function `pre_mk` provides both an initial state and the startup code from libraries. The startup code is intended to be passed `dostring`.

13 ⟨*lua.mli* 1⟩+≡ ◁7b 14a▷

```
module type EVALUATOR = sig
  module Value : VALUE
  module Ast   : AST with module Value = Value
  type state = Value.state
  type value = Value.value
  exception Error of string
  type compiled = unit -> value list
  val compile : srcdbg:(Luasrcmap.map * bool) -> Ast.chunk list -> state -> compiled
  type startup_code = (string -> unit) -> unit
  val pre_mk  : unit -> state * startup_code (* produce a fresh, initialized state *)
  val error   : string -> 'a    (* error fallback *)

  val getglobal : state -> value -> value
    (* get the named global variable *)
  val fallback : string -> state -> value list -> value list
    (* invoke named fallback on given state and arguments *)
  val with_stack  : Value.srcloc -> state -> ('a -> 'b) -> 'a -> 'b
    (* evaluate function with given srcloc on activation stack *)

  val setfallback : state -> string -> value -> value
    (* sets fallback, returns previous one *)
  val register_globals :           (string * value) list -> state -> unit
    (* registers values as named global variables *)
  val register_module  : string -> (string * value) list -> state -> unit
    (* register_module t l inserts members of l into global table t,
       creating t if needed *)
end

module MakeEval
    (T : Lib.USERTYPE)
    (L : Lib.USERCODE with type 'a userdata' = 'a T.t)
    : EVALUATOR with type 'a Value.userdata' = 'a T.t
```

Defines:
- `compile`, used in chunks 65 and 97.
- `compiled`, never used.
- `error`, used in chunks 42–44, 47, 93, 95, 97, 102, 104, 107, 109, 110, and 113.
- `fallback`, used in chunks 42b, 43a, 45–47, 50a, 82b, and 95.
- `getglobal`, used in chunks 43a, 52a, 66, and 95.
- `pre_mk`, used in chunks 41 and 95.
- `register_globals`, used in chunks 95, 103, 111c, and 116b.
- `register_module`, used in chunk 100.
- `setfallback`, used in chunk 95.
- `startup_code`, never used.
- `state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
- `value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

`with_stack`, used in chunks 44c, 46, and 97.
Uses `activation` 80a, `bool` 3a 76a 85a 99, `chunk` 7a 64a 127, `list` 3d 76d 86b 99, `map` 83, `srcloc` 1 1 74 74 80a 81a, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, `unit` 3a 76a 85a, and `userdata'` 80a.

Because building an interpreter requires a library and a type, and because not every client will want to create a special library, we provide empty libraries and types.

14a ⟨*lua.mli* 1⟩+≡ ◁13 14b▷
```
module Empty : sig
  module Type : Lib.USERTYPE
  module Library : Lib.USERCODE
end
```

We also provide three standard libraries: `Luastrlib : Lib.BARECODE`, `Luamathlib : Lib.BARECODE`, and `Luaiolib : Lib.FULL`. We don't mention these here because we want you to be able to leave them out if you won't use them.

Finally, to make a full-blown interpreter, you need to supply a parser. We then add the basic library. With the basic library in place, you get the function `mk`, which makes a fresh state, including running any startup code.

14b ⟨*lua.mli* 1⟩+≡ ◁14a 14c▷
```
module type INTERP = sig
  include EVALUATOR
  module Parser : Luaparser.S with type chunk = Ast.chunk
  val do_lexbuf : sourcename:string -> state -> Lexing.lexbuf -> value list
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
  val mk        : unit -> state
end
module MakeInterp (MakeParser : Parser.MAKER) (I : EVALUATOR)
    : INTERP with module Value = I.Value
```
Defines:
`do_lexbuf`, never used.
`dofile`, used in chunk 95.
`dostring`, used in chunk 95.
`mk`, used in chunks 65, 95, and 97.
Uses `chunk` 7a 64a 127, `list` 3d 76d 86b 99, `state` 1 13 39 41 70 71 74 80a 93 112b, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

Once we've made an interpreter, if we like, we can run it using the command-line arguments. If there are no arguments, it runs interactively from standard input.

14c ⟨*lua.mli* 1⟩+≡ ◁14b
```
module Run (I : INTERP) : sig end  (* runs interpreter on Sys.argv *)
```

Here is an example of how to build an interpreter that uses the three standard libraries:

```
module T = Lua.Lib.Combine.T1 (Luaiolib.T)
module X = Lua.Lib.WithType (T)
module L = Lua.Lib.Combine.C3 (Luaiolib.Make (T.T1))
                              (X(Luastrlib.M)) (X(Luamathlib.M))
module I = Lua.MakeInterp (Lua.Parser.MakeStandard) (Lua.MakeEval (T) (L))
module Go = Lua.Run(I)
```

Here is another example, this time of an interpreter that uses the I/O library, an "optimization backplane," and support for graph coloring. The `Luacolorgraph` library shares the backplane and graph-coloring types.

```
module T = Lua.Lib.Combine.T3 (Luaiolib.T) (Luabackplane.T) (Luacolorgraph.T)
module X = Lua.Lib.WithType (T)
module L = Lua.Lib.Combine.C5
             (Luaiolib.Make (T.TV1))
             (X(Luastrlib.M))
             (X(Luamathlib.M))
             (Luabackplane.Make(T.TV2))
             (Luacolorgraph.Make(T.TV2)(T.TV3))
module I = Lua.MakeInterp (Lua.Parser.MakeStandard) (Lua.MakeEval (T) (L))
module Go = Lua.Run(I)
```

2 Implementation

16 ⟨*lua.ml* 16⟩≡
```
module type VALUE = Luavalue.S
module type USERDATA = Luavalue.USERDATA

module Lib = Lualib
module Parser = Luaparser
module type AST = Luaast.S

module type EVALUATOR = Luainterp.S
module type INTERP = sig
  include EVALUATOR
  module Parser : Luaparser.S with type chunk = Ast.chunk
  val do_lexbuf : sourcename:string -> state -> Lexing.lexbuf -> value list
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
  val mk        : unit -> state
end
module Run (I : INTERP) = Luarun.Make (I)
module MakeEval = Luainterp.Make
module MakeInterp = Luabaselib.Add

module Empty = Lualib.Empty

let scanner map buf = Luascanner.token buf map
```
Defines:
`do_lexbuf`, never used.
`dofile`, used in chunk 95.
`dostring`, used in chunk 95.
`mk`, used in chunks 65, 95, and 97.
`scanner`, never used.

Uses `chunk` 7a 64a 127, `list` 3d 76d 86b 99, `map` 83, `state` 1 13 39 41 70 71 74 80a 93 112b, `string` 3a 76a 84a 99, `token` 7a, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

17 ⟨*similar specifications for* `C1` *to* `C9` 17⟩≡ (11c 22c) 34▷

```
module C1 (C1 : USERCODE)
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C2 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C3 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C4 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C5 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C6 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C7 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C8 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C9 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
```

```
      (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
    : USERCODE with type 'a userdata' = 'a C1.userdata'
```

Uses `userdata'` 80a.

18 ⟨*similar specifications for* `T1` *to* `T9` 18⟩≡ (11c 22c) 37▷

```
  module T1 (T1 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t
  module T2 (T1 : USERTYPE) (T2 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t
  module T3 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t
  module T4 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T5 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T6 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE)  :
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T7 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T8 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T9 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
```

3 Modules for building Lua libraries

To build a Lua interpreter, one must specify the type of userdata, and one must specify what libraries are to be included. Complexities arise when the library code depends on the type of userdata. For example, the I/O library must be able to project userdata to values of type `in_channel` and `out_channel`, representing open files.

The approved technique occurs in three stages:

1. Combine all the types using `Lualib.Combine.T`n, where n is the number of different types of userdata supported.
2. Use the `COMBINED_TYPE` module returned to inform all the libraries how to get "views" of the types they depend on. This may mean passing one or more views to each library.
3. Combine all the libraries into a single library using `Lualib.Combine.T`m, where m is the number of libraries.

The combined types and libraries can then be used to build an interpreter.

While this scheme is a bit more elaborate than a scheme in which types and libraries are bundled together, it makes it possible for each library to depend on any set of types—essential for complex interpreters.

The types that are combined all match `USERTYPE`.

19 ⟨*signatures* 19⟩≡ (22b 23c 40 41 79 126) 20a▷

```
module type USERTYPE = sig
  type 'a t                            (* type parameter will be Lua value *)
  val tname : string  (* name of this type, for projection errors *)
  val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  val to_string : ('a -> string) -> 'a t -> string
end
```

Defines:
`eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
`tname`, used in chunks 24a, 25, 28, 84a, and 115.
`to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
Uses `bool` 3a 76a 85a 99, `name` 127, `projection` 2b 75b 83, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

A “combined type” is composed of up to 10 individual types. We provide a “view” of each type, which is essentially the ability to convert between values of the combined type and values of the individual type. We could provide this ability as a pair of type `('a t, 'a combined) Luavalue.ep`, but it is more convenient for the clients to provide it as a function. Given a view `T : TYPEVIEW` and a suitable module `V : Luavalue.S`, clients will have code such as

```
let map = T.makemap V.userdata V.projection
```

20a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁19 20b▷

```
module type TYPEVIEW = sig
  type 'a combined
  type 'a t  (* the individual type of which this is a view *)
  val makemap : ('a combined, 'b, 'b) Luavalue.ep -> ('b -> string -> 'a t)
                -> ('a t, 'b, 'b) Luavalue.ep
end
```

Defines:
`makemap`, used in chunks 25, 28, 31, 99, and 113.
Uses `string` 3a 76a 84a 99.

When up to ten types are combined into a single type, the result provides views of all ten:

20b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁20a 21a▷

```
module type COMBINED_CORE = sig
  type 'a also_t
  module type VIEW = TYPEVIEW with type 'a combined = 'a also_t
  module TV1  : VIEW
  module TV2  : VIEW
  module TV3  : VIEW
  module TV4  : VIEW
  module TV5  : VIEW
  module TV6  : VIEW
  module TV7  : VIEW
  module TV8  : VIEW
  module TV9  : VIEW
  module TV10 : VIEW
end
module type COMBINED_VIEWS = sig
  type 'a t
  include COMBINED_CORE with type 'a also_t = 'a t
end
module type COMBINED_TYPE = sig
  include USERTYPE
  include COMBINED_CORE with type 'a also_t = 'a t
end
```

A library module needs zero or more views, plus an *interpreter core*, which enables libraries to use some of the capabilities of the interpreter.

21a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁20b 21b▷

```
module type CORE = sig
  module V : Luavalue.S
  val error : string -> 'a  (* error fallback *)
  val getglobal : V.state -> V.value -> V.value
  val fallback : string -> V.state -> V.value list -> V.value list
  val setfallback : V.state -> string -> V.value -> V.value
    (* sets fallback, returns previous one *)
  val apply : V.value -> V.state -> V.value list -> V.value list
  val register_globals :           (string * V.value) list -> V.state -> unit
  val register_module  : string -> (string * V.value) list -> V.state -> unit
end
```

Defines:
`apply`, used in chunks 5a, 6a, 55c, 56a, 78a, 79a, and 89–91.
`error`, used in chunks 42–44, 47, 93, 95, 97, 102, 104, 107, 109, 110, and 113.
`fallback`, used in chunks 42b, 43a, 45–47, 50a, 82b, and 95.
`getglobal`, used in chunks 43a, 52a, 66, and 95.
`register_globals`, used in chunks 95, 103, 111c, and 116b.
`register_module`, used in chunk 100.
`setfallback`, used in chunk 95.
Uses `list` 3d 76d 86b 99, `state` 1 13 39 41 70 71 74 80a 93 112b, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

All a library does is add values to the initial state of an interpreter. These values are the functions, etc, provided by the library. If the library needs to attach private data to the state of the interpreter, it can either use closures to capture this state (see the I/O library for an example) or it can keep the state in the interpreter's global variables. If per-interpreter state must be shared among two or more libraries, global variables are the only option.

A *bare* library does not use any user types—only the basic ones built into every Lua interpreter. That means it works with any core at all.

21b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁21a 22a▷

```
module type BARECODE =
  functor (C : CORE) -> sig
    val init : C.V.state -> unit
  end
```

Defines:
`init`, used in chunks 24, 26, 27, 41, 95, 100, 102, 103, 111, and 116b.
Uses `state` 1 13 39 41 70 71 74 80a 93 112b and `unit` 3a 76a 85a.

A *typeful* library won't work with just any core—it works only with particular cores. In practice, it will work for cores in which the userdata type is the `combined` type of the view on which the typeful library depends.

22a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁21b 39▷
```
module type USERCODE = sig
  type 'a userdata'  (* the userdata' tycon of the core on which lib depends *)
  module M : functor (C : CORE with type 'a V.userdata' = 'a userdata') -> sig
    val init : C.V.state -> unit
  end
end
```
Defines:
`init`, used in chunks 24, 26, 27, 41, 95, 100, 102, 103, 111, and 116b.
Uses `state` 1 13 39 41 70 71 74 80a 93 112b, `unit` 3a 76a 85a, and `userdata'` 80a.

For simplicity, we combine only typeful libraries. This means we occasionally need to extend a bare library to make it typeful. We put the type first, not the library, because the partial application is useful

22b ⟨*lualib.mli* 22b⟩≡ 22c▷
```
⟨signatures 19⟩
module WithType (T : USERTYPE) (L : BARECODE) : USERCODE with type 'a userdata' = 'a T.t
```
Uses `userdata'` 80a.

Herewith the module `Combine`, which contains members for combining up to 10 types and up to 10 libraries.

22c ⟨*lualib.mli* 22b⟩+≡ ◁22b 23a▷
```
module Combine : sig
  module T10 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)
             (T5 : USERTYPE) (T6 : USERTYPE) (T7 : USERTYPE) (T8 : USERTYPE)
             (T9 : USERTYPE) (T10 : USERTYPE)
    : COMBINED_TYPE with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t
                    with type 'a TV3.t = 'a T3.t with type 'a TV4.t = 'a T4.t
                    with type 'a TV5.t = 'a T5.t with type 'a TV6.t = 'a T6.t
                    with type 'a TV7.t = 'a T7.t with type 'a TV8.t = 'a T8.t
                    with type 'a TV9.t = 'a T9.t with type 'a TV10.t = 'a T10.t
  ⟨similar specifications for T1 to T9 18⟩
  module C10 (C1 : USERCODE)
    (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
    (C10: USERCODE with type 'a userdata' = 'a C1.userdata') :
    USERCODE with type 'a userdata' = 'a C1.userdata'
  ⟨similar specifications for C1 to C9 17⟩
end
```
Uses `userdata'` 80a.

If a user needs more than 10 types, it's necessary to lift a nested view.

23a ⟨*lualib.mli* 22b⟩+≡ ◁22c 23b▷

```
module Lift (T : COMBINED_TYPE) (View : TYPEVIEW with type 'a t = 'a T.t) :
  COMBINED_VIEWS with type 'a t = 'a View.combined
     and type 'a TV1.t = 'a T.TV1.t
     and type 'a TV2.t = 'a T.TV2.t
     and type 'a TV3.t = 'a T.TV3.t
     and type 'a TV4.t = 'a T.TV4.t
     and type 'a TV5.t = 'a T.TV5.t
     and type 'a TV6.t = 'a T.TV6.t
     and type 'a TV7.t = 'a T.TV7.t
     and type 'a TV8.t = 'a T.TV8.t
     and type 'a TV9.t = 'a T.TV9.t
     and type 'a TV10.t = 'a T.TV10.t
```

Finally, this is a convenient place to export empty libraries and types, for those interpreters that won't use libraries or types.

23b ⟨*lualib.mli* 22b⟩+≡ ◁23a

```
module Empty : sig
  module Type : COMBINED_TYPE
  module Library : USERCODE with type 'a userdata' = 'a Type.t
end
```

Uses `userdata'` 80a.

4 Implementation

If you understand the interfaces, there's only minor interest here. We actually define combining modules for only the case $n = 10$, then define some `Unused` types and libraries to use as placeholders for smaller values of n. The rest is bookkeeping.

23c ⟨*lualib.ml* 23c⟩≡ 24a▷

⟨*signatures* 19⟩

24a ⟨*lualib.ml* 23c⟩+≡ ◁23c 24b▷
```
  module Unused = struct
    module Type : USERTYPE = struct
      type 'a t = unit
      let tname = "unused type"
      let eq _ x y = true
      let to_string _ _ = "<this can't happen -- value of unused type>"
    end (* Type *)

    module Bare =
        functor (C : CORE) -> struct
          let init g = ()
        end (*Unused.Bare*)

    module Typeful (L : USERCODE) =
      struct
        type 'a userdata' = 'a L.userdata'
        module M (C : CORE with type 'a V.userdata' = 'a userdata') = struct
          let init g = ()
        end (*M*)
      end (*Unused.Typeful*)
  end

  module Combine = struct
    ⟨Combine contents 25⟩
  end
```
Uses `eq` 1 6b 8a 19 74 79b 80b, `init` 10b 11a 21b 22a, `tname` 6b 8a 19 79b, `to_string` 1 6b 8a 19 74 79b 81b, `unit` 3a 76a 85a, `userdata'` 80a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

24b ⟨*lualib.ml* 23c⟩+≡ ◁24a 26▷
```
  module WithType (T : USERTYPE) (L : BARECODE) : USERCODE with type 'a userdata' = 'a T.t
  =
      struct
        type 'a userdata' = 'a T.t
        module M (C : CORE with type 'a V.userdata' = 'a userdata') = struct
          module M' = L (C)
          let init g = M'.init g
        end (*M*)
      end (*WithType*)
```
Uses `init` 10b 11a 21b 22a and `userdata'` 80a.

25 ⟨Combine *contents* 25⟩≡ (24a) 27 ▷

```
    module T10 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)
               (T5 : USERTYPE) (T6 : USERTYPE) (T7 : USERTYPE) (T8 : USERTYPE)
               (T9 : USERTYPE) (T10 : USERTYPE)
     : COMBINED_TYPE with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t
                     with type 'a TV3.t = 'a T3.t with type 'a TV4.t = 'a T4.t
                     with type 'a TV5.t = 'a T5.t with type 'a TV6.t = 'a T6.t
                     with type 'a TV7.t = 'a T7.t with type 'a TV8.t = 'a T8.t
                     with type 'a TV9.t = 'a T9.t with type 'a TV10.t = 'a T10.t =
    struct
      type 'a t =
        | T1 of 'a T1.t
        | T2 of 'a T2.t
        | T3 of 'a T3.t
        | T4 of 'a T4.t
        | T5 of 'a T5.t
        | T6 of 'a T6.t
        | T7 of 'a T7.t
        | T8 of 'a T8.t
        | T9 of 'a T9.t
        | T10 of 'a T10.t
      type 'a also_t = 'a t
      let allnames = [T1.tname; T2.tname; T3.tname; T4.tname; T5.tname;
                      T6.tname; T7.tname; T8.tname; T9.tname; T10.tname]
      let tname = String.concat " or " (List.filter ((<>) Unused.Type.tname) allnames)
      let tname = match tname with "" -> Unused.Type.tname | n -> n

      let eq eqvs x y = match x, y with
      | T1 x, T1 y -> T1.eq eqvs x y
      | T2 x, T2 y -> T2.eq eqvs x y
      | T3 x, T3 y -> T3.eq eqvs x y
      | T4 x, T4 y -> T4.eq eqvs x y
      | T5 x, T5 y -> T5.eq eqvs x y
      | T6 x, T6 y -> T6.eq eqvs x y
      | T7 x, T7 y -> T7.eq eqvs x y
      | T8 x, T8 y -> T8.eq eqvs x y
      | T9 x, T9 y -> T9.eq eqvs x y
      | T10 x, T10 y -> T10.eq eqvs x y
      | _, _ -> false

      let to_string vs x = match x with
      | T1 x -> T1.to_string vs x
      | T2 x -> T2.to_string vs x
      | T3 x -> T3.to_string vs x
      | T4 x -> T4.to_string vs x
      | T5 x -> T5.to_string vs x
      | T6 x -> T6.to_string vs x
      | T7 x -> T7.to_string vs x
      | T8 x -> T8.to_string vs x
      | T9 x -> T9.to_string vs x
      | T10 x -> T10.to_string vs x

    module type VIEW = TYPEVIEW with type 'a combined = 'a t
    module V = Luavalue
    module TV1 = struct
      type 'a combined = 'a also_t
      type 'a t = 'a T1.t
      let makemap (upper : ('a combined, 'b, 'b) V.ep) fail =
        { V.embed   = (fun x -> upper.V.embed (T1 x))
        ; V.project = (fun x -> match upper.V.project x with
                       | T1 x -> x
                       | _ -> fail x T1.tname)
        ; V.is      = (fun x -> upper.V.is x &&
                                match upper.V.project x with T1 x -> true | _ -> false)
        }
    end
    ⟨nested T2..T10 in Combine.T10 28⟩
    end (* Combine.T10 *)
    ⟨definitions of Combine.T1 through Combine.T9 38⟩
```

Uses `concat` 49a, `eq` 1 6b 8a 19 74 79b 80b, `makemap` 8b 20a, `tname` 6b 8a 19 79b,
`to_string` 1 6b 8a 19 74 79b 81b, and `upper` 109.

26 ⟨*lualib.ml* 23c⟩+≡ ◁24b 31▷

```
  module Empty = struct
    module Type = Combine.T1 (Unused.Type)
    module Library =
      struct
        type 'a userdata' = 'a Type.t
        module M (C : CORE with type 'a V.userdata' = 'a userdata') = struct
          let init g = ()
        end (*M*)
      end (*Empty.Library*)
  end
```

Uses `init` 10b 11a 21b 22a and `userdata'` 80a.

27 ⟨Combine *contents* 25⟩+≡ (24a) ◁25 36▷
```
module C10 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C10 : USERCODE with type 'a userdata' = 'a C1.userdata') :
  USERCODE with type 'a userdata' = 'a C1.userdata'   =
struct
  type 'a userdata' = 'a C1.userdata'
  module M  (C : CORE with type 'a V.userdata' = 'a userdata') = struct
    module M1 = C1.M(C)
    module M2 = C2.M(C)
    module M3 = C3.M(C)
    module M4 = C4.M(C)
    module M5 = C5.M(C)
    module M6 = C6.M(C)
    module M7 = C7.M(C)
    module M8 = C8.M(C)
    module M9 = C9.M(C)
    module M10 = C10.M(C)
    let init g =
      begin
        M1.init  g;
        M2.init  g;
        M3.init  g;
        M4.init  g;
        M5.init  g;
        M6.init  g;
        M7.init  g;
        M8.init  g;
        M9.init  g;
        M10.init g
      end
  end (*Combine.C10.M*)
end (* Combine.C10*)
```
Uses `init` 10b 11a 21b 22a and `userdata'` 80a.

