1. Preface. There are many programming languages and this one is mine. `Lossless` is a general purpose programming language with a lisp/scheme look and feel. If you have received this document in electronic form it should have been accompanied with sources and a `README` file describing how to build and test `Lossless` on various platforms. If you received this document as a hard copy I would be very interested to know how and why, as it's woefully incomplete and I have not printed any hard copies, even for proofreading, but in that case you'll have to type in the code yourself.

Alternatively `Lossless` exists primarily as a package of source distributed through the web. The primary home for the source code is at http://zeus.jtan.com/~chohag/lossless/ describing the git repository at http://zeus.jtan.com/~chohag/lossless.git. Additionally the sources are occasionaly packaged up in a tarball along with the processed C sources so that TEX is optional.

If `Lossless` were complete it would include a comprehensive suite of documentation. This particular document would fit in that suite as a sort of appendix — available for people who are familiar with using `Lossless` (or not) to look into and learn or improve its implementation — so it is neither comprehensive nor pedagogical.

Of course `Lossless` is not in fact complete and this document is its only one. It is laid out as follows:

1 Introduction to the C implementation and error handling.
What source files there are and why, how to build them, how to run `Lossless` and/or link to and use it dynamically.
Includes a description of the rules governing variable and function (etc.) names.

2 Memory & data structures.
How to allocate and release (free) memory, and the mostly-functional[1] code to implement the few object required to operate the virtual machine.

3 Bare stubs to support I/O and threading.
Just enough of an object wrapping around a file descriptor to read and assemble source code, and enough of a threads implentation to put mutexes around critical process-global objects.

4a A bytecode-interpreting virtual machine.
Decode, Execute, Repeat. Like a REPL for robots, which don't read.

4b A bytecode assembler.
Far more effort than it seems like it should require, even this strictly line-based parser and assembler is the largest, most complicated piece of this implementation.

5 Test suite.
Some things it's better to begin with than try to retro-fit to an existing product. As well as thread support from day one `Lossless` also begins its existence with a comprehensive test suite.

6 Leftovers.
Vaguely unimportant things that don't belong anywhere else, and the index.

It should be evident by now that your author is not particularly skilled in the TEXnical arts but rather is using it as a means to an end. I will also endeavour to talk only in the third person as I try to avoid referring to the author anyway.

[1] Functional in the mathematical sense not (only) in that they are fully operational.

2. Implementation. Although its appearance as a PDF file (or similar) is unconventional `Lossless` is a fairly trivial C program with no dependencies other than the standard library (and TEX which is huge, semi-optional and also has minimal dependencies).

`Lossless`' source comes in the these files:

`README`	Unpacking the source
`README.deploy`	Packing the source
`lossless.w`	The C parts of `Lossless` (including *primitives*)
`barbaroi.ll`	The `Lossless` parts of `Lossless` (not primitive)
`evaluate.la`	Assembly source of the `Lossless` interpreter
`man/man`n`l/*`	Neglected unix manual page documentation
`man/man`n`l/intro.`n`l`	A description of each section's contents
`man/man9l/TEMPLATE.9l`	Manual page source template
`perl/*`	Neglected proof-of-concept library wrapper
`Makefile`	Build Plumbing
`bin/bin2c`	— „ —
`bin/reindex`	— „ —
`.gitignore`	Administrivia & Notes
`LICENSE`	— „ —
`llfig.mp`	— „ —
`PLAN`	— „ —
`PLAN.man`	— „ —

When `Lossless` is built these files are generated. Alternatively if you have downloaded the packaged sources rather than cloning a source repository then they come pre-built inside it and you don't need TEX to compile `Lossless`.

`lossless.c`	The C parts of `Lossless`
`lossless.h`	A header file for linking to `lossless.o` or its dynamic equivalent
`testless.c`	C code shared between test units
`testless.h`	— „ —
`t/*.c`	A comprehensive suite of test units

Building generates a bit of mess in the working directory which `make clean` removes (mostly) and of course each C source file is compiled to a corresponding static library (`*.o`). The following files are also created:

`lossless`	`Lossless`
`initialise.o`	The contents of `barbaroi.ll` and `evaluate.la`
`memless.o`	`lossless.o` for tests of the memory allocator
`liblossless.so`	Shared library for linking to `Lossless` dynamically[1]

As well as `clean` and the default `all` there are other interesting `make` targets:

`dist`	Build a distributable release
`test`	Build and lauch the comprehensive test suite
`lossless.pdf`	This document
`man/mandoc.db`	`Lossless`' manual database

3. The bulk of `Lossless`' source is in `lossless.w`. This `CWEB` source file is compiled by `ctangle` to produce C sources or by `cweave` to produce TEX sources. The TEX sources are compiled by TEX to produce this document `lossless.pdf` while the C sources are compiled by a C compiler into static and shared libraries.

The main library file produced is the static `lossless.o` and there's a variant which has hooks in the memory allocator named `memless.o`. Also produced from `lossless.w` is `testless.o` which is linked into each test script alongside `memless.o` and contains the functions shared by `Lossless`' test suite.

All of this is taken care of in the `Makefile`, compatible with BSD and GNU make simultaneously.

[1] Windows support is practically non-existent, but is planned and would call this `lossless.dll`.

4. The only intermediate source file of interest outside the `Lossless` build process is `lossless.h` which is necessary to link to the `Lossless` shared library or write extensions, although for practical reasons it contains many more definitions than library users require (that is: all of them).

The contents of `lossless.h` are described by the code block attached to this section. Code blocks may be named such as this one with a filename presented in a monospace font (`@(filename@>=` in `CWEB` source) and cause `CWEB` to concatenate all such sections to produce the file named. In fact `lossless.h` is generated by just this one section however it includes references to other sections by name (such as the next one, ⟨ System headers 6 ⟩) which are themselves concatenated and inserted in place.

The special section name "Preprocessor definitions" consists of all the #**define** lines (`@d` in `CWEB`) that preceed code sections.

Finally there are sections that do have a block of code but one without a name. These become `lossless.c` and begin `@c` in `CWEB`.

5. ⟨ `lossless.h` 5 ⟩ ≡

```
#ifndef LL_LOSSLESS_H
#define LL_LOSSLESS_H
  ⟨ System headers 6 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Essential types 23 ⟩
  ⟨ Type definitions 11 ⟩
  ⟨ Function declarations 19 ⟩
  ⟨ External C symbols 14 ⟩
  ⟨ Hacks and warts 9 ⟩
#endif
```

6. These system headers are required globally and included in the main header `lossless.h`. Perhaps they shouldn't be.

⟨ System headers 6 ⟩ ≡

```
#include <err.h>
#include <errno.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdint.h>
#include <unistd.h>
```

This code is cited in section 4.

This code is used in section 5.

7. This is the start of `lossless.c` — system header files which are not required by the definitions in `lossless.h` followed by `Lossless`' global variables (and then every other un-named section of code).

```
#include <assert.h>     /* Definitely not for library code. */
#include <stdio.h>      /* To be removed when Lossless I/O is usable. */
#include <stdlib.h>
#include <string.h>     /* Bulk memory transfers, and strlen. */
#include "lossless.h"
#ifdef LLTEST
#include <stdarg.h>
#include "testless.h"
#endif
  ⟨ Global variables 13 ⟩
```

8. Non-C source code which is compiled by `Lossless` as it starts up is baked into the binary through `initialise.c`, where it is kept separate from the compiled C in order to ease future plans for growth.

```
⟨ initialise.c 8 ⟩ ≡
#include <assert.h>
#include <string.h>
#include <stdio.h>      /* To be removed when Lossless I/O is usable. */
#include "lossless.h"
  ⟨ Data for initialisation 12 ⟩
```

See also sections 22 and 126.

9. This is the first of the warts which don't have a sensible place to live. The definition of `unused` silences compiler warnings when function arguments are unused. `shared` & `unique` distinguish between global variables which are accessible to all and private to each individual thread. TODO: They should not be here.

```
#define shared      /* Global variable visible to all threads. */
#define unique __thread      /* Global variable private to each thread. */
```

```
⟨ Hacks and warts 9 ⟩ ≡
#ifdef __GNUC__      /* & clang */
#define unused __attribute__((__unused__))
#else
#define unused      /* noisy compiler */
#endif
```

See also sections 63 and 75.

This code is used in section 5.

10. Errors. All code comes with errors and before going any further the errors that can occur in `Lossless` are defined. The simplest of mechanisms is used for functions which could fail, which is nearly all of them: a status code representing what went wrong is returned. The actual value being returned is saved to an address pointed to by the last argument.

The macros below check the error status code of such a function and "give up" somehow, depending on what is most appropriate. The macros were created in an ad-hoc fashion as the need arose and now only *orabort*, *orassert*, *orreturn* and *ortrap* are used — mostly the latter two.

#define *failure_p*(O) $((O) \neq$ `LERR_NONE`$)$
#define *just_abort*(E, M) *err*$((E),$ `"%s:␣%u"`$, (M), (E))$
#define *do_or_abort*(R, I) **do** { **if** $($*failure_p*$(((R) \leftarrow (I))))$ *just_abort*$((R), \#I);$ } **while** (0)
#define *do_or_return*(R, I) **do** { **if** $($*failure_p*$((R) \leftarrow (I)))$ **return** $(R);$ } **while** (0)
#define *do_or_trap*(R, I) **do** { **if** $($*failure_p*$((R) \leftarrow (I)))$ **goto** *Trap*; } **while** (0)
#define *orabort*(I) *do_or_abort*$(reason, I)$
#define **orreturn**(I) *do_or_return*$(reason, I)$
#define **ortrap**(I) *do_or_trap*$(reason, I)$

#define *do_or_assert*(R, I) **do** { $(R) \leftarrow (I);$ *assert*$(\#I \wedge \neg$*failure_p*$(R));$ } **while** (0)
#define *orassert*(I) **do** { $reason \leftarrow (I);$ *assert*$(\#I \wedge \neg$*failure_p*$(reason));$ } **while** (0)

11. These are all the ways that an `Lossless` internal function can fail. Some of these codes were defined when `Lossless` still had a custom parser and remain here in case it ever comes back.

⟨ Type definitions 11 ⟩ ≡

```
typedef enum {
  LERR_NONE,
  LERR_ADDRESS,      /* An invalid value is used as an address. */
  LERR_AMBIGUOUS,    /* An expression's ending is unclear. */
  LERR_BUSY,     /* A resource is busy. */
  LERR_DOUBLE_TAIL,      /* Two . elements in a list. */
  LERR_EMPTY_TAIL,     /* A . without a tail expression. */
  LERR_EOF,     /* End of file or stream. */
  LERR_EXISTS,     /* New binding conflicts. */
  LERR_FINISHED,     /* A thread or process is already finished. */
  LERR_HEAVY_TAIL,     /* A . with more than one tail expression. */
  LERR_IMMUTABLE,     /* Attempt to mutate a read-only object. */
  LERR_IMPROPER,     /* An improper list was encountered. */
  LERR_INCOMPATIBLE,     /* Operation on an incompatible operand. */
  LERR_INSTRUCTION,     /* Attempted to interpret an invalid instruction. */
  LERR_INTERNAL,     /* Bug in Lossless. */
  LERR_INTERRUPT,     /* An operation was interrupted. */
  LERR_IO,     /* "An" I/O error. */
  LERR_LEAK,     /* System resource (eg. file handle) lost. */
  LERR_LIMIT,     /* A software-defined limit has been reached. */
  LERR_LISTLESS_TAIL,      /* List tail-syntax (.) not in a list. */
  LERR_MISMATCH,     /* Closing bracket did not match open bracket. */
  LERR_MISSING,     /* A hash table or environment lookup failed. */
  LERR_NONCHARACTER,      /* Scanning UTF-8 encoding failed. */
  LERR_OOM,     /* Out of memory. */
  LERR_OUT_OF_BOUNDS,      /* Bounds exceeded. */
  LERR_OVERFLOW,     /* Attempt to access past the end of a buffer. */
  LERR_SELF,     /* An attempt to wait for oneself. */
  LERR_SYNTAX,     /* Unrecognisable syntax (insufficient alone). */
  LERR_SYSTEM,     /* A system error, check errno. */
  LERR_THREAD,     /* Failed to correctly join a thread. */
  LERR_UNCLOSED_OPEN,      /* Missing ), ] or }. */
  LERR_UNCOMBINABLE,     /* Attempted to combine a non-program. */
  LERR_UNDERFLOW,     /* A stack was popped too far. */
  LERR_UNLOCKED,     /* Attempt to use an unlocked critical resource. */
  LERR_UNIMPLEMENTED,      /* A feature is not implemented. */
  LERR_UNOPENED_CLOSE,     /* Premature ), ] or }, or (close!). */
  LERR_UNPRINTABLE,     /* Failed serialisation attempt. */
  LERR_UNSCANNABLE,     /* Parser encountered LEXICAT_INVALID. */
  LERR_USER,     /* A user-defined error. */
  LERR_LENGTH
} error_code;
```

See also sections 28, 34, 46, 82, 91, 98, 114, 121, 135, 136, 141, 142, 180, 203, and 285.

This code is used in section 5.

12. This is the same list with the run-time label each will be bound to.

⟨ Data for initialisation 12 ⟩ ≡

```
shared char *Error_Label[LERR_LENGTH] ← {
    [LERR_FINISHED] ← "already-finished",
    [LERR_AMBIGUOUS] ← "ambiguous-syntax",
    [LERR_THREAD] ← "bad-join",
    [LERR_EXISTS] ← "conflicted-binding",
    [LERR_DOUBLE_TAIL] ← "double-tail",
    [LERR_EOF] ← "end-of-file",
    [LERR_IMMUTABLE] ← "immutable",
    [LERR_IMPROPER] ← "improper-list",
    [LERR_ADDRESS] ← "invalid-address",
    [LERR_INSTRUCTION] ← "invalid-instruction",
    [LERR_INCOMPATIBLE] ← "incompatible-operand",
    [LERR_INTERRUPT] ← "interrupted",
    [LERR_IO] ← "io",      /* Helpful. */
    [LERR_INTERNAL] ← "lossless-error",
    [LERR_MISMATCH] ← "mismatched-brackets",
    [LERR_MISSING] ← "missing",
    [LERR_NONCHARACTER] ← "noncharacter",
    [LERR_NONE] ← "no-error",
    [LERR_LISTLESS_TAIL] ← "non-list-tail",
    [LERR_OUT_OF_BOUNDS] ← "out-of-bounds",
    [LERR_OOM] ← "out-of-memory",
    [LERR_OVERFLOW] ← "overflow",
    [LERR_USER] ← "pebdac",      /* Problem Exists Between Developer And Compiler. */
    [LERR_BUSY] ← "resource-busy",
    [LERR_LEAK] ← "resource-leak",
    [LERR_LIMIT] ← "software-limit",
    [LERR_SYNTAX] ← "syntax-error",
    [LERR_SYSTEM] ← "system-error",
    [LERR_HEAVY_TAIL] ← "tail-mid-list",
    [LERR_UNCLOSED_OPEN] ← "unclosed-list",
    [LERR_UNCOMBINABLE] ← "uncombinable",
    [LERR_UNDERFLOW] ← "underflow",
    [LERR_UNIMPLEMENTED] ← "unimplemented",
    [LERR_UNLOCKED] ← "unlocked-resource",
    [LERR_UNOPENED_CLOSE] ← "unopened-list",
    [LERR_UNPRINTABLE] ← "unprintable",
    [LERR_UNSCANNABLE] ← "unscannable-lexeme",
    [LERR_EMPTY_TAIL] ← "unterminated-tail",
    [LERR_SELF] ← "wait-for-self",
};
```

See also sections 132, 139, 145, 278, and 281.

This code is used in section 8.

13. Each type of error has a unique object created to represent it at run-time. These objects are saved in the *Error* array as they are initialised.

Run-time objects will be described in greater detail after the memory layout has been defined.

#define *error_id_c*(*O*) (*A*(*O*)¬*sin*)
#define *error_label_c*(*O*) (*A*(*O*)¬*dex*)
#define *error_object*(*O*) (&*Error*[*fixed_value*(*error_id_c*(*O*))])

⟨ Global variables 13 ⟩ ≡
shared cell *Error*[LERR_LENGTH];

See also sections 17, 35, 47, 99, 106, 115, 122, 129, 137, 143, 146, 277, and 286.

This code is used in section 7.

14. ⟨ External C symbols 14 ⟩ ≡
extern shared cell *Error*[ ];

See also sections 18, 36, 48, 100, 107, 116, 123, 130, 138, 144, 147, 279, 282, and 287.

This code is used in section 5.

15. An error object is a tagged pair with the index into *Error* in one half and a symbol created from the above label in the other.

⟨ Initialise error symbols 15 ⟩ ≡
for (*i* ← 0; *i* < LERR_LENGTH; *i*++) {
orreturn (*new_symbol_cstr*(*Error_Label*[*i*], &*ltmp*)); /* New symbol. */
orreturn (*new_atom*(*fix*(*i*), *ltmp*, FORM_ERROR, &*Error*[*i*])); /* Error object; ID & label. */
orreturn (*env_save_m*(*Root*, *error_label_c*(*Error*[*i*]), *Error*[*i*], *false*)); /* Run-time inding. */
}

This code is used in section 126.

16. Naming things. Harder than the halting problem, this has also not been solved. Much improvement could be made to the names currently in use.

Macro constants are named in `ALL_CAPS`.

Global variables are named using *Title_Case*.

Everything else is named in *lower_case* except macro variables, which are a single capital letter.

Some variable names always mean the same thing in their local context:

o: The object being considered; the first or only argument to a function.

O: The same, in a macro.

reason: The local error code value.

ret: A pointer to the location to save the value being returned.

Function/macro naming conventions:

`new_`*noun*: Allocate memory for and return an object (and in one case, although it should arguably be two, `alloc_`*noun*).

`noun_`*verb*: Perform some action on or with an object.

These may be specialised further, eg. *new_segment* vs. *new_segment_copy*.

Prefix `get_`: Discard errors and return a value directly.

Prefix `set_`: Mutate an object.

Suffix `_p`: Predicate (returns a C boolean).

Suffix `_m`: Mutates some state, usually the object *o*.

Suffix `_imp`: The real implementation of some routine.

Suffix `_c`: An internal variant of a function accepting or returning C-formatted data.

Suffix `_ref`: Lookup by index in an array-like object.

An object's attribute mutator is named *object*`_set_`*attribute*`_m`.

Multiple suffixes are appended in an unspecified but — I think — strict order.

Other conventions.

Many objects are based on a memory allocation (known as a *segment*) and consist of a header followed by an arbitrary number of bytes. The macro to evaluate the C struct representing the object is named *object*`_pointer`; the base of the data is named *object*`_base`.

"Length" is the length of an object in whatever its base unit is, not bytes. If the number of bytes needs to be tracked too it is the "width".

17. Memory. `Lossless` arranges memory allocations in a hierarchy. The lowest level wraps the system allocation routines with error detection. At this level there is no automatic memory management and each allocation must be tracked and released explicitly.

The next layer organises allocations as a *segment* with a header including a (doubly[1]) linked list of segments along with other metadata. Segments are managed automatically and freed by the garbage collector when no longer in use.

The final layer classifies one or more segments as *heap*. These heap areas hold the *atoms* which make up individual objects and the necessary metadata to allocate them.

It's not clear that the `*_Ready` variables serve any practical purpose. TODO: review.

```
⟨ Global variables 13 ⟩ +≡
  char *malloc_options ← "S";      /* Enable malloc security features. */
  shared bool Memory_Ready ← false;      /* Main memory routines are ready. */
  unique bool Thread_Ready ← false;      /* Thread initialisation if finished. */
  shared bool Runtime_Ready ← false;      /* Lossless' run-time is ready. */
  shared bool VM_Ready ← false;      /* Environment populated & evaluator linked. */
```

18.
```
⟨ External C symbols 14 ⟩ +≡
  extern shared bool Memory_Ready, Runtime_Ready, VM_Ready;
  extern unique bool Thread_Ready;
```

19.
```
⟨ Function declarations 19 ⟩ ≡
  error_code init_mem(void);
  error_code alloc_mem(void *, size_t, size_t, void **);
  error_code free_mem(void *);
```
See also sections 31, 37, 49, 62, 79, 84, 101, 108, 117, 124, 150, 162, 181, 195, 232, 288, and 389.

This code is used in section 5.

20. The allocation routine simply checks the return value of *realloc* (or *aligned_alloc* if an aligned allocation was requested) and returns an appropriate error status.

`LLTEST` can be defined to replace the system allocator with one suitable for use in the test suite.

```
  error_code alloc_mem(void *old, size_t length, size_t align, void **ret)
  {
    void *r;
#ifdef LLTEST
    ⟨ Testing memory allocator 313 ⟩
#endif
    if (¬align) r ← realloc(old, length);
    else {
      assert(old ≡ Λ);
      if (length < align) length ← align;
      else if (length % align) return LERR_INCOMPATIBLE;
      r ← aligned_alloc(align, length);
    }
    if (r ≡ Λ)
      switch (errno) {
      case EINVAL: return LERR_INCOMPATIBLE;      /* align ≠ n^2 */
      case ENOMEM: return LERR_OOM;
      default: return LERR_INTERNAL;
      }
    *ret ← r;
    return LERR_NONE;
  }
```

[1] TODO: Why?

21. **error_code** *free_mem*(**void** ∗*o*)

```
{
#ifdef LLTEST
    ⟨ Testing memory deallocator 314 ⟩
#endif
    free(o);
    return LERR_NONE;
}
```

22. `Lossless` begins here. *init_mem* must be called exactly once and before accessing using any other part of `Lossless`. For the most part this allocates the primary heap and other areas of memory used at run-time, and initialises the mutexes which keep threads out of each others' memory.

```
⟨ initialise.c 8 ⟩ +≡
  error_code init_mem(void)
  {
    cell ltmp;
    segment ∗stmp;
    int i;
    error_code reason;
    assert(¬Memory_Ready ∧ ¬Runtime_Ready);
    ⟨ Initialise memory allocator 50 ⟩
    ⟨ Initialise heap 38 ⟩
    ⟨ Initialise symbol table 102 ⟩
    ⟨ Initialise program linkage 125 ⟩
    ⟨ Initialise foreign linkage 118 ⟩
    Memory_Ready ← true;
    orreturn (init_osthread( ));      /∗ Finish memory initialisation for the first thread. ∗/
    ⟨ Initialise threading 289 ⟩
    ⟨ Initialise run-time environment 109 ⟩
    Runtime_Ready ← true;
    return LERR_NONE;
  }
```

23. Portability. Not all the world's a VAX. 32-bit machines are already long in the tooth as `Lossless` is being written so it's only natural to also include support for 16-bit machines. These sections set various C constants and types so that a `Lossless` atom is exactly the size of two (data) pointers, regardless of how big such a pointer is.

Unfortunately it will be seen later that 16-bit support is lacking in the bytecode/interpreter which has not been improved to take such small machines into account.

```
#define TAG_BITS 8
#define TAG_BYTES 1
#define WORD_MAX INTPTR_MAX
#define WORD_MIN INTPTR_MIN
#define INTERN_MAX (ATOM_BYTES − 1)
#define FIXED_SHIFT 4
#define FIXED_MIN (ASR(INTPTR_MIN, FIXED_SHIFT))
#define FIXED_MAX (ASR(INTPTR_MAX, FIXED_SHIFT))
#define FIXED_BITS (CELL_BITS − FIXED_SHIFT)
```

⟨ Essential types 23 ⟩ ≡

```
  typedef int8_t byte;
  typedef intptr_t cell;
  typedef intptr_t word;
```

```
#if UINTPTR_MAX ≥ 0xffffffffffffffffUL
```

⟨ Define a 64-bit addressing environment 24 ⟩

```
#elif UINTPTR_MAX ≥ 0xffffffffUL
```

⟨ Define a 32-bit addressing environment 25 ⟩

```
#elif UINTPTR_MAX ≥ 0xffffUL
```

⟨ Define a 16-bit addressing environment 26 ⟩

```
#else
#error Tiny computer.
#endif
```

This code is used in section 5.