28 ⟨*nested* T2..T10 *in* Combine.T10 28⟩≡ (25)
```
module TV2 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T2.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T2 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T2 x -> x
                  | _ -> fail x T2.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                           match upper.V.project x with T2 x -> true | _ -> false)
    }
end

module TV3 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T3.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T3 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T3 x -> x
                  | _ -> fail x T3.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                           match upper.V.project x with T3 x -> true | _ -> false)
    }
end

module TV4 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T4.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T4 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T4 x -> x
                  | _ -> fail x T4.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                           match upper.V.project x with T4 x -> true | _ -> false)
    }
end

module TV5 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T5.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T5 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T5 x -> x
                  | _ -> fail x T5.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                           match upper.V.project x with T5 x -> true | _ -> false)
    }
end

module TV6 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T6.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T6 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T6 x -> x
                  | _ -> fail x T6.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                            match upper.V.project x with T6 x -> true | _ -> false)
    }
end

module TV7 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T7.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T7 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T7 x -> x
                  | _ -> fail x T7.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                            match upper.V.project x with T7 x -> true | _ -> false)
    }
end

module TV8 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T8.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T8 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T8 x -> x
                  | _ -> fail x T8.tname)
    ; V.is      = (fun x -> upper.V.is x &&
                            match upper.V.project x with T8 x -> true | _ -> false)
    }
end

module TV9 = struct
  type 'a combined = 'a also_t
  type 'a t = 'a T9.t
  let makemap upper fail =
    { V.embed   = (fun x -> upper.V.embed (T9 x))
    ; V.project = (fun x -> match upper.V.project x with
                  | T9 x -> x
                  | _ -> fail x T9.tname)
```

```
      ; V.is      = (fun x -> upper.V.is x &&
                              match upper.V.project x with T9 x -> true | _ -> false)
      }
    end

    module TV10 = struct
      type 'a combined = 'a also_t
      type 'a t = 'a T10.t
      let makemap upper fail =
        { V.embed   = (fun x -> upper.V.embed (T10 x))
        ; V.project = (fun x -> match upper.V.project x with
                        | T10 x -> x
                        | _ -> fail x T10.tname)
        ; V.is      = (fun x -> upper.V.is x &&
                              match upper.V.project x with T10 x -> true | _ -> false)
        }
    end
```

Uses `makemap` 8b 20a, `tname` 6b 8a 19 79b, and `upper` 109.

31 ⟨*lualib.ml* 23c⟩+≡ ◁26

```
module Lift (T : COMBINED_TYPE) (View : TYPEVIEW with type 'a t = 'a T.t) :
  COMBINED_VIEWS with type 'a t = 'a View.combined
     with type 'a TV1.t = 'a T.TV1.t
     with type 'a TV2.t = 'a T.TV2.t
     with type 'a TV3.t = 'a T.TV3.t
     with type 'a TV4.t = 'a T.TV4.t
     with type 'a TV5.t = 'a T.TV5.t
     with type 'a TV6.t = 'a T.TV6.t
     with type 'a TV7.t = 'a T.TV7.t
     with type 'a TV8.t = 'a T.TV8.t
     with type 'a TV9.t = 'a T.TV9.t
     with type 'a TV10.t = 'a T.TV10.t =
  struct
    type 'a t = 'a View.combined
    type 'a also_t = 'a t
    module type VIEW = TYPEVIEW with type 'a combined = 'a also_t
    module Lift (T : T.VIEW) : VIEW with type 'a t = 'a T.t = struct
      type 'a combined = 'a also_t
      type 'a t = 'a T.t
      let makemap upper fail =
        let fail' x y = ignore(fail x y); assert false in
        let upper = View.makemap upper fail' in
        T.makemap upper fail
    end

    module TV1  = Lift(T.TV1)
    module TV2  = Lift(T.TV2)
    module TV3  = Lift(T.TV3)
    module TV4  = Lift(T.TV4)
    module TV5  = Lift(T.TV5)
    module TV6  = Lift(T.TV6)
    module TV7  = Lift(T.TV7)
    module TV8  = Lift(T.TV8)
    module TV9  = Lift(T.TV9)
    module TV10 = Lift(T.TV10)
  end (* Lift *)
```

Uses `makemap` 8b 20a and `upper` 109.

This Icon code was used to generate C1 through C9

32 ⟨*lspecl.icn* 32⟩≡

```
procedure main()
  write("<<similar specifications for [[C1]] to [[C9]]>>=")
  every spec (1 to 9)
  write("@")
  write("<<[[Combine]] contents>>=")
  every def (1 to 9)
end

procedure spec(k, succ)
  write("  module L", k, " (C1 : USERCODE)")
  every write ("    (L", 2 to k, " : USERCODE with type 'a userdata' = 'a C1.userdata')")
  write("  : USERCODE with type 'a userdata' = 'a C1.userdata'", \succ | "")
end

procedure def(k)
  spec(k, "   = ")
  writes("    C10")
  every writes(" (L", 1 to k, ")")
  every k+1 to 10 do writes(" (Unused.Typeful(C1))")
  write()
end
```

Uses `<<` 89 and `userdata'` 80a.

This Icon code generated T1 through T9.

33 ⟨*tspecl.icn* 33⟩≡
```
procedure main()
  write("<<similar specifications for [[T1]] to [[T9]]>>=")
  every spec (1 to 9)
  write("@")
  write("<<definitions of [[Combine.T1]] through [[Combine.T9]]>>=")
  every def (1 to 9)
end

procedure spec(k, succ)
  writes("module T", k)
  every writes(" (T", 1 to k, " : USERTYPE)")
  write("  : COMBINED_TYPE")
  every i := 1 to k do
      writes(" with type 'a TV", i, ".t = 'a T", i, ".t")
  write(\succ | "")
end

procedure def(k)
  spec(k, "   = ")
  writes("    T10")
  every writes(" (T", 1 to k, ")")
  every k+1 to 10 do writes(" (Unused.Type)")
  write()
end
```
Uses << 89.

34 ⟨*similar specifications for* C1 *to* C9 17⟩+≡ (11c 22c) ◁17

```
module C1 (C1 : USERCODE)
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C2 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C3 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C4 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C5 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C6 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C7 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C8 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'
module C9 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
```

```
      (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
    : USERCODE with type 'a userdata' = 'a C1.userdata'
```

Uses `userdata'` 80a.

36 ⟨Combine *contents* 25⟩+≡ (24a) ◁27

```
module C1 (C1 : USERCODE)
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typ
module C2 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unuse
module C3 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (C3) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (U
module C4 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (C3) (C4) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C
module C5 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (C3) (C4) (C5) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Type
module C6 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (C3) (C4) (C5) (C6) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused
module C7 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
: USERCODE with type 'a userdata' = 'a C1.userdata'   =
  C10 (C1) (C2) (C3) (C4) (C5) (C6) (C7) (Unused.Typeful(C1)) (Unused.Typeful(C1)) (Unused.Typeful(C1))
module C8 (C1 : USERCODE)
  (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
  (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
```

```
      (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
    : USERCODE with type 'a userdata' = 'a C1.userdata'   =
      C10 (C1) (C2) (C3) (C4) (C5) (C6) (C7) (C8) (Unused.Typeful(C1)) (Unused.Typeful(C1))
    module C9 (C1 : USERCODE)
      (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C3 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C4 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C5 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C6 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C7 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C8 : USERCODE with type 'a userdata' = 'a C1.userdata')
      (C9 : USERCODE with type 'a userdata' = 'a C1.userdata')
    : USERCODE with type 'a userdata' = 'a C1.userdata'   =
      C10 (C1) (C2) (C3) (C4) (C5) (C6) (C7) (C8) (C9) (Unused.Typeful(C1))
```

Uses `userdata'` 80a.

37 ⟨*similar specifications for* `T1` *to* `T9` 18⟩+≡ (11c 22c) ◁18

```
  module T1 (T1 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t
  module T2 (T1 : USERTYPE) (T2 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t
  module T3 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t
  module T4 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T5 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T6 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE)  :
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T7 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T8 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
  module T9 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
```

38 ⟨*definitions of* `Combine.T1` *through* `Combine.T9` 38⟩≡ (25)

```
  module T1 (T1 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t   =
      T10 (T1) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Typ
  module T2 (T1 : USERTYPE) (T2 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t   =
      T10 (T1) (T2) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unuse
  module T3 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t   =
      T10 (T1) (T2) (T3) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (U
  module T4 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Ty
  module T5 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE)  : COMBINED_TYPE
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (T5) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type)
  module T6 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE)  :
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (T5) (T6) (Unused.Type) (Unused.Type) (Unused.Type) (Unused.Type)
  module T7 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (T5) (T6) (T7) (Unused.Type) (Unused.Type) (Unused.Type)
  module T8 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (T5) (T6) (T7) (T8) (Unused.Type) (Unused.Type)
  module T9 (T1 : USERTYPE) (T2 : USERTYPE) (T3 : USERTYPE) (T4 : USERTYPE) (T5 : USERTYPE) (T6 : USERTYPE) (T
   with type 'a TV1.t = 'a T1.t with type 'a TV2.t = 'a T2.t with type 'a TV3.t = 'a T3.t with type 'a TV4.t =
      T10 (T1) (T2) (T3) (T4) (T5) (T6) (T7) (T8) (T9) (Unused.Type)
```

5 Core interpreter for Lua statements and expressions

This signature is intended as a convenient port of contact for an interpreter client. Much of what's here is actually brought in from the `Value` submodule. The main things every external client will want are `compile`, to compile Lua code, and `state`, to get an initial state. Other stuff, such as `getglobal`, `error`, etc., are primarily for use by libraries.

39 ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁22a 73▷

```
module type S = sig
  module Value : Luavalue.S
  module Ast   : Luaast.S with module Value = Value
  type state = Value.state
  type value = Value.value
  exception Error of string
  type compiled = unit -> value list
  val compile : srcdbg:(Luasrcmap.map * bool) -> Ast.chunk list -> state -> compiled
  type startup_code = (string -> unit) -> unit
  val pre_mk  : unit -> state * startup_code (* produce a fresh, initialized state *)
  val error   : string -> 'a   (* error fallback *)

  val getglobal : state -> value -> value
    (* get the named global variable *)
  val fallback  : string -> state -> value list -> value list
    (* invoke named fallback on given state and arguments *)
  val with_stack  : Value.srcloc -> state -> ('a -> 'b) -> 'a -> 'b
    (* evaluates function with given srcloc on activation stack *)

  val setfallback : state -> string -> value -> value
    (* sets fallback, returns previous one *)
  val register_globals :           (string * value) list -> state -> unit
    (* registers values as named global variables *)
  val register_module  : string -> (string * value) list -> state -> unit
    (* register_module t l inserts members of l into global table t,
       creating t if needed *)
end
```

Defines:
`compile`, used in chunks 65 and 97.
`compiled`, never used.
`error`, used in chunks 42–44, 47, 93, 95, 97, 102, 104, 107, 109, 110, and 113.
`fallback`, used in chunks 42b, 43a, 45–47, 50a, 82b, and 95.
`getglobal`, used in chunks 43a, 52a, 66, and 95.
`pre_mk`, used in chunks 41 and 95.
`register_globals`, used in chunks 95, 103, 111c, and 116b.
`register_module`, used in chunk 100.
`setfallback`, used in chunk 95.
`startup_code`, never used.
`state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

`with_stack`, used in chunks 44c, 46, and 97.
Uses `activation` 80a, `bool` 3a 76a 85a 99, `chunk` 7a 64a 127, `list` 3d 76d 86b 99, `map` 83, `srcloc` 1 1 74 74 80a 81a, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, and `unit` 3a 76a 85a.

To build an interpreter, one supplies module `T`, which gives the type of userdata, and module `L`, which gives a (combined) library.

40 ⟨*luainterp.mli* 40⟩≡

```
⟨signatures 19⟩
module Make (T : Luavalue.USERDATA)
            (L : Lualib.USERCODE with type 'a userdata' = 'a T.t) :
    S with type 'a Value.userdata'  = 'a T.t
```

Uses `userdata'` 80a.

5.1 Caveats

Builtin `type` doesn't return a tag.

5.2 Implementation

Much of the implementation is strung together from elsewhere. Once we have a userdata type, we can make a value module. Then, once we know what a value is, we can make an AST (we need a value to represent the a literal node). The new stuff is embedded in module `I`. Owing to some careless management of the name space, this module forms both the interpreter core and also some of the top-level stuff. It shows up in both places because we need the interpreter core before we can build the library (`L.M(Core)`), and we need the library before we can build the `state` function, because the initialization has to include the initialization of the library.

41 ⟨*luainterp.ml* 41⟩≡

```
exception Bogus
⟨signatures 19⟩
type answer = unit * string * bool
module Make  (T : Luavalue.USERDATA)
             (L : Lualib.USERCODE with type 'a userdata' = 'a T.t) :
    S with type 'a Value.userdata'  = 'a T.t = struct
  module Value = Luavalue.Make(T)
  module Ast   = Luaast.Make (Value)
  module I = struct
    type state = Value.state
    type value = Value.value
    (* begin with internal abbreviations *)
    module A = Ast
    module V = Ast.Value
    ⟨interp utility functions 47c⟩
    ⟨interp with_stack 62b⟩
    ⟨interp toplevel 42a⟩
    type compiled = unit -> value list
    ⟨compiler 51⟩
    ⟨interp fallbacks 43a⟩
    ⟨interp registration 66⟩
    ⟨interp tests 65⟩
  end
  module Core = struct
    include I
  end
  module L' = L.M(Core)
  include I
  type startup_code = (string -> unit) -> unit
  let pre_mk () = (* raw state + fallbacks + library initialization + startup code*)
    let g = V.state() in
    begin
      add_fallbacks g;
      L'.init g;
      g, V.initcode g
    end
  end (* Make *)
```

Defines:
  `answer`, used in chunks 46, 62b, 69, and 97.
  `compiled`, never used.
  `startup_code`, never used.
  `state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
  `value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

Uses `add_fallbacks` 43b, `bool` 3a 76a 85a 99, `init` 10b 11a 21b 22a, `initcode` 1 74 93, `list` 3d 76d 86b 99, `pre_mk` 13 39, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, and `userdata'` 80a.

5.2.1 Fallbacks

The error fallback is of primary interest because it's what we call when things go wrong. The error fallback is *always* supposed to raise an exception; if it does return a value something is horribly wrong. We get to the error fallback from Caml by calling the `error` function; if it doesn't raise an exception, we complain bitterly.

42a ⟨*interp toplevel* 42a⟩≡ (41) 44b ▷

```
  exception Errorfallback of V.value list
  let error s = raise (Errorfallback [V.String s])
```

Defines:
  `error`, used in chunks 42–44, 47, 93, 95, 97, 102, 104, 107, 109, 110, and 113.

Uses `list` 3d 76d 86b 99 and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

42b ⟨*definition of* `catcherrorfallback` 42b⟩≡ (44c)

```
  let catcherrorfallback g vs =
    ignore (fallback "error" g vs);
    raise (Error "Error fallback returned a value")
```

Defines:
  `catcherrorfallback`, used in chunks 46 and 66.

Uses `error` 10a 13 21a 39 42a, `fallback` 10a 13 21a 39 44c, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

Here are the definitions of the builtin fallbacks. Most cause errors, except the index fallbacks, which return nil by default.

43a ⟨*interp fallbacks* 43a⟩≡ (41) 43b▷

```
let errorfallback s g = fun args -> fallback "error" g [V.String s]
let arithfallback g   = function
  | [V.Number x; V.Number y; V.String s] when s = "pow" -> [V.Number (x ** y)]
  | args -> errorfallback "unexpected type at conversion to number" g args
let funcfallback g  = function
    | f::args   ->
        let args' = String.concat ", " (List.map V.to_string args) in
        let call  = Printf.sprintf "%s(%s)" (V.to_string f) args' in
        fallback "error" g [V.String ("call expr is "^call)]
    | args      -> fallback "error" g [V.String "call expr not a function"]

let fbs g =
  [ "arith",     arithfallback g
  ; "order",     errorfallback "unexpected type at comparison" g
  ; "concat",    errorfallback "unexpected type at conversion to string" g
  ; "index",     (fun args -> [V.Nil])
  ; "getglobal", (fun args -> [V.Nil])
  ; "gettable",  errorfallback "indexed expression not a table" g
  ; "settable",  errorfallback "indexed expression not a table" g
  ; "function",  funcfallback g
  ; "error",     default_error_fallback g
  ]
```

Defines:
`arithfallback`, never used.
`errorfallback`, never used.
`fbs`, used in chunk 43b.
`funcfallback`, never used.
Uses `arith` 48a, `call` 55c 127, `concat` 49a, `default_error_fallback` 44b, `error` 10a 13 21a 39 42a, `fallback` 10a 13 21a 39 44c, `getglobal` 10a 13 21a 39 50b, `index` 50a, `map` 83, `number` 84b, `order` 48c, `settable` 50a, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, and `to_string` 1 6b 8a 19 74 79b 81b.

Function `add_fallbacks` adds the default fallbacks to an interpreter. It's used above, at initialization time.

43b ⟨*interp fallbacks* 43a⟩+≡ (41) ◁43a 44a▷

```
let add_fallbacks g =
  List.iter (fun (k, f) -> Hashtbl.add g.V.fallbacks k (V.caml_func f)) (fbs g)
```

Defines:
`add_fallbacks`, used in chunk 41.
Uses `caml_func` 1 74 81a, `fbs` 43a, and `iter` 117 120.

A user can change a fallback by calling `setfallback`, which updates the fallback and returns the old fallback, if any.

44a ⟨*interp fallbacks* 43a⟩+≡ (41) ◁43b
```
  let setfallback g fbname fb =
    let fb' = try Hashtbl.find g.V.fallbacks fbname with Not_found -> V.Nil in
    let _   = Hashtbl.replace g.V.fallbacks fbname fb in
    fb'
```
Defines:
  `setfallback`, used in chunk 95.
Uses `find` 2a 75a 107 117 120 and `replace` 117 120.

The default error fallback expects one error message as argument. It prints the message and a stack trace, and then it raises the `Error` exception.

44b ⟨*interp toplevel* 42a⟩+≡ (41) ◁42a 44c▷
```
  exception Error of string
  let default_error_fallback g args =
    let msg = match args with V.String s :: _ -> s | _ -> "??error w/o message??" in
    prerr_string "lua: ";
    prerr_endline msg;
    prerr_endline "Stack trace:";
    currentloc_tostack g;
    List.iter (fun a -> prerr_string "  ";
               List.iter prerr_string (V.activation_strings g a);
               prerr_endline "") g.V.callstack;
    raise (Error msg)
```
Defines:
  `default_error_fallback`, used in chunks 43a, 44c, and 68.
Uses `activation_strings` 1 74 82b, `currentloc_tostack` 67a, `error` 10a 13 21a 39 42a,
  `iter` 117 120, and `string` 3a 76a 84a 99.

The `fallback` function invokes a fallback. It can't use the ordinary `call` because this could lead to an infinite loop.

If a fallback is a function, it is called. Otherwise, the `function` fallback is called. But if the `function` fallback is not a function, things are very badly wrong, and we call the *default* (not the current) error fallback. (Using the default error fallback guarantees termination.)

44c ⟨*interp toplevel* 42a⟩+≡ (41) ◁44b 46▷
```
  ⟨state dumping 68⟩
  let rec fallback fbname g args =
    let call f g args = match f with
    | V.Function (info, f) -> with_stack info g f args
    | v when fbname <> "function" -> fallback "function" g (v :: args)
    | v -> default_error_fallback g [V.String "‘function’ fallback not a function"] in
    let fbval = try Hashtbl.find g.V.fallbacks fbname
                with Not_found -> ⟨fallback failure 45⟩ in
    call fbval g args
  ⟨definition of catcherrorfallback 42b⟩
```
Defines:
  `fallback`, used in chunks 42b, 43a, 45–47, 50a, 82b, and 95.
Uses `call` 55c 127, `default_error_fallback` 44b, `find` 2a 75a 107 117 120,
  and `with_stack` 13 39 62b.

It should never happen that we don't have a fallback. But it can happen if a client registers an impure function as pure. If registered as pure, the function promises not to access machine state. In fact, we give it an empty state, and if the function breaks its promise, it can't find any fallbacks, and it may trigger this message.

45 ⟨*fallback failure* 45⟩≡ (44c)

```
begin
  prerr_string "no fallback named ‘";
  prerr_string fbname;
  prerr_endline "’ (probably registered an impure function as pure)";
  dump_state g;
  assert false (* can’t have any unknown fallbacks *)
end
```

Uses `dump_state` 68 and `fallback` 10a 13 21a 39 44c.

5.2.2 Function application

We're using the classic eval/apply model. Almost everybody uses the `apply` function, which if given a function, applies it. If `apply` is given anything else, it falls back to the "function" fallback.

If the types don't match and the application raises `Projection`, we have to do something. This problem has no counterpart in the C version, in which it's up to each individual C routine to worry about type mismatches.

> *I'm not sure that the* `Invalid_argument` *exception should be caught here. It is hard to find the cause of the problem without a stack trace. —CL*
>
> *It is unclear whether CL means he wants a Caml stack trace or a Lua stack trace. —NR*

Using `with_stack` here instead of in the definition of `lambda` ensures that we get a stack-trace entry even for Caml functions.

46 ⟨*interp toplevel* 42a⟩+≡ (41) ◁44c 47d▷

`apply : value -> state -> value list -> value list`

```
let apply f g args = match f with
  | V.Function (info, f) ->
        ( try (with_stack info g f args) with
        | V.Projection (v, what) -> ⟨projection error fallback 47a⟩
        | Errorfallback vs -> catcherrorfallback g vs
       (*** need the stack trace
        | Invalid_argument msg -> ⟨argument error fallback 47b⟩
        ***)
        )
  | v -> fallback "function" g (v :: args)
(*unboxval*)

let indent = ref 0
let apply' f g args =
  let str = proj_string g in
  let ind = String.make (!indent) '-' in
  Printf.eprintf
    "%sApply %s(%s)\n" ind (str f) (String.concat ", " (List.map str args));
  indent := !indent + 2;
  let answer = try apply f g args with e -> indent := !indent - 2; raise e in
  indent := !indent - 2;
  Printf.eprintf "%sResults %s(%s) = \n\t%s\n" ind
    (str f) (String.concat ", " (List.map str args))
    (String.concat ", " (List.map str answer));
  answer
```

Defines:
- `apply`, used in chunks 5a, 6a, 55c, 56a, 78a, 79a, and 89–91.
- `apply'`, never used.
- `indent`, never used.

Uses `answer` 41, `catcherrorfallback` 42b, `concat` 49a, `fallback` 10a 13 21a 39 44c, `map` 83, `proj_string` 47c, and `with_stack` 13 39 62b.

47a ⟨*projection error fallback* 47a⟩≡ (46)

```
fallback "error" g
         [V.String ("cannot convert value " ^ proj_string g v ^ " to " ^ what)]
```

Uses `error` 10a 13 21a 39 42a, `fallback` 10a 13 21a 39 44c, `proj_string` 47c, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

47b ⟨*argument error fallback* 47b⟩≡ (46)

```
fallback "error" g [V.String ("Function raised Invalid_argument " ^ msg)]
```

Uses `error` 10a 13 21a 39 42a and `fallback` 10a 13 21a 39 44c.

47c ⟨*interp utility functions* 47c⟩≡ (41) 67a▷

```
let proj_string g v =
  let what = match v with
  | V.Table t ->
      let l = try (V.list V.value).V.project v with _ -> [] in
      let not_nil = function V.Nil -> false | _ -> true in
      if Luahash.population t = List.length l && List.for_all not_nil l then
        "{ " ^ String.concat ", " (List.map V.to_string l) ^ " }"
      else
        V.to_string v
  | _ -> V.to_string v in
  let spr = Printf.sprintf in
  match V.objname g v with
  | Some (V.Fallback n)     -> spr "%s (fallback %s)" what n
  | Some (V.Global n)       -> spr "'%s %s'" what n
  | Some (V.Element (s, v)) -> spr "'%s %s[%s]'" what s (V.to_string v)
  | None -> what
```

Defines:
`proj_string`, used in chunks 46 and 47a.