24. ⟨ Define a 64-bit addressing environment 24 ⟩ ≡

```
#define CELL_BITS 64 /* Total size of a cell. */
#define CELL_BYTES 8
#define CELL_SHIFT 4 /* How many low bits of a pointer are zero. */
#define ATOM_BITS 128 /* Total size of an atom. */
#define ATOM_BYTES 16
#define HALF_MIN INT32_MIN
#define HALF_MAX INT32_MAX
  typedef int32_t half; /* Records the size of memory objects. */
```

This code is used in section 23.

25. ⟨ Define a 32-bit addressing environment 25 ⟩ ≡

```
#define CELL_BITS 32
#define CELL_BYTES 4
#define CELL_SHIFT 3
#define ATOM_BITS 64
#define ATOM_BYTES 8
#define HALF_MIN INT16_MIN
#define HALF_MAX INT16_MAX
  typedef int16_t half;
```

This code is used in section 23.

26. It's unclear how useful this could be given that it's already 2022 but it costs little to include it.

⟨ Define a 16-bit addressing environment 26 ⟩ ≡

```
#define CELL_BITS  16
#define CELL_BYTES  2
#define CELL_SHIFT  2
#define ATOM_BITS  32
#define ATOM_BYTES  4
#define HALF_MIN  INT8_MIN
#define HALF_MAX  INT8_MAX
  typedef int8_t half;
```

This code is used in section 23.

27. Atoms. `Lossless` objects are referred to by pointers called *cells*, which point directly to an atom in its heap. Because each atom is exactly 4, 8 or 16 bytes wide (depending on machine size) each one has an address which is a multiple of 4, 8 or 16 and so any cell which points to an atom must have an even address. This means that if a cell value is *odd* then it can't possibly be a valid pointer to an atom; it won't point to a valid, allocated address.

The following values (including zero, which isn't actually odd) define constants which can be placed in a cell without the need to perform a heap allocation. The `FIXED` constant is a little different: instead of representing 'a fixed' if a cell's value is exactly 15 (1111 in binary), as with the other constants, if the bottom 4 bits of a cell are set then the rest of the cell encodes as much of a signed integer as will fit.

`UNDEFINED` is a sentinel marker which should never be realised as a run-time object.

```
#define NIL ((cell) 0)      /* Nothing, the empty list, (). */
#define LFALSE ((cell) 1)      /* Boolean false, #f or #F. */
#define LTRUE ((cell) 3)      /* Boolean true, #t or #T. */
#define VOID ((cell) 5)      /* Even less than nothing — the "no explicit value" value. */
#define LEOF ((cell) 7)      /* Value obtained off the end of a file or other stream. */
#define INVALID0 ((cell) 9)
#define INVALID1 ((cell) 11)
#define UNDEFINED ((cell) 13)      /* The value of a variable that isn't there. */
#define FIXED ((cell) 15)      /* A small fixed-width integer. */

#define null_p(O) ((O) ≡ NIL)      /* Might not be Λ. */
#define special_p(O) (null_p(O) ∨ ((O)) & 1)
#define boolean_p(O) ((O) ≡ LFALSE ∨ (O) ≡ LTRUE)
#define false_p(O) ((O) ≡ LFALSE)
#define true_p(O) ((O) ≡ LTRUE)
#define void_p(O) ((O) ≡ VOID)
#define eof_p(O) ((O) ≡ LEOF)
#define undefined_p(O) ((O) ≡ UNDEFINED)
#define fixed_p(O) (((O) & FIXED) ≡ FIXED)      /* Mask out the value bits. */
#define defined_p(O) (¬undefined_p(O))
#define valid_p(O) (fixed_p(O) ∨ ((((O) & FIXED) ≡ (O)) ∧ (O) ≠ INVALID0 ∧ (O) ≠ INVALID1))

#define predicate(O) ((O) ? LTRUE : LFALSE)
```

28. As already indicated an atom (from Greek *atomos* "indivisible") is the size of two pointers. Depending on what the atom represents these may represent pointers to other atoms (or constants) or opaque data.

Associated with each atom is a tag. Each tag is 8 bits wide. Two bits (`LTAG_LIVE` and `LTAG_TODO`) are used by the garbage collector. The remaining bits define the atom (it's atomic number, if you will) and the first two of these — of interest to the garbage collector in particular — indicate whether or not each half is a pointer to another atom (`LTAG_PSIN` and `LTAN_PDEX`). The lower 6 bits are known as the atoms *format*.

The macros below examine and update an atom's tag after locating it (see below for the definition of `ATOM_TO_TAG` which does the locating), the A, C and T macros are for typing convenience and A in particular is used extensively to access the atom via the union **atom**.

To make it clear that atoms are used to implement more than just "cons cells" the traditional names *car* and *cdr* are not used internally, instead the Latin terms sinister and dexter (shortened sin and dex) are used. These terms were deliberately chosen to be unfamiliar and evoke no sense of priority or order.

```
#define LTAG_LIVE 0x80    /* Atom has been reached from a register. */
#define LTAG_TODO 0x40    /* Atom has been partially scanned. */
#define LTAG_PSIN 0x20    /* Atom's sin half points to an atom. */
#define LTAG_PDEX 0x10    /* Atom's dex half points to an atom. */
#define LTAG_BOTH (LTAG_PSIN | LTAG_PDEX)
#define LTAG_FORM (LTAG_BOTH | 0x0f)
#define LTAG_NONE 0x00

#define TAG(O) (ATOM_TO_TAG((O)))
#define TAG_SET_M(O,V) (ATOM_TO_TAG((O)) ← (V))

#define ATOM_LIVE_P(O) (TAG(O) & LTAG_LIVE)
#define ATOM_CLEAR_LIVE_M(O) (TAG_SET_M((O),TAG(O) & ~LTAG_LIVE))
#define ATOM_SET_LIVE_M(O) (TAG_SET_M((O),TAG(O) | LTAG_LIVE))
#define ATOM_MORE_P(O) (TAG(O) & LTAG_TODO)
#define ATOM_CLEAR_MORE_M(O) (TAG_SET_M((O),TAG(O) & ~LTAG_TODO))
#define ATOM_SET_MORE_M(O) (TAG_SET_M((O),TAG(O) | LTAG_TODO))
#define ATOM_FORM(O) (TAG(O) & LTAG_FORM)
#define ATOM_SIN_DATUM_P(O) (TAG(O) & LTAG_PSIN)
#define ATOM_DEX_DATUM_P(O) (TAG(O) & LTAG_PDEX)

#define A(O) ((atom *)(O))
#define C(O) ((cell)(O))
#define T(O) (ATOM_FORM(O))
```

⟨ Type definitions 11 ⟩ +≡

```
typedef uint8_t cell_tag;
typedef union {
  struct {
    cell sin, dex;
  };
  struct {
    void *yin, *yang;
  };
  struct {
    void *number;      /* a segment, not defined yet. */
    word value;
  };
  struct {
    int8_t length;     /* Only 4 bits needed */
    byte buffer[INTERN_MAX];
  };
} atom;
```

29. All of the atom formats that `Lossless` recognises.

```
#define FORM_NONE (LTAG_NONE | 0x00) /* Unallocated. */
#define FORM_COLLECTED (LTAG_NONE | 0x01) /* Garbage collector tombstone. */
#define FORM_HASHTABLE (LTAG_NONE | 0x02) /* Key:value (or just key) store. */
#define FORM_HEAP (LTAG_NONE | 0x03) /* Preallocated storage for atoms. */
#define FORM_INTEGER (LTAG_NONE | 0x04) /* Large integer. */
#define FORM_RUNE (LTAG_NONE | 0x05) /* Unicode code point. */
#define FORM_SEGMENT (LTAG_NONE | 0x06) /* Large memory allocation. */
#define FORM_SEGMENT_INTERN (LTAG_NONE | 0x07) /* Tiny memory allocation. */
#define FORM_SYMBOL (LTAG_NONE | 0x08) /* Symbol. */
#define FORM_SYMBOL_INTERN (LTAG_NONE | 0x09) /* Tiny symbol. */

#define FORM_ARRAY (LTAG_PDEX | 0x00) /* Zero or more sequential cells. */
#define FORM_ASSEMBLY (LTAG_PDEX | 0x01) /* (Partially) assembled bytecode. */
#define FORM_CSTRUCT (LTAG_PDEX | 0x02) /* A C struct. */
#define FORM_FILE_HANDLE (LTAG_PDEX | 0x03) /* File descriptor or equivalent. */
#define FORM_POINTER (LTAG_PDEX | 0x04)
#define FORM_STATEMENT (LTAG_PDEX | 0x05) /* A single assembled statement. */

#define FORM_PAIR (LTAG_BOTH | 0x00) /* Two pointers (a "cons cell"). */
#define FORM_ARGUMENT (LTAG_BOTH | 0x01) /* An assembly statement argument. */
#define FORM_CLOSURE (LTAG_BOTH | 0x02) /* Applicative or operative closure. */
#define FORM_ENVIRONMENT (LTAG_BOTH | 0x03) /* Run-time environment. */
#define FORM_ERROR (LTAG_BOTH | 0x04) /* An error (above). */
#define FORM_OPCODE (LTAG_BOTH | 0x05) /* A virtual machine's operator. */
#define FORM_PRIMITIVE (LTAG_BOTH | 0x06) /* A Lossless operator. */
#define FORM_REGISTER (LTAG_BOTH | 0x07) /* A virtual machine register. */
#define FORM_SYNTAX (LTAG_BOTH | 0x08) /* A syntactic element (''; ,). */
```

30. Each format has a corresponding test. Some also share implementation or are otherwise related.

```
#define form(O) (ATOM_FORM(O))
#define form_p(O,F) (¬special_p(O) ∧ form(O) ≡ FORM_##F)
#define pair_p(O) (form_p((O), PAIR))
#define argument_p(O) (form_p((O), ARGUMENT))
#define array_p(O) (form_p((O), ARRAY))
#define assembly_p(O) (form_p((O), ASSEMBLY))
#define collected_p(O) (form_p((O), COLLECTED))
#define cstruct_p(O) (form_p((O), CSTRUCT))
#define environment_p(O) (form_p((O), ENVIRONMENT))
#define error_p(O) (form_p((O), ERROR))
#define file_handle_p(O) (form_p((O), FILE_HANDLE))
#define hashtable_p(O) (form_p((O), HASHTABLE))
#define heap_p(O) (form_p((O), HEAP))
#define opcode_p(O) (form_p((O), OPCODE))
#define pointer_p(O) (form_p((O), POINTER))
#define register_p(O) (form_p((O), REGISTER))
#define rune_p(O) (form_p((O), RUNE))
#define statement_p(O) (form_p((O), STATEMENT))
#define syntax_p(O) (form_p((O), SYNTAX))
#define segment_intern_p(O) (form_p((O), SEGMENT_INTERN))
#define segment_stored_p(O) (form_p((O), SEGMENT))
#define segment_p(O) (segment_intern_p(O) ∨ segment_stored_p(O))
#define symbol_intern_p(O) (form_p((O), SYMBOL_INTERN))
#define symbol_stored_p(O) (form_p((O), SYMBOL))
#define symbol_p(O) (symbol_intern_p(O) ∨ symbol_stored_p(O))
#define intern_p(O) (symbol_intern_p(O) ∨ segment_intern_p(O))
#define integer_heap_p(O) (form_p((O), INTEGER))
#define integer_p(O) (fixed_p(O) ∨ integer_heap_p(O))
#define arraylike_p(O) (array_p(O) ∨ hashtable_p(O) ∨ assembly_p(O) ∨ statement_p(O))
#define primitive_p(O) (form_p((O), PRIMITIVE))
#define closure_p(O) (form_p((O), CLOSURE))
#define program_p(O) (closure_p(O) ∨ primitive_p(O))
```

31. ⟨ Function declarations 19 ⟩ +≡

```
error_code new_atom_imp(heap *, cell, cell, cell_tag, cell *);
```

32. New atoms are allocated and with their constituent parts set "atomically". Pairs are by far the most common atom created and get their own macro.

```
#define cons(A,D,R) (new_atom((A),(D), FORM_PAIR, (R)))      /* cAr, cDr, R */
#define new_atom(S,D,T,R)      /* Sinister, Dexter, Tag, R */
        (new_atom_imp(Heap_Thread, (cell)(S), (cell)(D), (T), (R)))

error_code new_atom_imp(heap *where, cell nsin, cell ndex, cell_tag ntag, cell *ret)
{
  error_code reason;
  assert(heap_mine_p(where) ∨ heap_shared_p(where));
  orreturn (heap_root(where)→fun→alloc(where, ret));
  TAG_SET_M(*ret, ntag);
  A(*ret)→sin ← nsin;
  A(*ret)→dex ← ndex;
  return LERR_NONE;
}
```

33. Heap. An atom is allocated within a heap object, each of which is the size of an operating system *page*. A single heap consists of one or more heap objects linked together. Hereafter the term "heap" generally refers to an individual heap object to avoid saying "heap object" all the time.

Like every other allocation a heap is a segment and care is taken to ensure that the segment's header is not *added* to the allocation as it normally would be because each heap must be precisely aligned in memory.

The segment header is still present though at the bottom of the allocated range, followed immediately by the heap's own header.

The first set of macros here calculate the inner-header sizes, the amount of space left for atoms (`HEAP_LEFTOVER`) and thus the total size of the header proper (`HEAP_BOOKEND`) and padding (added together in `HEAP_HEADER`). `HEAP_LENGTH` is how many atoms are available in a heap.

The remaining macros mask off the high or low bits of an atom's address to find the heap an atom is within, or its index within that heap, respectively, and other aspects of the atom/heap as per the macro's name.

`HEAP_TO_LAST` evaluates to the address of an atom *past* the boundary of the heap.

```
#define SYSTEM_PAGE_LENGTH sysconf(_SC_PAGESIZE) /* An Operating System page length. */
#define HEAP_CHUNK (SYSTEM_PAGE_LENGTH) /* Size of a heap page (bytes). */
#define HEAP_MASK (HEAP_CHUNK - 1) /* Bits which will always be 0. */
#define HEAP_BOOKEND (sizeof(segment) + sizeof(heap)) /* Full header size. */
#define HEAP_LEFTOVER ((HEAP_CHUNK - HEAP_BOOKEND)/(TAG_BYTES + ATOM_BYTES))
#define HEAP_LENGTH ((int) HEAP_LEFTOVER) /* Heap data size (bytes). */
#define HEAP_HEADER ((HEAP_CHUNK/ATOM_BYTES) - HEAP_LENGTH) /* (bytes) */
#define ATOM_TO_ATOM(O) ((atom *)(O))
#define ATOM_TO_HEAP(O) (SEGMENT_TO_HEAP(ATOM_TO_SEGMENT(O))) /* The atom's heap. */
#define ATOM_TO_INDEX(O) (((((intptr_t)(O)) & HEAP_MASK) >> CELL_SHIFT) - HEAP_HEADER)
  /* The offset of an atom within a heap. */
#define ATOM_TO_SEGMENT(O) ((segment *)(((intptr_t)(O)) & ~HEAP_MASK))
  /* The segment containing an atom. */
#define HEAP_TO_SEGMENT(O) (ATOM_TO_SEGMENT(O)) /* The segment containing a heap. */
#define SEGMENT_TO_HEAP(O) ((heap *)(O)->base) /* The heap part of a segment. */
#define HEAP_TO_LAST(O) ((atom *)(((intptr_t) HEAP_TO_SEGMENT(O)) + HEAP_CHUNK))
  /* The (invalid) atom after the last valid atom within a heap. */
#define ATOM_TO_TAG(O) (ATOM_TO_HEAP(O)->tag[ATOM_TO_INDEX(O)]) /* The atom's tag. */
```

34. A heap is one or more heap objects linked together. The first such object in the chain is called the *root heap* and each subsequent heap points back to this root heap.

In place of the pointer to the root heap in an ordinary heap object, the root heap points to a **heap_pun** object. Like a regular heap object this punned object begins with a pointer. Unlike a regular heap object where this is a pointer is to the next free atom (or past the end of the heap) in the punned object pointed to by the root heap it holds the value `HEAP_PUN_FLAG` (-1).

This sort of dirty hack is not repeated again.

After the fake free pointer the punned object has pointers to the heap's allocator and other functions.

```
⟨ Type definitions 11 ⟩ +≡
  struct heap {
    atom *free;      /* Next unallocated atom. */
    struct heap *next, *other;      /* Next & twin heap pages. */
    struct heap_pun *root;      /* Root page or heap_access. */
    cell_tag tag[];      /* Atoms' tags. */
  };
  typedef struct heap heap;
  struct heap_pun {
    atom *free;
    struct heap *next, *other;
    struct heap_access *fun;
    cell_tag tag[];
  };
  typedef struct heap_pun heap_pun;
  typedef error_code(*init_heap_fn)(heap *, heap *, heap *, heap *);
  typedef error_code(*heap_enlarge_fn)(heap *, heap **);
  typedef bool(*heap_enlarge_p_fn)(heap_pun *, heap *);
  typedef error_code(*heap_alloc_fn)(heap *, cell *);
  struct heap_access {      /* TODO: This can be a thread local variable. */
    void *free;      /* Named free to look like a heap object. */
    init_heap_fn init;
    heap_enlarge_fn enlarge;
    heap_enlarge_p_fn enlarge_p;
    heap_alloc_fn alloc;
  };
  typedef struct heap_access heap_access;
```

35. There may be several heaps active in `Lossless`. One heap is created initially and is where allocations usually happen. This heap is saved in *Heap_Thread*, which is a "thread-local" variable. This means that each operating system thread has its own *Heap_Thread*, and thus its own heap.

The other heaps are initialised at run-time.

Note that almost none of this is actually implemented and only *Heap_Thread* is used.

```
#define HEAP_PUN_FLAG -1      /* Fake 'free pointer' sentinel to identify the root heap. */
#define heap_root_p(O) ((O)→root→free ≡ (void *) HEAP_PUN_FLAG)
#define heap_root(O) (heap_root_p(O) ? (heap_pun *)(O) : (O)→root)
⟨ Global variables 13 ⟩ +≡
  shared heap *Heap_Shared ← Λ;      /* Process-wide shared heap. */
  unique heap *Heap_Thread ← Λ;      /* Per-thread private heap. */
  unique heap *Heap_Trap ← Λ;      /* Per-thread heap for trap handler. */
```

36. ⟨ External C symbols 14 ⟩ +≡

```
  extern shared heap *Heap_Shared;
  extern unique heap *Heap_Thread, *Heap_Trap;
```

37. ⟨ Function declarations 19 ⟩ +≡

```
bool heap_mine_p(heap *);
bool heap_shared_p(heap *);
bool heap_trapped_p(heap *);
bool heap_other_p(heap *);
error_code init_heap_compacting(heap *, heap *, heap *, heap *);
error_code init_heap_sweeping(heap *, heap *, heap *, heap *);
bool heap_enlarge_p(heap_pun *, heap *);
error_code heap_enlarge(heap *, heap **);
error_code heap_alloc_freelist(heap *, cell *);
error_code heap_alloc_pointer(heap *, cell *);
```

38. This is the first allocation that `Lossless` will make and the only time the list of allocations will ever be empty. By setting its `next` and `prev` pointers to itself *claim_segment* can safely 'insert' this into what looks like a list of segment allocations.

The negative allocation length of −`HEAP_CHUNK` indicates to *alloc_segment* that the segment header should not increase the length of the allocation.

There is no need to lock *Allocations_Lock*.

Allocating the **heap_access** object here like this is an awful hack which will go away eventually.

⟨ Initialise heap 38 ⟩ ≡

```
orabort(alloc_segment(-HEAP_CHUNK, HEAP_CHUNK, &stmp));
Heap_Thread ← SEGMENT_TO_HEAP(stmp);
orabort(init_heap_sweeping(Heap_Thread, Λ, Λ, Λ));
orabort(alloc_mem(Λ, sizeof(heap_access), sizeof(void *),
    (void **) &((heap_pun *) Heap_Thread)→fun));    /* This is nasty... */
((heap_pun *) Heap_Thread)→fun→free ← (void *) HEAP_PUN_FLAG;
((heap_pun *) Heap_Thread)→fun→init ← init_heap_sweeping;
((heap_pun *) Heap_Thread)→fun→enlarge ← heap_enlarge;
((heap_pun *) Heap_Thread)→fun→enlarge_p ← heap_enlarge_p;
((heap_pun *) Heap_Thread)→fun→alloc ← heap_alloc_freelist;
orabort(new_atom(NIL, NIL, FORM_NONE, &ltmp));
Allocations ← HEAP_TO_SEGMENT(Heap_Thread);
Allocations→next ← Allocations→prev ← Allocations;
orabort(claim_segment(HEAP_TO_SEGMENT(Heap_Thread), ltmp, FORM_HEAP));
Heap_Shared ← Heap_Trap ← Λ;
```

This code is used in section 22.

39. Tests to query whether the current thread can access a heap (and thus, after using `ATOM_TO_HEAP` on its pointer, an atom).

```
bool heap_mine_p(heap *o)
{
  return (heap *) heap_root(o) ≡ Heap_Thread;
}
bool heap_shared_p(heap *o)
{
  return (heap *) heap_root(o) ≡ Heap_Shared;
}
bool heap_trapped_p(heap *o)
{
  return (heap *) heap_root(o) ≡ Heap_Trap;
}
bool heap_other_p(heap *o)
{
  return (heap *) heap_root(o) ≠ Heap_Thread ∧ (heap *) heap_root(o) ≠ Heap_Shared;
}
```

40. The heap pun trick allows the allocation and garbage collection algorithms to be chosen dynamically. There are two algorithms built in to `Lossless` named after the type of garbage collection each uses: sweeping and compacting.

After scanning for atoms which are in use, the garbage collector "sweeps" through a sweeping heap collecting the remaining unused atoms into a "free list". Alternatively the atoms in a compacting heap are *moved* by the garbage collector into a new heap from which atoms are allocated by incrementing a pointer until it overflows past the top of the heap.

The garbage collector algorithms have been removed from `Lossless` for now for being a suspect whenever memory corruption bugs were being hunted down.

```
#define initialise_atom(H, F) do {      /* Heap, Free */
        (H)→free--;      /* Move to the previous atom. */
        ATOM_TO_TAG((H)→free) ← FORM_NONE;      /* Free the atom. */
        (H)→free→sin ← NIL;      /* Cleanse the atom of sin. */
        if (F) (H)→free→dex ← (cell)((H)→free + 1);      /* Link the atom to the free list. */
        else (H)→free→dex ← NIL;      /* Scrub up the rest of the atom. */
      } while (0)
error_code init_heap_sweeping(heap *new, heap *prev, heap *other, heap *root)
{
  int i;
  assert(prev ≡ Λ ∨ heap_mine_p(prev) ∨ heap_shared_p(prev));
  assert(other ≡ Λ);
  assert(root ≡ Λ ∨ heap_mine_p((heap *) root) ∨ heap_shared_p((heap *) root));
  new→free ← HEAP_TO_LAST(new);      /* HEAP_TO_LAST returns a pointer after the last atom. */
  initialise_atom(new, false);      /* The last atom in the free list points to NIL. */
  for (i ← 1; i < HEAP_LENGTH; i++)
    initialise_atom(new, true);      /* The remaining atoms are linked together. */
  new→root ← (heap_pun *) root;
  new→other ← other;
  if (prev ≡ Λ) new→next ← Λ;
  else {
    new→next ← prev→next;
    prev→next ← new;
  }
  return LERR_NONE;
}
```

41. Each heap object in a compacting heap is allocated alongside its twin, to which atoms will be moved by the garbage collector.

TODO: This (and its missing GC etc.) remains untested.

```
error_code init_heap_compacting(heap *new, heap *prev, heap *other, heap *root)
{
  int i;
  assert(prev ≡ Λ ∨ heap_mine_p(prev) ∨ heap_shared_p(prev));
  assert(other ≠ Λ);
  assert(root ≡ Λ ∨ heap_mine_p(root) ∨ heap_shared_p(root));
  new→free ← HEAP_TO_LAST(new);
  other→free ← HEAP_TO_LAST(other);
  for (i ← 0; i < HEAP_LENGTH; i++) {
    initialise_atom(new, false);
    initialise_atom(other, false);
  }
  new→root ← (heap_pun *) root;
  new→other ← other; other→other ← new;      /* Link each page to its twin. */
  if (prev ≡ Λ) new→next ← other→next ← Λ;
  else {
    if ((new→next ← prev→next) ≠ Λ) {
      assert(new→next→other→next ≡ prev→other);
      new→next→other→next ← new→other;
    }
    other→next ← prev→other;
    prev→next ← new;
  }
  return LERR_NONE;
}
```

42. Garbage collection is not automatic. One of the functions associated with each heap is this function or one like it which reports whether to try and allocate a new heap object rather than resorting to garbage collection.

```
bool heap_enlarge_p(heap_pun *root unused, heap *at unused)
{
  return true;
}
```

43. If no part of a heap has any atoms free then the heap is enlarged by allocating a new page (or two) and linking it into the heap list.

TODO: Split this in two like the rest.

```
error_code heap_enlarge(heap *old, heap **ret)
{
  heap *new, *other;
  heap_pun *root;
  segment *snew, *sother;
  cell hnew, hother;
  error_code reason;
  assert(heap_mine_p(old) ∨ heap_shared_p(old));
  root ← heap_root(old);
  if (old→other ≡ Λ) {
    orreturn (alloc_segment(−HEAP_CHUNK, HEAP_CHUNK, &snew));
    new ← SEGMENT_TO_HEAP(snew);
    orreturn (root→fun→init(new, old, Λ, (heap *) root));
    orreturn (root→fun→alloc(new, &hnew));
    pthread_mutex_lock(&Allocations_Lock);
    orreturn (claim_segment(snew, hnew, FORM_HEAP));
    pthread_mutex_unlock(&Allocations_Lock);
  }
  else {
    orreturn (alloc_segment(−HEAP_CHUNK, HEAP_CHUNK, &snew));
    orreturn (alloc_segment(−HEAP_CHUNK, HEAP_CHUNK, &sother));
    new ← SEGMENT_TO_HEAP(snew);
    other ← SEGMENT_TO_HEAP(sother);
    orreturn (root→fun→init(new, old, other, (heap *) root));
    orreturn (root→fun→alloc(new, &hnew));
    orreturn (root→fun→alloc(new, &hother));
    pthread_mutex_lock(&Allocations_Lock);
    reason ← claim_segment(snew, hnew, FORM_HEAP);
    if (¬failure_p(reason)) reason ← claim_segment(sother, hother, FORM_HEAP);
    pthread_mutex_unlock(&Allocations_Lock);
    if (failure_p(reason)) return reason;
  }
  *ret ← new;
  return LERR_NONE;
}
```

44. A sweeping heap points to the next available atom or `NIL`. Allocation is a matter of removing it from the list and cleaning it up.

```
error_code heap_alloc_freelist(heap *where, cell *ret)
{
  bool tried;
  heap *h, *next;
  error_code reason;
  assert(heap_mine_p(where) ∨ heap_shared_p(where));
  tried ← false;
again: next ← where;
  while (next ≠ Λ) {
    h ← next;
    if (¬null_p(h→free)) {
      *ret ← (cell) h→free;
      h→free ← (atom *)(h→free→dex);
      ((atom *) *ret)→dex ← NIL;
      return LERR_NONE;
    }
    next ← h→next;
  }
  if (tried ∨ ¬heap_root(where)→fun→enlarge_p(heap_root(where), h)) return LERR_OOM;
  orreturn (heap_root(where)→fun→enlarge(h, &where));
  tried ← true;
  goto again;
}
```

45. Allocation from a compacting heap is done by incrementing a pointer if it's not already past the end of the heap. There is no need to clean the atom and this algorithm is fractionally faster than using the free list.

```
error_code heap_alloc_pointer(heap *where, cell *ret)
{
  bool tried;
  heap *h, *next;
  error_code reason;
  assert(heap_mine_p(where) ∨ heap_shared_p(where));
  tried ← false;
again: next ← where;
  while (next ≠ Λ) {
    h ← next;
    if (ATOM_TO_HEAP(h→free) ≡ where) {
      *ret ← (cell) h→free++;
      return LERR_NONE;
    }
    next ← h→next;
  }
  if (tried ∨ ¬heap_root(where)→fun→enlarge_p(heap_root(where), h)) return LERR_OOM;
  orreturn (heap_root(where)→fun→enlarge(h, &where));
  tried ← true;
  goto again;
}
```

46. Segments. Every allocation in `Lossless` is a segment or is within a segment: an arbitrary-size memory allocation. Three objects are used internally to define segments:

A *pointer* is anything with a C pointer in its sinister half and an ignored cell in its dexter half.

A *segment* is such a pointer which points to an allocation.

An *interned segment* is an allocation that's small enough to fit within the atom that would otherwise be a pointer. This can only be achieved for objects which don't need the segment header data of which there are two: plain segments and symbols.

Every segment (except interned segments) is included in a global list via its `next` and `prev` pointers.

Note that `NULL` (Λ) and `NIL` are different, although they will likely both have the numeric value zero.

```
#define pointer(O) (A(O)→yin)
#define pointer_datum(O) (A(O)→dex)
#define pointer_set_m(O,V) (A(O)→yin ← (V))
#define pointer_set_datum_m(O,V) (A(O)→dex ← (V))
#define null_pointer_p(O) (pointer(O) ≡ Λ)

#define segment_object(O) ((segment *) pointer(O))
#define segment_base(O) (intern_p(O) ? A(O)→buffer : segment_object(O)→base)
#define segment_length_c(O) (intern_p(O) ? A(O)→length : segment_object(O)→length)

#define SEGMENT_MAX HALF_MAX
```

⟨ Type definitions 11 ⟩ +≡

```
struct segment {
  struct segment *next, *prev;
  cell owner;
  half length, scan;
  byte base[];
};
typedef struct segment segment;
```

47. Any thread can allocate a segment (or clean them up during garbage collection). *Allocations_Lock* is a mutex which ensure that no two threads attempt to do so at the same time.

⟨ Global variables 13 ⟩ +≡

```
shared segment *Allocations ← Λ;
shared pthread_mutex_t Allocations_Lock;
```

48. ⟨ External C symbols 14 ⟩ +≡

```
extern shared segment *Allocations;
extern shared pthread_mutex_t Allocations_Lock;
```

49. ⟨ Function declarations 19 ⟩ +≡

```
error_code alloc_segment(half, intptr_t, segment **);
error_code claim_segment(segment *, cell, cell_tag);
error_code new_pointer(address, cell *);
error_code new_segment_imp(heap *, half, intptr_t, cell_tag, cell_tag, cell *);
error_code segment_peek(cell, half, int, bool, cell *);
error_code segment_poke(cell, half, int, bool, cell);
error_code segment_resize_m(cell, half);
```

50. ⟨ Initialise memory allocator 50 ⟩ ≡

```
orabort(init_osthread_mutex(&Allocations_Lock, false, false));
```

This code is used in section 22.

51. Before moving on, it is helpful to be able to create pointer objects which aren't pointing to segments.

```
error_code new_pointer(address o, cell *ret)
{
  return new_atom((cell) o, NIL, FORM_POINTER, ret);
}
```

52. The main stage of allocating a segment is to obtain the memory from the operating system and fill in the header absent links to other segments.

```
error_code alloc_segment(half length, intptr_t align, segment **ret)
{
  word rlength;
  segment *new;
  error_code reason;
  assert(length ≡ −HEAP_CHUNK ∨ (length ≥ 0 ∧ length ≤ SEGMENT_MAX));
  if (length < 0) rlength ← HEAP_CHUNK;
  else rlength ← length + sizeof(segment);
  orreturn (alloc_mem(Λ, rlength, align, (void **) &new));
  if (length < 0) new→length ← HEAP_LENGTH;
  else new→length ← length;
  *ret ← new;
  return LERR_NONE;
}
```

53. Saving the allocation in the global list is done here after the caller has locked the *Allocations_Lock* mutex. The atom and its tag are updated here after the list has been updated but while the lock is still held to ensure that nothing tries to follow pointers that aren't there yet.

```
error_code claim_segment(segment *area, cell owner, cell_tag ntag)
{
  assert(Allocations ≠ Λ);
  area→next ← Allocations;
  area→prev ← Allocations→prev;
  Allocations→prev→next ← area;
  Allocations→prev ← area;
  area→owner ← owner;
  A(owner)→yin ← area;
  A(owner)→dex ← NIL;
  TAG_SET_M(owner, ntag);      /* Do this last so the atom remains opaque until ready. */
  return LERR_NONE;
}
```

54. To create the new segment first the length and proposed tag are checked to see if a full allocation is needed. If so the allocations are all performed and claiming the lock is left until the last possible moment.

```
#define new_segment(L, A, R)      /* Length, Align, R */
        new_segment_imp(Heap_Thread, (L), (A), FORM_SEGMENT, FORM_SEGMENT_INTERN, (R))
  error_code new_segment_imp(heap *where, half length, intptr_t align, cell_tag ntag, cell_tag
          itag, cell *ret)
  {
    cell holder;
    segment *area;
    error_code reason;
    if (itag ≠ FORM_NONE) {
      if (length > INTERN_MAX) goto new_allocation;
      orreturn (new_atom_imp(where, NIL, NIL, itag, ret));
      A(*ret)¬length ← length;
    }
    else {
    new_allocation: orreturn (new_atom_imp(where, NIL, NIL, FORM_NONE, &holder));
      orreturn (alloc_segment(length, align, &area));
      pthread_mutex_lock(&Allocations_Lock);
      reason ← claim_segment(area, holder, ntag);
      pthread_mutex_unlock(&Allocations_Lock);
      if (failure_p(reason)) return reason;
      *ret ← holder;
    }
    return LERR_NONE;
  }
```

55. A segment can be resized "in-place" however this may mean converting to or from an interned segment if the allocation size crosses the `INTERN_MAX` boundary.

```
  error_code segment_resize_m(cell o, half nlength)
  {
    half i, olength;
    word rlength;
    byte *new, *old;
    segment *embiggen;
    error_code reason;
    assert((segment_p(o) ∨ arraylike_p(o)));
    olength ← segment_length_c(o);
    if (nlength ≡ olength) return LERR_NONE;      /* Not an error. */
    if (¬segment_p(o) ∨ (nlength | olength) > INTERN_MAX) {⟨ Resize an allocated segment 58 ⟩}
    else if (nlength ≤ INTERN_MAX) {
      if (olength ≤ INTERN_MAX) A(o)¬length ← nlength;
      else {⟨ Intern a previously allocated segment 56 ⟩}
    }
    else {⟨ Allocate a segment for a previously interned segment 57 ⟩}
    return LERR_NONE;
  }
```

56. If a plain segment is being reduced enough the tag of the atom is changed and the allocation is left without an owner to be cleaned up by the garbage collector. The contents are copied as far as they will fit.

```
⟨ Intern a previously allocated segment 56 ⟩ ≡
  TAG_SET_M(o, FORM_SEGMENT_INTERN);      /* Do this first to turn the atom opaque. */
  old ← segment_object(o)→base;
  new ← A(o)→buffer;
  for (i ← 0; i < nlength; i++) new[i] ← old[i];
  A(o)→length ← nlength;
```

This code is used in section 55.

57. When a segment has outgrown its internment a new allocation is made and the amount of space they shared is copied into it before modifying the the atom's tag and contents to point to it.

```
⟨ Allocate a segment for a previously interned segment 57 ⟩ ≡
  olength ← segment_length_c(o);
  old ← segment_base(o);
  orreturn (alloc_segment(nlength, 0, &embiggen));
  new ← embiggen→base;
  for (i ← 0; i < olength; i++) new[i] ← old[i];
  pthread_mutex_lock(&Allocations_Lock);
  orreturn (claim_segment(embiggen, o, FORM_SEGMENT));
  pthread_mutex_unlock(&Allocations_Lock);
```

This code is used in section 55.

58. Resizing a segment which was not and will not be interned is performed by the backend memory allocator as far as the allocation and its contents are concerned. The atom is changed to point to the "new" allocation however there is no "old" allocation — this is taken care of by the allocator — except that the address may have changed.

TODO: Check whether the lock should be held (or the allocation removed from the list while locked) for the duration of *alloc_mem*.

```
⟨ Resize an allocated segment 58 ⟩ ≡
  rlength ← nlength + sizeof(segment);
  old ← (byte *) pointer(o);
  if (pthread_mutex_lock(&Allocations_Lock) ≠ 0) return LERR_INTERNAL;
  orreturn (alloc_mem(pointer(o), rlength, 0, (void **) &embiggen));
  if (embiggen ≠ (segment *) old) {
    embiggen→next→prev ← embiggen;
    embiggen→prev→next ← embiggen;
  }
  pointer_set_m(o, embiggen);
  pthread_mutex_unlock(&Allocations_Lock);
  embiggen→length ← nlength;
```

This code is used in section 55.

59. Data within a segment is read in words of 1, 2, 4 or 8 bytes. The address to read from need not be aligned to a multiple of the size of word being read, which may cause a bus fault on some architectures.

The word is interpreted as unsigned.

```
error_code segment_peek(cell o, half index,      /* Always byte address? */
int width,      /* 1, 2, 4, 8 */
bool lilliput, cell *ret)
{
  byte *s;
  uintmax_t v;
  error_code reason;
  assert(¬heap_other_p(ATOM_TO_HEAP(o)));
  assert(segment_p(o));
  assert(index ≥ 0 ∧ index < segment_length_c(o));
  assert(width ≡ 1 ∨ width ≡ 2 ∨ width ≡ 4 ∨ width ≡ 8);
  s ← segment_base(o);
  if (lilliput)
    switch (width) {
    case 1: v ← ((uint8_t *) s)[index]; break;
    case 2: v ← le16toh(*((uint16_t *)(s + index))); break;
    case 4: v ← le32toh(*((uint32_t *)(s + index))); break;
    case 8: v ← le64toh(*((uint64_t *)(s + index))); break;
    }
  else
    switch (width) {
    case 1: v ← ((uint8_t *) s)[index]; break;
    case 2: v ← be16toh(*((uint16_t *)(s + index))); break;
    case 4: v ← be32toh(*((uint32_t *)(s + index))); break;
    case 8: v ← be64toh(*((uint64_t *)(s + index))); break;
    }
  if (v > WORD_MAX) {
    orreturn (new_int(2, false, ret));
    int_buffer_c(*ret)[1] ← *(word *) &v;
    return LERR_NONE;
  }
  else return new_int_c(v, ret);
}
```

60. Writing to a segment is the same but backwards.

```
error_code segment_poke_m(cell o, half index,      /* Always byte address? */
int width,      /* 1, 2, 4, 8 */
bool lilliput, cell lvalue)
{
  byte *s;
  uintmax_t cvalue;
  assert(¬heap_other_p(ATOM_TO_HEAP(o)));
  assert(segment_p(o));
  assert(index ≥ 0 ∧ index < segment_length_c(o));
  assert(width ≡ 1 ∨ width ≡ 2 ∨ width ≡ 4 ∨ width ≡ 8);
  assert(integer_p(lvalue));
  assert(false);
  cvalue ← A(lvalue)→value & ((2 ⊕ (8 * width)) − 1);
  s ← segment_base(o);
  if (lilliput)
    switch (width) {
    case 1: ((uint8_t *) s)[index] ← cvalue; break;
    case 2: *((uint16_t *)(s + index)) ← htole16(cvalue); break;
    case 4: *((uint32_t *)(s + index)) ← htole32(cvalue); break;
    case 8: *((uint64_t *)(s + index)) ← htole64(cvalue); break;
    }
  else
    switch (width) {
    case 1: ((uint8_t *) s)[index] ← cvalue; break;
    case 2: *((uint16_t *)(s + index)) ← htobe16(cvalue); break;
    case 4: *((uint32_t *)(s + index)) ← htobe32(cvalue); break;
    case 8: *((uint64_t *)(s + index)) ← htobe64(cvalue); break;
    }
  return LERR_NONE;
}
```

61. Objects.

62. Integers. Few mathematical routines are required in the core of `Lossless`, chiefly the ability to add and subtract small numbers, however thought must be given to how memory will be organised for large integers so that they're compatible with the subset implemented in here.

To this end there are two integer formats used internally by `Lossless`: *fixed width* integers which do not use any allocated storage and *variable width* integers which do.

Fixed width integers are described briefly above in the introduction to atoms — 4 bits of an address are reserved to indicate that a value is really a fixed width integer and the remaining bits (12, 28 or 60) encode a signed integer value. Two macros *fix* and *fixed_value* store and access the value of a fixed width integer.

Variable width integers are stored in multiples of *words*, which are signed integers the same size as a cell. If a single word is enough then the space within the atom is used without the need for a segment allocation.

This chapter also includes an assortment of numerical routines not directly related to integer objects.

```
#define INT_LENGTH_MAX (HALF_MAX/sizeof(cell))
#define int_vcast(O) ((word *) &A(O)→value)
#define int_scast(O) ((word *) segment_base(O))
#define int_buffer_c(O) (null_pointer_p(O) ? int_vcast(O) : int_scast(O))
#define int_length_c(O) (null_pointer_p(O) ? 1 : segment_length_c(O)/sizeof(word))
#define int_negative_p(O) ((integer_heap_p(O) ? int_buffer_c(O)[0] : fixed_value(O)) < 0)
#define fix(V) (FIXED | ASL((V), FIXED_SHIFT))
#define fixed_value(V) (ASR((V), FIXED_SHIFT))
```

⟨ Function declarations 19 ⟩ +≡

```
error_code new_int_c(intmax_t, cell *);
error_code new_int(intmax_t, bool, cell *);
bool cmpis_p(cell, cell);
bool int_eq_p(cell, cell);
bool int_eq_p_imp(cell, cell);
error_code int_length(cell, cell *);
error_code int_to_symbol(cell, cell *);
error_code int_value(cell, word *);
error_code int_cmp(cell, cell, cell *);
error_code int_normalise(cell, cell *);
error_code int_add(cell, cell, cell *);
error_code int_sub(cell, cell, cell *);
error_code int_mul(cell, cell, cell *);
int high_bit(uintmax_t);
```

63. To ensure that the sign bits are maintained when shifting numbers left and right the macros `ASL` and `ASR` ensure that an arithmetic (sign-preserving) shift is used regardless of the architecture.

⟨ Hacks and warts 9 ⟩ +≡

```
#define ASL(V, I) ((V) ≪ (I))
#if ((−1) ≫ 1) ≡ −1
#define ASR(V, I) ((V) ≫ (I))
#else
#define ASR(V, I)((V) ≥ 0) ? ((V) ← (V) ≫ (I)) : ((V) ← ∼((∼(V)) ≫ (I)))
#endif
```

64. Returns a value representing the highest bit set in a number.

```
int high_bit(uintmax_t o)
{
  int i ← CELL_BITS;
  while (--i) if (o & (1ULL ≪ i)) return i;
  assert(o ≡ 0 ∨ o ≡ 1);
  return o − 1;
}
```

65. If a C integer is small enough to fit within the space of a fixed integer then *new_int_c* returns one without allocation, otherwise the integer is being created from a word[1] so of course it will fit within the single word available to an atom.

```
error_code new_int_c(intmax_t value, cell *ret)
{
  if (value ≥ FIXED_MIN ∧ value ≤ FIXED_MAX) {
    *ret ← fix(value);
    return LERR_NONE;
  }
  return new_atom((cell) Λ, (cell) value, FORM_INTEGER, ret);
}
```

66. Integers larger than a single word are created by initialising the storage with *new_int* and setting it to an initial value of 0 or -1 (all bits set).

```
error_code new_int(intmax_t length, bool negative, cell *ret)
{
  error_code reason;
  assert(length > 1 ∧ length < (intmax_t) INT_LENGTH_MAX);
  orreturn (new_segment_imp(Heap_Thread, length * sizeof(cell), sizeof(word), FORM_INTEGER,
      FORM_NONE, ret));
  memset(segment_base(*ret), −negative, length);
  return LERR_NONE;
}
```

67. The converse of *new_int_c* is *int_value* which extracts the value from any integer into a C variable, if it fits. The macro **or_int_value_bounds** is used when the integer is about to be used to check that a value is within the boundaries of, eg., an array and the error LERR_OUT_OF_BOUNDS is more appropriate than the usual LERR_LIMIT.

```
#define or_int_value_bounds(O, R)
        if ((reason ← int_value((O), (R))) ≡ LERR_LIMIT) return LERR_OUT_OF_BOUNDS;
        else if (failure_p(reason)) return reason      /* nb. no semicolon. */
error_code int_value(cell o, word *ret)
{
  assert(integer_p(o));
  if (fixed_p(o)) *ret ← fixed_value(o);
  else if (null_pointer_p(o)) *ret ← A(o)→value;
  else return LERR_LIMIT;
  return LERR_NONE;
}
```

[1] Technically **intmax_t** might not be the same as **intptr_t**; this should be looked into.

68. The length of an integer is the number of words of storage it uses.

```
error_code int_length(cell o, cell *ret)
{
  assert(integer_p(o));
  if (fixed_p(o)) *ret ← 0;
  else if (null_pointer_p(o)) *ret ← 1;
  else return new_int_c(int_length_c(o), ret);
  return LERR_NONE;
}
```

69. Normalising an integer removes excess leading zero/sign words. This is optional but can save space.

```
error_code int_normalise(cell o, cell *ret)
{
  half length;
  word *pint;
  error_code reason;
  assert(integer_p(o));
  if (fixed_p(o)) {
    *ret ← o;
    return LERR_NONE;
  }
  length ← int_length_c(o);
  pint ← int_buffer_c(o);
  if (int_negative_p(o))
    while (length > 1 ∧ *pint ≡ −1) length−−, pint++;
  else
    while (length > 1 ∧ ¬*pint) length−−, pint++;
  if (length > 1) {
    orreturn (new_segment_imp(Heap_Thread, length * sizeof(cell), sizeof(word), FORM_INTEGER,
        FORM_NONE, ret));
    memmove(int_buffer_c(*ret), pint, length);
    return LERR_NONE;
  }
  else return new_int_c(*pint, ret);
}
```

70. Like symbols numbers are uniquely themselves but unlike symbols they're not symbols. When an integer needs to be used in a place where a symbol is expected (eg. as the key to a hashtable entry) this function constructs a symbol from the integer's value. The binary representation of the number becomes the label of the symbol so in most cases the symbol will be unprintable.