Uses `concat` 49a, `fallback` 10a 13 21a 39 44c, `length` 84b, `list` 3d 76d 86b 99, `map` 83, `objname` 1 1 74 74 82a 82a, `population` 117 120, `to_string` 1 6b 8a 19 74 79b 81b, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

To use a fallback in a situation where we expect exactly one result, we adjust the results.

47d ⟨*interp toplevel* 42a⟩+≡ (41) ◁46 48a▷

```
fb1   : string -> state -> value list -> value
```

```
let fb1 name state args = match fallback name state args with
  | [] -> V.Nil
  | h :: t -> h
(*unboxval*)
```

Defines:
`fb1`, used in chunks 48–50.

Uses `fallback` 10a 13 21a 39 44c, `name` 127, and `state` 1 13 39 41 70 71 74 80a 93 112b.

5.2.3 Implementations of the infix operators

A binary arithmetic operator converts its arguments to floating-point numbers. If anything goes wrong, it calls the “arith” fallback.

48a ⟨*interp toplevel* 42a⟩+≡ (41) ◁47d 48b▷

```
arith : string -> (float -> float -> float) -> value -> value -> state -> value
```

```
  let arith opname op =
    let opname = V.String opname in (* allocate early and share *)
    let f x y g = try
      let x = V.float.V.project x in
      let y = V.float.V.project y in
      V.float.V.embed (op x y)
    with V.Projection (_, _) -> fb1 "arith" g [x; y; opname]
    in f
  (*unboxval*)
```

Defines:
`arith`, used in chunks 43a, 48b, and 49b.
Uses `fb1` 47d, `float` 3a 76a 84a 111b, and `op` 127.

Negation is similar except that it takes but one argument.

48b ⟨*interp toplevel* 42a⟩+≡ (41) ◁48a 48c▷

```
  let negate x g = try
    let x = V.float.V.project x in
    V.float.V.embed (~-. x)
    with V.Projection (_, _) -> fb1 "arith" g [x; V.Nil; V.String "umn"]
```

Defines:
`negate`, used in chunk 49b.
Uses `arith` 48a, `fb1` 47d, and `float` 3a 76a 84a 111b.

An ordering operation takes two forms: a numeric form and a string form. Its result is a Boolean.

48c ⟨*interp toplevel* 42a⟩+≡ (41) ◁48b 49a▷

```
order : string -> fcmp -> scmp -> value -> value -> state -> value
```

```
  type fcmp = float  -> float  -> bool
  type scmp = string -> string -> bool
  let order opname nop sop =
    let opname = V.String opname in
    let f x y g =
      match x, y with
      | V.Number x, V.Number y -> V.bool.V.embed (nop x y)
      | _ -> try let x = V.string.V.project x in
                 let y = V.string.V.project y in
                 V.bool.V.embed (sop x y)
             with V.Projection (_, _) -> fb1 "order" g [x; y; opname]
    in f
  (*unboxval*)
```

Defines:
`fcmp`, never used.
`order`, used in chunks 43a, 49b, 117, and 120.
`scmp`, never used.
Uses `bool` 3a 76a 85a 99, `fb1` 47d, `float` 3a 76a 84a 111b, and `string` 3a 76a 84a 99.

Concatenation converts to string.

49a ⟨*interp toplevel* 42a⟩+≡ (41) ◁48c 49b▷

```
let concat x y g =
  try let x = V.string.V.project x in
      let y = V.string.V.project y in
      V.string.V.embed (x ^ y)
  with V.Projection (_, _) -> fb1 "concat" g [x; y]
```

Defines:
`concat`, used in chunks 25, 43a, 46, 47c, 49b, 95, 99, and 102.
Uses `fb1` 47d and `string` 3a 76a 84a 99.

5.2.4 Compilation of infix syntax

The `binop` function takes abstract syntax and returns its denotation as a function. The `unop` function is similar. The short-circuit operators can't be handled this way.

49b ⟨*interp toplevel* 42a⟩+≡ (41) ◁49a 50a▷

`binop : A.op -> value -> value -> state -> value`

```
let binop = function
  | A.Plus   -> arith "add" (+.)
  | A.Minus  -> arith "sub" (-.)
  | A.Times  -> arith "mul" ( *. )
  | A.Div    -> arith "div" ( /. )
  | A.Pow    -> fun x y g -> fb1 "arith" g [x; y; V.String "pow"]
  | A.Lt     -> order "lt" (<)  (<)
  | A.Le     -> order "le" (<=) (<=)
  | A.Gt     -> order "gt" (>)  (>)
  | A.Ge     -> order "ge" (>=) (>=)
  | A.Eq     -> fun x y g -> V.bool.V.embed (V.eq x y)
  | A.Ne     -> fun x y g -> V.bool.V.embed (not (V.eq x y))
  | A.And    -> assert false (* short circuit *)
  | A.Or     -> assert false (* short circuit *)
  | A.Concat -> concat
  | A.Not    -> assert false (* unary *)

let unop = function
  | A.Minus  -> negate
  | A.Not    -> fun v g -> (match v with V.Nil -> V.Number 1.0 | _ -> V.Nil)
  | _        -> assert false (* all other operators are binary *)
(*unboxval*)
```

Defines:
`binop`, used in chunk 54a.
`unop`, used in chunk 54a.
Uses `arith` 48a, `binary` 65, `bool` 3a 76a 85a 99, `concat` 49a, `eq` 1 6b 8a 19 74 79b 80b, `fb1` 47d, `lt` 65, `negate` 48b, and `order` 48c.

5.2.5 Implementation of table operations

Tables are as you would expect, with fallbacks as required.

50a ⟨*interp toplevel* 42a⟩+≡ (41) ◁49b 50b▷
```
let index g t key = match t with
| V.Table t ->
    (match V.Table.find t key with
    | V.Nil -> fb1 "index" g [V.Table t; key]
    | v -> v)
| _ -> fb1 "gettable" g [t; key]

let settable g t key v = match t with
| V.Table t -> V.Table.bind t key v
| _ -> ignore (fallback "settable" g [t; key; v])
```
Defines:
`index`, used in chunks 43a, 52b, 56a, and 120.
`settable`, used in chunks 43a and 61.
Uses `bind` 2a 75a, `fallback` 10a 13 21a 39 44c, `fb1` 47d, and `find` 2a 75a 107 117 120.

5.2.6 Access to globals

Global lookup is much like table lookup, but they use a different fallback. Setting a global requires no fallback.

50b ⟨*interp toplevel* 42a⟩+≡ (41) ◁50a 50c▷
```
let getglobal g k =
  match V.Table.find g.V.globals k with
  | V.Nil -> fb1 "getglobal" g [k]
  | v -> v
let setglobal g k v = V.Table.bind g.V.globals k v
```
Defines:
`getglobal`, used in chunks 43a, 52a, 66, and 95.
`setglobal`, used in chunks 61, 66, and 95.
Uses `bind` 2a 75a, `fb1` 47d, and `find` 2a 75a 107 117 120.

5.2.7 Access to locals and temporaries

Local variables and temporaries share a single array. Access is by integer index, not by name, through functions `setlocal` and `getlocal`. To get multiple arguments for a function call, we have `getlocals`.

50c ⟨*interp toplevel* 42a⟩+≡ (41) ◁50b 56b▷
```
let setlocal locals n v = Array.set locals n v    (* could be made unsafe *)
let getlocal locals n   = Array.get locals n

let rec getlocals locals n count =
  if count = 0 then []
  else getlocal locals n :: getlocals locals (n+1) (count-1)
```
Defines:
`getlocal`, used in chunks 52–56, 59, and 61.
`getlocals`, used in chunks 55c, 56a, and 60.
`setlocal`, used in chunks 52a, 53, 55a, 56a, and 61.

5.3 Compilation

Compilation uses continuation-passing style, and to support type constraints, we define the type of continuation. A standard continuation takes an array that holds locals and temporaries. An extended continuation, which is used only in the case that the number of values used is not known until runtime, also takes a list of "extra results" (every result beyond the first).

51 ⟨*compiler* 51⟩≡ (41) 57▹

```
type a = A of a  (* used in place of type variable in type constraints *)
type 'a cont  = V.value array -> 'a  (* for exp1 *)
type 'a xcont = V.value list -> V.value array -> 'a  (* for exp, explist *)
```

Defines:
  `cont`, used in chunks 58, 60d, and 69.
  `xcont`, never used.
Uses `exp` 55a 65 84b 127, `exp1` 52a, `explist` 55b, `list` 3d 76d 86b 99,
  and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

5.3.1 Compiling expressions

We use two different expression compilers: in a context where exactly one value is expected, we use `exp1`; in a context where the number of values expected could vary, we use `exp`. The terms `exp` and `exp1` are taken from the grammar for Lua.

To avoid allocation, we let locals and temporaries share one array. The semantics of `exp1` are as follows:

- As early parameters, it takes a function `localref`, expression `e`, environment `rho`, continuation `theta`, and index `loc` (the location in the locals into which to put the result). The application of `exp1` to these parameters constitutes "compile time."
- This function returns a new continuation: a dynamic function that maps local state to answers, and as a side effect deposits the result of evaluating the expression in `locals.(loc)`. The application of this nameless function to global and local state, which produces an answer, constitutes "run time."
- At compile time, the `localref` function is called with each `loc` used. This function allows us to discover the maximum location that will be used at run time. We use this information to allocate a "locals" array of exactly the right size.

It is critically important that when `exp1` calls itself recursively, the call not be hidden under a lambda. Otherwise, we could inadvertently delay compilation until run time, with possibly disastrous results (e.g., access of a nonexistent local). It helps to remember that every `theta` should be a compile-time value.

As an example, we see that for a variable, we do the name lookup at compile time. At run time, we do either a by-name lookup for a global or a simple array lookup for a local. We "finish" by writing the result of the expression (in this case the value of the variable) into the location `loc` that is passed as a parameter.

52a ⟨*definitions of* `exp`, `explist`, *and friends* 52a⟩≡ (57) 52b▷

```
let append argv rest = match rest with [] -> argv | _ -> argv @ rest in
        (* optimizes common case *)
let rec exp1 localref =
  let rec exp1 rho e loc theta =
    let finish v l = setlocal l loc v; theta l in
    match e with
    | A.Var x -> localref loc;
                 (match rho x with
                  | Global  -> fun l -> finish (getglobal g (V.String x)) l
                  | Local n -> fun l -> finish (getlocal l n) l)
```

Defines:
`append`, used in chunks 55c, 56a, 60, 93, 99, and 113.
`exp1`, used in chunks 51–56, 59, and 61.
Uses `getglobal` 10a 13 21a 39 50b, `getlocal` 50c, and `setlocal` 50c.

A literal is straightforward. An indexing expression compiles the table and index into two locations, then builds a new continuation `theta`, which fetches from these locations, computes the indexing expression, and finishes.

52b ⟨*definitions of* `exp`, `explist`, *and friends* 52a⟩+≡ (57) ◁52a 53▷

```
    | A.Lit v -> localref loc; fun l -> finish v l
    | A.Index (tab, key) ->
        let tabloc = loc in
        let keyloc = loc + 1 in
        let theta l = finish (index g (getlocal l tabloc) (getlocal l keyloc)) l in
        exp1 rho tab tabloc (exp1 rho key keyloc theta)
```

Uses `exp1` 52a, `getlocal` 50c, and `index` 50a.

In a table literal, the keys are static, but the values are dynamic. Luckily, however, we can bind away each value as soon as it is produced, so we need only one temporary location for values. Internally we would like to use a special `theta` that takes *three* dynamic parameters: global, local, and table. But this would require polymorphic recursion.

Local function `listbind` deals with the list-like part of the syntax, and `bind` deals with the record-like part of the syntax. Because lists always come first, we call `listbind` which then calls `bind` when it runs out of bindings.

53 ⟨*definitions of* `exp`*,* `explist`*, and friends* 52a⟩+≡ (57) ◁52b 54a▷

```
  | A.Table (lists, bindings) ->
      localref loc;                    (* needed if table is empty *)
      let tabloc = loc in
      let vloc   = loc + 1 in
      let tbl l = match getlocal l tabloc with V.Table t -> t | _ -> assert false in
      let rec listbind n theta = function
        | [] -> bind theta bindings
        | h::t ->
            let theta = listbind (n +. 1.0) theta t in
            let theta = fun l -> V.Table.bind (tbl l) (V.Number n) (getlocal l vloc);
                                 theta l
            in  exp1 rho h vloc theta
              (* PERHAPS FOR LAST ELEMENT IN LIST, SHOULD CAPTURE *ALL* RESULTS? *)
      and bind theta = function
        | [] -> theta
        | (n, h) :: t ->
            let theta = bind theta t in
            let theta = fun l -> V.Table.bind (tbl l) (V.String n) (getlocal l vloc);
                                 theta l
            in  exp1 rho h vloc theta in
      let size  = List.length bindings + List.length lists in
      let theta = listbind 1.0 theta lists in
      fun l ->
        let t = V.Table.create size in
        setlocal l tabloc (V.Table t);
        theta l
```

Uses `bind` 2a 75a, `create` 2a 75a 117 120, `exp1` 52a, `getlocal` 50c, `length` 84b, `setlocal` 50c, and `table` 4a 77a 80a 87a.

The short-circuit binary operators require special trickery. We use the `short_circuit` function to combine true and false continuations. The standard binary operators provide a good illustration of how the location technique works.

54a ⟨*definitions of* `exp`, `explist`, *and friends* 52a⟩+≡ (57) ◁53 54b▷

```
    | A.Binop (e1, op, e2) ->
        let short_circuit theta_t theta_f = fun l ->
          match getlocal l loc with
          | V.Nil -> theta_f l
          | _ ->     theta_t l in
        ( match op with
        | A.And -> exp1 rho e1 loc (short_circuit (exp1 rho e2 loc theta) theta)
        | A.Or  -> exp1 rho e1 loc (short_circuit theta (exp1 rho e2 loc theta))
        | _ ->
            let loc1 = loc in
            let loc2 = loc + 1 in
            let op = binop op in
            exp1 rho e1 loc1 (
            exp1 rho e2 loc2 (
            fun l -> finish (op (getlocal l loc1) (getlocal l loc2) g) l)))
    | A.Unop (op, e) ->
        let op = unop op in
        exp1 rho e loc (fun l -> finish (op (getlocal l loc) g) l)
```

Uses `binop` 49b, `exp1` 52a, `getlocal` 50c, `op` 127, and `unop` 49b.

A call could be a function call or a method call. Because in the general case, a call can return multiple results, we delegate the compilation to `exp`. Because *this* call occurs in a single-result context, we pass `exp` a continuation that ignores any "extra" results.

54b ⟨*definitions of* `exp`, `explist`, *and friends* 52a⟩+≡ (57) ◁54a 55a▷

```
    | A.Call c -> exp localref rho e loc (fun _ -> theta)
  in  exp1
```

Uses `exp` 55a 65 84b 127 and `exp1` 52a.

When we compile an expression in a full, multi-result context, we `finish` with a *list* of expressions. The first expression is written to `loc`; any remaining expressions are passed to `theta` before the local state.

The only expression that can actually produce multiple results is a call. Any other expression is compiled by `exp1`. The details are delegated to function `call`, which is shared with the compiler for statements.

55a ⟨*definitions of* `exp`*,* `explist`*, and friends* 52a⟩+≡ (57) ◁54b 55b▷

```
and exp localref rho e loc theta =
  let finish  vs l = match vs with
  | v :: vs -> setlocal l loc v; theta vs l
  | []      -> setlocal l loc V.Nil; theta [] l in
  match e with
  | A.Call c -> localref loc; call localref c rho loc finish
  | _ -> exp1 localref rho e loc (theta [])
```

Defines:
`exp`, used in chunks 51, 54b, 55b, and 69.
Uses `call` 55c 127, `exp1` 52a, and `setlocal` 50c.

Function `explist` compiles code to evaluate a list of expressions, each linked to the next by its continuation `theta`. The last expression is compiled in a multi-result context. Function `explist` tracks the desired location of each result.

55b ⟨*definitions of* `exp`*,* `explist`*, and friends* 52a⟩+≡ (57) ◁55a 55c▷

```
and explist localref rho es loc theta = match es with
| [] -> theta []
| [e] -> exp localref rho e loc theta
| e :: es -> exp1 localref rho e loc (explist localref rho es (loc+1) theta)
```

Defines:
`explist`, used in chunks 51, 55c, 56a, and 60.
Uses `exp` 55a 65 84b 127 and `exp1` 52a.

For a call we compile the function and its arguments, creating a continuation that pulls `fv` (the function value) and `argv` (the list of arguments) out of the locals, then uses `apply` to apply the function. The `theta` passed in is a somewhat different one from the expression `theta`; before taking the state, it takes the list of *all* values returned.

55c ⟨*definitions of* `exp`*,* `explist`*, and friends* 52a⟩+≡ (57) ◁55b 56a▷

```
and call localref c rho loc theta = match c with
| A.Funcall (f, args) ->
    let argcount = List.length args in
    let argloc = loc + 1 in
    exp1 localref rho f loc (
    explist localref rho args argloc
      (fun vs l ->
        let fv   = getlocal l loc in
        let argv = getlocals l argloc argcount in
        theta (apply fv g (append argv vs)) l))
```

Defines:
`call`, used in chunks 43a, 44c, 55a, and 60c.
Uses `append` 52a, `apply` 10a 21a 46, `exp1` 52a, `explist` 55b, `getlocal` 50c, `getlocals` 50c, and `length` 84b.

Because of the semantics of method call, we use more locations, and the code is more elaborate, but the ideas are the same.

56a ⟨*definitions of* `exp`*,* `explist`*, and friends* 52a⟩+≡ (57) ◁55c

```
  | A.Methcall (obj, meth, args) ->
      let mloc     = loc in
      let selfloc  = mloc + 1 in
      let argloc   = selfloc + 1 in
      let argcount = List.length args + 1 in
      let meth     = V.String meth in
      exp1 localref rho obj selfloc (
        let theta_m = explist localref rho args argloc
            (fun vs l ->
              let fv = getlocal l loc in
              let argv = getlocals l selfloc argcount in
              theta (apply fv g (append argv vs)) l) in
        fun l -> setlocal l mloc (index g (getlocal l selfloc) meth); theta_m l)
```

Uses `append` 52a, `apply` 10a 21a 46, `exp1` 52a, `explist` 55b, `getlocal` 50c, `getlocals` 50c, `index` 50a, `length` 84b, and `setlocal` 50c.

5.3.2 Compiling statements

One oddity of Lua is that a statement can declare new local variables. At compile time, we keep a list of all local variables that have been declared. This list forms an environment that enables us to map a local-variable name to its index (by the `lookup` function). The function `extend` accepts a statement and uses it to extend the environment.

56b ⟨*interp toplevel* 42a⟩+≡ (41) ◁50c

```
  let rec extend rho = function
    | A.Stmt' (_, s) -> extend rho s
    | A.Local (vs, _) -> List.rev_append vs rho
    | _ -> rho
```

Defines:
`extend`, used in chunks 60d and 64b.

Here's where we compile an entire block. The function `block_compiler` is a sort of pre-compiler; it strings things together. It returns two functions: `bcomp` can be used to compile a block, and afterward, `local_size` can be used to find out how many locals are used in the block.

57 ⟨*compiler* 51⟩+≡ (41) ◁51 63▷

```
exp1    : (int -> unit) -> (string -> var) -> A.exp      -> int -> a  cont -> a cont
exp     : (int -> unit) -> (string -> var) -> A.exp      -> int -> a xcont -> a cont
explist : (int -> unit) -> (string -> var) -> A.exp list -> int -> a xcont -> a cont
block   : string list -> A.block -> 'a cont -> (value list -> 'a) -> 'a cont
```

```
⟨old debugging code 69⟩
let block_compiler srcmap g =
  ⟨definitions of exp, explist, and friends 52a⟩ in
  let high_local_limit = ref 0 in
  let localref n = if n >= !high_local_limit then high_local_limit := n+1 in
  let local_size () = !high_local_limit in
  let bcomp ~debug =
      ⟨definitions of block, stmt, and friends 58⟩ in
(*inboxval*)
      block
  in  bcomp, local_size
```

Defines:
`block_compiler`, used in chunks 63 and 64b.
Uses `block` 60d 127.

When compiling a statement, we provide two continuations: a standard continuation `theta` and a return continuation `ret`. Both are compile-time values (static parameters). This setup approximates direct threaded code. The type of `theta` is locals to answer, and the type of `ret` is value list to answer.

When we get a source-code marker, we might have to update the current location in the machine state. We pay the overhead only if debugging is turned on, which decision is made at *compile* time. An earlier version of this code had a mistake in which the debugging call to `stmt` was hidden under a lambda. We avoid that problem here by defining new *static* continuations that use the *dynamic* contents of the ref call `restore`.

58 ⟨*definitions of* `block`*,* `stmt`*, and friends* 58⟩≡ (57) 59a▷

```
let rec stmt rho s (theta: 'a cont)  (ret:value list -> 'a) = match s with
  | A.Stmt' (charpos, s) ->
      if debug then
        (* might make interesting example for paper *)
        let where = Luasrcmap.location srcmap charpos in
        let restore = ref (fun () -> ()) in  (* will restore currentloc *)
        let theta' l = (!restore(); theta l) in
        let ret' ans = (!restore(); ret ans) in
        let stheta = stmt rho s theta' ret' in
        fun l -> let n = g.V.currentloc in
                 ( restore := (fun () -> g.V.currentloc <- n)
                 ; g.V.currentloc <- Some where
                 ; stheta l
                 )
            (* hard to maintain current line if exn raised ... *)
      else
        stmt rho s theta ret
```

Defines:
`stmt`, used in chunk 60.
Uses `cont` 51, `list` 3d 76d 86b 99, `location` 127, `ret` 65, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

A little ref-cell jujitsu is needed to make loops work. Again, we give the body a *static* continuation which uses the *dynamic* contents of a ref cell. Here we actually update the ref cell immediately after the compilation, closing the recursive loop.

59a ⟨*definitions of* `block`*,* `stmt`*, and friends* 58⟩+≡ (57) ◁58 59b▷
```
  | A.WhileDo (cond, body) ->
      let loop_cont = ref theta in  (* to become loop continuation *)
      let goto_head l = !loop_cont l in
      let condloc = List.length rho in
      let body = block rho body goto_head ret in
      let loop =
        exp1 localref (lookup rho) cond condloc
          (fun l -> if notnil (getlocal l condloc) then body l else theta l) in
      let _ = loop_cont := loop in
      loop
  | A.RepeatUntil (body, cond) ->
      let loop_test = ref theta in (* to become loop-end continuation *)
      let goto_test l = !loop_test l in
      let condloc = List.length rho in
      let body = block rho body goto_test ret in
      let loop =
        exp1 localref (lookup rho) cond condloc
          (fun l -> if notnil (getlocal l condloc) then theta l else body l) in
      let _ = loop_test := loop in
      body
```
Uses `block` 60d 127, `exp1` 52a, `getlocal` 50c, `length` 84b, `lookup` 67b, `loop` 65, `notnil` 67c, and `ret` 65.

The only complication with the conditional is the complication built into the abstract syntax.