Leading zero/sign words are skipped, so unnormalised and normalised integers convert to the same symbol.

```
error_code int_to_symbol(cell o, cell *ret)
{
  bool negative;
  word *ib, *last, value;
  assert(integer_p(o));
  if (fixed_p(o)) {
    value ← fixed_value(o);
small_integer:
    return new_symbol_buffer((byte *) &value, sizeof(word), Λ, ret);
  }
  else if (null_pointer_p(o)) {
    value ← A(o)→value;
    goto small_integer;
  }
  else {
    ib ← int_buffer_c(o);
    last ← ib + int_length_c(o);
    negative ← *ib < 0;
    value ← 0;
    for ( ; *ib ≡ −negative; ib++) {
      if (ib ≡ last) goto small_integer;
      else if (*ib ≠ −negative) break;
    }
    return new_symbol_buffer((byte *) ib, (last − ib) * sizeof(word), Λ, ret);
  }
}
```

71. The `Lossless` core needs integer support for some simple arithmetic (below) and these comparison routines. This first comparison routine implements `is?`, which is similar in spirit but not in scope to `eq?` in Scheme, the difference chiefly being that integers are compared *numerically* while all other objects' *identities* are compared.

```
bool cmpis_p(cell yin, cell yang)
{
  if (integer_heap_p(yin) ∧ integer_heap_p(yang)) return int_eq_p_imp(yin, yang);
  else return yin ≡ yang;
}
```

72. On the other hand `Lossless`' equality routine is *only* applicable to integers.

```
bool int_eq_p(cell yin, cell yang)
{
  assert(integer_p(yin));
  assert(integer_p(yang));
  if (fixed_p(yin) ∧ fixed_p(yang)) return yin ≡ yang;
  else if (integer_heap_p(yin) ∧ integer_heap_p(yang)) return int_eq_p_imp(yin, yang);
  else return false;
}
```

73. Integers are compared for equality word-by-word starting with the least significant. To account for potentially unnormalised integers of different lengths the excess words of the longer integer are compared to 0, or -1 if the integers are negative.

```
bool int_eq_p_imp(cell yin, cell yang)
{
  bool negative;
  word *pg, *pn;
  half lg, ln;
  assert(integer_heap_p(yin));
  assert(integer_heap_p(yang));
  if ((ln ← int_length_c(yin)) ≡ 1) pn ← &(A(yin)→value);
  else pn ← ((word *) segment_base(yin)) + ln − 1;
  if ((lg ← int_length_c(yang)) ≡ 1) pg ← &(A(yang)→value);
  else pg ← ((word *) segment_base(yang)) + lg − 1;
  while (1) {
    negative ← *pg < 0;
    if (*pn−− ≠ *pg−−) return false;
    ln−−;
    lg−−;
    if (ln ≡ 0) goto finish_yin;
    negative ← *(pn + 1) < 0;
    if (lg ≡ 0) goto finish_yang;
  }
finish_yang: lg ← ln; pg ← pn;
finish_yin:
  if (negative) {
    for ( ; lg > 0; lg−−)
      if (*pg−− ≠ −1) return false;
  }
  else {
    for ( ; lg > 0; lg−−)
      if (*pg−−) return false;
  }
  return true;
}
```

74. This routine returns -1, 0 or 1 respectively if *yin* is less than, equal to or greater than *yang*.

```
#define int_cmp_hack(N) (((N) * 2) + 1)    /* -1 becomes -1, 0 becomes 1 */
  error_code int_cmp(cell yin, cell yang, cell *ret)
  {
    half lyin, lyang;
    word negative, *pyin, *pyang, vyin, vyang;
    assert(integer_p(yin));
    assert(integer_p(yang));
    if (fixed_p(yin)) {
      vyin ← fixed_value(yin);
      pyin ← &vyin;
      lyin ← 1;
    }
    else {
      pyin ← int_buffer_c(yin);
      lyin ← int_length_c(yin);
    }
    if (fixed_p(yang)) {
      vyang ← fixed_value(yang);
      pyang ← &vyang;
      lyang ← 1;
    }
    else {
      pyang ← int_buffer_c(yang);
      lyang ← int_length_c(yang);
    }
    negative ← −(*pyin < 0);
    if (negative ∧ *pyang ≥ 0) {    /* yin < yang */
      *ret ← fix(−1);
      return LERR_NONE;
    }
    else if (¬negative ∧ *pyang < 0) {    /* yin > yang */
      *ret ← fix(1);
      return LERR_NONE;
    }
    while (lyin > 1 ∧ *pyin ≡ negative) pyin++, lyin−−;    /* Skip leading 0s. */
    while (lyang > 1 ∧ *pyang ≡ negative) pyang++, lyang−−;
    if (lyin < lyang) {
      *ret ← fix(−int_cmp_hack(negative));
      return LERR_NONE;
    }
    else if (lyin > lyang) {
      *ret ← fix(int_cmp_hack(negative));
      return LERR_NONE;
    }
    while (lyin−−) {    /* Same length, same sign; first difference wins. */
      vyin ← *pyin++;
      vyang ← *pyang++;
      if (vyin ≡ vyang) continue;
      if (vyin < vyang) *ret ← fix(−int_cmp_hack(negative));
      else *ret ← fix(int_cmp_hack(negative));
      return LERR_NONE;
    }
    *ret ← fix(0);
    return LERR_NONE;
  }
```

75. Only the three basic algebraic routines of addition, subtraction and multiplication are necessary. Non-standard functions built in to GCC and Clang, among others, are used in place of the regular C operators. These trap overflow using appropriate CPU instructions which is shorter and faster than relying on explicit checks of the operands and/or result.

TODO: These hacks should be actual hacks which check for the presence of the built-in routines and define substitutes if they are not.

⟨ Hacks and warts 9 ⟩ +≡
#define *ckd_add*(r, x, y) *__builtin_add_overflow*$((x), (y), (r))$
#define *ckd_sub*(r, x, y) *__builtin_sub_overflow*$((x), (y), (r))$
#define *ckd_mul*(r, x, y) *__builtin_mul_overflow*$((x), (y), (r))$

76. The addition routine works with all integers. It creates a buffer one word longer than the longest of the two addends and then adds the two arguments into it one word at a time starting at the least significant word. The result is then normalised and returned.

One integer is copied into the newly allocated space and then the other added to it rather than both being added together. I can't think of a good reason why, now.

Because fixed-width integers always have four bits to spare they can be added together much more quickly than going through the full variable-width algorithm.

```
error_code int_add(cell yin, cell yang, cell *ret)
{
  cell result;
  half lyin, lyang;
  word carry, *next, *presult, *pyin, *pyang, vyin, vyang;
  error_code reason;
  assert(integer_p(yin));
  assert(integer_p(yang));
  if (fixed_p(yin)) {
    vyin ← fixed_value(yin);
    if (fixed_p(yang)) return new_int_c(vyin + fixed_value(yang), ret);
    pyin ← &vyin;
    lyin ← 1;
  }
  else {
    pyin ← int_buffer_c(yin);
    lyin ← int_length_c(yin);
  }
  if (fixed_p(yang)) {
    vyang ← fixed_value(yang);
    pyang ← &vyang;
    lyang ← 1;
  }
  else {
    pyang ← int_buffer_c(yang);
    lyang ← int_length_c(yang);
  }
  if (lyin > lyang) orreturn (new_int(lyin + 1, int_negative_p(yin), &result));
  else orreturn (new_int(lyang + 1, int_negative_p(yin), &result));
  presult ← int_buffer_c(result);
  next ← presult + int_length_c(result);
  memmove(next − lyin, pyin, lyin);
  carry ← 0;
  for ( ; lyang; lyang−−) {
    next−−;
    carry ← ckd_add(next, *next, carry);
    carry |= ckd_add(next, *next, pyang[lyang − 1]);
  }
  if (carry) *(−−next) += carry;
  return int_normalise(result, ret);
}
```

77. The subtraction routine is the same except for using subtraction instead of addition.

```
error_code int_sub(cell yin, cell yang, cell *ret)
{
  cell result;
  half lyin, lyang;
  word carry, *next, *presult, *pyin, *pyang, vyin, vyang;
  error_code reason;
  assert(integer_p(yin));
  assert(integer_p(yang));
  if (fixed_p(yin)) {
    vyin ← fixed_value(yin);
    if (fixed_p(yang)) return new_int_c(vyin − fixed_value(yang), ret);
    pyin ← &vyin;
    lyin ← 1;
  }
  else {
    pyin ← int_buffer_c(yin);
    lyin ← int_length_c(yin);
  }
  if (fixed_p(yang)) {
    vyang ← fixed_value(yang);
    pyang ← &vyang;
    lyang ← 1;
  }
  else {
    pyang ← int_buffer_c(yang);
    lyang ← int_length_c(yang);
  }
  if (lyin > lyang) orreturn (new_int(lyin + 1, int_negative_p(yin), &result));
  else orreturn (new_int(lyang + 1, int_negative_p(yin), &result));
  presult ← int_buffer_c(result);
  next ← presult + int_length_c(result);
  memmove(next − lyin, pyin, lyin);
  carry ← 0;
  for ( ; lyang; lyang−−) {
    next−−;
    carry ← ckd_sub(next, *next, carry);
    carry |= ckd_sub(next, *next, pyang[lyang − 1]);
  }
  if (carry) *(−−next) −= carry;
  return int_normalise(result, ret);
}
```

78. Multiplication accepts any integer as an argument but large integers are not supported. If both multiplicands are fixed-width integers then normal multiplication is attempted and if the result still fits within a fixed width integer it's returned. Otherwise the error `LERR_UNSUPPORTED` is raised which is expected to be trapped and full multiplication implemented at a higher level.

```
error_code int_mul(cell yin, cell yang, cell *ret)
{
  word result;
  assert(integer_p(yin));
  assert(integer_p(yang));
  if (yin ≡ fix(0) ∨ yang ≡ fix(0)) {
    *ret ← fix(0);
    return LERR_NONE;
  }
  else if (yin ≡ fix(1)) {
    *ret ← yang;
    return LERR_NONE;
  }
  else if (yang ≡ fix(1)) {
    *ret ← yin;
    return LERR_NONE;
  }
  if (¬fixed_p(yin) ∨ ¬fixed_p(yang)) return LERR_UNIMPLEMENTED;
  if (ckd_mul(&result, fixed_value(yin), fixed_value(yang))) return LERR_UNIMPLEMENTED;
  return new_int_c(result, ret);
}
```

79. Arrays. A sequence of zero or more cells is an *array*. It's assumed that callers accessing an array's elements have checked that their index value is within bounds.

Arrays in `Lossless` are segments and due to how segments are stored the atom pointing to the array data has a spare cell with no purpose. To make use of this spare space it's called the array's *offset* and holds an integer which can be subtracted from a normal array reference index to find the real offset in memory (ie. from zero)[1]. Nothing in `Lossless`' core uses this feature except to expose the value for calculation with at a higher level.

```
#define ARRAY_MAX HALF_MAX
#define array_base(O) ((cell *) segment_base(O))
#define array_length_c(O) (segment_length_c(O)/(half) sizeof(cell))
#define array_offset_c(O) (pointer_datum(O))
```

⟨ Function declarations 19 ⟩ +≡

```
error_code new_array_imp(half, cell, cell, cell_tag, cell *);
error_code array_resize_m(cell, half, cell);
```

80. Most arrays are normal arrays with — at creation their slots are initialised to `NIL` and they're created with *new_array*.

Some objects are defined in terms of underlying an array store and begin with their contents uninitialised.

```
#define new_array(L, O, R) /* Length, Offset, R */
        new_array_imp((L), (O), NIL, FORM_ARRAY, (R))
error_code new_array_imp(half length, cell offset, cell fill, cell_tag form, cell *ret)
{
  error_code reason;
  assert(length ≥ 0 ∧ length ≤ ARRAY_MAX);
  assert(integer_p(offset));
  orreturn (new_segment_imp(Heap_Thread, length * sizeof(cell), sizeof(cell), form, FORM_NONE,
      ret));
  pointer_set_datum_m(*ret, offset);
  if (defined_p(fill))
    while (length > 0) array_base(*ret)[--length] ← fill;
  return LERR_NONE;
}
```

81. Arrays can be resized in-place. Ultimately this relies on the system memory allocator resizing an allocation without changing (or by copying) the shared data and then initialising any remaining new cells, usually to `NIL`.

```
error_code array_resize_m(cell o, half nlength, cell fill)
{
  half olength;
  error_code reason;
  assert(arraylike_p(o));
  assert(nlength ≥ 0 ∧ nlength ≤ ARRAY_MAX);
  olength ← array_length_c(o);
  orreturn (segment_resize_m(o, nlength * sizeof(cell)));
  if (defined_p(fill))
    while (nlength > olength) array_base(o)[--nlength] ← fill;
  return LERR_NONE;
}
```

[1] I don't anticipate this feature finding much use but the space is there.

82. Hashtables. A significant user of arrays is the *hashtable* for associating a value with a key. A hashtable works by calculating the *hash* value of the key to locate an initial array index and then decreasing[1] the index until the correct key or an unused array slot is located.

The hash value calculated for each key is an unsigned 32 bit integer. A similar function is used depending on whether the length of the buffer is known or is a zero terminated C-string.

⟨ Type definitions 11 ⟩ +≡

```
typedef uint32_t hash;
```

83.

```
hash hash_buffer(byte *buf, half length)
{
  hash r ← 0;
  half i;
  assert(length ≥ 0);
  r ← 0;
  for (i ← 0; i < length; i++) r ← 33 * r + (unsigned char)(*buf++);
  return r;
}
hash hash_cstr(byte *buf, half *length)
{
  hash r ← 0;
  byte *p ← buf;
  while (*p ≠ '\0') r ← 33 * r + (unsigned char)(*p++);
  *length ← p − buf;
  return r;
}
```

[1] Or increasing or indeed any consistent algorithm.

84. The array underlying a hashtable is always created with a total of 2^n slots, plus a *footer*, so that hashtable entries are always based on an offset of zero, of two additional cells. When a new hashtable is created one of these cells is set to the number of slots in the array which can be used, set to 70% of the total slots available (rounded down). The other is set to zero representing the number of entries which have been removed from the hashtable or *blocked*.

A value of 70% ensures that there will still be "holes" even when the hashtable is nearly full, limiting how much of the array needs to be scanned to find the correct slot. The smallest hashtable above zero which can be created has 16 slots and in this case 15 slots are made available to leave a single one-slot hole, anticipating, with prejudice not benchmarks, that at such a small size a full array scan will not be expensive.

```
#define HASHTABLE_TINY 16
#define HASHTABLE_MAX ((HALF_MAX ≫ 1) + 1)
#define HASHTABLE_MAX_FREE (hashtable_default_free(HASHTABLE_MAX))
#define hashtable_default_free(L) (((L) ≡ HASHTABLE_TINY)
    ? (HASHTABLE_TINY − 1)      /* Guarantee at least one NIL. */
    : ((7 ∗ (1ULL ≪ high_bit(L)))/10))      /* ⌊70%⌋ */
#define hashtable_length_c(O) (array_length_c(O) − 2)
#define hashtable_base(O) (array_base(O))
#define hashtable_blocked_c(O) (fixed_value(array_base(O)[array_length_c(O) − 2]))
#define hashtable_blocked_p(O) (hashtable_blocked_c(O) ≥ 1)
#define hashtable_free_c(O) (fixed_value(array_base(O)[array_length_c(O) − 1]))
#define hashtable_free_p(O) (hashtable_free_c(O) > 0)
#define hashtable_unused_c(O) (hashtable_free_c(O) + hashtable_blocked_c(O))
#define hashtable_used_c(O) (hashtable_length_c(O) − hashtable_unused_c(O))
#define hashtable_set_blocked_m(O, V) (array_base(O)[array_length_c(O) − 2] ← fix(V))
#define hashtable_set_free_m(O, V) (array_base(O)[array_length_c(O) − 1] ← fix(V))
```

⟨ Function declarations 19 ⟩ +≡

```
hash hash_cstr(byte ∗, half ∗);
hash hash_buffer(byte ∗, half);
bool hashtable_match_paired(cell, void ∗);
bool hashtable_match_raw(cell, void ∗);
error_code copy_hashtable(cell, cell ∗);
error_code copy_hashtable_imp(cell, cell);
error_code hashtable_enlarge_m(cell);
error_code hashtable_erase_m(cell, cell, bool);
error_code hashtable_reduce_m(cell);
error_code hashtable_save_m(cell, cell, bool);
error_code hashtable_scan(cell, hash, void ∗, half ∗);
error_code hashtable_search(cell, cell, cell ∗);
error_code hashtable_search_raw(cell, hash, byte ∗, half, cell ∗);
error_code hashtable_to_list(cell, int, cell ∗);
error_code new_hashtable(half, cell ∗);
hash hashtable_key_paired(cell);
hash hashtable_key_raw(cell);
```

85. At base a new hashtable is an array set to `NIL` with the free and blocked slots set appropriately. This primary constructor increases the desired length (which is an optional argument to the constructor at run time) to the next power of two and calculates the initial free count.

```
#define new_hashtable_imp(L, F, R) do { /* Length, Free, R */
        assert(((L) ≡ 0 ∧ (F) ≡ 0) ∨ ((F) < (L)));
        orreturn (new_array_imp((L) + 2, fix(0), NIL, FORM_HASHTABLE, (R)));
        hashtable_set_blocked_m(*(R), 0);
        hashtable_set_free_m(*(R), (F));
      } while (0)
error_code new_hashtable(half slots, cell *ret)
{
  half nfree, rlength;
  error_code reason;
  assert(slots ≥ 0 ∧ slots ≤ (half) hashtable_default_free(HASHTABLE_MAX));
  if (slots ≡ 0) rlength ← nfree ← 0;
  else {
    if (slots ≤ (half) hashtable_default_free(HASHTABLE_TINY)) {
      rlength ← HASHTABLE_TINY;
      nfree ← hashtable_default_free(HASHTABLE_TINY);
    }
    else {
      rlength ← 1 ≪ high_bit(slots);
      nfree ← hashtable_default_free(rlength);
      while (nfree < slots) {
        rlength ≪= 1;
        nfree ← hashtable_default_free(rlength);
      }
      if (rlength > HASHTABLE_MAX) return LERR_LIMIT;
    }
    assert(nfree ≥ slots ∧ nfree < rlength);
  }
  new_hashtable_imp(rlength, nfree, ret);
  return LERR_NONE;
}
```

86. Ordinarily the object stored in a hashtable slot is a pair containing the key and the value. In one case the object being stored is the key itself and there is no pair. These two functions are used when comparing with a value in an occupied slot so obtain the key symbol's hash.

```
hash hashtable_key_paired(cell o)
{
  assert(pair_p(o) ∧ symbol_p(A(o)→sin));
  return symbol_hash_c(A(o)→sin);
}
hash hashtable_key_raw(cell o)
{
  assert(symbol_p(o));
  return symbol_hash_c(o);
}
```

87. Some of the algorithms used to implement hashtables copy the contents of one hashtable into another. The copying part is performed by *copy_hashtable_imp* after the new hashtable has been constructed, where it is guaranteed to have been created large enough to fit it all. The algorithm scans each array slot in the source hashtable and if there's a live entry in it inserts it into the new hashtable. The number of free slots in the new hashtable is updated afterwards.

The scanning algorithm at the heart of this function is a duplicate of the main scanning algorithm in *hashtable_scan*.

```
error_code copy_hashtable_imp(cell old, cell new)
{    /* nb. new is not a "cell *" and must have already been allocated. */
  half i, j, nfree;
  cell pos, value;
  hash hval;
  hash(*hashfn)(cell);
  assert(hashtable_p(old));
  assert(hashtable_p(new));
  nfree ← hashtable_free_c(new);
  assert(nfree ≥ hashtable_used_c(old));
  if (old ≡ Symbol_Table) hashfn ← hashtable_key_raw;
  else hashfn ← hashtable_key_paired;
  for (i ← 0; i < hashtable_length_c(old); i++) {
    value ← hashtable_base(old)[i];
    if (¬null_p(value) ∧ defined_p(value)) {    /* Slot in old. */
      hval ← hashfn(value);
      j ← hval % hashtable_length_c(new);
      while (1) {    /* Find a (guaranteed) slot in new. */
        pos ← hashtable_base(new)[j];
        if (null_p(pos)) break;
        if (j ≡ 0) j ← hashtable_length_c(new) − 1;
        else j−−;
      }
      hashtable_base(new)[j] ← value;
      nfree−−;
    }
  }
  hashtable_set_free_m(new, nfree);
  return LERR_NONE;
}
```

88. A function which can be exposed at run-time to simply copy a hashtable is an obvious, light wrapper around the above.

```
error_code copy_hashtable(cell o, cell *ret)
{
  cell new;
  error_code reason;
  orreturn (new_hashtable(hashtable_used_c(o), &new));
  orassert(copy_hashtable_imp(o, new));
  *ret ← new;
  return LERR_NONE;
}
```

89. To insert a new entry into a hashtable that's full the hashtable is first enlarged using the copying algorithm above. To maintain the fiction that the hashtable is enlarged in-place the atom (passed in via *o*) holding the original hashtable is mutated to reference the new one.

```
error_code hashtable_enlarge_m(cell o)
{
  atom tmp;
  cell new;
  half nfree, nlength;
  error_code reason;
  assert(hashtable_p(o));
  if (hashtable_length_c(o) ≡ 0) nlength ← HASHTABLE_TINY;
  else nlength ← hashtable_length_c(o) * 2;
  if (nlength > HASHTABLE_MAX) return LERR_LIMIT;
  nfree ← hashtable_default_free(nlength);
  new_hashtable_imp(nlength, nfree, &new);
  copy_hashtable_imp(o, new);
  tmp ← *A(new);      /* No need to swap the tag. */
  *A(new) ← *A(o);
  *A(o) ← tmp;
  segment_object(new)→owner ← new;
  segment_object(o)→owner ← o;
  return LERR_NONE;
}
```

90. Going the other way, when enough entries have been removed from a hashtable it can be reduced to the next size down. This is essentially the same as *hashtable_enlarge_m* (with a slightly more complex check to determine the size to reduce to) except that whereas the previous function will always attempt to enlarge the hashtable, *hashtable_reduce_m* will silently do nothing if there's no need to.

```
error_code hashtable_reduce_m(cell o)
{
  atom tmp;
  cell new;
  half nfree, nlength, olength;
  error_code reason;
  assert(hashtable_p(o));
  olength ← hashtable_length_c(o);
  assert(olength > 0);
  if (hashtable_used_c(o) ≡ 0) nfree ← nlength ← 0;
  else if (olength ≡ HASHTABLE_TINY) return LERR_NONE;
  else {
    assert(olength % 2 ≡ 0);
    nlength ← olength/2;
    nfree ← hashtable_default_free(nlength);
    if (hashtable_used_c(o) > nfree) return LERR_NONE;
  }
  new_hashtable_imp(nlength, nfree, &new);
  copy_hashtable_imp(o, new);
  tmp ← *A(new);      /* No need to swap the tag. */
  *A(new) ← *A(o);
  *A(o) ← tmp;
  segment_object(new)→owner ← new;
  segment_object(o)→owner ← o;
  return LERR_NONE;
}
```

91. Ordinary hashtables are used to associate a symbol with any `Lossless` object, except in one case. As will be explained in the chapter on symbols there is one hashtable called the symbol table which associates each symbol with itself.

This hashtable is searched when attempting to define a new symbol from a memory buffer to see if it has already been created so the key which is passed to the search function is not a symbol but a pointer to this pre-symbol structure.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    byte *buf;
    half length;
  } hashtable_raw;
```

92. And of course there are two comparison callback functions, one for each type of hashtable.

```
  bool hashtable_match_paired(cell each, void *ctx)
  {
    assert(pair_p(each) ∧ symbol_p(A(each)→sin));
    assert(symbol_p((cell) ctx));
    return A(each)→sin ≡ C(ctx);
  }
  bool hashtable_match_raw(cell each, void *ctx)
  {
    hashtable_raw *proto ← ctx;
    int i;
    assert(symbol_p(each));
    if (symbol_length_c(each) ≠ proto→length) return false;
    for (i ← 0; i < proto→length; i++)
      if (symbol_buffer_c(each)[i] ≠ proto→buf[i]) return false;
    return true;
  }
```

93. The central part of a hashtable is this algorithm to determine the array index at which the object is, or should be.

To find the correct index take key's hash modulo the length of the hashtable. If there's an object there and it matches that being sought then the search is over. Alternatively there may be a `NIL` indicating the key was not present in this hashtable and either this is its correct location or the hashtable is full.

If neither condition was true decrease the index by one and look again, wrapping around to the back of the array when passing zero. The correct object or `NIL` is guaranteed to come eventually.

When removing an entry from a hashtable it's replaced with `UNDEFINED` until the hashtable is reduced. It can't be replaced with `NIL` or future searches which were previously blocked by it will see the new hole.

Unusually *hashtable_scan* will always set the return value, either to the correct index or -1. The error code indicates whether or not the key was found at all by raising `LERR_MISSING` regardless of whether the hashtable is full.

```
error_code hashtable_scan(cell o, hash hval, void *ctx, half *ret)
{
  cell at;
  bool(*matchfn)(cell, void *);
  assert(hashtable_p(o));
  if (hashtable_length_c(o) ≡ 0) {
    *ret ← −1;
    return LERR_MISSING;
  }
  if (o ≡ Symbol_Table) matchfn ← hashtable_match_raw;
  else matchfn ← hashtable_match_paired;
  *ret ← hval % hashtable_length_c(o);      /* Default index value. */
  while (1) {      /* At least one NIL is guaranteed to be present. */
    at ← hashtable_base(o)[*ret];
    if (null_p(at) ∨ (defined_p(at) ∧ matchfn(at, ctx))) break;
    if (*ret ≡ 0) *ret ← hashtable_length_c(o) − 1;
    else (*ret)−−;
  }
  if (null_p(at)) {
    if (¬hashtable_free_p(o)) *ret ← −1;
    return LERR_MISSING;
  }
  else return LERR_NONE;
}
```

94. Searching either kind of hashtable is the same except for packing the raw buffer data into a **hashtable_raw** structure. An absent value is not an error here. This rather odd twisting of the API stems from a time when these functions returned the value half of the pair and could be improved (TODO).

```
error_code hashtable_search(cell o, cell label, cell *ret)
{
  half idx;
  error_code reason;
  assert(hashtable_p(o));
  assert(symbol_p(label));
  reason ← hashtable_scan(o, symbol_hash_c(label), (void *) label, &idx);
  if (reason ≡ LERR_MISSING) {
    *ret ← UNDEFINED;
    return LERR_NONE;
  }
  else if (failure_p(reason)) return reason;
  *ret ← hashtable_base(o)[idx];
  return LERR_NONE;
}
error_code hashtable_search_raw(cell o, hash hval, byte *buf, half length, cell *ret)
{
  hashtable_raw proto;
  half idx;
  error_code reason;
  proto.length ← length;
  proto.buf ← buf;
  reason ← hashtable_scan(o, hval, (void *) &proto, &idx);
  if (reason ≡ LERR_MISSING) {
    *ret ← UNDEFINED;
    return LERR_NONE;
  }
  else if (failure_p(reason)) return reason;
  *ret ← hashtable_base(o)[idx];
  return LERR_NONE;
}
```

95. Inserting anew and replacing an existing associating in a hashtable is the same except for whether its presence or absence is an error. When inserting if the hashtable is full and the key not found then the hashtable is enlarged.

```
error_code hashtable_save_m(cell o, cell datum, bool replace)
{
  hashtable_raw proto;
  half idx;
  hash hval;
  void *ctx;
  error_code reason;
  assert(hashtable_p(o));
  if (o ≡ Symbol_Table) {
    assert(symbol_p(datum));
    hval ← symbol_hash_c(datum);
    proto.length ← −1;      /* Match nothing. */
    ctx ← &proto;
  }
  else {
    assert(pair_p(datum) ∧ symbol_p(A(datum)→sin));
    hval ← symbol_hash_c(A(datum)→sin);
    ctx ← (void *) A(datum)→sin;
  }
again: reason ← hashtable_scan(o, hval, ctx, &idx);
  if (reason ≡ LERR_MISSING) {
    if (replace) return reason;
    else if (idx ≡ −1) {
      orreturn (hashtable_enlarge_m(o));
      goto again;
    }
  }
  else if (failure_p(reason)) return reason;
  else if (¬replace) return LERR_EXISTS;
  assert(¬failure_p(reason) ∨ hashtable_free_p(o));
  assert(idx ≥ 0 ∧ idx < hashtable_length_c(o));
  hashtable_base(o)[idx] ← datum;
  hashtable_set_free_m(o, hashtable_free_c(o) − 1);
  return LERR_NONE;
}
```

96. Removing an entry from a hashtable replaces it with `UNDEFINED` and increases the count of blocked slots. If it crosses a threshold *hashtable_reduce_m* to the next size down will reduce the size of the hashtable, removing it (and any others).

```
error_code hashtable_erase_m(cell o, cell label, bool relax)
{
  half idx;
  error_code reason;
  assert(hashtable_p(o));
  assert(symbol_p(label));
  reason ← hashtable_scan(o, symbol_hash_c(label), (void *) label, &idx);
  if (reason ≡ LERR_MISSING) return relax ? LERR_NONE : LERR_MISSING;
  assert(¬failure_p(reason));
  hashtable_base(o)[idx] ← UNDEFINED;
  hashtable_set_blocked_m(o, hashtable_blocked_c(o) + 1);
  return hashtable_reduce_m(o);
}
```

97. This page intentionally left blank.

```
#define HASHTABLE_ASIS 0
#define HASHTABLE_LABEL 1
#define HASHTABLE_VALUE 2
#if 0
  error_code hashtable_to_list(cell o, int preprocess, cell *ret)
  {
    assert(hashtable_p(o));
    acc ← NIL;
    if (preprocess ≡ HASHTABLE_LABEL) pp ← lsinx;
    else if (preprocess ≡ HASHTABLE_VALUE) pp ← ldexx;
    else pp ← lasisx;
    for (i ← 0; i < hashtable_length_c(o); i++) {
      next ← hashtable_base(o)[i];
      if (null_p(next) ∨ ¬defined_p(next)) continue;
      orreturn (cons(pp(next), acc, &acc));
    }
    *ret ← acc;
    return LERR_NONE;
  }
#endif
```

98. Symbols. Most of the time symbols are one or more printable letters or other characters, used in particular to bind objects to a readable name, but in fact they can be zero or more bytes of any value, even zero or any reserved or special character. This sequence of bytes is the symbol's *label*. A symbol also has a *hash* associated with it calculated over the bytes of this label.

If a symbol's label is short enough then it's interned in a heap atom rather than allocating a segment, and the hash value is not cached but recalculated whenever it's needed.

Every symbol is saved in the symbol table. When a new symbol is being created from a span of memory its hash value is calculated and then the symbol table is searched to see if that symbol has already been created, which is returned instead of creating a new one.

```
#define SYMBOL_MAX (HALF_MAX − sizeof(symbol))
#define symbol_object(O) ((symbol *) segment_base(O))
#define symbol_buffer_c(O) (symbol_intern_p(O) ? A(O)→buffer : symbol_object(O)→label)
#define symbol_hash_c(O)
        (symbol_intern_p(O) ? hash_buffer(A(O)→buffer, A(O)→length) : symbol_object(O)→hash_value)
#define symbol_length_c(O)
        (symbol_intern_p(O) ? A(O)→length : segment_length_c(O) − (half) sizeof(symbol))
```

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    hash hash_value;      /* Cache the hash value of long symbols. */
    byte label[];      /* This is bytes not characters. */
  } symbol;
```

99.
```
⟨ Global variables 13 ⟩ +≡
  shared cell Symbol_Table ← NIL;
```

100.
```
⟨ External C symbols 14 ⟩ +≡
  extern shared cell Symbol_Table;
```

101.
```
⟨ Function declarations 19 ⟩ +≡
  error_code new_symbol_buffer(byte *, half, bool *, cell *);
  error_code new_symbol_imp(hash, byte *, half, bool *, cell *);
  error_code new_symbol_segment(cell, half, half, cell *);
```

102.
```
⟨ Initialise symbol table 102 ⟩ ≡
  orabort(new_hashtable(0, &Symbol_Table));
```
This code is used in section 22.

103. Most new symbols are created using one of these three front ends, *new_symbol_cstr* for a buffer with a terminating zero byte which is *not* included in the symbol and *new_symbol_const* for symbols created from constant C-strings who's length is known to the C compiler.

```
#define new_symbol_cstr(O, R) new_symbol_buffer((byte *)(O), −1, Λ, (R))
#define new_symbol_const(O, R) new_symbol_buffer((byte *)(O), sizeof (O) − 1, Λ, (R))
  error_code new_symbol_buffer(byte *buf, half length, bool *fresh, cell *ret)
  {
    hash hval;
    assert(length ≥ −1 ∧ length < (half) SYMBOL_MAX);
    if (length ≡ −1) hval ← hash_cstr((byte *) buf, &length);
    else hval ← hash_buffer((byte *) buf, length);
    return new_symbol_imp(hval, buf, length, fresh, ret);
  }
```

104. TODO: Turn this into another macro.

```
error_code new_symbol_segment(cell o, half offset, half length, cell *ret)
{
  assert(segment_p(o));
  assert(offset ≥ 0);
  assert(length ≥ 0);
  assert(length + offset ≤ segment_length_c(o));
  return new_symbol_buffer(segment_base(o) + offset, length, Λ, ret);
}
```

105. To create a symbol from a buffer who's length and hash are known first search the symbol table to see if a symbol with that label already exists and return it, otherwise allocate a new (possibly interned) segment, save it in the symbol table and return that.

```
error_code new_symbol_imp(hash hval, byte *buf, half length, bool *fresh, cell *ret)
{
  cell sym;
  bool ignore;
  error_code reason;
  assert(length ≥ 0 ∧ length < (half) SYMBOL_MAX);
  if (fresh ≡ Λ) fresh ← &ignore;
  orreturn (hashtable_search_raw(Symbol_Table, hval, buf, length, &sym));
  if (defined_p(sym)) {
    *fresh ← false;
    *ret ← sym;
    return LERR_NONE;
  }
  *fresh ← true;
  if (length ≥ INTERN_MAX) {
    orreturn (new_segment_imp(Heap_Thread, length + sizeof(symbol), 0, FORM_SYMBOL,
        FORM_NONE, &sym));
    symbol_object(sym)¬hash_value ← hval;
  }
  else {
    orreturn (new_atom(NIL, NIL, FORM_SYMBOL_INTERN, &sym));
    A(sym)¬length ← length;
  }
  memmove(symbol_buffer_c(sym), buf, length);
  orreturn (hashtable_save_m(Symbol_Table, sym, false));
  *ret ← sym;
  return LERR_NONE;
}
```

106. Environment. All of the objects presented so far are united by the common theme of being implemented using low level constructs to "look behind the curtain" of `Lossless`' objects. Environment objects are the odd one out by being based on plain `Lossless` objects, they are included here because they are fundamental. In fact all of other objects mostly exists here to provide the support that environment objects need.

An environment is built out of a pair. One half of the pair points to a *previous* environment (circular connections are not allowed; an environment cannot point back to itself) and the other to a hashtable of symbol-to-value bindings.

Any environment who's previous environment is `NIL` is a *root environment*. One of these is defined during `Lossless`' initialisation, referred to as *the* root environment and saved in *Root*.

```
#define env_layer(O) (A(O)↣dex)
#define env_previous(O) (A(O)↣sin)
#define env_root_p(O) (environment_p(O) ∧ null_p(env_previous(O)))
⟨ Global variables 13 ⟩ +≡
  shared cell Root ← NIL;
  unique cell Environment ← NIL;
```

107.
```
⟨ External C symbols 14 ⟩ +≡
  extern shared cell Root;
  extern unique cell Environment;
```

108.
```
⟨ Function declarations 19 ⟩ +≡
  error_code new_env(cell, cell *);
  cell env_get_root(cell);
  error_code env_save_m(cell, cell, cell, bool);
  error_code env_search(cell, cell, cell *);
```

109.
```
⟨ Initialise run-time environment 109 ⟩ ≡
  orabort(new_empty_env(&Root));
  Environment ← Root;
```

See also section 151.

This code is used in section 22.

110. A new environment is created by extending an existing environment, a new root or empty environment is created by extending `NIL`.

```
#define new_empty_env(R) (new_env(NIL, (R)))
  error_code new_env(cell o, cell *ret)
  {
    cell tmp;
    error_code reason;
    assert(null_p(o) ∨ environment_p(o));
    orreturn (new_hashtable(0, &tmp));
    orreturn (new_atom(o, tmp, FORM_ENVIRONMENT, ret));
    return LERR_NONE;
  }
```

111. Locate an environment's root however deep in the hierarchy.

```
  cell env_get_root(cell o)
  {
    assert(environment_p(o));
    while (¬env_root_p(o)) o ← env_previous(o);
    return o;
  }
```

112. Inserting or replacing a binding in an environment.

```
#define env_save_m_imp(E, D, R)      /* Environment, Datum, Replace? */
        hashtable_save_m(env_layer(E), (D), (R))
  error_code env_save_m(cell o, cell label, cell value, bool replace)
  {
    cell tmp;
    error_code reason;
    assert(environment_p(o));
    assert(symbol_p(label));
    assert(defined_p(value));
    orreturn (cons(label, value, &tmp));
    return env_save_m_imp(o, tmp, replace);
  }
```

113. Searching an environment for a binding searches each level of the hierarchy in turn until the binding is found or the root environment is passed. The value that was bound, and not the pair saved into the hashtable, is returned.

```
  error_code env_search(cell o, cell label, cell *ret)
  {
    cell tmp;
    error_code reason;
    assert(environment_p(o));
    assert(symbol_p(label));
    for ( ; ¬null_p(o); o ← env_previous(o)) {
      orreturn (hashtable_search(env_layer(o), label, &tmp));
      if (defined_p(tmp)) {
        assert(pair_p(tmp));
        *ret ← A(tmp)¬dex;
        return LERR_NONE;
      }
    }
    return LERR_MISSING;
  }
```

114. SCOW. Very little thought has gone into this object so not much description will either. There is one built in scow for threads, to hold a **pthread_t**.

```
#define LSCOW_PTHREAD_T 0
```

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    int length, align;
  } scow;
```

115.
```
⟨ Global variables 13 ⟩ +≡
  shared scow *SCOW_Attributes ← Λ;
  shared int SCOW_Length ← 1;      /* To fit pthread_t */
```

116.
```
⟨ External C symbols 14 ⟩ +≡
  extern shared scow *SCOW_Attributes;
  extern shared int SCOW_Length;
```

117.
```
⟨ Function declarations 19 ⟩ +≡
  error_code register_scow(int, int, int *);
  error_code new_scow(int, intmax_t, cell *);
  bool scow_id_p(cell, int);
  half scow_length(cell);
```

118.
```
⟨ Initialise foreign linkage 118 ⟩ ≡
  orabort(alloc_mem(Λ, SCOW_Length * sizeof(scow), 0, (void **) &SCOW_Attributes));
```

This code is used in section 22.

119. I/O. None.

120. Runes (Characters). Not defined.

#define *ascii_p*(O) $((O) \geq 0x\texttt{00} \wedge (O) \leq 0x\texttt{7f})$
#define *ascii_space_p*(O) $((O) \equiv \texttt{'␣'} \vee (O) \equiv \texttt{'\t'} \vee (O) \equiv \texttt{'\n'})$
#define *ascii_digit_p*(O) $((O) \geq \texttt{'0'} \wedge (O) \leq \texttt{'9'})$
#define *ascii_hex_p*(O) $(((O)\ \&\ 0x\texttt{df}) \geq \texttt{'A'} \wedge ((O)\ \&\ 0x\texttt{df}) \leq \texttt{'F'})$
#define *ascii_printable_p*(O) $((O) \geq 0x\texttt{21} \wedge (O) \leq 0x\texttt{7e})$
#define *ascii_upcase_p*(O) $((O) \geq \texttt{'A'} \wedge (O) \leq \texttt{'Z'})$
#define *ascii_downcase_p*(O) $((O) \geq \texttt{'a'} \wedge (O) \leq \texttt{'z'})$
#define *ascii_upcase*(O) $(\mathit{ascii\_downcase\_p}(O)\ ?\ (O) - 0x\texttt{20} : (O))$
#define *ascii_downcase*(O) $(\mathit{ascii\_upcase\_p}(O)\ ?\ (O) + 0x\texttt{20} : (O))$

121. Virtual Machine.
Table ID = XX YYYY,YYYY YYZZ,ZZZZ
XX shifted left `OBJECTDB_SPLIT_GAP`.

```
#define ADDRESS_INVALID INTPTR_MAX
#define OBJECTDB_LENGTH (1 ≪ 16)
```

⟨ Type definitions 11 ⟩ +≡

```
typedef uintptr_t address;      /* void * would also be acceptable but for arithmetic. */
typedef int32_t instruction;
```

122. ⟨ Global variables 13 ⟩ +≡

```
shared cell Program_ObjectDB ← NIL;
shared half Program_ObjectDB_Free ← 0;
shared cell Program_Export_Table ← NIL;      /* Pairs of (name . index). */
shared address *Program_Export_Base ← Λ;     /* Array indexed in to. */
shared half Program_Export_Free ← 0;         /* Next available array slot. */
shared pthread_mutex_t Program_Lock;         /* Global lock for all of the above. */
unique address Ip ← ADDRESS_INVALID;         /* Current (or previous) instruction. */
shared address Empty_Trap_Handler[LERR_LENGTH] ← {0};
unique address *Trap_Handler ← Empty_Trap_Handler;
unique address Trap_Ip ← ADDRESS_INVALID;
unique error_code Trapped ← LERR_NONE;
```

```
#ifdef LLTEST
shared long Interpret_Count ← 0, Interpret_Limit ← 0;
#endif
```

123. ⟨ External C symbols 14 ⟩ +≡

```
extern shared cell Program_ObjectDB, Program_Export_Table;
extern shared half Program_ObjectDB_Free, Program_Export_Free;
extern shared address *Program_Export_Base;
extern shared pthread_mutex_t Program_Lock;
extern shared address Empty_Trap_Handler[];
extern unique address Ip, *Trap_Handler, Trap_Ip;
extern unique error_code Trapped;
extern shared long Interpret_Count, Interpret_Limit;
```

124.

```
#define CODE_PAGE_LENGTH 0x10000L
#define CODE_PAGE_MASK (CODE_PAGE_LENGTH − 1)
#define instruction_page(O) ((O) & ∼CODE_PAGE_MASK)
```

⟨ Function declarations 19 ⟩ +≡

```
error_code init_vm(void);
error_code vm_locate_entry(cell, word *, address *);
```

125. ⟨ Initialise program linkage 125 ⟩ ≡

```
for (i ← 0; i < LERR_LENGTH; i++) Empty_Trap_Handler[i] ← ADDRESS_INVALID;
Trap_Handler ← (address *) Empty_Trap_Handler;
orabort(init_osthread_mutex(&Program_Lock, false, false));
orabort(new_array(0, fix(0), &Program_ObjectDB));
orabort(alloc_mem(Λ, CODE_PAGE_LENGTH, CODE_PAGE_LENGTH, (void **) &Program_Export_Base));
assert((address) Program_Export_Base ≡ instruction_page((address) Program_Export_Base));
orabort(new_hashtable(0, &Program_Export_Table));
```

This code is used in section 22.

126. ⟨ `initialise.c` 8 ⟩ +≡

```
error_code init_vm(void)
{
  address atmp, adefault;
  cell copy[3], eval[3], list[3], optional[3];
  cell ltmp, jexit, sig_copy, sig_eval, sig_list, sig_optional;
  cell sig[SIGNATURE_LENGTH];
  char btmp[1024], *bptr;      /* Way more space than necessary. */
  int i, j, k;
  error_code reason;
  assert(¬VM_Ready);
  ⟨ Initialise error symbols 15 ⟩
  ⟨ Initialise assembler symbols 133 ⟩
  ⟨ Initialise Lossless primitives 148 ⟩
  ⟨ Initialise evaluator and other bytecode 280 ⟩
  ⟨ Link Lossless primitives to installed bytecode 149 ⟩
  ⟨ Initialise Lossless procedures 283 ⟩
  VM_Ready ← true;
  return LERR_NONE;
}
```

127.

```
error_code vm_locate_entry(cell label, word *index, address *ret)
{
  cell loffset;
  word coffset;
  error_code reason;
  assert(symbol_p(label));
  orreturn (hashtable_search(Program_Export_Table, label, &loffset));
  if (undefined_p(loffset)) return LERR_MISSING;
  orreturn (int_value(A(loffset)¬dex, &coffset));
  assert(coffset ≥ 0 ∧ coffset < Program_Export_Free);
  if (index ≠ Λ) *index ← coffset;
  *ret ← Program_Export_Base[coffset];
  return LERR_NONE;
}
```

128. Registers. There are still too many general registers. They have no run-time state associated with them except a name.

#define *register_id_c*(*O*) (*A*(*O*)¬*sin*)
#define *register_label_c*(*O*) (*A*(*O*)¬*dex*)

#define *LR_r0* 0
#define *LR_r1* 1
#define *LR_r2* 2
#define *LR_r3* 3
#define *LR_r4* 4
#define *LR_r5* 5
#define *LR_r6* 6
#define *LR_r7* 7
#define *LR_r8* 8
#define *LR_r9* 9
#define *LR_r10* 10
#define *LR_r11* 11
#define *LR_r12* 12
#define *LR_r13* 13
#define *LR_r14* 14
#define `LR_GENERAL` 15

129. Some global state isn't represented by a register, perhaps these should be: *Root*, *SCOW_Attributes*, *Threads*. More?

```
#define LR_Scrap 15
#define LR_Accumulator 16
#define LR_Argument_List 17
#define LR_Control_Link 18 /* Special: push/pop */
#define LR_Environment 19 /* Typed: environment? */
#define LR_Expression 20
#define LR_Root 21
#define LR_Arg1 22
#define LR_Arg2 23
#define LR_Result 24
#define LR_Trap_Arg1 25
#define LR_Trap_Arg2 26
#define LR_Trap_Result 27
#define LR_CELL 27
#define LR_Ip 28 /* RO: Pseudo int */
#define LR_Trap_Handler 29 /* RO: Pseudo array */
#define LR_Trap_Ip 30 /* RO: Pseudo int */
#define LR_Trapped 31 /* RO: Pseudo bool */
#define LR_SPECIAL 31
#define LR_LENGTH 32 /* 2^5 */
```

⟨ Global variables 13 ⟩ +≡

```
unique cell *Register[LR_CELL + 1] ← {Λ}; /* The registers. */
shared cell Register_Table[LR_LENGTH] ← {NIL}; /* Run-time register objects. */
unique cell Scrap ← NIL;
unique cell Accumulator ← NIL;
unique cell Argument_List ← NIL;
unique cell Control_Link ← NIL;
unique cell Expression ← NIL;
unique cell Trap_Arg1 ← NIL;
unique cell Trap_Arg2 ← NIL;
unique cell Trap_Result ← NIL;
unique cell VM_Arg1 ← NIL;
unique cell VM_Arg2 ← NIL;
unique cell VM_Result ← NIL;
unique cell General[LR_GENERAL + 1] ← {NIL};
```

130. ⟨ External C symbols 14 ⟩ +≡

```
extern shared cell Register_Table[];
extern unique cell General[], *Register[];
extern unique cell Accumulator, Argument_List, Control_Link, Expression, Scrap, Trap_Arg1,
    Trap_Arg2, Trap_Result, VM_Arg1, VM_Arg2, VM_Result;
```

131. ⟨ (Re-)Initialise thread register pointers 131 ⟩ ≡
```
Register[LR_Scrap] ← &Scrap;
Register[LR_Accumulator] ← &Accumulator;
Register[LR_Argument_List] ← &Argument_List;
Register[LR_Environment] ← &Environment;
Register[LR_Root] ← &Root;
Register[LR_Expression] ← &Expression;
Register[LR_Control_Link] ← &Control_Link;
Register[LR_Arg1] ← &VM_Arg1;
Register[LR_Arg2] ← &VM_Arg2;
Register[LR_Result] ← &VM_Result;
Register[LR_Trap_Arg1] ← &Trap_Arg1;
Register[LR_Trap_Arg2] ← &Trap_Arg2;
Register[LR_Trap_Result] ← &Trap_Result;
for (i ← 0; i ≤ LR_GENERAL; i++) Register[i] ← General + i;
```
This code is used in section 290.

132. ⟨ Data for initialisation 12 ⟩ +≡
```
shared char *Register_Label[LR_LENGTH] ← {[LR_Scrap] ← "VM:Temp",
    [LR_Accumulator] ← "VM:Accumulator",
    [LR_Argument_List] ← "VM:Argument-List",
    [LR_Control_Link] ← "VM:Control-Link",
    [LR_Environment] ← "VM:Environment",
    [LR_Root] ← "VM:Root",
    [LR_Expression] ← "VM:Expression",
    [LR_Trap_Arg1] ← "VM:Trap-Arg1",
    [LR_Trap_Arg2] ← "VM:Trap-Arg2",
    [LR_Trap_Result] ← "VM:Trap-Result",
    [LR_Arg1] ← "VM:Arg1",
    [LR_Arg2] ← "VM:Arg2",
    [LR_Result] ← "VM:Result",
    [LR_Trap_Ip] ← "VM:Trap-Ip",
    [LR_Trapped] ← "VM:Trapped",
    [LR_Trap_Handler] ← "VM:Trap-Handler",
    [LR_Ip] ← "VM:Ip",

[LR_r0] ← "VM:R0",
    [LR_r1] ← "VM:R1",
    [LR_r2] ← "VM:R2",
    [LR_r3] ← "VM:R3",
    [LR_r4] ← "VM:R4",
    [LR_r5] ← "VM:R5",
    [LR_r6] ← "VM:R6",
    [LR_r7] ← "VM:R7",
    [LR_r8] ← "VM:R8",
    [LR_r9] ← "VM:R9",
    [LR_r10] ← "VM:R10",
    [LR_r11] ← "VM:R11",
    [LR_r12] ← "VM:R12",
    [LR_r13] ← "VM:R13",
    [LR_r14] ← "VM:R14",
    };
```

133. ⟨ Initialise assembler symbols 133 ⟩ ≡

```
for (i ← 0; i < LR_LENGTH; i++) {
  orreturn (new_symbol_cstr(Register_Label[i], &ltmp));
  orreturn (new_atom(fix(i), ltmp, FORM_REGISTER, Register_Table + i));
  orreturn (env_save_m(Root, ltmp, Register_Table[i], false));
}
```

See also sections 134 and 140.

This code is used in section 126.

134. Six commonly-used registers are bound to an additional short names for convenience.

⟨ Initialise assembler symbols 133 ⟩ +≡

```
orabort(new_symbol_const("VM:Acc", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Accumulator], false));
orabort(new_symbol_const("VM:Args", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Argument_List], false));
orabort(new_symbol_const("VM:Clink", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Control_Link], false));
orabort(new_symbol_const("VM:Env", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Environment], false));
orabort(new_symbol_const("VM:Expr", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Expression], false));
orabort(new_symbol_const("VM:Tmp", &ltmp));
orabort(env_save_m(Root, ltmp, Register_Table[LR_Scrap], false));
```

135. Opcodes.

#define *opcode_id_c*(*O*) (*A*(*O*)→*sin*)
#define *opcode_label_c*(*O*) (*A*(*O*)→*dex*)
#define *opcode_object*(*O*) (&*Op*[*fixed_value*(*opcode_id_c*(*O*))])
#define *opcode_signature_c*(*O*) (&*opcode_object*(*O*)→*arg0*)

⟨ Type definitions 11 ⟩ +≡

```
typedef struct {
  cell owner;
  char arg0;
  char arg1;
  char arg2;
} opcode_table;
```

136. ⟨ Type definitions 11 ⟩ +≡

```
typedef enum {
  OP_HALT,      /* Instruction 0 for uninitialised memory. */
  OP_DUMP, OP_ADD, OP_ADDRESS, OP_ARRAY_P, OP_BODY, OP_CAR, OP_CDR, OP_CLOSURE, OP_CLOSURE_P,
      OP_CMP, OP_CMPEQ_P, OP_CMPGE_P, OP_CMPGT_P, OP_CMPIS_P, OP_CMPLE_P, OP_CMPLT_P,
      OP_CONS, OP_DEFINE_M, OP_DELIMIT, OP_ENVIRONMENT_P, OP_EXISTS_P, OP_EXTEND,
      OP_INTEGER_P, OP_JOIN, OP_JUMP, OP_JUMPIF, OP_JUMPNOT, OP_LENGTH, OP_LOAD, OP_LOOKUP,
      OP_MUL, OP_OPEN, OP_PAIR_P, OP_PEEK, OP_PEEK2, OP_PEEK4, OP_PEEK8, OP_PEND, OP_POKE2_M,
      OP_POKE4_M, OP_POKE8_M, OP_POKE_M, OP_PRIMITIVE_P, OP_REPLACE_M, OP_RESUMPTION_P,
      OP_SEGMENT_P, OP_SIGNATURE, OP_SPORK, OP_SUB, OP_SYMBOL, OP_SYMBOL_P, OP_SYNTAX,
      OP_SYNTAX_P, OP_TABLE, OP_TRAP, OPCODE_LENGTH
} opcode;
```

137.