59b ⟨*definitions of* `block`*,* `stmt`*, and friends* 58⟩+≡ (57) ◁59a 60a▷
```
  | A.If (c, t, alts, f) ->
      let alts = (c, t) :: alts in
      let f = block rho (match f with None -> [] | Some ss -> ss) theta ret in
      let condloc = List.length rho in
      let add (cond, body) f =
        let body = block rho body theta ret in
        exp1 localref (lookup rho) cond condloc (
          fun l -> if notnil (getlocal l condloc) then body l else f l) in
      List.fold_right add alts f
```
Uses `block` 60d 127, `exp1` 52a, `getlocal` 50c, `length` 84b, `lookup` 67b, `notnil` 67c, and `ret` 65.

Returning compiles a list of expressions, supplying a static continuation that grabs the results from the local-variable array and passes them to the static return continuation.

60a ⟨*definitions of* block, stmt, *and friends* 58⟩+≡ (57) ◁59b 60b▷
```
  | A.Return es ->
      let loc = List.length rho in
      let result_count = List.length es in
      explist localref (lookup rho) es loc
      (fun vs l -> ret (append (getlocals l loc result_count) vs))
```
Uses append 52a, explist 55b, getlocals 50c, length 84b, lookup 67b, and ret 65.

A local declaration has the same dynamic effect as an assignment, which must be compiled in an extended compile-time environment.

60b ⟨*definitions of* block, stmt, *and friends* 58⟩+≡ (57) ◁60a 60c▷
```
  | A.Local (vs, es) ->
      stmt (List.rev_append vs rho) (A.Assign (List.map (fun x -> A.Lvar x) vs, es))
      theta ret
```
Uses map 83, ret 65, and stmt 58 127.

Assignments are quite tricky because of the multiple-return-value semantics of Lua calls. This stuff probably needs better documentation.

60c ⟨*definitions of* block, stmt, *and friends* 58⟩+≡ (57) ◁60b 60d▷
```
  | A.Assign (vs, es) ->
      let rhscount = List.length es in
      lvars localref (lookup rho) (List.length rho) vs (fun setlvs loc ->
        explist localref (lookup rho) es loc (fun vs l ->
          setlvs l (append (getlocals l loc rhscount) vs);
          theta l))
  | A.Callstmt c ->
      call localref c (lookup rho) (List.length rho) (fun _ l -> theta l)
```
Uses append 52a, call 55c 127, explist 55b, getlocals 50c, length 84b, lookup 67b, and lvars 62a.

The block function compiles statements in sequence. It manages both the linking of continuations and the extension of the environment after each statement s, since an s could contain a local declaration.

60d ⟨*definitions of* block, stmt, *and friends* 58⟩+≡ (57) ◁60c 61▷
```
and block rho body (theta:'a cont) (ret:V.value list -> 'a) = match body with
  | [] -> theta
  | s :: ss -> stmt rho s (block (extend rho s) ss theta ret) ret
```
Defines:
block, used in chunks 57, 59, 63, and 64.
Uses cont 51, extend 56b, list 3d 76d 86b 99, ret 65, stmt 58 127, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

5.3.3 Compiling assignments

How are we to deal with assigment? One problem is that the number of right-hand sides cannot always be bound statically. But since the number of *left* hand sides is bound statically, maybe we can pull a bit of a cheat there.

So here's the idea: pass a continuation builder for putting all the right-hand sides in place, then continue with them.

What's the control flow like? Well, first we evaluate the lvalue, which may mean using some temps and then pulling them immediately. Then we do the next lvalue and so on. Then the next step is the rvalues, and for each rvalue, we must have some sort of dynamic thingummy built up. When I get this figured out, it should go in the paper!

Argument `nextlvar` passed to `lvar` takes a setter for the current lvar and also the index of the next free location in the temporary space. The setter takes global and local state, plus a value, and it is executed only for side effect. Argument `finish` passed to `lvars` is similar, except its setter takes a list of values.

61 ⟨*definitions of* `block`*,* `stmt`*, and friends* 58⟩+≡ (57) ◁60d 62a▷
```
  and lvar localref rho lv lhsloc nextlvar =
    match lv with
    | A.Lvar x ->
        let setx = match rho x with
        | Global  -> fun l v -> setglobal g (V.String x) v
        | Local n -> fun l v -> setlocal l n v in
        nextlvar setx lhsloc
    | A.Lindex (t, key) ->
        let keyloc = lhsloc + 1 in
        let setidx = exp1 localref rho t lhsloc (exp1 localref rho key keyloc (fun l ->
              let t   = getlocal l lhsloc in
              let key = getlocal l keyloc in
              (fun v -> settable g t key v))) in
        nextlvar setidx (lhsloc+2)
```
Defines:
  `lvar`, used in chunk 62a.
Uses `exp1` 52a, `getlocal` 50c, `setglobal` 50b 113, `setlocal` 50c, and `settable` 50a.

62a ⟨*definitions of* block, stmt, *and friends* 58⟩+≡ (57) ◁61

```
and lvars localref rho loc lvs finish = match lvs with
  | [] -> finish (fun l vs -> ()) loc
  | h :: t ->
      lvar localref rho h loc (fun setter loc ->
        lvars localref rho loc t (fun setlvs loc ->
          let setlvs l vs =
            let v, vs = match vs with h::t -> h, t | [] -> V.Nil, [] in
            setter l v;
            setlvs l vs in
          finish setlvs loc))
```

Defines:
  lvars, used in chunk 60c.
Uses lvar 61.

5.3.4 Compiling chunks and functions

Chunks are executed by compiling functions.

Executing a function requires pushing that function's information on the machine's call stack. The with_stack function handles the stack: it pushes new info, and it pops on both normal and exceptional termination. Too bad Caml doesn't have try-finally.

62b ⟨*interp* with_stack 62b⟩≡ (41)

```
let indent = ref 0
let with_stack info g f x =
  let _ = currentloc_tostack g in
  let _ = g.V.callstack <- (info, None) :: g.V.callstack in
  let _ = currentloc_fromstack g in
(*
let _ = (prerr_string (String.make (!indent) ' ');
         prerr_string "=> ";
             List.iter prerr_string (V.activation_strings g (info, None));
             prerr_endline "") in
*)
  let pop () = g.V.callstack <- List.tl g.V.callstack; currentloc_fromstack g
(*
; indent := !indent - 1; prerr_string (String.make (!indent) ' ')
; prerr_string "[]\n"
*)
  in
(*let _ = indent := !indent + 1 in*)
  let answer = try f x with e -> (pop(); raise e) in
  let _ = pop() in
  answer
```

Defines:
  indent, never used.
  with_stack, used in chunks 44c, 46, and 97.
Uses activation_strings 1 74 82b, answer 41, currentloc_fromstack 67a,
  currentloc_tostack 67a, and iter 117 120.

Here's the compilation of a function. We compile the body with continuations that produce the list of results of the function. Falling off produces the empty list, and returning `results` produces those results.

We return a dynamic function suitable for use as a Lua function value. It takes state and arguments, allocates an array for locals and temporaries,[1] initializes the array with the values of the actual parameters, then applies body continuation to the state to get the answer. We don't have to maintain the stack because that is done in function application above.

63 ⟨*compiler* 51⟩+≡ (41) ◁57 64a▷

```
ool -> Luasrcmap.location -> string list -> bool -> A.block -> state -> V.srcloc * V.func
  let value_list = V.list V.value
  let lambda (src, debug) (file, line, col) args varargs body state =
    let rho = let args' = List.rev args in if varargs then "arg" :: args' else args' in
    let block, count = block_compiler src state in
    let body = block ~debug rho body (fun l -> []) (fun results -> results) in
    let n = max (count()) (List.length rho) in
    let srcloc = V.srcloc file line in
    srcloc,
    fun argv ->
      let locals = Array.make n V.Nil in
      let rec walk n formals actuals = match formals with
        | [] -> if varargs then Array.set locals n (value_list.V.embed actuals)
        | f :: fs ->
            let a, a's = match actuals with [] -> V.Nil, [] | h :: t -> h, t in
            (Array.set locals n a; walk (n+1) fs a's)  in
      let _ = walk 0 args argv in
      body locals
  (*unboxval*)
```

Defines:
  `lambda`, used in chunk 64a.
  `value_list`, never used.
Uses `block` 60d 127, `block_compiler` 57, `file` 99 113, `length` 84b, `list` 3d 76d 86b 99, `results` 5a 78a 89, `srcloc` 1 1 74 74 80a 81a, `state` 1 13 39 41 70 71 74 80a 93 112b, `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c, and `varargs` 127.

[1]We don't allocate space for unused locals, but we do allocate space for formals, even if formals are not used. The reason is we hope they're used in the common case, and we'd rather not add the additional check to every formal parameter.

Chunk compilation is not too exciting. We have a statement or a function.

64a ⟨*compiler* 51⟩+≡ (41) ◁63 64b▷
```
  let func (info, f) = V.Function (info, f)
  let chunk ((smap, dbg) as srcdbg) block rho g = function
    | A.Debug _ -> assert false (* must never get here *)
    | A.Statement s -> block rho [s]
    | A.Fundef (pos, f, args, varargs, body) ->
        let v = func (lambda srcdbg (Luasrcmap.location smap pos) args varargs body g) in
        block rho [A.Stmt'(pos, A.Assign ([f], [A.Lit v]))]
    | A.Methdef (pos, obj, meth, args, varargs, body) ->
        let args = "self" :: args in
        let v = func (lambda srcdbg (Luasrcmap.location smap pos) args varargs body g) in
        block rho [A.Stmt'(pos, A.Assign ([A.Lindex (obj, A.Lit (V.String meth))],
                                           [A.Lit v]))]
```
Defines:
`chunk`, used in chunks 13, 14b, 16, 39, 64b, and 94.
`func`, used in chunks 1, 74, and 99.
Uses `block` 60d 127, `lambda` 63, `location` 127, and `varargs` 127.

Compiling a list of chunks is a lot like compiling a function, except we also have to keep track of whether debugging is turned on.

64b ⟨*compiler* 51⟩+≡ (41) ◁64a
```
  let extendchunk rho = function
    | A.Statement s -> extend rho s
    | _ -> rho

  let compile ~srcdbg cs g =
    let block, count = block_compiler (fst srcdbg) g in
    let ret = fun results -> results in
    let rec chunks ((smap, debug) as srcdbg) rho = function
      | [] -> fun l -> []
      | A.Debug dbg :: t -> chunks (smap, dbg) rho t
      | h :: t -> chunk srcdbg (block ~debug) rho g h
                  (chunks srcdbg (extendchunk rho h) t) ret in
    let theta = chunks srcdbg [] cs in
    let locals = Array.make (count()) V.Nil in
    fun () -> theta locals
```
Defines:
`compile`, used in chunks 65 and 97.
`extendchunk`, never used.
Uses `block` 60d 127, `block_compiler` 57, `chunk` 7a 64a 127, `chunks` 7a, `extend` 56b, `results` 5a 78a 89, and `ret` 65.

5.4 Some very simple tests

These tests are probably just about worthless now, but early in the game they were quite a help.

65 ⟨*interp tests* 65⟩≡ (41)
```
  let nil = A.Lit V.Nil
  let three = A.Lit (V.Number 3.0)

  let ret = A.Return ([nil; three])

  let test_state = V.state ()

  let bogusmap = Luasrcmap.mk ()
  let stmts l = compile ~srcdbg:(bogusmap, false) (List.map (fun s -> A.Statement s) l)

  let num n = A.Lit (V.Number (float n))
  let rtest = stmts [ret]
  let sum = stmts [A.Return ([A.Binop (three, A.Plus, three)])]
  let exp = stmts [A.Return ([A.Binop (three, A.Times, A.Binop (num 2, A.Minus, three))])]
  let x = A.Var "x"
  let gets x e = A.Assign ([A.Lvar x], [e])
  let binary op e1 e2 = A.Binop (e1, op, e2)
  let lt = binary A.Lt
  let times = binary A.Times
  let loop = [ gets "x" (num 10)
               ; A.WhileDo (lt x (num 100), [gets "x" (times (num 2) x)])
               ; A.Return [x]]

  let test _ =
    [ "return nil and three", rtest test_state
    ; "sum of three and three", sum test_state
    ; "expression minus three", exp test_state
    ; "result of loop", stmts loop test_state
    ]
```
Defines:
- `binary`, used in chunk 49b.
- `bogusmap`, never used.
- `exp`, used in chunks 51, 54b, 55b, and 69.
- `gets`, never used.
- `loop`, used in chunks 59a and 71.
- `lt`, used in chunk 49b.
- `nil`, used in chunks 3d, 66, 76d, 81b, 95, and 113.
- `num`, never used.
- `ret`, used in chunks 58–60 and 64b.
- `rtest`, never used.
- `stmts`, never used.
- `sum`, used in chunk 120.
- `test`, never used.
- `test_state`, never used.
- `three`, never used.
- `times`, never used.

Uses `compile` 13 39 64b, `float` 3a 76a 84a 111b, `map` 83, `mk` 14b 16 70 71 94, `op` 127, `result` 5a 78a 89, and `state` 1 13 39 41 70 71 74 80a 93 112b.

5.4.1 Client registration functions

66 ⟨*interp registration* 66⟩≡ (41)

```
let register_global g k v =
  match getglobal g k with
  | V.Nil -> setglobal g k v
  | _ -> Printf.kprintf failwith "Global variable '%s' is already set" (V.to_string k)

let register_globals l g = List.iter (fun (k, v) -> register_global g (V.String k) v) l

let register_module tabname members g =
  let t = getglobal g (V.String tabname) in
  let t = match t with
  | V.Nil       -> V.Table.create (List.length members)
  | V.Table t   -> t
  | _           -> catcherrorfallback g
                   [V.String ("Global value " ^ tabname ^ " is not (table or nil)")] in
  let _ = register_global g (V.String tabname) (V.Table t) in
  let bind (k, v) = match V.Table.find t (V.String k) with
  | V.Nil -> V.Table.bind t (V.String k) v
  | _ ->
      Printf.kprintf failwith "Duplicate '%s' registered in module '%s'" k tabname in
  List.iter bind members
```

Defines:
`register_global`, never used.
`register_globals`, used in chunks 95, 103, 111c, and 116b.
`register_module`, used in chunk 100.

Uses `bind` 2a 75a, `catcherrorfallback` 42b, `create` 2a 75a 117 120, `find` 2a 75a 107 117 120, `getglobal` 10a 13 21a 39 50b, `iter` 117 120, `length` 84b, `nil` 65, `setglobal` 50b 113, `table` 4a 77a 80a 87a, `to_string` 1 6b 8a 19 74 79b 81b, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

5.4.2 Utility functions

Managing the current location The state of the interpreter includes a stack of locations, but the current location is not stored on the top of the stack but in the special field `currentloc`. This representation is supposed to improve performance (though it has never been measured). Functions `currentloc_tostack` and `currentloc_fromstack` both leave the cache consistent with the stack, in both cases by making the eponymous assignment.

67a ⟨*interp utility functions* 47c⟩+≡ (41) ◁47c 67b▷

```
let currentloc_tostack g =
  match g.V.callstack with
  | (info, _) :: t -> g.V.callstack <- (info, g.V.currentloc) :: t
  | [] -> ()

let currentloc_fromstack g =
  match g.V.callstack with
  | (info, where) :: _ -> g.V.currentloc <- where
  | [] -> ()
```

Defines:
`currentloc_fromstack`, used in chunk 62b.
`currentloc_tostack`, used in chunks 44b and 62b.

Access to variables A variable is global or local. A local variable is known by its index. The mapping from name to `var` is done at compile time; global and local lookups are done at

67b ⟨*interp utility functions* 47c⟩+≡ (41) ◁67a 67c▷

```
type var = Global | Local of int
let lookup rho x =
  let rec look = function
    | [] -> Global
    | h :: t when h = x -> Local (List.length t)
    | h :: t -> look t
  in look rho
```

Defines:
`lookup`, used in chunks 59, 60, and 69.
`var`, used in chunk 69.
Uses `int` 3a 76a 85a 99 and `length` 84b.

Miscellany

67c ⟨*interp utility functions* 47c⟩+≡ (41) ◁67b

```
let notnil = function
  | V.Nil -> false
  | _ -> true
```

Defines:
`notnil`, used in chunk 59.

Debugging support

68 ⟨*state dumping* 68⟩≡ (44c)
```
let dump_state g =
  let err = prerr_string in
  let rec value = function
    | V.Table t -> tab t ""
    | v -> err (V.to_string v)
  and tab t sfx =
    err "{"; Luahash.iter (fun k d -> err " "; value k; err "="; value d; err ",") t;
    err "}"; err sfx in
  let stab t sfx =
    err "{"; Hashtbl.iter (fun k d -> err " "; err k; err "="; value d; err ",") t;
    err "}"; err sfx in
  err "state is: \n";
  err "  globals =\n    ";
  tab g.V.globals "\n";
  err "  fallbacks =\n    ";
  stab g.V.fallbacks "\n";
  default_error_fallback g [V.String "Stack trace is:"]
```

Defines:
`dump_state`, used in chunk 45.
Uses `default_error_fallback` 44b, `iter` 117 120, `state` 1 13 39 41 70 71 74 80a 93 112b, `to_string` 1 6b 8a 19 74 79b 81b, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

69 ⟨*old debugging code* 69⟩≡ (57)
```
  let wrap f x = let l = f x in (prerr_string "function returned ";
                               prerr_int (List.length l);
                               prerr_endline " values";
                               l)

  let expname = function
    | A.Var v -> v
    | _ -> "?"
  let funname = function
    | A.Lvar v -> v
    | A.Lindex (e, A.Lit (V.String s)) -> expname e ^ "." ^ s
    | A.Lindex (e, e') -> expname e ^ "[" ^ expname e' ^ "]"

  (*
  let (_ : (string -> var) -> A.exp -> int -> 'a cont -> 'a cont) = exp' (fun _ -> ())

  let (pexp : (string -> var) -> A.exp -> int -> answer cont -> answer cont) = exp' (fun _ -> ())
  *)
  (*
  let show_locals rho =
    prerr_string "============\n";
    List.iter (fun x ->
      List.iter prerr_string ([x; " is "] @
                             match lookup rho x with
                             | Local n -> [" local variable "; string_of_int n; "\n"]
                             | Global -> ["global\n"])) rho;
    prerr_endline ""
  *)
```
Defines:
`expname`, never used.
`funname`, used in chunk 113.
`show_locals`, never used.
`wrap`, never used.

Uses `answer` 41, `cont` 51, `exp` 55a 65 84b 127, `int` 3a 76a 85a 99, `iter` 117 120, `length` 84b, `lookup` 67b, `string` 3a 76a 84a 99, and `var` 67b.

6 Standalone Lua interpreters

Runs the given interpreter at startup, using `Sys.argv` to decide what to do.

70 ⟨*luarun.mli* 70⟩≡

```
module type INTERP = sig
  module Value : Luavalue.S
  type value = Value.value
  type state = Value.state
  val mk        : unit -> state
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
end
module Make (I : INTERP) : sig end
```

Defines:
`dofile`, used in chunk 95.
`dostring`, used in chunk 95.
`mk`, used in chunks 65, 95, and 97.
`state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

Uses `list` 3d 76d 86b 99, `string` 3a 76a 84a 99, and `unit` 3a 76a 85a.

71 ⟨*luarun.ml* 71⟩≡

```
module type INTERP = sig
  module Value : Luavalue.S
  type value = Value.value
  type state = Value.state
  val mk        : unit -> state
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
end
module Make (I : INTERP) = struct
  module V = I.Value
  let state = I.mk()
  let dumpstate = ref false
  let showresults =
    let rec loop n = function
      | h :: t -> print_string "Result "; print_int n; print_string " = ";
                  print_endline (V.to_string h); loop (n+1) t
      | [] -> ()
    in loop 1
  let run infile = ignore (I.dofile state infile)
  let run_interactive infile =
    let rec loop n pfx =
      let line = input_line infile in
      if String.length line > 0 && String.get line (String.length line - 1) = '\\' then
        loop n (pfx ^ String.sub line 0 (String.length line - 1) ^ "\n")
      else
        begin
          ignore (I.dostring state (pfx ^ line ^ "\n"));
          flush stdout; flush stderr;
          loop (n+1) ""
        end
    in  try loop 1 "" with End_of_file -> ()
  let rec args = function
    | "-dump" :: a's -> (dumpstate := true; args a's)
    | "-new"  :: a's -> args a's
    | [] -> run_interactive stdin
    | files -> List.iter run files

  let _ = args (List.tl (Array.to_list (Sys.argv)))

  let _ = if !dumpstate then
    begin
      print_endline "final state: ";
      Luahash.iter (fun k v -> print_string "  ";
        print_string (V.to_string k); print_string " |-> ";
        print_endline (V.to_string v)) state.V.globals
    end
end
```

Defines:

`dofile`, used in chunk 95.
`dostring`, used in chunk 95.
`mk`, used in chunks 65, 95, and 97.
`state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

Uses `infile` 99 113, `iter` 117 120, `length` 84b, `list` 3d 76d 86b 99, `loop` 65, `string` 3a 76a 84a 99, `to_string` 1 6b 8a 19 74 79b 81b, and `unit` 3a 76a 85a.

7 Lua values, parameterized by user data

As in Lua, we keep the value space simple and small. Unlike Lua, we have only one kind of function. The type of `userdata` is not specified here; it is intended to be a functor parameter.

This interface is a key for clients because it not only specifies what a Lua value is, but also provides higher-order embedding/projection pairs so that Caml values can be mapped to Lua values and back again.

We begin with embedding/projection, which is shared among all modules matching `Luavalue.S`. We need a synonym because to re-export `ep` and `ep` would be consider a circular type defintion.

73 ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁39 74▷

```
type ('a, 'b, 'c) ep = { embed : 'a -> 'b; project : 'b -> 'a; is : 'c -> bool }
type ('a, 'b, 'c) synonym_for_ep = ('a, 'b, 'c) ep
  = { embed : 'a -> 'b; project : 'b -> 'a; is : 'c -> bool }
```

Uses `bool` 3a 76a 85a 99.

74 ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁73 75a▷

```
module type S = sig
  type 'a userdata'
  type srcloc
  type initstate
  type value
    = Nil
    | Number   of float
    | String   of string
    | Function of srcloc * func
    | Userdata of userdata
    | Table    of table
  and func  = value list -> value list (* can also side-effect state *)
  and table = (value, value) Luahash.t
  and userdata  = value userdata'
  and state = { globals : table
              ; fallbacks : (string, value) Hashtbl.t
              ; mutable callstack : activation list
              ; mutable currentloc : Luasrcmap.location option (* supersedes top of stack *)
              ; startup : initstate
              }
  and activation = srcloc * Luasrcmap.location option

  val caml_func : func -> value (* each result unique *)
  val lua_func  : file:string -> linedefined:int -> func -> value
  val srcloc    : file:string -> linedefined:int -> srcloc (* must NOT be reused *)
  val eq        : value -> value -> bool
  val to_string : value -> string
  val activation_strings : state -> activation -> string list
  type objname = Fallback of string | Global of string | Element of string * value
  val objname : state -> value -> objname option
     (* 'fallback', 'global', or 'element', name *)

  val state : unit -> state (* empty state, without even fallbacks *)
  val at_init : state -> string list -> unit  (* run code at startup time *)
  val initcode : state -> (string -> unit) -> unit (* for the implementation only *)
```

Defines:
- `activation_strings`, used in chunks 44b and 62b.
- `at_init`, never used.
- `caml_func`, used in chunks 43b, 88, 90, 91a, 95, and 102.
- `eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
- `initcode`, used in chunk 41.
- `initstate`, never used.
- `lua_func`, never used.
- `objname`, used in chunks 47c, 82b, and 95.
- `srcloc`, used in chunks 13, 39, 63, 82b, and 97.
- `state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
- `to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
- `value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

Uses `activation` 80a, `bool` 3a 76a 85a 99, `file` 99 113, `float` 3a 76a 84a 111b, `func` 5a 64a 78a 80a 90, `int` 3a 76a 85a 99, `list` 3d 76d 86b 99, `location` 127, `name` 127, `option` 3b 76b 86a, `result` 5a 78a 89, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, `unit` 3a 76a 85a, `userdata` 3a 76a 80a 84a, and `userdata'` 80a.