```
#define NARG 0    /* No argument. */
#define AADD 1    /* An address. */
#define ALOB 2    /* An Lossless object. */
#define ALOT 3    /* A tiny Lossless object. */
#define AREG 4    /* A register. */
#define ARGH 5    /* A trap code (symbol, encoded as an 8-bit iny). */
```

⟨ Global variables 13 ⟩ +≡

```
shared opcode_table Op[OPCODE_LENGTH] ← {
    [OP_DUMP] ← {NIL, AREG, ALOB, NARG},
    [OP_ENVIRONMENT_P] ← {NIL, AREG, ALOB, NARG},
    [OP_RESUMPTION_P] ← {NIL, AREG, ALOB, NARG},
    [OP_PRIMITIVE_P] ← {NIL, AREG, ALOB, NARG},
    [OP_CLOSURE_P] ← {NIL, AREG, ALOB, NARG},
    [OP_INTEGER_P] ← {NIL, AREG, ALOB, NARG},
    [OP_REPLACE_M] ← {NIL, AREG, ALOB, NARG},
    [OP_SEGMENT_P] ← {NIL, AREG, ALOB, NARG},
    [OP_SIGNATURE] ← {NIL, AREG, ALOB, NARG},
    [OP_DEFINE_M] ← {NIL, AREG, ALOB, NARG},
    [OP_EXISTS_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_SYMBOL_P] ← {NIL, AREG, ALOB, NARG},
    [OP_SYNTAX_P] ← {NIL, AREG, ALOB, NARG},
    [OP_ADDRESS] ← {NIL, AREG, ALOB, NARG},
    [OP_ARRAY_P] ← {NIL, AREG, ALOB, NARG},
    [OP_CLOSURE] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPEQ_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPGE_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPGT_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPIS_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPLE_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_CMPLT_P] ← {NIL, AREG, ALOT, ALOT},
    [OP_DELIMIT] ← {NIL, AREG, NARG, NARG},
    [OP_JUMPNOT] ← {NIL, AREG, AADD, NARG},
    [OP_POKE2_M] ← {NIL, AREG, ALOT, ALOT},
    [OP_POKE4_M] ← {NIL, AREG, ALOT, ALOT},
    [OP_POKE8_M] ← {NIL, AREG, ALOT, ALOT},
    [OP_EXTEND] ← {NIL, AREG, ALOB, NARG},
    [OP_JUMPIF] ← {NIL, AREG, AADD, NARG},
    [OP_LENGTH] ← {NIL, AREG, ALOB, NARG},
    [OP_LOOKUP] ← {NIL, AREG, ALOT, ALOT},
    [OP_PAIR_P] ← {NIL, AREG, ALOB, NARG},
    [OP_POKE_M] ← {NIL, AREG, ALOT, ALOT},
    [OP_SYMBOL] ← {NIL, AREG, ALOT, ALOT},
    [OP_SYNTAX] ← {NIL, AREG, ALOT, ALOT},
    [OP_PEEK2] ← {NIL, AREG, ALOT, ALOT},
    [OP_PEEK4] ← {NIL, AREG, ALOT, ALOT},
    [OP_PEEK8] ← {NIL, AREG, ALOT, ALOT},
    [OP_SPORK] ← {NIL, AREG, AADD, NARG},
    [OP_TABLE] ← {NIL, AREG, ALOB, NARG},
    [OP_BODY] ← {NIL, AREG, ALOB, NARG},
    [OP_CONS] ← {NIL, AREG, ALOT, ALOT},
    [OP_HALT] ← {NIL, NARG, NARG, NARG},
    [OP_JOIN] ← {NIL, AREG, ALOB, NARG},
    [OP_JUMP] ← {NIL, AADD, NARG, NARG},
    [OP_LOAD] ← {NIL, AREG, ALOB, NARG},
    [OP_OPEN] ← {NIL, AREG, ALOB, NARG},
```

```
    [OP_PEEK] ← {NIL, AREG, ALOT, ALOT},
    [OP_PEND] ← {NIL, AREG, AADD, NARG},
    [OP_TRAP] ← {NIL, ARGH, NARG, NARG},
    [OP_ADD] ← {NIL, AREG, ALOT, ALOT},
    [OP_CAR] ← {NIL, AREG, ALOB, NARG},
    [OP_CDR] ← {NIL, AREG, ALOB, NARG},
    [OP_CMP] ← {NIL, AREG, ALOT, ALOT},
    [OP_MUL] ← {NIL, AREG, ALOT, ALOT},
    [OP_SUB] ← {NIL, AREG, ALOT, ALOT},
};
```

138. ⟨ External C symbols 14 ⟩ +≡

```
extern shared opcode_table Op[];
```

139. ⟨ Data for initialisation 12 ⟩ +≡

```
shared char *Opcode_Label[OPCODE_LENGTH] ← {
    [OP_DUMP] ← "VM:DUMP",
    [OP_ENVIRONMENT_P] ← "VM:ENVIRONMENT?",
    [OP_RESUMPTION_P] ← "VM:RESUMPTION?",
    [OP_PRIMITIVE_P] ← "VM:PRIMITIVE?",
    [OP_CLOSURE_P] ← "VM:CLOSURE?",
    [OP_INTEGER_P] ← "VM:INTEGER?",
    [OP_REPLACE_M] ← "VM:REPLACE!",
    [OP_SEGMENT_P] ← "VM:SEGMENT?",
    [OP_SIGNATURE] ← "VM:SIGNATURE",
    [OP_DEFINE_M] ← "VM:DEFINE!",
    [OP_EXISTS_P] ← "VM:EXISTS?",
    [OP_SYMBOL_P] ← "VM:SYMBOL?",
    [OP_SYNTAX_P] ← "VM:SYNTAX?",
    [OP_ADDRESS] ← "VM:ADDRESS",
    [OP_ARRAY_P] ← "VM:ARRAY?",
    [OP_CLOSURE] ← "VM:CLOSURE",
    [OP_CMPEQ_P] ← "VM:CMPEQ?",
    [OP_CMPGE_P] ← "VM:CMPGE?",
    [OP_CMPGT_P] ← "VM:CMPGT?",
    [OP_CMPIS_P] ← "VM:CMPIS?",
    [OP_CMPLE_P] ← "VM:CMPLE?",
    [OP_CMPLT_P] ← "VM:CMPLT?",
    [OP_DELIMIT] ← "VM:DELIMIT",
    [OP_JUMPNOT] ← "VM:JUMPNOT",
    [OP_POKE2_M] ← "VM:POKE2!",
    [OP_POKE4_M] ← "VM:POKE4!",
    [OP_POKE8_M] ← "VM:POKE8!",
    [OP_EXTEND] ← "VM:EXTEND",
    [OP_JUMPIF] ← "VM:JUMPIF",
    [OP_LENGTH] ← "VM:LENGTH",
    [OP_LOOKUP] ← "VM:LOOKUP",
    [OP_PAIR_P] ← "VM:PAIR?",
    [OP_POKE_M] ← "VM:POKE!",
    [OP_SYMBOL] ← "VM:SYMBOL",
    [OP_SYNTAX] ← "VM:SYNTAX",
    [OP_PEEK2] ← "VM:PEEK2",
    [OP_PEEK4] ← "VM:PEEK4",
    [OP_PEEK8] ← "VM:PEEK8",
    [OP_SPORK] ← "VM:SPORK",
    [OP_TABLE] ← "VM:TABLE",
    [OP_BODY] ← "VM:BODY",
    [OP_CONS] ← "VM:CONS",
    [OP_HALT] ← "VM:HALT",
    [OP_JOIN] ← "VM:JOIN",
    [OP_JUMP] ← "VM:JUMP",
    [OP_LOAD] ← "VM:LOAD",
    [OP_OPEN] ← "VM:OPEN",
    [OP_PEEK] ← "VM:PEEK",
    [OP_PEND] ← "VM:PEND",
    [OP_TRAP] ← "VM:TRAP",
    [OP_ADD] ← "VM:ADD",
    [OP_CAR] ← "VM:CAR",
    [OP_CDR] ← "VM:CDR",
    [OP_CMP] ← "VM:CMP",
```

```
    [OP_MUL] ← "VM:MUL",
    [OP_SUB] ← "VM:SUB",
};
```

140. ⟨ Initialise assembler symbols 133 ⟩ +≡

```
for (i ← 0; i < OPCODE_LENGTH; i++) {
  orabort(new_symbol_cstr(Opcode_Label[i], &ltmp));
  orabort(new_atom(fix(i), ltmp, FORM_OPCODE, &Op[i].owner));
  orabort(env_save_m(Root, ltmp, Op[i].owner, false));
}
```

141. Run-time primitives.

```
⟨ Type definitions 11 ⟩ +≡
  typedef error_code (*primitive_fn)(cell, cell *);
  typedef struct {
    cell signature;
    cell owner;
    address wrapper;
    primitive_fn action;
  } primitive;
```

142. **#define** *primitive_object*(*O*) (&*Primitive*[*fixed_value*(*A*(*O*)→*sin*)])
#define *primitive_address_c*(*O*) (*primitive_object*(*O*)→*wrapper*)
#define *primitive_signature_c*(*O*) (*primitive_object*(*O*)→*signature*)

```
⟨ Type definitions 11 ⟩ +≡
  typedef enum {
    PRIMITIVE_ADD, PRIMITIVE_ARRAY_LENGTH, PRIMITIVE_ARRAY_OFFSET, PRIMITIVE_ARRAY_P,
        PRIMITIVE_ARRAY_REF, PRIMITIVE_ARRAY_RESIZE_M, PRIMITIVE_ARRAY_SET_M,
        PRIMITIVE_BOOLEAN_P, PRIMITIVE_CAR, PRIMITIVE_CDR, PRIMITIVE_CONS,
        PRIMITIVE_CURRENT_ENVIRONMENT, PRIMITIVE_DEFINE_M, PRIMITIVE_DO, PRIMITIVE_EVAL,
        PRIMITIVE_FALSE_P, PRIMITIVE_INTEGER_P, PRIMITIVE_IF, PRIMITIVE_IS_P,
        PRIMITIVE_LAMBDA, PRIMITIVE_MUL, PRIMITIVE_NEW_ARRAY, PRIMITIVE_NEW_SEGMENT,
        PRIMITIVE_NEW_SYMBOL_SEGMENT, PRIMITIVE_NULL_P, PRIMITIVE_PAIR_P, PRIMITIVE_QUOTE,
        PRIMITIVE_ROOT_ENVIRONMENT, PRIMITIVE_SEGMENT_LENGTH, PRIMITIVE_SEGMENT_P,
        PRIMITIVE_SEGMENT_RESIZE_M, PRIMITIVE_SET_M, PRIMITIVE_SUB, PRIMITIVE_SYMBOL_KEY,
        PRIMITIVE_SYMBOL_P, PRIMITIVE_SYMBOL_SEGMENT, PRIMITIVE_TRUE_P, PRIMITIVE_VOID_P,
        PRIMITIVE_VOV, PRIMITIVE_LENGTH
  } primitive_code;
  enum {
    SIGNATURE_0, SIGNATURE_1, SIGNATURE_2, SIGNATURE_3, SIGNATURE_C, SIGNATURE_CL,
        SIGNATURE_EL, SIGNATURE_EO, SIGNATURE_ECL, SIGNATURE_ECO, SIGNATURE_L,
        SIGNATURE_LENGTH
  };
```

143. **#define** PO(P, S, F) $[(P)] \leftarrow \{(S), \texttt{NIL}, \texttt{ADDRESS\_INVALID}, (F)\}$

⟨ Global variables 13 ⟩ +≡

```
shared primitive Primitive[PRIMITIVE_LENGTH] ← {PO(PRIMITIVE_ADD,SIGNATURE_2,Λ),
PO(PRIMITIVE_ARRAY_LENGTH,SIGNATURE_1,Λ),
PO(PRIMITIVE_ARRAY_OFFSET,SIGNATURE_1,Λ),
PO(PRIMITIVE_ARRAY_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_ARRAY_REF,SIGNATURE_2,Λ),
PO(PRIMITIVE_ARRAY_RESIZE_M,SIGNATURE_2,Λ),
PO(PRIMITIVE_ARRAY_SET_M,SIGNATURE_3,Λ),
PO(PRIMITIVE_BOOLEAN_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_CAR,SIGNATURE_1,Λ),
PO(PRIMITIVE_CDR,SIGNATURE_1,Λ),
PO(PRIMITIVE_CONS,SIGNATURE_2,Λ),
PO(PRIMITIVE_CURRENT_ENVIRONMENT,SIGNATURE_0,Λ),
PO(PRIMITIVE_DEFINE_M,SIGNATURE_ECL,Λ),
PO(PRIMITIVE_DO,SIGNATURE_L,Λ),
PO(PRIMITIVE_EVAL,SIGNATURE_EO,Λ),
PO(PRIMITIVE_FALSE_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_INTEGER_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_IF,SIGNATURE_ECO,Λ),
PO(PRIMITIVE_IS_P,SIGNATURE_2,Λ),
PO(PRIMITIVE_LAMBDA,SIGNATURE_CL,Λ),
PO(PRIMITIVE_MUL,SIGNATURE_2,Λ),
PO(PRIMITIVE_NEW_ARRAY,SIGNATURE_3,Λ),
PO(PRIMITIVE_NEW_SEGMENT,SIGNATURE_2,Λ),
PO(PRIMITIVE_NEW_SYMBOL_SEGMENT,SIGNATURE_3,Λ),
PO(PRIMITIVE_NULL_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_PAIR_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_QUOTE,SIGNATURE_C,Λ),
PO(PRIMITIVE_ROOT_ENVIRONMENT,SIGNATURE_0,Λ),
PO(PRIMITIVE_SEGMENT_LENGTH,SIGNATURE_1,Λ),
PO(PRIMITIVE_SEGMENT_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_SEGMENT_RESIZE_M,SIGNATURE_2,Λ),
PO(PRIMITIVE_SET_M,SIGNATURE_ECL,Λ),
PO(PRIMITIVE_SUB,SIGNATURE_2,Λ),
PO(PRIMITIVE_SYMBOL_KEY,SIGNATURE_1,Λ),
PO(PRIMITIVE_SYMBOL_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_SYMBOL_SEGMENT,SIGNATURE_1,Λ),
PO(PRIMITIVE_TRUE_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_VOID_P,SIGNATURE_1,Λ),
PO(PRIMITIVE_VOV,SIGNATURE_CL,Λ),
};
```

144. ⟨ External C symbols 14 ⟩ +≡

```
extern shared primitive Primitive[];
```

145. ⟨ Data for initialisation 12 ⟩ +≡

```
shared char *Primitive_Label[PRIMITIVE_LENGTH] ← {[PRIMITIVE_ADD] ← "+",      /* X */
[PRIMITIVE_ARRAY_LENGTH] ← "array/length",      /* X */
[PRIMITIVE_ARRAY_OFFSET] ← "array/offset",      /* X */
[PRIMITIVE_ARRAY_P] ← "array?", [PRIMITIVE_ARRAY_REF] ← "array/ref",      /* X */
[PRIMITIVE_ARRAY_RESIZE_M] ← "array/resize!",      /* X */
[PRIMITIVE_ARRAY_SET_M] ← "array/set!",      /* X */
[PRIMITIVE_BOOLEAN_P] ← "boolean?", [PRIMITIVE_CAR] ← "car", [PRIMITIVE_CDR] ← "cdr",
    [PRIMITIVE_CONS] ← "cons", [PRIMITIVE_CURRENT_ENVIRONMENT] ← "current-environment",
    [PRIMITIVE_DEFINE_M] ← "define!",      /* X */
[PRIMITIVE_DO] ← "do",      /* X */
[PRIMITIVE_EVAL] ← "eval", [PRIMITIVE_FALSE_P] ← "false?",
    [PRIMITIVE_INTEGER_P] ← "integer?",      /* X */
[PRIMITIVE_IF] ← "if", [PRIMITIVE_IS_P] ← "is?", [PRIMITIVE_LAMBDA] ← "lambda",
    [PRIMITIVE_MUL] ← "*",      /* X */
[PRIMITIVE_NEW_ARRAY] ← "new-array",      /* X */
[PRIMITIVE_NEW_SEGMENT] ← "new-segment",      /* X */
[PRIMITIVE_NEW_SYMBOL_SEGMENT] ← "segment->symbol",      /* X */
[PRIMITIVE_NULL_P] ← "null?", [PRIMITIVE_PAIR_P] ← "pair?", [PRIMITIVE_QUOTE] ← "quote",
    [PRIMITIVE_ROOT_ENVIRONMENT] ← "root-environment",
    [PRIMITIVE_SEGMENT_LENGTH] ← "segment/length",      /* X */
[PRIMITIVE_SEGMENT_P] ← "segment?", [PRIMITIVE_SEGMENT_RESIZE_M] ← "segment/resize!",
  /* X */
[PRIMITIVE_SET_M] ← "set!", [PRIMITIVE_SUB] ← "-",      /* X */
[PRIMITIVE_SYMBOL_KEY] ← "symbol/key",      /* X */
[PRIMITIVE_SYMBOL_P] ← "symbol?", [PRIMITIVE_SYMBOL_SEGMENT] ← "symbol/segment",
  /* X */
[PRIMITIVE_TRUE_P] ← "true?", [PRIMITIVE_VOID_P] ← "void?", [PRIMITIVE_VOV] ← "vov", };
```

146. ⟨ Global variables 13 ⟩ +≡

```
shared address Interpret_Closure ← ADDRESS_INVALID;
```

147. ⟨ External C symbols 14 ⟩ +≡

```
extern shared address Interpret_Closure;
```

148. ⟨ Initialise `Lossless` primitives 148 ⟩ ≡

```
orreturn (new_symbol_const("eval",&sig_eval));
orreturn (cons(sig_eval,NIL,&sig_eval));
orreturn (new_symbol_const("copy",&sig_copy));
orreturn (cons(sig_copy,NIL,&sig_copy));
orreturn (new_symbol_const("copy-list",&sig_list));
orreturn (cons(sig_list,NIL,&sig_list));
orreturn (new_symbol_const("optional",&sig_optional));
orreturn (cons(sig_optional,NIL,&sig_optional));
orreturn (cons(fix(0),sig_copy,copy+0));
orreturn (cons(fix(1),sig_copy,copy+1));
orreturn (cons(fix(2),sig_copy,copy+2));
orreturn (cons(fix(0),sig_list,list+0));
orreturn (cons(fix(1),sig_list,list+1));
orreturn (cons(fix(2),sig_list,list+2));
orreturn (cons(fix(0),sig_eval,eval+0));
orreturn (cons(fix(1),sig_eval,eval+1));
orreturn (cons(fix(2),sig_eval,eval+2));
orreturn (cons(fix(3),sig_eval,eval+3));
orreturn (cons(fix(0),sig_optional,optional+0));
orreturn (cons(fix(1),sig_optional,optional+1));
orreturn (cons(fix(2),sig_optional,optional+2));
sig[SIGNATURE_0] ← NIL;
orreturn (cons(eval[0],NIL,sig+SIGNATURE_1));
orreturn (cons(eval[1],NIL,sig+SIGNATURE_2));
orreturn (cons(eval[0],sig[SIGNATURE_2],sig+SIGNATURE_2));
orreturn (cons(eval[2],NIL,sig+SIGNATURE_3));
orreturn (cons(eval[1],sig[SIGNATURE_3],sig+SIGNATURE_3));
orreturn (cons(eval[0],sig[SIGNATURE_3],sig+SIGNATURE_3));
orreturn (cons(copy[0],NIL,sig+SIGNATURE_C));
orreturn (cons(list[0],NIL,sig+SIGNATURE_L));
orreturn (cons(list[1],NIL,sig+SIGNATURE_CL));
orreturn (cons(copy[0],sig[SIGNATURE_CL],sig+SIGNATURE_CL));
orreturn (cons(list[1],NIL,sig+SIGNATURE_EL));
orreturn (cons(eval[0],sig[SIGNATURE_EL],sig+SIGNATURE_EL));
orreturn (cons(list[2],NIL,sig+SIGNATURE_ECL));
orreturn (cons(copy[1],sig[SIGNATURE_ECL],sig+SIGNATURE_ECL));
orreturn (cons(eval[0],sig[SIGNATURE_ECL],sig+SIGNATURE_ECL));
orreturn (cons(optional[1],NIL,sig+SIGNATURE_EO));
orreturn (cons(eval[0],sig[SIGNATURE_EO],sig+SIGNATURE_EO));
orreturn (cons(optional[2],NIL,sig+SIGNATURE_ECO));
orreturn (cons(copy[1],sig[SIGNATURE_ECO],sig+SIGNATURE_ECO));
orreturn (cons(eval[0],sig[SIGNATURE_ECO],sig+SIGNATURE_ECO));
for (i ← 0; i < PRIMITIVE_LENGTH; i++) {
  Primitive[i].signature ← sig[Primitive[i].signature];
  orreturn (new_symbol_cstr(Primitive_Label[i],&ltmp));
  orreturn (new_atom(fix(i),ltmp,FORM_PRIMITIVE,&Primitive[i].owner));
  orreturn (env_save_m(Root,ltmp,Primitive[i].owner,false));
}
```

This code is used in section 126.

149. `#define PRIMITIVE_PREFIX "!Primitive/"`
`#define PRIMITIVE_DEFAULT "!Primitive.Default"`
`#define PRIMITIVE_INTERPRET "!Interpret-Closure"`

⟨ Link `Lossless` primitives to installed bytecode 149 ⟩ ≡

```
k ← 11 + 7;  /* strlen(PRIMITIVE_WRAPPER) */
memmove(btmp, PRIMITIVE_DEFAULT, k);
orreturn (new_symbol_buffer((byte *) btmp, k, Λ, &ltmp));
orreturn (vm_locate_entry(ltmp, Λ, &adefault));
k ← 11;  /* strlen(PRIMITIVE_PREFIX) */
btmp[k − 1] ← '/';  /* memmove(btmp, PRIMITIVE_PREFIX, k) */
for (i ← 0; i < PRIMITIVE_LENGTH; i++) {
  bptr ← btmp + k;
  for (j ← 0; Primitive_Label[i][j]; j++, bptr++) *bptr ← Primitive_Label[i][j];
  orreturn (new_symbol_buffer((byte *) btmp, k + j, Λ, &ltmp));
  reason ← vm_locate_entry(ltmp, Λ, &atmp);
  if (reason ≡ LERR_MISSING) Primitive[i].wrapper ← adefault;
  else if (failure_p(reason)) return reason;
  else Primitive[i].wrapper ← atmp;
}
orreturn (new_symbol_const(PRIMITIVE_INTERPRET, &ltmp));
orreturn (vm_locate_entry(ltmp, Λ, &Interpret_Closure));
```

This code is used in section 126.

150. Stack. Based on list or array.

⟨ Function declarations 19 ⟩ +≡
```
error_code init_stack_array(cell *);
error_code init_stack_list(cell *);
error_code stack_array_enlarge(cell *);
error_code stack_array_peek(cell *, cell *);
error_code stack_array_pop(cell *, cell *);
error_code stack_array_push(cell *, cell);
error_code stack_array_reduce(cell *);
error_code stack_list_pop_imp(cell *, bool, cell *);
error_code stack_list_push(cell *, cell);
```

151. ⟨ Initialise run-time environment 109 ⟩ +≡
```
orreturn (init_stack_array(&Control_Link));
```

152.
```
error_code init_stack_list(cell *ret)
{
  *ret ← NIL;
  return LERR_NONE;
}
```

153.
```
error_code stack_list_push(cell *stack, cell value)
{
  error_code reason;
  orreturn (cons(value, *stack, stack));
  return LERR_NONE;
}
```

154.
```
#define stack_list_pop(S, R)  stack_list_pop_imp((S), true, (R))
#define stack_list_peek(S, R)  stack_list_pop_imp((S), false, (R))
error_code stack_list_pop_imp(cell *stack, bool popping, cell *ret)
{
  if (null_p(*stack)) return LERR_UNDERFLOW;
  else if (¬pair_p(*stack)) return LERR_INCOMPATIBLE;
  *ret ← A(*stack)¬sin;
  if (popping) *stack ← A(*stack)¬dex;
  return LERR_NONE;
}
```

155.
```
error_code init_stack_array(cell *ret)
{
  error_code reason;
  orreturn (new_array(1 + (64 * CELL_BYTES), fix(0), ret));
  array_base(*ret)[0] ← fix(1);
  return LERR_NONE;
}
```

156.
```
#define stack_array_p(O)
    (array_p(O) ∧ array_length_c(O) > 0 ∧ fixed_p(array_base(O)[0]) ∧ fixed_value(array_base(O)[0]) >
        0 ∧ fixed_value(array_base(O)[0]) ≤ array_length_c(O))
```

157. **error_code** *stack_array_push*(**cell** ∗*stack*, **cell** *value*)

```
{
  half sp;
  assert(stack_array_p(∗stack));
  sp ← fixed_value(array_base(∗stack)[0]);
  if (sp ≡ array_length_c(∗stack)) return LERR_OVERFLOW;
  array_base(∗stack)[sp++] ← value;
  array_base(∗stack)[0] ← fix(sp);
  return LERR_NONE;
}
```

158. **error_code** *stack_array_pop*(**cell** ∗*stack*, **cell** ∗*ret*)

```
{
  half sp;
  assert(stack_array_p(∗stack));
  sp ← fixed_value(array_base(∗stack)[0]);
  if (sp ≡ 1) return LERR_UNDERFLOW;
  ∗ret ← array_base(∗stack)[−−sp];
  array_base(∗stack)[0] ← fix(sp);
  return LERR_NONE;
}
```

159. **error_code** *stack_array_peek*(**cell** ∗*stack*, **cell** ∗*ret*)

```
{
  half sp;
  assert(stack_array_p(∗stack));
  sp ← fixed_value(array_base(∗stack)[0]);
  if (sp ≡ 1) return LERR_UNDERFLOW;
  ∗ret ← array_base(∗stack)[sp − 1];
  return LERR_NONE;
}
```

160. **error_code** *stack_array_enlarge*(**cell** ∗*stack*)

```
{
  half nlength;
  assert(stack_array_p(∗stack));
  nlength ← array_length_c(∗stack) + 64;
  return array_resize_m(∗stack, nlength, NIL);
}
```

161. **error_code** *stack_array_reduce*(**cell** ∗*stack*)

```
{
  half nlength;
  assert(stack_array_p(∗stack));
  assert(array_length_c(∗stack) > 65);
  nlength ← array_length_c(∗stack) + 64;
  return array_resize_m(∗stack, nlength, NIL);
}
```

162. Interpreter.

Argument flavours:

ARGH, NARG, NARG TRAP Encode as AREG, ALOG ALOT, ALOT, ALOT DEFINE! "" NARG, NARG, NARG HALT "" AREG, NARG, NARG DELIMIT ""

AADD, NARG, NARG JUMP + WIDEs AREG, AADD, NARG

AREG, ALOB, NARG AREG, ALOT, ALOT

Majority of instructions are one-arg or two-arg (plus destination):

One arg: 16 or 19 bits Objects can be integers, special, from registers (popping) or via a table 2 instruction variants, integer or indirect If indirect reg(p) in an arg or 15 bits to identify a table offset

Two arg: 4 instruction variants, one for each arg being integer or not can encode whether arg is int in the spare bits of the dest reg means 1 variant in opcode space or in other words, instruction variance is in 2nd byte if integer, value is signed 8 bit otherwise highest bit is register or special if register, next bit is popping then 5 or 6 bit reg if special most bits are wasted, no fixnums Dest byte only destination; 3 extra bits

Majority remainder are addressing: Conditional, forking or goto Use 'destination' reg for condition so no possibility for popping 2 variants, relative or indirect indirect is reg or table, possibly 3rd variant if not spare bit

8 bits for instruction 3 bits for variance 5 bits for destination 16 bits vary

Some ops have no destination, eg. HALT, TRAP, JUMP.

nb. use of char * and array deref is the best kind of incorrect.

```
#define IB(I,B) ((int32_t)(((unsigned char *) &(I))[B]))
#define IINT(I) ((int32_t)((IB((I),2) << 8) | IB((I),3)))
#define OP(I) (IB((I),0))

#define FLAGS(I) (betoh32((I) & htobe32(LBC_FLAGS)))
#define INTP(I,B) (IB((I),1) & (1 << (9 - (B))))
#define SINT(I) ((int16_t)(betoh32(I) & 0xffff))
#define UINT(I) ((uint16_t)(betoh32(I) & 0xffff))
#define SBIG(I) ((int32_t)(UBIG(I) | SIGN(I)))
#define SIGN(I) ((0xffc00000 * ((betoh32(I) & 0x00200000) >> 21)))
#define UBIG(I) ((uint32_t)(betoh32(I) & 0x003fffff))
#define REGP(I,B) ((B) == 0 || !(IB((I),(B)+1) & 0x80))
#define REG(I,B) (IB((I),(B)+1) & 0x1f)
#define POP(I,B) (IB((I),(B)+1) & 0x20)
#define VAL(I,B) ((int)(IB((I),(B)+1) & 0x7f))
#define LBC_OPCODE 0xff000000
#define LBC_FLAGS 0x00c00000
#define LBC_TARGET 0x003f0000
#define LBC_TARGET_POPPING 0x00200000

#define LBC_ADDRESS_REGISTER 0x00000000 /* Zero. */
#define LBC_ADDRESS_RELATIVE 0x00800000
#define LBC_ADDRESS_INDIRECT 0x00400000
#define LBC_ADDRESS_ABSOLUTE 0x00c00000 /* Unused */

#define LBC_OBJECT_REGISTER 0x00000000 /* Zero. */
#define LBC_OBJECT_INTEGER 0x00800000
#define LBC_OBJECT_TABLE 0x00400000
#define LBC_OBJECT_RESERVED 0x00c00000 /* Unused */

#define LBC_FIRST_INTEGER 0x00800000
#define LBC_FIRST_SPECIAL 0x00008000
#define LBC_FIRST_POPPING 0x00002000

#define LBC_SECOND_INTEGER 0x00400000
#define LBC_SECOND_SPECIAL 0x00000080
#define LBC_SECOND_POPPING 0x00000020
```

⟨ Function declarations 19 ⟩ +≡

```
error_code interpret(void);
error_code interpret_address16(instruction, address *);
```

```
  error_code interpret_address24(instruction, address *);
  error_code interpret_argument(instruction, int, cell *);
  error_code interpret_integer(instruction, int, cell *);
  error_code interpret_register(instruction, int, cell *);
  error_code interpret_save(instruction, cell);
  error_code interpret_solo_argument(instruction, cell *);
  error_code interpret_special(instruction, int, cell *);
```

163.

```
#define pins(O) for (int _i ← 0; _i < 4; _i++) printf("%02hhx", ((char *)(O))[_i])
#define psym(O)
        for (half _i ← 0; _i < symbol_length_c(O); _i++) putchar(symbol_buffer_c(O)[_i]);
  error_code interpret(void)
  {
    address link;
    instruction ins;      /* Register? A copy of the current instruction. */
    int width;
    error_code reason;
    Trapped ← LERR_NONE;
    while (¬failure_p(Trapped)) {
      reason ← LERR_NONE;
      if (Ip < 0 ∨ Ip ≥ ADDRESS_INVALID) {
        ins ← 0;
        reason ← LERR_ADDRESS;
        goto Trap;
      }
      ins ← *(instruction *) Ip;
      Ip += sizeof(instruction);
    Reinterpret:
#ifdef LLTEST
#if 0
        printf("%7lu␣%p:␣", Interpret_Count, (void *)(Ip − sizeof(instruction)));
        pins(&ins);
        putchar('␣');
        psym(opcode_label_c(Op[OP(ins)].owner));
        putchar('\n');
#endif
        Interpret_Count++;
        if (Interpret_Limit ∧ Interpret_Count ≥ Interpret_Limit) return LERR_LENGTH;
            /* Cheeky. */
#endif
        switch (OP(ins)) {⟨ Carry out an operation 172 ⟩}
    }
  Halt: return reason;
  }
```

164. **error_code** *interpret_argument*(**instruction** *ins*, **int** *argc*, **cell** ∗*ret*)

```
{
  bool intp;
  assert(argc ≥ 0 ∧ argc ≤ 2);
  switch (argc) {
  case 0: intp ← false; break;
  case 1: intp ← FLAGS(ins) & LBC_FIRST_INTEGER; break;
  case 2: intp ← FLAGS(ins) & LBC_SECOND_INTEGER; break;
  }
  if (intp) return interpret_integer(ins, argc, ret);
  else if (REGP(ins, argc)) return interpret_register(ins, argc, ret);
  else return interpret_special(ins, argc, ret);
}
```

165. **error_code** *interpret_integer*(**instruction** *ins*, **int** *argc*, **cell** ∗*ret*)

```
{
  return new_int_c(IB(ins, argc + 1), ret);
}
```

166. **error_code** *interpret_special*(**instruction** *ins*, **int** *argc*, **cell** ∗*ret*)

```
{
  static cell look[] ← {NIL, LFALSE, LTRUE, VOID, LEOF, INVALID0, INVALID1, UNDEFINED};
  cell value;
  if (VAL(ins, argc) < 0 ∨ VAL(ins, argc) > sizeof (look)) return LERR_INCOMPATIBLE;
  value ← look[VAL(ins, argc)];
  if (fixed_p(value) ∨ ¬special_p(value) ∨ ¬valid_p(value)) return LERR_INCOMPATIBLE;
  ∗ret ← value;
  return LERR_NONE;
}
```

167. For *reading*.

```
error_code interpret_register(instruction ins, int argc, cell *ret)
{
  assert(argc ≥ 0 ∧ argc ≤ 2);
  switch (REG(ins, argc)) {
  case LR_Trap_Handler:
    return LERR_INCOMPATIBLE;
  case LR_Ip:
    if (POP(ins, argc + 1)) return LERR_INCOMPATIBLE;
    return new_pointer(Ip, ret);
  case LR_Trap_Ip:
    if (POP(ins, argc + 1)) return LERR_INCOMPATIBLE;
    return new_pointer(Trap_Ip, ret);
  case LR_Trapped:
    if (POP(ins, argc + 1)) return LERR_INCOMPATIBLE;
    *ret ← Error[Trapped];
    return LERR_NONE;
  case LR_Control_Link:
    if (POP(ins, argc)) return stack_array_pop(Register[REG(ins, argc)], ret);
    else return stack_array_peek(Register[REG(ins, argc)], ret);
  default:
    if (argc ∧ POP(ins, argc)) return stack_list_pop(Register[REG(ins, argc)], ret);
    else *ret ← *Register[REG(ins, argc)];
    return LERR_NONE;
  }
}
```

168.

```
error_code interpret_solo_argument(instruction ins, cell *ret)
{
  long index;
  int16_t value;
  switch (FLAGS(ins)) {
  case LBC_OBJECT_REGISTER:
    if (REGP(ins, 1)) return interpret_register(ins, 1, ret);
    else return interpret_special(ins, 1, ret);
  case LBC_OBJECT_INTEGER:
    value ← SINT(ins);
    return new_int_c(value, ret);
  case LBC_OBJECT_TABLE:
    index ← UINT(ins);
    if (index > Program_ObjectDB_Free) return LERR_OUT_OF_BOUNDS;
    *ret ← array_base(Program_ObjectDB)[index];
    return LERR_NONE;
  default: return LERR_INTERNAL;
  }
}
```

169. **error_code** *interpret_address16*(**instruction** *ins*, **address** ∗*ret*)

```
{
  address from, to, ivia;
  cell rvia;
  error_code reason;
  from ← Ip − sizeof(instruction);
  switch (FLAGS(ins)) {
  case LBC_ADDRESS_ABSOLUTE:
    return LERR_UNIMPLEMENTED;
  case LBC_ADDRESS_INDIRECT:
    ivia ← UINT(ins);
    if (ivia ≥ (address) Program_Export_Free) return LERR_OUT_OF_BOUNDS;
    to ← Program_Export_Base[ivia];
    break;
  case LBC_ADDRESS_REGISTER:
    orreturn (interpret_register(ins, 1, &rvia));
    if (¬pointer_p(rvia)) return LERR_INCOMPATIBLE;
    to ← (address) pointer(rvia);
    break;
  case LBC_ADDRESS_RELATIVE:
    to ← SINT(ins) + from;
    break;
  }
  ∗ret ← to;
  return LERR_NONE;
}
```

170. The same as *interpret_address16* but using ARGT & *resign24* to obtain a 24-bit value.

```
error_code interpret_address24(instruction ins, address ∗ret)
{
  address from, to, ivia;
  cell rvia;
  error_code reason;
  from ← Ip − sizeof(instruction);
  switch (FLAGS(ins)) {
  case LBC_ADDRESS_ABSOLUTE:
    return LERR_UNIMPLEMENTED;
  case LBC_ADDRESS_INDIRECT:
    ivia ← UBIG(ins);
    if (ivia ≥ (address) Program_Export_Free) return LERR_OUT_OF_BOUNDS;
    to ← Program_Export_Base[ivia];
    break;
  case LBC_ADDRESS_REGISTER:
    orreturn (interpret_register(ins, 0, &rvia));
    if (¬pointer_p(rvia)) return LERR_INCOMPATIBLE;
    to ← (address) pointer(rvia);
    break;
  case LBC_ADDRESS_RELATIVE:
    to ← SBIG(ins) + from;
    break;
  }
  ∗ret ← to;
  return LERR_NONE;
}
```

171. **error_code** *interpret_save*(**instruction** *ins*, **cell** *result*)

```
{
  switch (REG(ins, 0)) {
  case LR_Ip:      /* Could be mutable, but why? */
  case LR_Root:
  case LR_Trap_Handler:      /* TODO */
  case LR_Trap_Ip:
  case LR_Trapped:
    return LERR_IMMUTABLE;
  case LR_Control_Link:
    return stack_array_push(Register[REG(ins, 0)], result);
    break;
  case LR_Environment:
    if (¬environment_p(result)) return LERR_INCOMPATIBLE;
    *Register[REG(ins, 0)] ← result;
    return LERR_NONE;
  default:
    *Register[REG(ins, 0)] ← result;
    return LERR_NONE;
  }
}
```

172. ⟨ Carry out an operation 172 ⟩ ≡

```
default:
  if (OP(ins) ≥ 0 ∧ OP(ins) < OPCODE_LENGTH) reason ← LERR_UNIMPLEMENTED;
  else reason ← LERR_INSTRUCTION;
  goto Trap;
case OP_TRAP: ortrap (interpret_integer(ins, 0, &VM_Arg1));
  assert(fixed_p(VM_Arg1));
  reason ← fixed_value(VM_Arg1);
Trap: Trapped ← failure_p(reason) ? reason : LERR_INTERNAL;
  if (Trap_Handler[reason] ≡ ADDRESS_INVALID) goto Halt;
  else {
    Trapped ← LERR_NONE;
    Trap_Ip ← Ip;
    Trap_Arg1 ← VM_Arg1;
    Trap_Arg2 ← VM_Arg2;
    Trap_Result ← VM_Result;
    Ip ← Trap_Handler[reason];
  }
  break;
case OP_HALT:
  reason ← LERR_NONE;
  goto Halt;
```

See also sections 173, 174, 175, 176, 177, 178, 179, 187, 188, 189, 190, 191, 192, 193, and 391.

This code is used in section 163.

173. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_JUMP:
  ortrap (interpret_address24(ins, &Ip));
  break;
case OP_JUMPIF:
  ortrap (interpret_argument(ins, 0, &VM_Result));
  if (true_p(VM_Result)) ortrap (interpret_address16(ins, &Ip));
  break;
case OP_JUMPNOT:
  ortrap (interpret_argument(ins, 0, &VM_Result));
  if (false_p(VM_Result)) ortrap (interpret_address16(ins, &Ip));
  break;
```

174. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_LOAD:
  ortrap (interpret_solo_argument(ins, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_PEND:
  ortrap (interpret_address16(ins, &link));
  orassert(new_pointer(link, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

175. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_ARRAY_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(segment_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_CLOSURE_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(closure_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_ENVIRONMENT_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(environment_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_INTEGER_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(integer_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_PAIR_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(pair_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_PRIMITIVE_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(primitive_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_RESUMPTION_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← LFALSE;
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SEGMENT_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(segment_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SYMBOL_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(symbol_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SYNTAX_P: ortrap (interpret_solo_argument(ins, &VM_Arg1));
  VM_Result ← predicate(syntax_p(VM_Arg1));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

176. These two functions have different signatures.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_EXISTS_P:
case OP_LOOKUP:
  ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Table */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Label */
#if 0
  printf("search␣");
  psym(VM_Arg2);
  printf("\n");
#endif
  if (environment_p(VM_Arg1)) {
    reason ← env_search(VM_Arg1, VM_Arg2, &VM_Result);
    if (reason ≡ LERR_MISSING ∧ OP(ins) ≡ OP_EXISTS_P) VM_Result ← LFALSE;
    else if (failure_p(reason)) goto Trap;
    else if (OP(ins) ≡ OP_EXISTS_P) VM_Result ← LTRUE;
  }
  else {
    ortrap (hashtable_search(VM_Arg1, VM_Arg2, &VM_Result));
    if (OP(ins) ≡ OP_EXISTS_P) VM_Result ← predicate(defined_p(VM_Result));
  }
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_EXTEND:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  ortrap (new_env(VM_Arg1, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_TABLE:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(fixed_p(VM_Arg1));
  assert(fixed_value(VM_Arg1) ≥ 0 ∧ fixed_value(VM_Arg1) ≤ (word) HASHTABLE_MAX_FREE);
  ortrap (new_hashtable(fixed_value(VM_Arg1), &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_DEFINE_M:
case OP_REPLACE_M:
  ortrap (interpret_argument(ins, 0, &VM_Arg1));      /* Table */
  ortrap (interpret_solo_argument(ins, &VM_Arg2));    /* Datum */
  assert(environment_p(VM_Arg1) ∨ hashtable_p(VM_Arg1));
  if (environment_p(VM_Arg1))
    ortrap (env_save_m_imp(VM_Arg1, VM_Arg2, OP(ins) ≡ OP_REPLACE_M));
  else ortrap (hashtable_save_m(VM_Arg1, VM_Arg2, OP(ins) ≡ OP_REPLACE_M));
  break;
```

177. The CONS opcode calls cons.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_CONS:
  ortrap (interpret_argument(ins, 1, &VM_Arg1));
  ortrap (interpret_argument(ins, 2, &VM_Arg2));
  assert(defined_p(VM_Arg1) ∧ defined_p(VM_Arg2));      /* cons is a macro without an assertion. */
  ortrap (cons(VM_Arg1, VM_Arg2, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_CAR:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(ATOM_SIN_DATUM_P(VM_Arg1));
  VM_Result ← A(VM_Arg1)→sin;
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_CDR:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(ATOM_DEX_DATUM_P(VM_Arg1));
  VM_Result ← A(VM_Arg1)→dex;
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SYNTAX:
  ortrap (interpret_argument(ins, 1, &VM_Arg1));
  ortrap (interpret_argument(ins, 2, &VM_Arg2));
  assert(symbol_p(VM_Arg1));
  assert(defined_p(VM_Arg2));
  ortrap (new_atom(VM_Arg1, VM_Arg2, FORM_SYNTAX, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

178. Everything but numbers are `is?`-identical based on pointer equality la `eq?` in scheme. Integers and runes base identity on their *value* (and form) not their address. Numerically identical integers are is?-identical to each other, and runes are `is?`-identical to each other, but an integer will never `is?`-match a rune.

No idea what to say about floats yet.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_CMPIS_P:
  ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Yin */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Yang */
  VM_Result ← predicate(cmpis_p(VM_Arg1, VM_Arg2));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

179. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_CMP: case OP_CMPGT_P: case OP_CMPGE_P: case OP_CMPEQ_P: case OP_CMPLE_P:
  case OP_CMPLT_P: ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Yin */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Yang */
  ortrap (int_cmp(VM_Arg1, VM_Arg2, &VM_Result));
  switch (OP(ins)) {
  case OP_CMP: break;      /* This is fine. */
  case OP_CMPGT_P:      /* Yin (¡:-1),(=:0),(¿:+1) Yang? */
    VM_Result ← predicate(fixed_value(VM_Result) > 0); break;
  case OP_CMPGE_P: VM_Result ← predicate(fixed_value(VM_Result) ≥ 0); break;
  case OP_CMPEQ_P: VM_Result ← predicate(fixed_value(VM_Result) ≡ 0); break;
  case OP_CMPLE_P: VM_Result ← predicate(fixed_value(VM_Result) ≤ 0); break;
  case OP_CMPLT_P: VM_Result ← predicate(fixed_value(VM_Result) < 0); break;
  }
  ortrap (interpret_save(ins, VM_Result));
  break;
```

180. ⟨ Type definitions 11 ⟩ +≡

```
  enum {
    CLOSURE_ADDRESS, CLOSURE_BODY, CLOSURE_ENVIRONMENT, CLOSURE_SIGNATURE, CLOSURE_LENGTH
  };
```

181. ⟨ Function declarations 19 ⟩ +≡

```
  error_code new_closure(cell, cell, cell *);
  error_code closure_body(cell, cell *);
  error_code closure_address(cell, cell *);
  error_code closure_environment(cell, cell *);
  error_code closure_signature(cell, cell *);
```

182.

```
  error_code new_closure(cell sign, cell body, cell *ret)
  {
    cell start;
    error_code reason;
    assert(null_p(sign) ∨ pair_p(sign));
    assert(null_p(body) ∨ pair_p(body));
    orreturn (new_pointer(Interpret_Closure, &start));
    orreturn (new_array_imp(CLOSURE_LENGTH, fix(0), NIL, FORM_CLOSURE, ret));
    array_base(*ret)[CLOSURE_ADDRESS] ← start;
    array_base(*ret)[CLOSURE_BODY] ← body;
    array_base(*ret)[CLOSURE_SIGNATURE] ← sign;
    array_base(*ret)[CLOSURE_ENVIRONMENT] ← Environment;
    return LERR_NONE;
  }
```