If a library wants to register Lua code to be executed at startup time, it can call `at_init`. No library should ever call `initcode`; that function is reserved for the implementation, which uses it to run the registered code.

Lua tables are not quite like Caml tables, but they are close.

75a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁74 75b▷

```
module Table : sig
  val create : int -> table
  val find   : table -> key:value -> value   (* returns Nil if not found *)
  val bind   : table -> key:value -> data:value -> unit
  val of_list : (string * value) list -> table
end
```

Defines:
`bind`, used in chunks 50, 53, 66, 86b, 92, 95, and 113.
`create`, used in chunks 5b, 53, 66, 78b, 86b, 92, 93, and 104.
`find`, used in chunks 44, 50, 66, 85b, 86b, 91, 92, and 102.
`of_list`, used in chunks 87b, 92, 93, and 99.

Uses `int` 3a 76a 85a 99, `list` 3d 76d 86b 99, `string` 3a 76a 84a 99, `table` 4a 77a 80a 87a, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

Now, for embedding and projection. This is a key, as it completely replaces the stupid stack-based interface found in C. Instead, we use higher-order functions to whip up functionality as needed. If a projection fails, it raises `Projection` giving the value it tried to project and a string indicating what it tried to project to. We re-export `ep` to make it easier for clients to get the `embed` and `project` fields from the locations of their choice. WE MIGHT HAVE TO ADD FLEXIBILITY BY MAKING `type 'a map = state -> ('a, value) ep`.

75b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁75a 76a▷

```
exception Projection of value * string
val projection : value -> string -> 'a
type ('a, 'b, 'c) ep = ('a, 'b, 'c) synonym_for_ep
  = { embed : 'a -> 'b; project : 'b -> 'a; is : 'c -> bool }
type 'a map  = ('a, value, value) ep
type 'a mapf  (* used to build function maps that curry/uncurry *)
```

Defines:
`projection`, used in chunks 6b, 8a, 19, 79b, 99, 104, and 113.

Uses `bool` 3a 76a 85a 99, `map` 83, `mapf` 83, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

These functions enable us to convert the basic types.

76a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁75b 76b▷

```
    val float    : float  map
    val int      : int    map
    val bool     : bool   map
    val string   : string map
    val userdata : userdata map
    val unit     : unit   map
```

Defines:
`bool`, used in chunks 1, 2b, 6b, 8a, 13, 19, 39, 41, 48c, 49b, 73–75, 79b, 80a, 83, 91a, 117, 120, and 127.
`float`, used in chunks 1, 48, 65, 74, 80a, 95, 104, and 120.
`int`, used in chunks 1, 2a, 67b, 69, 74, 75a, 80a, 95, 102, 104, 111b, 113, 117, 120, and 127.
`string`, used in chunks 1, 2, 4, 6b, 8, 10a, 13, 14b, 16, 19–21, 39, 41, 43a, 44b, 48c, 49a, 69–71, 74, 75, 77, 79b, 80a, 82a, 83, 87b, 93–95, 97, 102, 104, 112a, 113, and 127.
`unit`, used in chunks 1, 2a, 10, 11a, 13, 14b, 16, 21, 22a, 24a, 39, 41, 70, 71, 74, 75a, 80a, 89, 93–95, 99, 102, 111b, 113, and 117.
`userdata`, used in chunks 1, 74, 95, 99, and 113.
Uses `map` 83.

To convert a value of option type, we represent `None` as `Nil`. Woe betide you if `Nil` is a valid value of your type! We won't see it.

76b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁76a 76c▷

```
    val option : 'a map -> 'a option map
```

Defines:
`option`, used in chunks 1, 74, 80a, 102, 113, and 127.
Uses `map` 83.

To project with a default value, we provide `default` v `t`, which behaves just as `t` except it projects `Nil` to v.

76c ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁76b 76d▷

```
    val default : 'a -> 'a map -> 'a map
```

Defines:
`default`, used in chunks 86b, 95, and 102.
Uses `map` 83.

To embed a list of values, we produce a table with a binding of the length to the name `n` and bindings of the values to the numbers $1..n$. To project a Lua table down to a list, we first look to see if the table binds the name `n`. If so, we take that to be the number of elements; otherwise we use the table's population. (In the latter case, lists cannot contain `nil`.) This way, users are free to include `n` or not as they choose.

76d ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁76c 77a▷

```
    val list    : 'a map -> 'a list map   (* does not project nil *)
    val optlist : 'a map -> 'a list map   (* projects nil to empty list *)
```

Defines:
`list`, used in chunks 1, 2a, 4, 5, 7a, 10a, 13, 14b, 16, 21a, 39, 41, 42a, 47c, 51, 58, 60d, 63, 70, 71, 74, 75a, 77, 78, 80a, 83, 88–91, 94, 102, 117, and 127.
`optlist`, never used.
Uses `map` 83 and `nil` 65.

N.B. `optlist t = default [] (list t)`.

If for some reason a Caml function operates on Lua values, we need an identity pair. We also enable functions that expect tables.

77a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁76d 77b▷

```
    val value  : value map
    val table  : table map
```

Defines:
`table`, used in chunks 1, 2a, 5a, 10a, 13, 39, 43a, 53, 66, 74, 75a, 78a, 81b, 86b, 87b, 91c, 95, 117, and 120.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.
Uses `map` 83.

A common case is to use a table as a record, with string keys and homogenous values.

77b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁77a 77c▷

```
    val record : 'a map -> (string * 'a) list map
```

Defines:
`record`, never used.
Uses `list` 3d 76d 86b 99, `map` 83, and `string` 3a 76a 84a 99.

Another common case is to represent an enumeration type using strings. The string passed to `enum` is the name of the type, which is used in projection errors. The list passed to `enum` must contain *every* value of type `'a`, which must be comparable using `=`. To do otherwise is to risk an assertion failure during embedding.

77c ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁77b 78a▷

```
    val enum   : string -> (string * 'a) list -> 'a map
```

Defines:
`enum`, never used.
Uses `list` 3d 76d 86b 99, `map` 83, and `string` 3a 76a 84a 99.

Here is the support for converting functions. First, if one wants a Lua function to be curried (as the Caml functions are), one can simply use `-->`. There's a small gotcha, in that we can't make `-->` right associative. That's OK, as it probably shouldn't be used for curried functions.

For curried functions that should take lists of arguments in Lua, we use `**->`, `pfunc`, `func`, `closure`, and `result`. The idea is this: if we have a Caml function type `t -> u -> v -> w`, we can turn this into a Lua function of three arguments by using the embedding/projection pair produced by

```
pfunc (t **-> u **-> v **-> result w)
```

78a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁77c 78b▷

```
    val ( -->  ) : 'a map  -> 'b map  -> ('a -> 'b) map
    val ( **-> ) : 'a map  -> 'b mapf -> ('a -> 'b) mapf
    val result   : 'a map  -> 'a mapf
    val resultvs : value list mapf                    (* functions returning value lists*)
    val resultpair:'a map  -> 'b map  -> ('a * 'b)       mapf
    val dots_arrow:'a map  -> 'b map  -> ('a list -> 'b) mapf     (* varargs functions *)
    val results  : ('a -> value list) -> (value list -> 'a) -> 'a mapf
                                    (* 'a represents multiple results (general case) *)
    val func     : 'a mapf -> 'a map                  (* function *)
    val closure  : 'a mapf -> 'a map                  (* function or table+apply method *)
    val efunc    : 'a mapf -> 'a -> value             (* efunc f = (closure f).embed *)
```

Defines:
`**->`, used in chunks 91a, 95, 99, 100, 102, 103, 111, 113, and 116b.
`-->`, used in chunks 99 and 100.
`closure`, never used.
`dots_arrow`, used in chunks 103 and 116b.
`efunc`, used in chunks 91a, 95, 99, 102, 111b, and 113.
`func`, used in chunks 1, 74, and 99.
`result`, used in chunks 1, 65, 74, 88, 90, 91a, 95, 99, 103, 111c, and 116b.
`resultpair`, never used.
`results`, used in chunks 63, 64b, and 117.
`resultvs`, used in chunks 95 and 102.
Uses `apply` 10a 21a 46, `list` 3d 76d 86b 99, `map` 83, `mapf` 83, `table` 4a 77a 80a 87a, `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c, and `varargs` 127.

Support for dynamically typed Lua functions Type-based dispatch defines several alternatives for a function and at each call, chooses the right function based on the types of the arguments.

78b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁78a 79a▷

```
    type alt                             (* an alternative *)
    val alt    : 'a mapf -> 'a -> alt    (* create an alternative *)
    val choose : alt list -> value       (* dispatch on type/number of args *)
```

Defines:
`alt`, used in chunk 113.
`choose`, used in chunk 113.
Uses `create` 2a 75a 117 120, `list` 3d 76d 86b 99, `mapf` 83, `number` 84b, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

It's also possible to combine multiple types for a single argument. The idea is just like parsing combinators, and we use notation from Jeroen Fokker's paper *Functional Parsers*. We can use the choice operator `<|>` to combine two maps. To project, map `t <|> t'` projects using `t` if `t` claims to recognize the argument. If `t` does not recognize the argument, the map projects using `t'`. To embed, map `t <|> t'` always embeds using `t'`.

We can use the continuation operator `>>=` to apply a function to a value after projection. To project, the map `t <@ f` applies `f` to the result of projecting with `t`. Because function `f` cannot be inverted, the map `t <@ f` is not capable of embedding. It is therefore useful primarily on the left-hand side of the `<|>` operator.

79a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁78b 79b▷

```
  val ( <|> ) : 'a map -> 'a map -> 'a map
  val ( <@ ) : 'a map -> ('a -> 'b) -> 'b map   (* apply continuation after project *)
end
```

Defines:
`<@`, never used.
`<|>`, never used.
Uses `apply` 10a 21a 46 and `map` 83.

For the rest of the interface, we can make Lua values by supplying an appropriate `userdata` type.

79b ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁79a 126a▷

```
module type USERDATA = sig
  type 'a t                              (* type parameter will be Lua value *)
  val tname : string  (* name of this type, for projection errors *)
  val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  val to_string : ('a -> string) -> 'a t -> string
end
```

Defines:
`eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.
`tname`, used in chunks 24a, 25, 28, 84a, and 115.
`to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.
Uses `bool` 3a 76a 85a 99, `name` 127, `projection` 2b 75b 83, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

79c ⟨*luavalue.mli* 79c⟩≡
⟨*signatures* 19⟩

```
module Make (U : USERDATA) : S with type 'a userdata'  = 'a U.t
```

Uses `userdata'` 80a.

7.1 Implementation

79d ⟨*luavalue.ml* 79d⟩≡
⟨*signatures* 19⟩

```
module Make (U : USERDATA) : S with type 'a userdata'  = 'a U.t
= struct
  ⟨value toplevel 80a⟩
end
```

Uses `userdata'` 80a.

I have to repeat the datatype definition.

80a ⟨*value toplevel* 80a⟩≡ (79d) 80b▷

```
type 'a userdata'  = 'a U.t
type srcloc = int * string * int (* unique id, filename, linedefined *)
type value
  = Nil
  | Number   of float
  | String   of string
  | Function of srcloc * func
  | Userdata of userdata
  | Table    of table
and func  = value list -> value list
and table = (value, value) Luahash.t
and userdata  = value userdata'
and state = { globals : table
            ; fallbacks : (string, value) Hashtbl.t
            ; mutable callstack : activation list
            ; mutable currentloc : Luasrcmap.location option (* supersedes top of stack *)
            ; startup : initstate
            }
and initstate =
  { mutable init_strings : (string -> unit) -> unit; mutable initialized : bool }
and activation = srcloc * Luasrcmap.location option
```

Defines:
`activation`, used in chunks 1, 13, 39, and 74.
`func`, used in chunks 1, 74, and 99.
`initstate`, never used.
`srcloc`, used in chunks 13, 39, 63, 82b, and 97.
`state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
`table`, used in chunks 1, 2a, 5a, 10a, 13, 39, 43a, 53, 66, 74, 75a, 78a, 81b, 86b, 87b, 91c, 95, 117, and 120.
`userdata`, used in chunks 1, 74, 95, 99, and 113.
`userdata'`, used in chunks 1, 11, 13, 17, 22–24, 26, 27, 32, 34, 36, 40, 41, 74, 79, 98a, 100, 112a, and 116b.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.
Uses `bool` 3a 76a 85a 99, `float` 3a 76a 84a 111b, `int` 3a 76a 85a 99, `list` 3d 76d 86b 99, `location` 127, `option` 3b 76b 86a, `string` 3a 76a 84a 99, and `unit` 3a 76a 85a.

We need suitable equality.

80b ⟨*value toplevel* 80a⟩+≡ (79d) ◁80a 80c▷

```
let rec eq x y = match x, y with
| Nil,           Nil            -> true
| Number x,      Number y       -> x = y
| String x,      String y       -> x = y
| Userdata x,    Userdata y     -> U.eq eq x y
| Table x,       Table y        -> x == y
| Function ((x, _, _), _),
           Function ((y, _, _), _) -> x = y
| _,             _              -> false
```

Defines:
`eq`, used in chunks 24a, 25, 49b, 82a, 92, 115, 117, and 120.

Once we have equality, we can make tables.

80c ⟨*value toplevel* 80a⟩+≡ (79d) ◁80b 81a▷
```
⟨table definition 92⟩
```

To make this work, every function gets a unique id. No exceptions.

81a ⟨*value toplevel* 80a⟩+≡ (79d) ◁80c 81b▷
```
let srcloc =
  let n = ref 0 in
  fun ~file ~linedefined:line -> (n := !n + 1; (!n, file, line))
let lua_func ~file ~linedefined:line f = Function (srcloc file line, f)
let caml_func = lua_func ~file:"(OCaml)" ~linedefined:(-1)
```
Defines:
- `caml_func`, used in chunks 43b, 88, 90, 91a, 95, and 102.
- `lua_func`, never used.
- `srcloc`, used in chunks 13, 39, 63, 82b, and 97.

Uses `file` 99 113.

81b ⟨*value toplevel* 80a⟩+≡ (79d) ◁81a 82a▷
```
let luastring_of_float x =
  let s = string_of_float x in
  if String.get s (String.length s - 1) = '.' then
    String.sub s 0 (String.length s - 1)
  else
    s

let rec to_string = function
  | Nil             -> "nil"
  | Number x        -> luastring_of_float x
  | String s        -> s
  | Function (_, _) -> "function"
  | Userdata u      -> U.to_string to_string u
  | Table t         -> "table"
```
Defines:
- `luastring_of_float`, used in chunk 84a.
- `to_string`, used in chunks 24a, 25, 43a, 47c, 66, 68, 71, 82b, 95, and 115.

Uses `length` 84b, `nil` 65, and `table` 4a 77a 80a 87a.

82a ⟨*value toplevel* 80a⟩+≡ (79d) ◁81b 82b▷

```
type objname = Fallback of string | Global of string | Element of string * value
let key_matching iter t needle =
  let r = ref None in
  iter (fun k v -> if eq needle v then r := Some k else ()) t;
  !r
let objname g needle =
  match key_matching Hashtbl.iter g.fallbacks needle with
  | Some s -> Some (Fallback s)
  | None -> match key_matching Luahash.iter g.globals needle with
    | Some (String s) -> Some (Global s)
    | _ ->
        let r = ref None in
        Luahash.iter (fun k v ->
          match !r with
          | None -> (match k, v with
            | String n, Table t ->
                (match key_matching Luahash.iter t needle with
                | Some v -> r := Some (Element (n, v))
                | None -> ())
            | k, v -> ())
          | Some _ -> ()) g.globals;
        !r
```

Defines:
`key_matching`, never used.
`objname`, used in chunks 47c, 82b, and 95.

Uses `eq` 1 6b 8a 19 74 79b 80b, `iter` 117 120, `string` 3a 76a 84a 99,
and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

82b ⟨*value toplevel* 80a⟩+≡ (79d) ◁82a 83▷

```
let activation_strings g ((uid, file, line) as srcloc, current) =
  let first tail = match objname g (Function (srcloc, fun _ -> assert false)) with
  | Some (Fallback n) -> "`" :: n :: "' fallback" :: tail
  | Some (Global n)   -> "function " :: n :: tail
  | Some (Element (t, String n)) -> "function " :: t :: "." :: n :: tail
  | Some (Element (t, v))    -> "function " :: t :: "[" :: to_string v :: "]" :: tail
  | None -> "unknown function" :: tail
  in let last = match current with
  | None -> " defined in file " :: file ::
             (if line > 0 then [" at line "; string_of_int line ] else [])
(*  | Some (f, l, c) when f = file ->
      [" at line "; string_of_int l; " column "; string_of_int c]
*)
  | Some (f, l, c) ->
      [" in file "; f; ", line "; string_of_int l; " column "; string_of_int c]
  in match line with
  | 0  -> "main of " :: file :: last
  | -1 -> first [" "; file]
  | _  -> first last
```

Defines:
  activation_strings, used in chunks 44b and 62b.
Uses fallback 10a 13 21a 39 44c, file 99 113, first 117 120, objname 1 1 74 74 82a 82a,
  srcloc 1 1 74 74 80a 81a, and to_string 1 6b 8a 19 74 79b 81b.

7.2 Embedding and projection

Now, for embedding and projection. More repeats.

The predicate for a value tells whether that value can be projected. The predicate for a function (`mapf`) tells whether a list of arguments would be accepted by that function. When higher-order functions come into play, these predicates are only an approximation, because they dispatch only on the top-level type parameter. So for example, every function parameter is treated as equivalent to any other. Thus, it is not possible to dispatch on the type of a function but only on whether it is a function or some other beast. A similar limitation applies to lists.

83 ⟨*value toplevel* 80a⟩+≡ (79d) ◁82b 84a▷

```
exception Projection of value * string
let projection v s = raise (Projection(v, s))
type ('a, 'b, 'c) ep = ('a, 'b, 'c) synonym_for_ep
  = { embed : 'a -> 'b; project : 'b -> 'a; is : 'c -> bool }
type 'a map  = ('a, value, value) ep
type 'a mapf = ('a, value list -> value list, value list) ep
```

Defines:
`map`, used in chunks 2–6, 13, 16, 39, 43a, 46, 47c, 60b, 65, 75–79, 86b, 87b, 89, 90, 97, 99, 102, and 107.
`mapf`, used in chunks 2b, 5, 75b, 78, 89, and 90.
`projection`, used in chunks 6b, 8a, 19, 79b, 99, 104, and 113.

Uses `bool` 3a 76a 85a 99, `list` 3d 76d 86b 99, `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

Here are the embedding/projection pairs for the simple types. There are subtle mismatches between Caml types and Lua types, so I do my best to patch them.

84a ⟨*value toplevel* 80a⟩+≡ (79d) ◁83 85a▷

```
let userdata = { embed = (fun x -> Userdata x)
               ; project = (function Userdata x -> x
                                  | v -> raise (Projection (v, U.tname)))
               ; is = (function Userdata _ -> true | _ -> false)
               }

let string = { embed = (fun s -> String s)
             ; project = (function String s -> s
                                 | Number x -> luastring_of_float x
                                 | v -> raise (Projection (v, "string")))
             ; is = (function String _ | Number _ -> true | _ -> false)
             }

let is_float_literal s =
  try Luafloat.length (Lexing.from_string s) = String.length s
  with Failure _ -> false
let pervasive_float = float
let float =
  { embed = (fun x -> Number x)
  ; project = (function Number x -> x
                      | String s when is_float_literal s -> float_of_string s
                      | v -> raise (Projection (v, "float")))
  ; is = (function Number _ -> true | String s -> is_float_literal s | _ -> false)
  }
```

Defines:
`float`, used in chunks 1, 48, 65, 74, 80a, 95, 104, and 120.
`is_float_literal`, never used.
`pervasive_float`, used in chunks 85a and 86b.
`string`, used in chunks 1, 2, 4, 6b, 8, 10a, 13, 14b, 16, 19–21, 39, 41, 43a, 44b, 48c, 49a, 69–71, 74, 75, 77, 79b, 80a, 82a, 83, 87b, 93–95, 97, 102, 104, 112a, 113, and 127.
`userdata`, used in chunks 1, 74, 95, 99, and 113.
Uses `length` 84b, `luastring_of_float` 81b, and `tname` 6b 8a 19 79b.

84b ⟨*luafloat.mll* 84b⟩≡

```
let digit       = ['0'-'9']
let sign        = ['+' '-']
let exp         = ['e''E'] sign? digit+
let number      = sign? digit+ exp?
                | sign? digit+ '.' digit+ exp?
rule length = parse number { Lexing.lexeme_end lexbuf } | _ { -1 }
```

Defines:
`digit`, never used.
`exp`, used in chunks 51, 54b, 55b, and 69.
`length`, used in chunks 47c, 53, 55c, 56a, 59, 60, 63, 66, 67b, 69, 71, 81b, 84a, 86b, 92, 97, 99, 102, 104, 107, 113, and 120.
`number`, used in chunks 5b, 43a, 78b, 95, 102, and 117.
`sign`, never used.
Uses `parse` 102.

85a ⟨*value toplevel* 80a⟩+≡ (79d) ◁84a 85b▷
```
let to_int x =
  let n = truncate x in
  if pervasive_float n = x then n else raise (Projection (Number x, "int"))

let int   = { embed = (fun n -> Number (pervasive_float n))
            ; project = (function Number x -> to_int x
                                | v -> raise (Projection (v, "int")))
            ; is = (function Number x -> pervasive_float (truncate x) = x | _ -> false)
            }

let bool  = { embed = (fun b -> if b then String "t" else Nil)
            ; project = (function Nil -> false | _ -> true)
            ; is = (fun _ -> true)
            }
let unit =  { embed = (fun () -> Nil)
            ; project = (function Nil -> () | v -> raise (Projection (v, "unit")))
            ; is = (function Nil -> true | _ -> false)
            }
```
Defines:
`bool`, used in chunks 1, 2b, 6b, 8a, 13, 19, 39, 41, 48c, 49b, 73–75, 79b, 80a, 83, 91a, 117, 120, and 127.
`int`, used in chunks 1, 2a, 67b, 69, 74, 75a, 80a, 95, 102, 104, 111b, 113, 117, 120, and 127.
`to_int`, used in chunk 86b.
`unit`, used in chunks 1, 2a, 10, 11a, 13, 14b, 16, 21, 22a, 24a, 39, 41, 70, 71, 74, 75a, 80a, 89, 93–95, 99, 102, 111b, 113, and 117.
Uses `pervasive_float` 84a.

85b ⟨*value toplevel* 80a⟩+≡ (79d) ◁85a 86a▷
```
let enum typename pairs =
  { embed = (fun v' -> try String (fst (List.find (fun (k, v) -> v = v') pairs))
                       with Not_found -> assert false)
  ; project = (function String k ->
                 (try List.assoc k pairs
                  with Not_found -> raise (Projection (String k, typename)))
               | v -> raise (Projection (v, typename)))
  ; is = (function String k -> List.mem_assoc k pairs | _ -> false)
  }
```
Defines:
`enum`, never used.
Uses `find` 2a 75a 107 117 120.

86a ⟨*value toplevel* 80a⟩+≡ (79d) ◁85b 86b▷

```
let option t = { embed = (function None -> Nil | Some x -> t.embed x)
               ; project = (function Nil -> None | v -> Some (t.project v))
               ; is = (function Nil -> true | v -> t.is v)
               }
let default d t =
  { embed = t.embed
  ; project = (function Nil -> d | v -> t.project v)
  ; is = (function Nil -> true | v -> t.is v)
  }
```

Defines:
`default`, used in chunks 86b, 95, and 102.
`option`, used in chunks 1, 74, 80a, 102, 113, and 127.

86b ⟨*value toplevel* 80a⟩+≡ (79d) ◁86a 87a▷

```
let list (ty : 'a map) =
  let table l =
    let n = List.length l in
    let t = Table.create n in
    let rec set_elems next = function
      | [] -> Table.bind t (String "n") (Number (pervasive_float n))
      | e :: es -> ( Table.bind t (Number next) (ty.embed e)
                   ; set_elems (next +. 1.0) es)
    in  (set_elems 1.0 l; Table t)
  in
  let untable (t:table) =
    let n = match Table.find t (String "n") with
    | Number x -> to_int x
    | _ -> Luahash.population t  in
    let rec elems i =
      if i > n then []
      else ty.project (Table.find t (Number (pervasive_float i))) :: elems (i + 1) in
    elems 1
  in { embed = table; project = (function Table t -> untable t
                                        | v -> raise (Projection (v, "list")));
       is = (function Table t -> true | _ -> false) }
let optlist ty = default [] (list ty)
```

Defines:
`list`, used in chunks 1, 2a, 4, 5, 7a, 10a, 13, 14b, 16, 21a, 39, 41, 42a, 47c, 51, 58, 60d, 63, 70, 71, 74, 75a, 77, 78, 80a, 83, 88–91, 94, 102, 117, and 127.
`optlist`, never used.
Uses `bind` 2a 75a, `create` 2a 75a 117 120, `default` 3c 76c 86a, `find` 2a 75a 107 117 120, `length` 84b, `map` 83, `next` 117 120, `pervasive_float` 84a, `population` 117 120, `table` 4a 77a 80a 87a, and `to_int` 85a.

87a ⟨*value toplevel* 80a⟩+≡ (79d) ◁86b 87b▷

```
  let value = { embed = (fun x -> x); project = (fun x -> x); is = (fun _ -> true) }
  let table = { embed = (fun x -> Table x)
              ; project = (function Table t -> t | v -> raise (Projection (v, "table")))
              ; is = (function Table t -> true | _ -> false)
              }
```

Defines:
`table`, used in chunks 1, 2a, 5a, 10a, 13, 39, 43a, 53, 66, 74, 75a, 78a, 81b, 86b, 87b, 91c, 95, 117, and 120.
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

87b ⟨*value toplevel* 80a⟩+≡ (79d) ◁87a 88▷

```
  let projectRecord ty v = match v with
  | Table t ->
      let rec addpairs (k, v) =
        (string.project v, ty.project v) ::
        try addpairs (Luahash.next t k) with Not_found -> [] in
      (try addpairs (Luahash.first t) with Not_found -> [])
  | _ -> raise (Projection (v, "table (as record)"))

  let record ty =
    { embed = (fun pairs ->
               Table (Table.of_list (List.map (fun (k, v) -> (k, ty.embed v)) pairs)))
    ; project = projectRecord ty
    ; is = table.is
    }
```

Defines:
`projectRecord`, never used.
`record`, never used.
Uses `first` 117 120, `map` 83, `next` 117 120, `of_list` 2a 75a, `string` 3a 76a 84a 99, and `table` 4a 77a 80a 87a.

Here is a simple function from one argument to one result. We implement the usual Lua game of "adjusting" the argument list.

88 ⟨*value toplevel* 80a⟩+≡ (79d) ◁87b 89▷

```
let take1 = function  (* take one value from a list of arguments *)
  | [] -> Nil
  | h::t -> h

let take2 = function [] -> Nil, Nil | v :: vs -> v, take1 vs

let const f s = f
let (-->) arg result =
  { embed =   (fun f ->
               caml_func (fun args -> [result.embed (f (arg.project (take1 args)))]))
  ; project = (function Function (_, f) ->
                           fun x -> result.project (take1 (f [arg.embed x]))
                       | v -> raise (Projection (v, "function")))
  ; is = (function Function (_, _) -> true | _ -> false)
  }
```

Defines:
`-->`, used in chunks 99 and 100.
`const`, never used.
`take1`, used in chunks 89 and 90.
`take2`, used in chunk 89.

Uses `caml_func` 1 74 81a, `list` 3d 76d 86b 99, `result` 5a 78a 89, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

What we have above would be enough, except that Caml likes functions to be curried and Lua likes them uncurried. We provide `**->`, `results`, and `func` for embedding and projecting curried functions. The functions are a bit subtle, but if you follow the types, you shouldn't be too baffled.

89 ⟨*value toplevel* 80a⟩+≡ (79d) ◁88 90▷

```
let ( **-> ) (firstarg : 'a map) (lastargs : 'b mapf) : ('a -> 'b) mapf =
  let apply (f : 'a -> 'b) args =
    let h, t = match args with [] -> Nil, [] | h :: t -> h, t in
    let f = f (firstarg.project h) in
    lastargs.embed f t
  in
  let unapp f' =
    fun (x : 'a) -> lastargs.project (function t -> f' (firstarg.embed x :: t)) in
  (* function can match even if args are defaulted, but not if too many args *)
  let is args =
    let h, t = match args with [] -> Nil, [] | h :: t -> h, t in
    firstarg.is h && lastargs.is t in
  { embed = apply; project = unapp; is = is }

let results (a_to_values : 'a -> value list) (a_of_values : value list -> 'a) =
  { embed   = (fun (a:'a) -> fun lua_args -> a_to_values a);
    project = (fun f_lua -> (a_of_values (f_lua []) : 'a));
    is = (function [] -> true | _ :: _ -> false)
  }

let (<<) f g = fun x -> f (g x)

let result r = results (fun v -> [r.embed v]) (r.project << take1)
let resultvs = results (fun l -> l) (fun l -> l)
let resultpair a b =
  let em (x, y) = [a.embed x; b.embed y] in
  let pr vs =
    let x, y = match vs with
    | [] -> Nil, Nil
    | [x] -> x, Nil
    | x :: y :: _ -> x, y in
    (a.project x, b.project y) in
  results em pr

(* other possibilities not exposed in interface *)
let result2 r1 r2 = results (fun (v1, v2) -> [r1.embed v1; r2.embed v2])
                            ((fun (l1, l2) -> r1.project l1, r2.project l2) << take2)
let runit =
  results (fun () -> [])
          (function [] -> () | h :: _ -> raise (Projection (h, "unit result")))
```

Defines:
`**->`, used in chunks 91a, 95, 99, 100, 102, 103, 111, 113, and 116b.
`<<`, used in chunks 32 and 33.
`result`, used in chunks 1, 65, 74, 88, 90, 91a, 95, 99, 103, 111c, and 116b.
`result2`, never used.

resultpair, never used.
results, used in chunks 63, 64b, and 117.
resultvs, used in chunks 95 and 102.
runit, never used.
Uses apply 10a 21a 46, list 3d 76d 86b 99, map 83, mapf 83, take1 88, take2 88,
unit 3a 76a 85a, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

90 ⟨*value toplevel* 80a⟩+≡ (79d) ◁89 91a▷

```
let dots_arrow (varargs : 'a map) (result : 'b map) : ('a list -> 'b) mapf =
  let apply (f : 'a list -> 'b) =
    fun (args : value list) ->
      [result.embed (f (List.map varargs.project args))] in
  let unapp (f' : value list -> value list) =
    fun (args : 'a list) ->
      result.project (take1 (f' (List.map varargs.embed args))) in
  { embed = apply; project = unapp; is = List.for_all varargs.is }


let func (arrow : 'a mapf) : ('a map) =
  { embed   = (fun (f : 'a) -> caml_func (arrow.embed f))
  ; project = (function Function (_, f) -> (arrow.project f : 'a)
                      | v -> raise (Projection (v, "function")))
  ; is = (function Function(_, _) -> true | _ -> false)
  }

let closure (arrow : 'a mapf) : ('a map) =
  { embed   = (fun (f : 'a) -> caml_func (arrow.embed f))
  ; project = (function Function (_, f) -> (arrow.project f : 'a)
                      | Table t as v -> (⟨project table t into a function 91c⟩)
                      | v -> raise (Projection (v, "function")))
  ; is = (function Function(_, _) -> true | Table t -> ⟨table t is a closure 91d⟩
                 | _ -> false)
  }

let efunc t f = (closure t).embed f
```

Defines:
closure, never used.
dots_arrow, used in chunks 103 and 116b.
efunc, used in chunks 91a, 95, 99, 102, 111b, and 113.
func, used in chunks 1, 74, and 99.
Uses apply 10a 21a 46, caml_func 1 74 81a, list 3d 76d 86b 99, map 83, mapf 83,
result 5a 78a 89, take1 88, value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c,
and varargs 127.

91a ⟨*value toplevel* 80a⟩+≡ (79d) ◁90 91b▷
```
  type alt = (value list -> value list) * (value list -> bool)
  let alt t f = (t.embed f, t.is)
  let choose alts =
    let run args =
      let f = try fst (List.find (fun (_, is) -> is args) alts)
              with Not_found ->
                let args = (list value).embed args in
                raise (Projection (args, "arguments matching alternatives")) in
      f args in
    caml_func run

  let lf = efunc (list value **-> result (list value)) List.rev
```
Defines:
  `alt`, used in chunk 113.
  `choose`, used in chunk 113.
  `lf`, never used.
Uses `**->` 5a 78a 89, `bool` 3a 76a 85a 99, `caml_func` 1 74 81a, `efunc` 5a 78a 90, `find` 2a 75a 107 117 120, `list` 3d 76d 86b 99, `result` 5a 78a 89, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

91b ⟨*value toplevel* 80a⟩+≡ (79d) ◁91a 93▷
```
  let ( <|> ) t t' =
    { project = (fun v -> if t.is v then t.project v else t'.project v)
    ; embed   = t'.embed
    ; is      = (fun v -> t.is v || t'.is v)
    }

  let ( <@ ) t k =
    { project = (fun v -> k (t.project v))
    ; embed   = (fun _ -> assert false)
    ; is      = t.is
    }
```
Defines:
  `<@`, never used.
  `<|>`, never used.

A table is a function if it has an apply method that is a function.

91c ⟨*project table* `t` *into a function* 91c⟩≡ (90)
```
  let f = try Table.find t (String "apply")
          with Not_found -> raise (Projection (v, "function"))  in
  match f with
  | Function (_, f) -> arrow.project (fun vs -> f (v :: vs))
  | v -> raise (Projection (v, "'apply' element of table as function"))
```
Uses `apply` 10a 21a 46, `find` 2a 75a 107 117 120, and `table` 4a 77a 80a 87a.

91d ⟨*table* t *is a closure* 91d⟩≡ (90)
```
  (try
    match Table.find t (String "apply") with
    | Function (_, _) -> true
    | _ -> false
  with Not_found -> false)
```
Uses apply 10a 21a 46 and find 2a 75a 107 117 120.

A Lua table is very nearly a Caml hash table, except it never has multiple elements.

92 ⟨*table definition* 92⟩≡ (80c)
```
  module Table = struct
    let create = Luahash.create eq
    let find t ~key:k = try Luahash.find t k with Not_found -> Nil
    let bind t ~key:k ~data:v =
      match v with
      | Nil -> Luahash.remove t k
      | _ -> Luahash.replace t k v
    let of_list l =
      let t = create (List.length l) in
      let _ = List.iter (fun (k, v) -> bind t (String k) v) l in
      t
  end
```
Uses bind 2a 75a, create 2a 75a 117 120, eq 1 6b 8a 19 74 79b 80b, find 2a 75a 107 117 120, iter 117 120, length 84b, of_list 2a 75a, remove 117 120, and replace 117 120.

7.3 Initialization at startup time

93 ⟨*value toplevel* 80a⟩+≡ (79d) ◁91b

```
module StringList = struct
  type t = (string -> unit) -> unit
  let empty f = ()
  let of_list l f = List.iter f l
  let append l1 l2 f = l1 f; l2 f
end

let state () =
 { globals = Table.create 50; fallbacks = Hashtbl.create 10; callstack = [];
   currentloc = None;
   startup = { init_strings = StringList.empty; initialized = false; }
 }

let at_init g ss =
  if g.startup.initialized then
    (prerr_endline
       "Internal Lua-ML error: called at_init after initialiation was complete";
     exit(1))
  else
    g.startup.init_strings <-
      StringList.append g.startup.init_strings (StringList.of_list ss)

let initcode g =
  if g.startup.initialized then
    (prerr_endline "Internal Lua-ML error: a naughty client called initcode";
     exit(1))
  else
    let code = g.startup.init_strings in
    begin
      g.startup.initialized <- true;
      g.startup.init_strings <- StringList.empty;
      code
    end
```

Defines:
  `at_init`, never used.
  `initcode`, used in chunk 41.
  `state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.
Uses `append` 52a, `create` 2a 75a 117 120, `error` 10a 13 21a 39 42a, `iter` 117 120,
  `of_list` 2a 75a, `string` 3a 76a 84a 99, and `unit` 3a 76a 85a.

94 ⟨*luabaselib.mli* 94⟩≡

```
module Add (MakeParser : Luaparser.MAKER) (I : Luainterp.S) : sig
  include Luainterp.S
  module Parser : Luaparser.S with type chunk = Ast.chunk
  val do_lexbuf : sourcename:string -> state -> Lexing.lexbuf -> value list
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
  val mk        : unit -> state  (* builds state and runs startup code *)
end with module Value = I.Value
```

Defines:
`do_lexbuf`, never used.
`dofile`, used in chunk 95.
`dostring`, used in chunk 95.
`mk`, used in chunks 65, 95, and 97.

Uses `chunk` 7a 64a 127, `list` 3d 76d 86b 99, `state` 1 13 39 41 70 71 74 80a 93 112b, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

95 ⟨*luabaselib.ml* 95⟩≡
```
module Add (MakeParser : Luaparser.MAKER) (I : Luainterp.S) = struct
  module Parser = MakeParser (I.Ast)
  module P = Parser
  module V = I.Value
  ⟨toplevel 97⟩
  let ( **-> ) = V.( **-> )
  let ( **->> ) x y = x **-> V.result y

  let next t key =
    let k, v =
      try match key with
      | V.Nil -> Luahash.first t
      | _   -> Luahash.next t key
      with Not_found -> V.Nil, V.Nil
    in [k; v]

  let objname g v =
    let tail = [] in
    let ss = match V.objname g v with
    | Some (V.Fallback n) -> "‘" :: n :: "’ fallback" :: tail
    | Some (V.Global n)   -> "function " :: n :: tail
    | Some (V.Element (t, V.String n)) -> "function " :: t :: "." :: n :: tail
    | Some (V.Element (t, v))    -> "function " :: t :: "[" :: V.to_string v :: "]" :: tail
    | None -> "unnamed " :: V.to_string v :: tail in
    String.concat "" ss



  let luabaselib g =
    [ "dofile",      V.efunc (V.string **-> V.resultvs) (dofile g)
    ; "dostring",    V.efunc (V.string **-> V.resultvs) (dostring g)
        (* should catch Sys_error and turn into an error fallback... *)
    ; "size",        V.efunc (V.table **->> V.int) Luahash.population
    ; "next",        V.efunc (V.table **->  V.value **-> V.resultvs) next
    ; "nextvar",     V.efunc (V.value **->  V.resultvs) (fun x -> next g.V.globals x)
    ; "tostring",    V.efunc (V.value **->> V.string) V.to_string
    ; "objname",     V.efunc (V.value **->> V.string) (objname g)
    ; "print",       V.caml_func
                       (fun args ->
                         List.iter (fun x -> print_endline (V.to_string x)) args;
                         flush stdout;
                         [])
    ; "tonumber",    V.efunc (V.float **->> V.float) (fun x -> x)
    ; "type",        V.efunc (V.value **->> V.string)
                     (function
                     | V.Nil            -> "nil"
                     | V.Number _       -> "number"
                     | V.String _       -> "string"
                     | V.Table _        -> "table"
```

```
                          | V.Function (_,_) -> "function"
                          | V.Userdata _     -> "userdata")
      ; "assert",         V.efunc (V.value **-> V.default "" V.string **->> V.unit)
                          (fun c msg -> match c with
                          | V.Nil -> I.error ("assertion failed: " ^ msg)
                          | _ -> ())
      ; "error",          V.efunc (V.string **->> V.unit) I.error
      ; "setglobal",      V.efunc (V.value **-> V.value **->> V.unit)
                          (fun k v -> V.Table.bind g.V.globals k v)
      ; "getglobal",      V.efunc (V.value **->> V.value) (I.getglobal g)
      ; "setfallback", V.efunc (V.string **-> V.value **->> V.value) (I.setfallback g)
      ]

    include I
    let mk () =
      let g, init = I.pre_mk () in
      I.register_globals (luabaselib g) g;
      init (fun s -> ignore (dostring g s));
      g
  end
```

Uses **-> 5a 78a 89, **->> 99, bind 2a 75a, caml_func 1 74 81a, concat 49a, default 3c 76c 86a, dofile 14b 16 70 71 94 97, dostring 14b 16 70 71 94 97, efunc 5a 78a 90, error 10a 13 21a 39 42a, fallback 10a 13 21a 39 44c, first 117 120, float 3a 76a 84a 111b, getglobal 10a 13 21a 39 50b, init 10b 11a 21b 22a, int 3a 76a 85a 99, iter 117 120, mk 14b 16 70 71 94, next 117 120, nil 65, number 84b, objname 1 1 74 74 82a 82a, population 117 120, pre_mk 13 39, register_globals 10a 13 21a 39 66, result 5a 78a 89, resultvs 5a 78a 89, setfallback 10a 13 21a 39 44a, setglobal 50b 113, string 3a 76a 84a 99, table 4a 77a 80a 87a, to_string 1 6b 8a 19 74 79b 81b, unit 3a 76a 85a, userdata 3a 76a 80a 84a, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

97 ⟨*toplevel* 97⟩≡ (95)

```
let lex map buf = Luascanner.token buf map
let do_lexbuf ~sourcename:filename g buf =
  let map = Luasrcmap.mk () in
  let _ = Luasrcmap.sync map 0 (filename, 1, 1) in
  try
    let chunks = P.chunks (lex map) buf in
    let pgm = I.compile ~srcdbg:(map, false) chunks g in
    match pgm () with
    | [] -> [I.Value.String "executed without errors"]
    | answers -> answers
  with
  | Parsing.Parse_error ->
      let file, line, _ = Luasrcmap.last map in begin
        prerr_string file;
        prerr_string ", line ";
        prerr_int line;
        prerr_endline ": syntax error";
        []
      end
  | I.Error s -> (prerr_endline "Lua interpreter halted with error"; [])
  | I.Value.Projection (v, w) -> (prerr_endline ("error projecting to " ^ w); [])


let dostring g s =
  let abbreviate s =
    if String.length s < 200 then s
    else String.sub s 0 60 ^ "..." in
  I.with_stack (V.srcloc ("dostring('" ^ abbreviate s ^ "')") 0) g
    (do_lexbuf ~sourcename:"<string>" g) (Lexing.from_string s)

let dofile g infile =
  try
    let f = match infile with "-" -> stdin | _ -> open_in infile in
    let close () = if infile <> "-" then close_in f else () in
    try
      let answer = I.with_stack (V.srcloc ("dofile('" ^ infile ^ "')") 0) g
                     (do_lexbuf ~sourcename:infile g) (Lexing.from_channel f)
      in  (close(); answer)
    with e -> (close (); raise e)
  with Sys_error msg -> [V.Nil; V.String ("System error: " ^ msg)]
```

Defines:
  do_lexbuf, never used.
  dofile, used in chunk 95.
  dostring, used in chunk 95.
  lex, never used.
Uses answer 41, chunks 7a, compile 13 39 64b, error 10a 13 21a 39 42a, file 99 113,
  infile 99 113, length 84b, map 83, mk 14b 16 70 71 94, srcloc 1 1 74 74 80a 81a,
  string 3a 76a 84a 99, token 7a, and with_stack 13 39 62b.

8 Excerpts from the Caml library, imported into Lua

98a ⟨*luacamllib.mli* 98a⟩≡

```
module Make (TV : Lua.Lib.TYPEVIEW with type 'a t = 'a Luaiolib.t)
    : Lua.Lib.USERCODE with type 'a userdata' = 'a TV.combined
```

Uses `userdata'` 80a.

98b ⟨*luacamllib.ml* 98b⟩≡ 100 ▷

```
module IO = Luaiolib
```

99 ⟨*Caml library support 99*⟩≡ (100)
```
  let file    = T.makemap V.userdata V.projection in
  let infile  = IO.in' file V.projection in
  let outfile = IO.out file V.projection in
  let ( **->> ) x y = x **-> V.result y in
  let a      = V.value in
  let b      = V.value in
  let list   = V.list in
  let string = V.string in
  let int    = V.int in
  let bool   = V.bool in
  let ef     = V.efunc in
  let caml_modules =
    let swap (x, y) = (y, x) in
    List.map (fun (m, vs) -> (m, V.Table (V.Table.of_list (List.map swap vs))))
    ["Filename",
         (let extension s =
           try
             let without = Filename.chop_extension s in
             let n = String.length without in
             String.sub s n (String.length s - n)
           with Invalid_argument _ -> "" in
         let chop s = try Filename.chop_extension s with Invalid_argument _ -> s in
         [ ef (string **-> string **->> V.bool) Filename.check_suffix, "check_suffix"
         ; ef (string **->> string) chop, "chop_extension"
         ; ef (string **->> string) extension, "extension"
         ; ef (string **-> string **->> string) Filename.concat, "concat"
         ; ef (string **->> string) Filename.basename, "basename"
         ; ef (string **->> string) Filename.dirname, "dirname"
         ; ef (string **-> string **->> string) Filename.temp_file, "temp_file"
         ; ef (string **->> string) Filename.quote, "quote"
         ])
    ; "List",
         [ ef (list a **->> int)                   List.length,     "length"
         ; ef (list a **->> list a)                List.rev,        "rev"
         ; ef (list a **-> list a **->> list a)    List.append,     "append"
         ; ef (list a **-> list a **->> list a)    List.rev_append, "rev_append"
         ; ef (list (list a) **->> list a)         List.concat,     "concat"
         ; ef ((a --> b) **-> list a **->> list b) List.map,        "map"
         ; ef ((a --> V.unit) **-> list a **->> V.unit) List.iter,  "iter"
         ; ef ((a --> b) **-> list a **->> list b)    List.rev_map, "rev_map"
         ; ef ((a --> bool) **-> list a **->> bool)   List.for_all, "for_all"
         ; ef ((a --> bool) **-> list a **->> bool)   List.exists,  "exists"
         ; ef ((a --> bool) **-> list a **->> list a) List.filter,  "filter"
         ; ef (V.func (a **-> a **->> int) **-> list a **->> list a) List.sort, "sort"
         ; ef (V.func (a **-> a **->> int) **-> list a **->> list a) List.stable_sort,
                         "stable_sort"
         ]
    ] in
```
Defines:

**->>, used in chunks 95, 102, 103, 111, 113, and 116b.
bool, used in chunks 1, 2b, 6b, 8a, 13, 19, 39, 41, 48c, 49b, 73–75, 79b, 80a, 83, 91a, 117, 120, and 127.
caml_modules, used in chunk 100.
ef, never used.
file, used in chunks 1, 63, 74, 81a, 82b, 97, 116a, 117, and 120.
infile, used in chunks 71 and 97.
int, used in chunks 1, 2a, 67b, 69, 74, 75a, 80a, 95, 102, 104, 111b, 113, 117, 120, and 127.
list, used in chunks 1, 2a, 4, 5, 7a, 10a, 13, 14b, 16, 21a, 39, 41, 42a, 47c, 51, 58, 60d, 63, 70, 71, 74, 75a, 77, 78, 80a, 83, 88–91, 94, 102, 117, and 127.
outfile, never used.
string, used in chunks 1, 2, 4, 6b, 8, 10a, 13, 14b, 16, 19–21, 39, 41, 43a, 44b, 48c, 49a, 69–71, 74, 75, 77, 79b, 80a, 82a, 83, 87b, 93–95, 97, 102, 104, 112a, 113, and 127.

Uses **-> 5a 78a 89, --> 5a 78a 88, append 52a, concat 49a, efunc 5a 78a 90, func 5a 64a 78a 80a 90, in' 112a 116a, iter 117 120, length 84b, makemap 8b 20a, map 83, of_list 2a 75a, out 112a 116a, projection 2b 75b 83, result 5a 78a 89, unit 3a 76a 85a, userdata 3a 76a 80a 84a, and value 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

100 ⟨*luacamllib.ml* 98b⟩+≡ ◁98b

```
module Make (T : Lua.Lib.TYPEVIEW with type 'a t = 'a Luaiolib.t)
    : Lua.Lib.USERCODE with type 'a userdata' = 'a T.combined =
  struct
    type 'a userdata' = 'a T.combined
    module M (C : Lua.Lib.CORE with type 'a V.userdata' = 'a userdata') =
     struct
       module V = C.V
       let ( **-> ) = V.( **-> )
       let ( --> ) = V.( --> )
       let init =
         ⟨Caml library support 99⟩
         C.register_module "Caml" caml_modules
     end (*M*)
  end (*Make*)
```

Uses **-> 5a 78a 89, --> 5a 78a 88, caml_modules 99, init 10b 11a 21b 22a, register_module 10a 13 21a 39 66, and userdata' 80a.

101 ⟨*luastrlib.mli* 101⟩≡

```
module M : Lua.Lib.BARECODE
```

102 ⟨*string builtins* 102⟩≡ (103)

```
  ⟨support for format 104⟩
  type 'a parse = int -> (int -> (unit -> 'a) -> 'a) -> (unit -> 'a) -> 'a

  let strindex = { V.embed   = (fun n -> V.int.V.embed (n+1))
                 ; V.project = (fun v -> V.int.V.project v - 1)
                 ; V.is = V.int.V.is
                 }

  let string_builtins =
    let invalid f x =
      try f x with Invalid_argument m -> I.error ("Invalid argument: " ^ m) in
    let wrap_inv = function
      | V.Function (l, f) -> V.Function(l, invalid f)
      | v -> raise (V.Projection (v, "function")) in
    let ifunc ty f = wrap_inv (V.efunc ty f) in
    ⟨support for matching 107⟩
    let quote_char c t = if alnum c then c :: t else '%' :: c :: t in
    let quote_pat p = List.fold_right quote_char (explode p) [] in
    let strfind s pat init plain =
      let int    i = V.int.V.embed    i in
      let string s = V.string.V.embed s in
      let pat = match plain with Some _ -> quote_pat pat | None -> explode pat in
      find pat s init
        (fun caps i j _ -> int (i+1) :: int j :: List.map string caps)
        (fun () -> [V.Nil]) in
    [ "strfind", V.efunc (V.string **-> V.string **-> V.default 0 strindex **->
                          V.option V.int **-> V.resultvs) strfind
    ; "strlen",  V.efunc (V.string **->> V.int) String.length
    ; "strsub",
      (V.efunc (V.string **-> strindex **-> V.option strindex **->> V.string))
      (fun s start last ->
        let maxlast = String.length s - 1 in
        let last = match last with None -> maxlast
                                 | Some n -> min n maxlast in
        let len = last - start + 1 in
        invalid (String.sub s start) len)
    ; "strlower", V.efunc (V.string **->> V.string) String.lowercase
    ; "strupper", V.efunc (V.string **->> V.string) String.uppercase
    ; "strrep",   V.efunc (V.string **-> V.int **->> V.string)
                  (fun s n ->
                    if n < 0 then
                      raise (Invalid_argument ("number of replicas " ^ string_of_int n ^
                                               " is negative"))
                    else
                      let rec list l = function 0 -> l | n -> list (s::l) (n-1) in
                      String.concat "" (list [] n))
    ; "ascii",   V.efunc (V.string **-> V.default 0 strindex **->> V.int)
                 (fun s i -> Char.code (String.get s i))
    ; "format",  ifunc (V.string **-> V.value *****->> V.string) format
    ; "gsub",    V.caml_func (fun _ -> I.error "string library does not implement gsub")
    ]
```

Defines:
`parse`, used in chunks 84b and 107.
`strindex`, never used.
`string_builtins`, used in chunk 103.

Uses `**->` 5a 78a 89, `**->>` 99, `alnum` 109, `caml_func` 1 74 81a, `concat` 49a, `default` 3c 76c 86a, `efunc` 5a 78a 90, `error` 10a 13 21a 39 42a, `explode` 107, `find` 2a 75a 107 117 120, `format` 104, `init` 10b 11a 21b 22a, `int` 3a 76a 85a 99, `length` 84b, `list` 3d 76d 86b 99, `map` 83, `number` 84b, `option` 3b 76b 86a, `resultvs` 5a 78a 89, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

103 ⟨*luastrlib.ml* 103⟩≡

```
module M (I : Lua.Lib.CORE) = struct
  module V = I.V
  let ( **-> ) = V.( **-> )
  let ( **->> ) x y = x **-> V.result y
  let ( *****->> ) = V.dots_arrow
  ⟨string builtins 102⟩
  let init = I.register_globals string_builtins
end
```

Uses `**->` 5a 78a 89, `**->>` 99, `dots_arrow` 5a 78a 90, `init` 10b 11a 21b 22a, `register_globals` 10a 13 21a 39 66, `result` 5a 78a 89, and `string_builtins` 102.

Code for `format` borrowed from standard `Printf` library.

104 ⟨*support for* `format` 104⟩≡ (102)
```
external format_int: string -> int -> string = "caml_format_int"
external format_float: string -> float -> string = "caml_format_float"
let add_quoted_string buf s =
  let escape delim c = c == delim || c == '\n' || c == '\\' in
  let delim = if String.contains s '\'' then '"' else '\'' in
  let add c =
    if escape delim c then (Buffer.add_char buf '\\'); Buffer.add_char buf c in
  Buffer.add_char buf delim;
  String.iter add s;
  Buffer.add_char buf delim

let bprintf_internal buf format =
  let rec doprn i args =
    if i >= String.length format then
      begin
        let res = Buffer.contents buf in
        Buffer.clear buf; (* just in case [bs]printf is partially applied *)
        res
      end
    else begin
      let c = String.get format i in
      if c <> '%' then begin
        Buffer.add_char buf c;
        doprn (succ i) args
      end else begin
        let j = skip_args (succ i) in
        (* Lua conversions: d i o u x X   e E    f g   c  s  p %   q *)
        (*                  ^ ^ ^ ^ ^ ^   ^ ^    ^ ^   ^  ^    ^     *)
        match String.get format j with
        | '%' ->
            Buffer.add_char buf '%';
            doprn (succ j) args
        | c ->
            let arg, args =
              match args with h :: t -> h, t
              | [] -> I.error
                      "Not enough arguments to string-library function `format'" in
            match c with
            | 's' ->
                let s = V.string.V.project arg in
                if j <= i+1 then
                  Buffer.add_string buf s
                else begin
                  let p =
                    try
                      int_of_string (String.sub format (i+1) (j-i-1))
                    with _ ->
                      invalid_arg
                        ("format: bad %s format ‘" ^ String.sub format i (j-i) ^ "’")in
                  if p > 0 && String.length s < p then begin
                    Buffer.add_string buf (String.make (p - String.length s) ' ');
                    Buffer.add_string buf s
                  end else if p < 0 && String.length s < -p then begin
                    Buffer.add_string buf s;
                    Buffer.add_string buf (String.make (-p - String.length s) ' ')
                  end else
                    Buffer.add_string buf s
                end;
                doprn (succ j) args
          | 'c' ->
            let c =
              try Char.chr (V.int.V.project arg)
              with Invalid_argument _ -> V.projection arg "Character code" in
            Buffer.add_char buf c;
            doprn (succ j) args
        | 'd' | 'i' | 'o' | 'x' | 'X' | 'u' ->
            let n = V.int.V.project arg in
            Buffer.add_string buf (format_int (String.sub format i (j-i+1)) n);
            doprn (succ j) args
        | 'f' | 'e' | 'E' | 'g' | 'G' ->
            let f = V.float.V.project arg in
            Buffer.add_string buf (format_float (String.sub format i (j-i+1)) f);
            doprn (succ j) args
        | 'p' ->
            I.error   ("string library does not implement format specifier '%" ^
                       String.make 1 c ^ "'")
        | 'q' ->
            if j <= i+1 then
              add_quoted_string buf (V.string.V.project arg)
            else
              I.error "length not permitted with format specifier '%q'";
            doprn (succ j) args
        | c ->
            I.error   ("bad format specifier '%" ^ String.make 1 c ^ "'")
      end
    end

  and skip_args j =
    match String.get format j with
      '0' .. '9' | ' ' | '.' | '-' -> skip_args (succ j)
    | c -> j

  in doprn 0

let format fmt args = bprintf_internal (Buffer.create 16) fmt args
```

Defines:
  add_quoted_string, never used.

`bprintf_internal`, never used.
`format`, used in chunk 102.
`format_float`, never used.
`format_int`, never used.

Uses `clear` 117 120, `create` 2a 75a 117 120, `error` 10a 13 21a 39 42a, `float` 3a 76a 84a 111b, `int` 3a 76a 85a 99, `iter` 117 120, `length` 84b, `projection` 2b 75b 83, and `string` 3a 76a 84a 99.

A recognizer takes as arguments a position and success and failure continuations. The implementation is based on standard parser combinators.

107 ⟨*support for matching* 107⟩≡ (102)

```
let explode s =
  let rec add n cs = if n = 0 then cs else add (n-1) (s.[n-1] :: cs) in
  add (String.length s) []  in
⟨character-matching functions 109⟩
let find pat s =
  let prerr_string s = () in
  let length = String.length s in
  let () = prerr_string "=========\n" in
  let lefts  = ref [] in
  let pairs = ref [] in
  let push l x = l := x :: !l in
  let pop l =
    match !l with n :: ns -> (l := ns; n) | [] -> I.error "unmatched )" in
  let lparen lp i succ fail =
    push lefts (lp, i); succ i (fun () -> ignore (pop lefts);  fail()) in
  let rparen i succ fail =
    let lp, start = pop lefts in
    push pairs (lp, start, i);
    succ i (fun () -> ignore (pop pairs); push lefts (lp, start); fail()) in
  let captures () =
    let rec insert ((i, l, r) as p) = function
      | [] -> [p]
      | (i', _, _) as p' :: ps -> if i < i' then p :: p' :: ps else p' :: insert p ps
    in let pairs = List.fold_right insert (!pairs) [] in
    List.map (fun (_, l, r) -> String.sub s l (r-l)) pairs in
  let atend i   succ fail = if i = length then succ i fail else fail () in
  let atstart i succ fail = if i = 0      then succ i fail else fail () in
  let opt  r i succ fail = r i succ (fun () -> succ i fail) in
  let (||) r r' i succ fail =
    r i succ (fun () -> r' i succ fail) in
  let (>>) r r' i succ fail =
    r i (fun i' resume -> r' i' succ resume) fail in
  let atzero r i succ fail = r i (succ 0) fail in
  let rec anywhere r i succ fail =
    r i (succ i) (fun () -> if i = length then fail ()
                            else anywhere r (i+1) succ fail) in
  let nonempty (r:'a parse) i succ fail =
    r i (fun i' resume -> if i = i' then resume () else succ i' resume) fail in
  let empty i succ fail = succ i fail in
  let rec star (r : 'a parse) = ((fun i -> ((nonempty r >> star r) || empty) i) : 'a parse) in
  let char p i succ fail =
    if (try p s.[i] with _ -> false) then succ (i+1) fail else fail () in
  let comp pat =
    let rec comp lps c cs =
      let rec finish (p, cs) =
        match cs with
```

```
          | '*' :: cs -> finish (star p, cs)
          | '?' :: cs -> finish (opt  p, cs)
          | []        -> p
          | c :: cs   -> p >> comp lps c cs in
        match c, cs with
        | '%', c :: cs -> finish (char (percent c), cs)
        | '$', []      -> atend
        | '.', cs      -> finish (char (fun _ -> true), cs)
        | '[', cs      -> let p, cs = cclass cs in finish(char p, cs)
        | '(', c :: cs -> lparen lps >> comp (lps+1) c cs
        | ')', c :: cs -> rparen     >> comp lps     c cs
        | ')', []      -> rparen
        | c  , cs      -> finish (char ((=) c), cs) in
      match pat with
      | [] -> fun i succ fail -> if i <= length then succ i i fail else fail ()
      | '^' :: [] -> atzero atstart
      | '^' :: c :: cs -> atzero (atstart >> comp 0 c cs)
      | c :: cs -> anywhere (comp 0 c cs) in
    let with_caps p i succ fail = p i (fun i res -> succ (captures()) i res) fail in
    with_caps (comp pat) in
```

Defines:
`explode`, used in chunk 102.
`find`, used in chunks 44, 50, 66, 85b, 86b, 91, 92, and 102.

Uses `cclass` 110, `error` 10a 13 21a 39 42a, `length` 84b, `map` 83, `parse` 102, and `percent` 109.

109 ⟨*character-matching functions* 109⟩≡ (107) 110 ▷

```
let andp p1 p2 c = p1 c && p2 c   in
let orp  p1 p2 c = p1 c || p2 c   in
let range l h c = l <= c && c <= h    in
let lower = range 'a' 'z'   in
let upper = range 'A' 'Z'   in
let digit = range '0' '9'   in
let space c = c = ' ' || c = '\t' || c = '\r' || c = '\n'   in
let letter = orp lower upper   in
let alnum  = orp letter digit   in
let non p c = not (p c)   in
let percent = function
  | 'a' -> letter | 'A' -> non letter
  | 'd' -> digit  | 'D' -> non digit
  | 'l' -> lower  | 'L' -> non lower
  | 's' -> space  | 'S' -> non space
  | 'u' -> upper  | 'U' -> non upper
  | 'w' -> alnum  | 'W' -> non alnum
  | c when non alnum c -> (=) c
  | _ -> I.error "bad % escape in pattern"   in
```

Defines:
- `alnum`, used in chunk 102.
- `andp`, never used.
- `digit`, never used.
- `letter`, never used.
- `lower`, never used.
- `non`, used in chunks 110 and 113.
- `orp`, used in chunk 110.
- `percent`, used in chunks 107 and 110.
- `range`, used in chunk 110.
- `space`, never used.
- `upper`, used in chunks 25, 28, 31, and 116a.

Uses `error` 10a 13 21a 39 42a.

110 ⟨*character-matching functions* 109⟩+≡ (107) ◁109
```
  let cclass cs =
    let orr p (p', cs) = orp p p', cs in
    let rec pos cs = match cs with
      | ']' :: cs -> orr ((=) ']') (pos2 cs)
      | cs -> pos2 cs
    and pos2 cs = match cs with
    | '-' :: cs -> orr ((=) '-') (pos3 cs)
    | ']' :: cs -> (fun _ -> false), cs
    | _         -> pos3 cs
    and pos3 cs = match cs with
    | '%' :: c :: cs -> orr (percent c) (pos3 cs)
    | ']' :: cs      -> (fun _ -> false), cs
    | c   :: '-' :: ']' :: cs -> orp ((=) c) ((=) '-'), cs
    | c   :: '-' :: c'  :: cs -> orr (range c c') (pos3 cs)
    | c   :: cs               -> orr ((=) c) (pos3 cs)
    | [] -> I.error "bad character class in pattern" in
    match cs with
    | '^' :: cs -> let p, cs = pos cs in non p, cs
    | _ -> pos cs   in
```
Defines:
  `cclass`, used in chunk 107.
Uses `error` 10a 13 21a 39 42a, `non` 109, `orp` 109, `percent` 109, and `range` 109.

111a ⟨*luamathlib.mli* 111a⟩≡
```
  module M : Lua.Lib.BARECODE
```

111b ⟨*math builtins* 111b⟩≡ (111c)
```
  let float = V.float
  let math_builtins =
    [ "abs",        V.efunc (float **->> float)            abs_float
    ; "acos",       V.efunc (float **->> float)            acos
    ; "asin",       V.efunc (float **->> float)            asin
    ; "atan",       V.efunc (float **->> float)            atan
    ; "atan2",      V.efunc (float **-> float **->> float) atan2
    ; "ceil",       V.efunc (float **->> float)            ceil
    ; "cos",        V.efunc (float **->> float)            cos
    ; "floor",      V.efunc (float **->> float)            floor
    ; "log",        V.efunc (float **->> float)            log
    ; "log10",      V.efunc (float **->> float)            log10
    ; "max",        V.efunc (float **-> float **->> float) max
    ; "min",        V.efunc (float **-> float **->> float) min
    ; "mod",        V.efunc (float **-> float **->> float) mod_float
    ; "sin",        V.efunc (float **->> float)            sin
    ; "sqrt",       V.efunc (float **->> float)            sqrt
    ; "tan",        V.efunc (float **->> float)            tan
    ; "random",     V.efunc (V.value **->> float)          (fun _ -> Random.float 1.0)
    ; "randomseed", V.efunc (V.int **->> V.unit)           Random.init
    ]
```
Defines:
  `float`, used in chunks 1, 48, 65, 74, 80a, 95, 104, and 120.
  `math_builtins`, used in chunk 111c.
Uses `**->` 5a 78a 89, `**->>` 99, `efunc` 5a 78a 90, `init` 10b 11a 21b 22a, `int` 3a 76a 85a 99,
  `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

111c ⟨*luamathlib.ml* 111c⟩≡
```
  module M (I : Lua.Lib.CORE) = struct
    module V = I.V
    let ( **-> ) = V.( **-> )
    let ( **->> ) x y = x **-> V.result y
    ⟨math builtins 111b⟩
    let init = I.register_globals math_builtins
  end
```
Uses `**->` 5a 78a 89, `**->>` 99, `init` 10b 11a 21b 22a, `math_builtins` 111b,
  `register_globals` 10a 13 21a 39 66, and `result` 5a 78a 89.

9 Lua I/O library

112a ⟨*luaiolib.mli* 112a⟩≡

```
type 'a t = In of in_channel | Out of out_channel
val out :
    ('a t, 'b, 'b) Luavalue.ep ->
    ('b -> string -> out_channel) ->
    (out_channel, 'b, 'b) Luavalue.ep
val in' :
    ('a t, 'b, 'b) Luavalue.ep ->
    ('b -> string -> in_channel) ->
    (in_channel, 'b, 'b) Luavalue.ep

module T : Lua.Lib.USERTYPE  with type 'a t = 'a t
module Make (TV : Lua.Lib.TYPEVIEW with type 'a t = 'a t)
    : Lua.Lib.USERCODE with type 'a userdata' = 'a TV.combined
```

Defines:
`in'`, used in chunks 99 and 113.
`out`, used in chunks 99 and 113.
Uses `string` 3a 76a 84a 99 and `userdata'` 80a.

112b ⟨*luaiolib.ml* 112b⟩≡ 115 ▷

```
type 'a t = In of in_channel | Out of out_channel
type 'a state = { mutable currentin  : in_channel
                ; mutable currentout : out_channel
                }
type 'a alias_for_t = 'a t
type 'a alias_for_state = 'a state
```

Defines:
`alias_for_state`, never used.
`alias_for_t`, used in chunk 115.
`state`, used in chunks 10, 11a, 14b, 16, 21, 22a, 47d, 63, 65, 68, and 94.

113 ⟨*i/o builtins* 113⟩≡ (116b)

```
  let file = T.makemap V.userdata V.projection in
  let infile  = in' file V.projection in
  let outfile = out file V.projection in

  let wrap_err = function
    | V.Function (l, f) ->
        V.Function(l, fun args -> try f args with Sys_error s -> [V.Nil; V.String s])
    | v -> raise (V.Projection (v, "function")) in

  (* errfunc -- a function that returns nil, string on error *)
  let errfunc   ty f = wrap_err (V.efunc ty f)  in
  let errchoose alts = wrap_err (V.choose alts) in

  (* succeed, succeed2: return non-nil on success *)
  let succeed (f : 'a -> unit) (x : 'a) = (f x; "OK") in
  let succeed2 f x y = ((f x y : unit); "OK") in

  let setglobal s v = V.Table.bind g.V.globals (V.String s) v in

  let readfrom =
    let setinput file =
      (io.currentin <- file; setglobal "_INPUT" (infile.V.embed file); file) in
    let from_string s =
      if String.get s 0 = '|' then
        setinput (Unix.open_process_in (String.sub s 1 (String.length s - 1)))
      else
        setinput (open_in s) in
    let from_other _ = C.error "bad args to readfrom" in
    [ V.alt (V.string **->> infile) from_string
    ; V.alt (V.unit   **->> infile) (fun () -> (close_in io.currentin; setinput stdin))
    ; V.alt (infile   **->> infile) setinput
    ; V.alt (V.value  **->> infile) from_other
    ]  in

  let open_out_append s =
    open_out_gen [Open_wronly; Open_creat; Open_trunc; Open_text] 0o666 s  in

  let open_out_string append s =
    match String.get s 0 with
    | '|' -> if append then raise (Sys_error "tried to appendto() a pipe")
             else Unix.open_process_out (String.sub s 1 (String.length s - 1))
    | _   -> if append then open_out_append s else open_out s in

  let writeto' append =
    let setoutput file =
      (io.currentout <- file; setglobal "_OUTPUT" (outfile.V.embed file); file) in
    let to_nil () = (close_out io.currentout; setoutput stdout) in
    let to_other _ =
      let funname = if append then "appendto" else "writeto" in
      C.error ("bad args to " ^ funname) in
    [ V.alt (V.string **->> outfile)  (fun s -> setoutput (open_out_string append s))
    ; V.alt (V.unit   **->> outfile)  to_nil
    ; V.alt (outfile  **->> outfile)  setoutput
    ; V.alt (V.value  **->> V.value)  to_other
    ]  in

  let read = function
    | None -> (try Some (input_line io.currentin) with End_of_file -> None)
    | Some _ -> C.error ("I/O library does not implement read patterns")  in

  let getopt x d = match x with Some v -> v | None -> d  in

  let date = function
    | Some _ -> C.error ("I/O library does not implement read patterns")
    | None ->
        let t = Unix.localtime (Unix.time ()) in
        let s = string_of_int in
        let mm = t.Unix.tm_mon + 1 in
        let yyyy = t.Unix.tm_year + 1900 in
        let dd = t.Unix.tm_mday in
        s mm ^ "/" ^ s dd ^ "/" ^ s yyyy in

  let tmpname () = Filename.temp_file "lua" "" in

  let write_strings file l = (List.iter (output_string file) l; flush file; 1) in

  let io_builtins =
    [ "readfrom",  errchoose readfrom
    ; "open_out",  V.efunc (V.string **->> outfile) (open_out_string false)
    ; "close_out", V.efunc (outfile  **->> V.unit)  close_out
    ; "open_in",   V.efunc (V.string **->> infile)  open_in
    ; "close_in",  V.efunc (infile   **->> V.unit)  close_in
    ; "writeto",   errchoose (writeto' false)
    ; "appendto",  errchoose (writeto' true)
    ; "remove",    errfunc (V.string **->> V.string) (succeed Sys.remove)
    ; "rename",    errfunc (V.string **-> V.string **->> V.string) (succeed2 Sys.rename)
    ; "tmpname",   V.efunc (V.unit **->> V.string) tmpname
    ; "read",      V.efunc (V.option V.string **->> V.option V.string) read
    ; "write",     errchoose
                   [ V.alt (V.string *****->> V.int)   (* eta-expand to delay eval *)
                                              (fun l -> write_strings io.currentout l)
                   ; V.alt (outfile **-> V.string *****->> V.int) write_strings
                   ]
    ; "date",      V.efunc (V.option V.string **->> V.string) date
    ; "exit",      V.efunc (V.option V.int **->> V.unit) (fun n -> exit (getopt n 0))
    ; "getenv",    V.efunc (V.string **->> V.option V.string)
                   (fun s -> try Some (Sys.getenv s) with Not_found -> None)
    ; "execute",   V.efunc (V.string **->> V.int) Sys.command
    ; "_STDIN",    infile.V.embed  stdin
    ; "_STDOUT",   outfile.V.embed stdout
    ; "_STDERR",   outfile.V.embed stderr
    ; "_INPUT",    infile.V.embed  io.currentin
    ; "_OUTPUT",   outfile.V.embed io.currentout
    ] in
```

Defines:
`date`, never used.
`errchoose`, never used.
`errfunc`, never used.
`file`, used in chunks 1, 63, 74, 81a, 82b, 97, 116a, 117, and 120.
`getopt`, never used.
`infile`, used in chunks 71 and 97.
`io_builtins`, used in chunk 116b.
`open_out_append`, never used.
`open_out_string`, never used.
`outfile`, never used.
`read`, never used.
`readfrom`, used in chunk 116b.
`setglobal`, used in chunks 61, 66, and 95.
`succeed`, never used.
`succeed2`, never used.
`tmpname`, never used.
`wrap_err`, never used.
`write_strings`, never used.
`writeto'`, never used.

Uses `**->` 5a 78a 89, `**->>` 99, `alt` 5b 5b 78b 78b 91a 91a, `append` 52a, `bind` 2a 75a, `choose` 5b 78b 91a, `efunc` 5a 78a 90, `error` 10a 13 21a 39 42a, `funname` 69, `in'` 112a 116a, `int` 3a 76a 85a 99, `iter` 117 120, `length` 84b, `makemap` 8b 20a, `nil` 65, `non` 109, `option` 3b 76b 86a, `out` 112a 116a, `projection` 2b 75b 83, `remove` 117 120, `string` 3a 76a 84a 99, `unit` 3a 76a 85a, `userdata` 3a 76a 80a 84a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

115 ⟨*luaiolib.ml* 112b⟩+≡ ◁112b 116a▷

```
  module T = struct
    type 'a t     = 'a alias_for_t
    let tname = "I/O channel"
    let eq _ x y = match x, y with
    | In x,    In y    -> x = y
    | Out x,   Out y   -> x = y
    | _, _ -> false
    let to_string vs = function
      | In _ -> "<input>"
      | Out _ -> "<output>"
  end
```

Uses `alias_for_t` 112b, `eq` 1 6b 8a 19 74 79b 80b, `tname` 6b 8a 19 79b, and `to_string` 1 6b 8a 19 74 79b 81b.

116a ⟨*luaiolib.ml* 112b⟩+≡ ◁115 116b▷

```
module V = Luavalue
let out upper fail =
      { V.embed   = (fun x -> upper.V.embed (Out x))
      ; V.project = (fun x -> match upper.V.project x with
                     | Out x -> x
                     | _ -> fail x "output file")
      ; V.is      = (fun x -> upper.V.is x && match upper.V.project x with
                                          | Out x -> true | _ -> false)
      }
let in' upper fail =
      { V.embed   = (fun x -> upper.V.embed (In x))
      ; V.project = (fun x -> match upper.V.project x with
                     | In x -> x
                     | _ -> fail x "input file")
      ; V.is      = (fun x -> upper.V.is x && match upper.V.project x with
                                          | In x -> true | _ -> false)
      }
```

Defines:
`in'`, used in chunks 99 and 113.
`out`, used in chunks 99 and 113.
Uses `file` 99 113 and `upper` 109.

116b ⟨*luaiolib.ml* 112b⟩+≡ ◁116a

```
module Make (T : Lua.Lib.TYPEVIEW with type 'a t = 'a t)
    : Lua.Lib.USERCODE with type 'a userdata' = 'a T.combined =
  struct
    type 'a userdata' = 'a T.combined
    module M (C : Lua.Lib.CORE with type 'a V.userdata' = 'a userdata') =
     struct
       module V = C.V
       let ( **-> ) = V.( **-> )
       let ( **->> ) x y = x **-> V.result y
       let ( *****->> ) = V.dots_arrow
       let init g =  (* g needed for readfrom, writeto, appendto *)
         let io = {currentin = stdin; currentout = stdout} in
         ⟨i/o builtins 113⟩
         C.register_globals io_builtins g
     end (*M*)
  end (*Make*)
```

Uses `**->` 5a 78a 89, `**->>` 99, `dots_arrow` 5a 78a 90, `init` 10b 11a 21b 22a, `io_builtins` 113, `readfrom` 113, `register_globals` 10a 13 21a 39 66, `result` 5a 78a 89, and `userdata'` 80a.

This is the standard OCaml hash table, except I've added `first` and `next` functions to support Lua's table-enumeration primitive, plus I've added a `population` function.

117 ⟨*luahash.mli* 117⟩≡
```
(***********************************************************************)
(*                                                                     *)
(*                           Objective Caml                            *)
(*                                                                     *)
(*            Xavier Leroy, projet Cristal, INRIA Rocquencourt         *)
(*                                                                     *)
(*  Copyright 1996 Institut National de Recherche en Informatique et   *)
(*  en Automatique.  All rights reserved.  This file is distributed    *)
(*  under the terms of the GNU Library General Public License.         *)
(*                                                                     *)
(***********************************************************************)

(* modified by Norman Ramsey to provide threading via a ‘next’ function *)

(* $Id: luahash.nw,v 1.8 2004-08-03 22:13:33 nr Exp $ *)

(* Hash tables are hashed association tables, with in-place modification. *)

(*** Generic interface *)

type (’a, ’b) t
        (* The type of hash tables from type [’a] to type [’b]. *)

val create : (’a -> ’a -> bool) -> int -> (’a,’b) t
        (* [Luahash.create eq n] creates a new, empty hash table, with
           initial size [n].  Function eq is used to compare equality of keys
           For best results, [n] should be on the
           order of the expected number of elements that will be in
           the table.  The table grows as needed, so [n] is just an
           initial guess. *)

val population : (’a, ’b) t -> int
        (* number of key-value pairs in a table (as distinct from its size) *)

val clear : (’a, ’b) t -> unit
        (* Empty a hash table. *)

val find : (’a, ’b) t -> ’a -> ’b
        (* [Luahash.find tbl x] returns the current binding of [x] in [tbl],
           or raises [Not_found] if no such binding exists. *)

val find_all : (’a, ’b) t -> ’a -> ’b list
        (* [Luahash.find_all tbl x] returns the list of all data
           associated with [x] in [tbl].
           The current binding is returned first, then the previous
```

```
            bindings, in reverse order of introduction in the table. *)

  val mem :  ('a, 'b) t -> 'a -> bool
        (* [Luahash.mem tbl x] checks if [x] is bound in [tbl]. *)

  val remove : ('a, 'b) t -> 'a -> unit
        (* [Luahash.remove tbl x] removes the current binding of [x] in [tbl],
           restoring the previous binding if it exists.
           It does nothing if [x] is not bound in [tbl]. *)

  val replace : ('a, 'b) t -> key:'a -> data:'b -> unit
        (* [Luahash.replace tbl x y] replaces the current binding of [x]
           in [tbl] by a binding of [x] to [y].  If [x] is unbound in [tbl],
           a binding of [x] to [y] is added to [tbl].
           This is functionally equivalent to [Luahash.remove tbl x]
           followed by [Luahash.add tbl x y], except that Luahash has no [add]. *)

  val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
        (* [Luahash.iter f tbl] applies [f] to all bindings in table [tbl].
           [f] receives the key as first argument, and the associated value
           as second argument. The order in which the bindings are passed to
           [f] is unspecified. Each binding is presented exactly once
           to [f]. *)

  val first : ('a, 'b) t -> 'a * 'b
  val next  : ('a, 'b) t -> 'a -> 'a * 'b
        (* Used to iterate over the contents of the table, Lua style.
           Raises Not_found when the contents are exhausted *)


  (*** The polymorphic hash primitive *)

  val hash : 'a -> int
        (* [Luahash.hash x] associates a positive integer to any value of
           any type. It is guaranteed that
                if [x = y], then [hash x = hash y].
           Moreover, [hash] always terminates, even on cyclic
           structures. *)
```

Defines:
`clear`, used in chunk 104.
`create`, used in chunks 5b, 53, 66, 78b, 86b, 92, 93, and 104.
`find`, used in chunks 44, 50, 66, 85b, 86b, 91, 92, and 102.
`find_all`, never used.
`first`, used in chunks 82b, 87b, and 95.
`hash`, never used.
`iter`, used in chunks 43b, 44b, 62b, 66, 68, 69, 71, 82a, 92, 93, 95, 99, 104, and 113.
`mem`, never used.
`next`, used in chunks 86b, 87b, and 95.
`population`, used in chunks 47c, 86b, and 95.
`remove`, used in chunks 92 and 113.

`replace`, used in chunks 44a and 92.
Uses `bool` 3a 76a 85a 99, `eq` 1 6b 8a 19 74 79b 80b, `file` 99 113, `int` 3a 76a 85a 99, `list` 3d 76d 86b 99, `number` 84b, `order` 48c, `results` 5a 78a 89, `table` 4a 77a 80a 87a, `unit` 3a 76a 85a, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.

120 ⟨*luahash.ml* 120⟩≡

```
(***********************************************************************)
(*                                                                     *)
(*                           Objective Caml                            *)
(*                                                                     *)
(*            Xavier Leroy, projet Cristal, INRIA Rocquencourt         *)
(*                                                                     *)
(*  Copyright 1996 Institut National de Recherche en Informatique et   *)
(*  en Automatique.  All rights reserved.  This file is distributed    *)
(*  under the terms of the GNU Library General Public License.         *)
(*                                                                     *)
(***********************************************************************)

(* $Id: luahash.nw,v 1.8 2004-08-03 22:13:33 nr Exp $ *)

(* Hash tables *)

let hash_param = Hashtbl.hash_param

let hash x = hash_param 10 100 x

(* We do dynamic hashing, and resize the table and rehash the elements
   when buckets become too long. *)

type ('a, 'b) t =
  { eq : 'a -> 'a -> bool;
    mutable population : int;
    mutable max_len: int;                     (* max length of a bucket *)
    mutable data: ('a, 'b) bucketlist array } (* the buckets *)

and ('a, 'b) bucketlist =
    Empty
  | Cons of 'a * 'b * ('a, 'b) bucketlist

let bucket_length l =
  let rec len k = function
    | Empty -> k
    | Cons(_, _, l) -> len (k+1) l
  in len 0 l

let dump_buckets h k i l =
  let nsize = Array.length h.data in
  let int = string_of_int in
  let hmod k = int (hash k mod nsize) in
  let rec dump = function
    | Empty -> ()
    | Cons (k', i', l) ->
        List.iter prerr_string ["New bucket hash = "; int (hash k'); " [mod=";
                                hmod k'; "]";
                                if h.eq k k' then " (identical " else " (different ";
                                "keys)\n"];
        dump l
  in List.iter prerr_string ["First bucket hash = "; string_of_int (hash k);
                             " [mod="; hmod k; "]\n"];
     dump l

let create eq initial_size =
  let s = if initial_size < 1 then 1 else initial_size in
  let s = if s > Sys.max_array_length then Sys.max_array_length else s in
  { eq = eq; max_len = 3; data = Array.make s Empty; population = 0 }

let clear h =
  h.population <- 0;
  for i = 0 to Array.length h.data - 1 do
    h.data.(i) <- Empty
  done

let dump_table_stats h =
  let flt x = Printf.sprintf "%4.2f" x in
  let int = string_of_int in
  let sum = ref 0 in
  let sumsq = ref 0 in
  let n   = ref 0 in
  let zs  = ref 0 in
  let ratio n m = float n /. float m in
  let inc r n = r := !r + n in
  let stats l =
    let k = bucket_length l in
    if k = 0 then inc zs 1
    else (inc sum k; inc sumsq (k*k); inc n 1) in
  for i = 0 to Array.length h.data - 1 do
    stats h.data.(i)
  done;
  let mean = ratio (!sum) (!n) in
  let variance =
    if !n > 1 then                         (* concrete math p 378 *)
      (float (!sumsq) -. float (!sum) *. mean) /. (float (!n - 1))
    else
      0.0 in
  let variance = if variance < 0.0 then 0.0 else variance in
  let stddev = sqrt variance in
  let stderr = stddev /. sqrt (float (!n)) in
  List.iter prerr_string ["Table has "; int (!zs); " empy buckets; ";
                          "avg nonzero length is "; flt (ratio (!sum) (!n));
                          " +/- "; flt stderr; " \n"]


let resize hashfun tbl =
  let odata = tbl.data in
  let osize = Array.length odata in
  let nsize = min (2 * osize + 1) Sys.max_array_length in
  if nsize <> osize then begin
    let ndata = Array.create nsize Empty in
    let rec insert_bucket = function
        Empty -> ()
      | Cons(key, data, rest) ->
          insert_bucket rest; (* preserve original order of elements *)
          let nidx = (hashfun key) mod nsize in
          ndata.(nidx) <- Cons(key, data, ndata.(nidx)) in
    for i = 0 to osize - 1 do
      insert_bucket odata.(i)
    done;
    tbl.data <- ndata;
  end;
  tbl.max_len <- 2 * tbl.max_len
(*  if tbl.max_len >= 48 then dump_table_stats tbl *)

let rec bucket_too_long n bucket =
  if n < 0 then true else
    match bucket with
      Empty -> false
    | Cons(_,_,rest) -> bucket_too_long (n - 1) rest

let remove h key =
  let rec remove_bucket = function
      Empty ->
        Empty
    | Cons(k, i, next) ->
        if h.eq k key then
          begin
            h.population <- h.population - 1;
            next
          end
        else
          Cons(k, i, remove_bucket next) in
  let i = (hash key) mod (Array.length h.data) in
  h.data.(i) <- remove_bucket h.data.(i)

let rec find_rec eq key = function
    Empty ->
      raise Not_found
  | Cons(k, d, rest) ->
      if eq key k then d else find_rec eq key rest

let find h key =
  match h.data.((hash key) mod (Array.length h.data)) with
    Empty -> raise Not_found
  | Cons(k1, d1, rest1) ->
      if h.eq key k1 then d1 else
      match rest1 with
```

```
          Empty -> raise Not_found
        | Cons(k2, d2, rest2) ->
            if h.eq key k2 then d2 else
            match rest2 with
              Empty -> raise Not_found
            | Cons(k3, d3, rest3) ->
                if h.eq key k3 then d3 else find_rec h.eq key rest3

  (* next element in table starting in bucket [index] *)
  let rec next_at h index =
    if index = Array.length h.data then
      raise Not_found
    else
      match h.data.(index) with
      | Empty -> next_at h (index+1)
      | Cons(k1, d1, _) -> (k1, d1)

  let rec following eq key fail =
    let finish = function
      | Empty -> fail ()
      | Cons (k, d, _) -> (k, d)
    in function
    | Empty -> assert false
    | Cons(k1, d1, rest1) ->
        if eq key k1 then finish rest1 else
        following eq key fail rest1

  let next h key =
    let index = (hash key) mod (Array.length h.data) in
    let finish = function
      | Empty -> next_at h (index+1)
      | Cons (k, d, _) -> (k, d)
    in
    match h.data.(index) with
      Empty -> next_at h (index+1)
    | Cons(k1, _, rest1) ->
        if h.eq key k1 then finish rest1 else
        match rest1 with
          Empty -> raise Not_found
        | Cons(k2, _, rest2) ->
            if h.eq key k2 then finish rest2 else
            match rest2 with
              Empty -> raise Not_found
            | Cons(k3, _, rest3) ->
                if h.eq key k3 then finish rest3
                else following h.eq key (fun () -> finish Empty) rest3

  let rec first_at h index =
    if index = Array.length h.data then
      raise Not_found
    else
      match h.data.(index) with
      | Empty -> first_at h (index+1)
      | Cons(k, d, _) -> (k, d)

  let first h = first_at h 0

  let find_all h key =
    let rec find_in_bucket = function
      Empty ->
        []
    | Cons(k, d, rest) ->
        if k = key then d :: find_in_bucket rest else find_in_bucket rest in
    find_in_bucket h.data.((hash key) mod (Array.length h.data))

  let replace h ~key ~data:info =
    let rec replace_bucket = function
        Empty ->
          raise Not_found
      | Cons(k, i, next) ->
          if k = key
          then Cons(k, info, next)
          else Cons(k, i, replace_bucket next) in
    let i = (hash key) mod (Array.length h.data) in
    let l = h.data.(i) in
  (*
    Log.bucket_length (bucket_length l);
    if bucket_length l > 5 then
      begin
        (match l with Cons (k, i, l) -> dump_buckets h k i l | _ -> ());
        prerr_string
            (if bucket_too_long h.max_len l then "bucket too long (> "
             else "bucket length OK (<= ");
        prerr_int h.max_len;
        prerr_string ")\n\n"
      end;
  *)
    try
      h.data.(i) <- replace_bucket l
    with Not_found ->
      begin
        let bucket = Cons(key, info, l) in
        h.data.(i) <- bucket;
        h.population <- h.population + 1;
        (*if bucket_too_long h.max_len bucket then resize hash h*)
        if h.population > Array.length h.data then resize hash h
      end

  let mem h key =
    let rec mem_in_bucket = function
    | Empty ->
        false
    | Cons(k, d, rest) ->
        k = key || mem_in_bucket rest in
    mem_in_bucket h.data.((hash key) mod (Array.length h.data))

  let iter f h =
    let rec do_bucket = function
        Empty ->
          ()
      | Cons(k, d, rest) ->
          f k d; do_bucket rest in
    let d = h.data in
    for i = 0 to Array.length d - 1 do
      do_bucket d.(i)
    done

  let population h =
    h.population
```

Defines:
  `bucket_length`, never used.
  `bucket_too_long`, never used.
  `clear`, used in chunk 104.
  `create`, used in chunks 5b, 53, 66, 78b, 86b, 92, 93, and 104.
  `dump_buckets`, never used.
  `dump_table_stats`, never used.
  `find`, used in chunks 44, 50, 66, 85b, 86b, 91, 92, and 102.
  `find_all`, never used.
  `find_rec`, never used.
  `first`, used in chunks 82b, 87b, and 95.
  `first_at`, never used.
  `following`, never used.
  `hash`, never used.
  `hash_param`, never used.
  `iter`, used in chunks 43b, 44b, 62b, 66, 68, 69, 71, 82a, 92, 93, 95, 99, 104, and 113.
  `mem`, never used.
  `next`, used in chunks 86b, 87b, and 95.
  `next_at`, never used.
  `population`, used in chunks 47c, 86b, and 95.
  `remove`, used in chunks 92 and 113.
  `replace`, used in chunks 44a and 92.
  `resize`, never used.
Uses `bool` 3a 76a 85a 99, `eq` 1 6b 8a 19 74 79b 80b, `file` 99 113, `float` 3a 76a 84a 111b,
  `index` 50a, `int` 3a 76a 85a 99, `length` 84b, `order` 48c, `sum` 65, and `table` 4a 77a 80a 87a.

10 Abstract syntax for Lua

126a ⟨*signatures* 19⟩+≡ (22b 23c 40 41 79 126) ◁79b

```
module type S = sig
  module Value : Luavalue.S
  type value = Value.value
  ⟨abstract syntax 127⟩
end
```

Defines:
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

126b ⟨*luaast.mli* 126b⟩≡

```
⟨signatures 19⟩
module Make (V : Luavalue.S) : S with module Value = V
```

126c ⟨*luaast.ml* 126c⟩≡

```
⟨signatures 19⟩
module Make (V : Luavalue.S) : S with module Value = V = struct
  module Value = V
  type value = Value.value
  ⟨abstract syntax 127⟩
end
```

Defines:
`value`, used in chunks 2, 5, 6b, 8a, 10a, 14b, 16, 19, 21a, 24a, 42, 47, 51, 58, 60d, 63, 66, 68, 75, 78, 79b, 82a, 83, 88–91, 94, 95, 99, 102, 111b, 113, 117, and 127.

127 ⟨*abstract syntax* 127⟩≡ (126)
```
  type name = string
  type location = int (* character position *)
  type stmt =
    | Stmt'       of location * stmt
    | Assign      of lval list * exp list
    | WhileDo     of exp * block
    | RepeatUntil of block * exp
    | If          of exp * block * (exp * block) list * block option
    | Return      of exp list
    | Callstmt    of call
    | Local       of name list * exp list
  and block = stmt list
  and lval =
    | Lvar   of name
    | Lindex of exp * exp
  and exp =
    | Var   of name
    | Lit   of value
    | Binop of exp * op * exp
    | Unop  of op * exp
    | Index of exp * exp
    | Table of exp list * (name * exp) list
    | Call  of call
  and call =
    | Funcall  of exp * exp list
    | Methcall of exp * name * exp list
  and op = And | Or | Lt | Le | Gt | Ge | Eq | Ne | Concat
         | Plus | Minus | Times | Div | Not | Pow

  type chunk =
    | Debug     of bool                    (* turn debugging on/off *)
    | Statement of stmt
    | Fundef    of location * lval       * name list * varargs * block
    | Methdef   of location * exp * name * name list * varargs * block
  and varargs = bool
```
Defines:
  `block`, used in chunks 57, 59, 63, and 64.
  `call`, used in chunks 43a, 44c, 55a, and 60c.
  `chunk`, used in chunks 13, 14b, 16, 39, 64b, and 94.
  `exp`, used in chunks 51, 54b, 55b, and 69.
  `location`, used in chunks 1, 58, 64a, 74, and 80a.
  `lval`, never used.
  `name`, used in chunks 1, 6b, 8a, 19, 47d, 74, and 79b.
  `op`, used in chunks 48a, 54a, and 65.
  `stmt`, used in chunk 60.
  `varargs`, used in chunks 5a, 63, 64a, 78a, and 90.
Uses `bool` 3a 76a 85a 99, `int` 3a 76a 85a 99, `list` 3d 76d 86b 99, `option` 3b 76b 86a,
  `string` 3a 76a 84a 99, and `value` 1 4a 13 39 41 70 71 74 77a 80a 87a 126a 126c.