183.

```
  error_code closure_address(cell o, cell *ret)
  {
    assert(closure_p(o));
    *ret ← array_base(o)[CLOSURE_ADDRESS];
    return LERR_NONE;
  }
```

184. **error_code** *closure_body*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(closure_p(o));
  *ret ← array_base(o)[CLOSURE_BODY];
  return LERR_NONE;
}
```

185. **error_code** *closure_environment*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(closure_p(o));
  *ret ← array_base(o)[CLOSURE_ENVIRONMENT];
  return LERR_NONE;
}
```

186. **error_code** *closure_signature*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(closure_p(o));
  *ret ← array_base(o)[CLOSURE_SIGNATURE];
  return LERR_NONE;
}
```

187. arg1: (nargs . reversed-formals) or (signature-list . reversed-formals)? arg2: body

⟨ Carry out an operation 172 ⟩ +≡

```
case OP_CLOSURE:
  ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Signature */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Body */
  ortrap (new_closure(VM_Arg1, VM_Arg2, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_OPEN:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(primitive_p(VM_Arg1) ∨ closure_p(VM_Arg1));
  if (primitive_p(VM_Arg1)) {      /* Root? */
    reason ← LERR_UNIMPLEMENTED;
    goto Trap;
  }
  else ortrap (closure_environment(VM_Arg1, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_ADDRESS:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(primitive_p(VM_Arg1) ∨ closure_p(VM_Arg1));
  if (primitive_p(VM_Arg1)) ortrap (new_pointer(primitive_address_c(VM_Arg1), &VM_Result));
  else ortrap (closure_address(VM_Arg1, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_BODY:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(primitive_p(VM_Arg1) ∨ closure_p(VM_Arg1));
  if (primitive_p(VM_Arg1)) {      /* Root? */
    reason ← LERR_UNIMPLEMENTED;
    goto Trap;
  }
  else ortrap (closure_body(VM_Arg1, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SIGNATURE:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  assert(primitive_p(VM_Arg1) ∨ closure_p(VM_Arg1));
  if (primitive_p(VM_Arg1)) VM_Result ← primitive_signature_c(VM_Arg1);
  else ortrap (closure_signature(VM_Arg1, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

188. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_LENGTH:
  ortrap (interpret_solo_argument(ins, &VM_Arg1));
  if (segment_p(VM_Arg1)) new_int_c(segment_length_c(VM_Arg1), &VM_Result);
  else if (symbol_p(VM_Arg1)) new_int_c(symbol_length_c(VM_Arg1), &VM_Result);
  else if (array_p(VM_Arg1)) new_int_c(array_length_c(VM_Arg1), &VM_Result);
  else if (fixed_p(VM_Arg1)) VM_Result ← fix(0);
  else if (integer_p(VM_Arg1)) ortrap (int_length(VM_Arg1, &VM_Result));
  else assert(¬"unreachable");
  ortrap (interpret_save(ins, VM_Result));
  break;
```

189. Unusually this opcode takes 3 arguments and always puts its result in the accumulator.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_SYMBOL:
  ortrap (interpret_argument(ins, 0, &VM_Result));
  ortrap (interpret_argument(ins, 1, &VM_Arg1));
  ortrap (interpret_argument(ins, 2, &VM_Arg2));
  assert(fixed_p(VM_Arg1));
  assert(fixed_p(VM_Arg2));
  ortrap (new_symbol_segment(VM_Result, fixed_value(VM_Arg1), fixed_value(VM_Arg2),
      &VM_Result));
  Accumulator ← VM_Result;
  break;
```

190.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_PEEK8: width ← 8;
  goto PEEK;
case OP_PEEK4: width ← 4;
  goto PEEK;
case OP_PEEK2: width ← 2;
  goto PEEK;
case OP_PEEK: width ← 1;
PEEK: ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Segment */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Offset */
  ortrap (segment_peek(VM_Arg1, fixed_value(VM_Arg2), width, false, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

191.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_POKE8_M: width ← 8;
  goto POKE;
case OP_POKE4_M: width ← 4;
  goto POKE;
case OP_POKE2_M: width ← 2;
  goto POKE;
case OP_POKE_M: width ← 1;
POKE: ortrap (interpret_argument(ins, 0, &VM_Result));      /* Segment */
  ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Offset */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Value */
  ortrap (segment_poke_m(VM_Result, fixed_value(VM_Arg1), width, false, VM_Arg2));
  break;
```

192.

```
⟨ Carry out an operation 172 ⟩ +≡
case OP_ADD: ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Yin */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Yang */
  ortrap (int_add(VM_Arg1, VM_Arg2, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
case OP_SUB: ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Yin */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Yang */
  ortrap (int_sub(VM_Arg1, VM_Arg2, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

193. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_MUL: ortrap (interpret_argument(ins, 1, &VM_Arg1));      /* Yin */
  ortrap (interpret_argument(ins, 2, &VM_Arg2));      /* Yang */
  ortrap (int_mul(VM_Arg1, VM_Arg2, &VM_Result));
  ortrap (interpret_save(ins, VM_Result));
  break;
```

194. Assembly Statement Parser.

195.

```
#define ARGUMENT_BACKWARD_ADDRESS 0
#define ARGUMENT_ERROR 1
#define ARGUMENT_FAR_ADDRESS 2
#define ARGUMENT_FORWARD_ADDRESS 3
#define ARGUMENT_OBJECT 4
#define ARGUMENT_REGISTER 5
#define ARGUMENT_REGISTER_POPPING 6
#define ARGUMENT_RELATIVE_ADDRESS 7
#define ARGUMENT_TABLE 8
#define ARGUMENT_LENGTH 9
#define STATEMENT_LOCAL_LABEL 0
#define STATEMENT_FAR_LABEL 1
#define STATEMENT_INSTRUCTION 2
#define STATEMENT_COVEN 3
#define STATEMENT_COMMENT 6
#define STATEMENT_LENGTH 7
#define pstas_source_byte(P, I) (segment_base((P)→source)[(P)→start + (I)])
⟨ Function declarations 19 ⟩ +≡
  error_code new_statement_imp(cell, cell *);
  error_code parse_segment_to_statement(cell, cell, cell, cell *, cell *);
  error_code pstas_any_symbol(statement_parser *, error_code(*)(statement_parser
      *, half, half), half);
  error_code pstas_argument(statement_parser *, half);
  error_code pstas_argument_address(statement_parser *, half);
  error_code pstas_argument_address_first(statement_parser *, half);
  error_code pstas_argument_encode_error(statement_parser *, half, half);
  error_code pstas_argument_encode_register(statement_parser *, bool, byte *, half, half);
  error_code pstas_argument_encode_symbol(statement_parser *, half, half);
  error_code pstas_argument_error(statement_parser *, half);
  error_code pstas_argument_far_address(statement_parser *, half);
  error_code pstas_argument_local_address(statement_parser *, half);
  error_code pstas_argument_number(statement_parser *, bool, int, bool, half);
  error_code pstas_argument_object(statement_parser *, bool, half);
  error_code pstas_argument_register(statement_parser *, half);
  error_code pstas_argument_signed_number(statement_parser *, bool, bool, half);
  error_code pstas_argument_special(statement_parser *, bool, half);
  error_code pstas_far_label(statement_parser *, half);
  error_code pstas_instruction(statement_parser *, half);
  error_code pstas_instruction_encode(statement_parser *, bool, byte *, half, half);
  error_code pstas_invalid(statement_parser *, half);
  error_code pstas_line_comment(statement_parser *, half);
  error_code pstas_local_label(statement_parser *, half);
  error_code pstas_maybe_no_argument(statement_parser *, half);
  error_code pstas_pre_argument_list(statement_parser *, half);
  error_code pstas_pre_instruction(statement_parser *, half);
  error_code pstas_pre_next_argument(statement_parser *, half);
  error_code pstas_pre_trailing_comment(statement_parser *, half);
  error_code pstas_real_comment(statement_parser *, half);
  error_code statement_append_comment_m(cell, cell);
  error_code statement_argument(cell, cell, cell *);
  error_code statement_comment(cell, cell *);
  error_code statement_far_label(cell, cell *);
  bool statement_has_comment_p(cell);
  bool statement_has_far_label_p(cell);
```

```
bool statement_has_instruction_p(cell);
bool statement_has_local_label_p(cell);
error_code statement_instruction(cell, cell *);
bool statement_integer_fits_p(cell, int, cell);
error_code statement_local_label(cell, cell *);
error_code statement_set_argument_m(cell, int, int, cell);
error_code statement_set_far_label_m(cell, cell);
error_code statement_set_instruction_m(cell, cell);
error_code statement_set_local_label_m(cell, cell);
```

196. `#define new_statement(R) new_statement_imp(NIL, (R))`

```
error_code new_statement_imp(cell op, cell *ret)
{
  error_code reason;
  assert(null_p(op) ∨ opcode_p(op));
  orreturn (new_array_imp(STATEMENT_LENGTH, fix(0), NIL, FORM_STATEMENT, ret));
  array_base(*ret)[STATEMENT_INSTRUCTION] ← op;
  return LERR_NONE;
}
```

197.

```
error_code statement_far_label(cell o, cell *ret)
{
  assert(statement_p(o));
  *ret ← array_base(o)[STATEMENT_FAR_LABEL];
  return LERR_NONE;
}
bool statement_has_far_label_p(cell o)
{
  assert(statement_p(o));
  return ¬null_p(array_base(o)[STATEMENT_FAR_LABEL]);
}
error_code statement_set_far_label_m(cell o, cell label)
{
  assert(statement_p(o));
  assert(symbol_p(label));
  assert(null_p(array_base(o)[STATEMENT_FAR_LABEL]));
  array_base(o)[STATEMENT_FAR_LABEL] ← label;
  return LERR_NONE;
}
```

198. **error_code** *statement_local_label*(**cell** *o*, **cell** ∗*ret*)
```
{
  assert(statement_p(o));
  ∗ret ← array_base(o)[STATEMENT_LOCAL_LABEL];
  return LERR_NONE;
}
bool statement_has_local_label_p(cell o)
{
  assert(statement_p(o));
  return ¬null_p(array_base(o)[STATEMENT_LOCAL_LABEL]);
}
error_code statement_set_local_label_m(cell o, cell label)
{
  assert(statement_p(o));
  assert(fixed_p(label) ∧ fixed_value(label) ≥ 0 ∧ fixed_value(label) ≤ 9);
  assert(null_p(array_base(o)[STATEMENT_LOCAL_LABEL]));
  array_base(o)[STATEMENT_LOCAL_LABEL] ← label;
  return LERR_NONE;
}
```

199. **error_code** *statement_instruction*(**cell** *o*, **cell** ∗*ret*)
```
{
  assert(statement_p(o));
  ∗ret ← array_base(o)[STATEMENT_INSTRUCTION];
  return LERR_NONE;
}
bool statement_has_instruction_p(cell o)
{
  assert(statement_p(o));
  return ¬null_p(array_base(o)[STATEMENT_INSTRUCTION]);
}
error_code statement_set_instruction_m(cell o, cell op)
{
  assert(statement_p(o));
  assert(opcode_p(op));
  assert(null_p(array_base(o)[STATEMENT_INSTRUCTION]));
  array_base(o)[STATEMENT_INSTRUCTION] ← op;
  return LERR_NONE;
}
```

200. **error_code** *statement_argument*(**cell** *o*, **cell** *id*, **cell** ∗*ret*)
```
{
  assert(statement_p(o));
  assert(fixed_p(id) ∧ fixed_value(id) ≥ 0 ∧ fixed_value(id) ≤ 2);
  ∗ret ← array_base(o)[STATEMENT_COVEN + fixed_value(id)];
  return LERR_NONE;
}
error_code statement_set_argument_m(cell o, int id, int cat, cell object)
{
  assert(statement_p(o));
  assert(id ≥ 0 ∧ id ≤ 2);
  assert(cat ≥ 0 ∧ cat < ARGUMENT_LENGTH);
  assert(¬special_p(o) ∨ valid_p(object));
  return new_atom(fix(cat), object, FORM_ARGUMENT, array_base(o) + STATEMENT_COVEN + id);
}
```

201. **bool** *statement_integer_fits_p*(**cell** *o*, **int** *argid*, **cell** *number*)
```
{
  char signature;
  cell op;
  assert(statement_p(o));
  assert(statement_has_instruction_p(o));
  assert(integer_p(number));
  statement_instruction(o, &op);
  signature ← opcode_signature_c(op)[argid];
  assert(signature ≡ ALOT ∨ signature ≡ ALOB);
  if (¬fixed_p(number)) return false;      /* TODO: Not correct on 16 bit */
  if (signature ≡ ALOT) return fixed_value(number) ≥ −64 ∧ fixed_value(number) ≤ 63;
  else return fixed_value(number) ≥ −32768 ∧ fixed_value(number) ≤ 32767;
}
```

202. **bool** *statement_has_comment_p*(**cell** *o*)
```
{
  assert(statement_p(o));
  return ¬null_p(array_base(o)[STATEMENT_COMMENT]);
}
```
error_code *statement_comment*(**cell** *o*, **cell** ∗*ret*)
```
{
  assert(statement_p(o));
  ∗ret ← array_base(o)[STATEMENT_COMMENT];
  return LERR_NONE;
}
```
error_code *statement_append_comment_m*(**cell** *o*, **cell** *text*)
```
{
  cell tmp;
  error_code reason;
  assert(statement_p(o));
  assert(pair_p(text));      /* Triplet: (segment offset length) */
  orreturn (cons(text, array_base(o)[STATEMENT_COMMENT], &tmp));
  array_base(o)[STATEMENT_COMMENT] ← tmp;
  return LERR_NONE;
}
```

203. ⟨ Type definitions 11 ⟩ +≡
```
typedef struct {
  cell partial;
  cell source;
  char ∗signature;
  int argument;
  half consume;
  half start;
  half length;
  half end;
} statement_parser;
```

204. **error_code** *parse_segment_to_statement*(**cell** *source*, **cell** *start*, **cell** *length*, **cell** ∗*consume*, **cell** ∗*ret*)

```
{
  byte first;
  word value;
  statement_parser pstate ← {0};
  error_code reason;
  assert(segment_p(source));
  pstate.source ← source;
  orassert(int_value(start, &value));
  assert(value ≥ 0 ∧ value < segment_length_c(source));
  pstate.start ← value;
  orassert(int_value(length, &value));
  assert(value ≥ 1 ∧ pstate.start + value ≤ segment_length_c(source));
  pstate.length ← value;
  pstate.end ← pstate.start + pstate.length;
  orreturn (new_statement(&pstate.partial));
  switch ((first ← pstas_source_byte(&pstate, 0))) {
  case '#': case ';': case '\'': reason ← pstas_line_comment(&pstate, 0);
    break;
  case '␣': case '\t': case '\n': reason ← pstas_pre_instruction(&pstate, 0);
    break;
  default:
    if (ascii_digit_p(first)) reason ← pstas_local_label(&pstate, 0);
    else if (ascii_printable_p(first)) reason ← pstas_far_label(&pstate, 0);
    else reason ← pstas_invalid(&pstate, 0);
    break;
  }
  if (failure_p(reason)) return reason;
  orreturn (new_int_c(pstate.consume, consume));
  ∗ret ← pstate.partial;
  return LERR_NONE;
}
```

205. **error_code** *pstas_invalid*(**statement_parser** ∗*pstate* **unused**, **half** *offset* **unused**)

```
{
  assert(¬"invalid\n");
  return LERR_UNIMPLEMENTED;
}
```

206. **error_code** *pstas_line_comment*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  byte leader;
  half i;
  leader ← pstas_source_byte(pstate, offset);
  i ← offset;
next_leader:
  if (i ≡ pstate→end) {
    pstate→consume ← i;
    return LERR_NONE;
  }
  else if (pstas_source_byte(pstate, i) ≡ leader) {
    i++;
    goto next_leader;
  }
next_space:
  if (i ≡ pstate→end) {
    pstate→consume ← i;
    return LERR_NONE;
  }
  switch (pstas_source_byte(pstate, i)) {
  case '␣': case '\t': i++;
    goto next_space;
  case '\n': pstate→consume ← i + 1;
    return LERR_NONE;
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) return pstas_real_comment(pstate, i);
    else return pstas_invalid(pstate, i);
  }
}
```

207. **error_code** *pstas_local_label*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  half hoffset;
  byte h;
  cell label;
  error_code reason;
  hoffset ← offset + 1;
  if (hoffset ≡ pstate→end) return pstas_invalid(pstate, hoffset);
  h ← pstas_source_byte(pstate, hoffset);
  if (h ≠ 'h' ∧ h ≠ 'H') return pstas_invalid(pstate, hoffset);
  label ← fix(pstas_source_byte(pstate, offset) − '0');
  orassert(statement_set_local_label_m(pstate→partial, label));
  return pstas_pre_instruction(pstate, hoffset + 1);
}
```

208. **error_code** *pstas_far_label*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  cell label;
  half i, j;
  error_code reason;
  j ← i ← offset + 1;
next_label:
  if (i ≡ pstate→end) {
  complete_line:
    orreturn (new_symbol_segment(pstate→source, pstate→start + offset, j − offset, &label));
    orassert(statement_set_far_label_m(pstate→partial, label));
    pstate→consume ← i;
    return LERR_NONE;
  }
  switch (pstas_source_byte(pstate, i)) {
  case '␣': case '\t':
    orreturn (new_symbol_segment(pstate→source, pstate→start + offset, i − offset, &label));
    orassert(statement_set_far_label_m(pstate→partial, label));
    return pstas_pre_instruction(pstate, i);
  case '\n': j ← i++;
    goto complete_line;
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      i++;
      goto next_label;
    }
    else return pstas_invalid(pstate, i);
  }
}
```

209. **error_code** *pstas_pre_instruction*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  if (offset ≡ pstate→end) {
    offset−−;
    goto finish_line;
  }
  if (pstas_source_byte(pstate, offset) ≡ '\n') goto finish_line;
  while (pstas_source_byte(pstate, offset) ≡ '␣' ∨ pstas_source_byte(pstate, offset) ≡ '\t') offset++;
  if (pstas_source_byte(pstate, offset) ≡ '\n') {
  finish_line: pstate→consume ← offset + 1;
    return LERR_NONE;
  }
  if (ascii_printable_p(pstas_source_byte(pstate, offset))) return pstas_instruction(pstate, offset);
  else return pstas_invalid(pstate, offset);
}
```

210. **error_code** *pstas_instruction*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  bool pop_p;
  byte b;
  byte label[24];
  half i, out;
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  pop_p ← (pstas_source_byte(pstate, offset) ≡ '=');
  i ← offset + pop_p;
  label[0] ← 'V';
  label[1] ← 'M';
  label[2] ← ':';
  out ← 3;
next_byte:
  if (out ≡ 24 ∨ i ≡ pstate→end) return pstas_instruction_encode(pstate, false, label, out, i);
  switch ((b ← pstas_source_byte(pstate, i))) {
  case '␣': case '\t': return pstas_instruction_encode(pstate, true, label, out, i);
  case '\n': return pstas_instruction_encode(pstate, false, label, out, i);
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      label[out++] ← ascii_upcase(b);
      i++;
      goto next_byte;
    }
    else return pstas_invalid(pstate, i);
  }
}
```

211. **error_code** *pstas_instruction_encode*(**statement_parser** ∗*pstate*, **bool** *more_p*, **byte** ∗*label*, **half** *length*, **half** *offset*)

```
{
  cell sop, lop;
  error_code reason;
  orreturn (new_symbol_buffer(label, length, Λ, &sop));
  reason ← env_search(Environment, sop, &lop);
  if (reason ≡ LERR_MISSING) return pstas_invalid(pstate, offset);
  if (failure_p(reason)) return reason;
  if (¬opcode_p(lop)) return pstas_invalid(pstate, offset);
  orassert(statement_set_instruction_m(pstate→partial, lop));
  if (more_p) return pstas_pre_argument_list(pstate, offset);
  else return pstas_maybe_no_argument(pstate, offset);
}
```

```
212. error_code pstas_maybe_no_argument(statement_parser *pstate, half offset)
{
  cell op;
  error_code reason;
  orassert(statement_instruction(pstate→partial, &op));
  assert(opcode_p(op));
  if (opcode_object(op)→arg0 ≡ NARG) {
    return pstas_pre_trailing_comment(pstate, offset);
    pstate→consume ← offset;
    return LERR_NONE;
  }
  else return pstas_invalid(pstate, offset);
}

213. error_code pstas_pre_argument_list(statement_parser *pstate, half offset)
{
  cell op;
  half i;
  error_code reason;
  i ← offset;
next_byte:
  if (i ≡ pstate→end) return pstas_maybe_no_argument(pstate, i);
  switch (pstas_source_byte(pstate, i)) {
  case '␣': case '\t': i++;
    goto next_byte;
  case '\n': return pstas_maybe_no_argument(pstate, i);
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      orassert(statement_instruction(pstate→partial, &op));
      pstate→signature ← opcode_signature_c(op);
      if (*pstate→signature ≡ NARG) return pstas_maybe_no_argument(pstate, i);
      pstate→argument ← 0;
      return pstas_argument(pstate, i);
    }
    else return pstas_invalid(pstate, i);
  }
}

214. error_code pstas_argument(statement_parser *pstate, half offset)
{
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  assert(*pstate→signature ≠ NARG);
  switch (*pstate→signature) {
  case AADD: return pstas_argument_address(pstate, offset);
  case ALOB: return pstas_argument_object(pstate, true, offset);
  case AREG: return pstas_argument_register(pstate, offset);
  case ALOT: return pstas_argument_object(pstate, false, offset);
  case ARGH: return pstas_argument_error(pstate, offset);
  default: return LERR_INTERNAL;
  }
}
```

215. **error_code** *pstas_argument_address*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  else if (pstas_source_byte(pstate, offset) ≡ '@')
    return pstas_argument_address_first(pstate, offset + 1);
  else return pstas_argument_register(pstate, offset);
}
```

216. **error_code** *pstas_argument_address_first*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  byte first;
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  first ← pstas_source_byte(pstate, offset);
  if (ascii_digit_p(first)) return pstas_argument_local_address(pstate, offset);
  else if (ascii_printable_p(first) ∧ first ≠ ',') return pstas_argument_far_address(pstate, offset);
  else return pstas_invalid(pstate, offset);
}
```

217. **error_code** *pstas_argument_local_address*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  int label;
  error_code reason;
  label ← pstas_source_byte(pstate, offset) − '0';
  offset++;
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  switch (pstas_source_byte(pstate, offset)) {
  case 'b': case 'B': orassert(statement_set_argument_m(pstate→partial, pstate→argument,
        ARGUMENT_BACKWARD_ADDRESS, fix(label)));
    return pstas_pre_next_argument(pstate, offset + 1);
  case 'f': case 'F': orassert(statement_set_argument_m(pstate→partial, pstate→argument,
        ARGUMENT_FORWARD_ADDRESS, fix(label)));
    return pstas_pre_next_argument(pstate, offset + 1);
  default: return pstas_invalid(pstate, offset);
  }
}
```

218. **error_code** *pstas_argument_far_address*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  cell label;
  half i;
  error_code reason;
  i ← offset;
next_byte:
  if (i ≡ pstate→end) {
  finish_address:
    orreturn (new_symbol_segment(pstate→source, pstate→start + offset, i − offset, &label));
    orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_FAR_ADDRESS,
        label));
    return pstas_pre_next_argument(pstate, i);
  }
  switch (pstas_source_byte(pstate, i)) {
  case ',': case '␣': case '\t': case '\n': goto finish_address;
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      i++;
      goto next_byte;
    }
    else return pstas_invalid(pstate, i);
  }
}
```

219. **error_code** *pstas_pre_next_argument*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  byte sep;
  pstate→argument++;
  pstate→signature++;
  if (offset ≡ pstate→end) sep ← '\n';
  else sep ← pstas_source_byte(pstate, offset);
  if (pstate→argument > 2 ∨ ∗pstate→signature ≡ NARG) {
    if (sep ≡ '␣' ∨ sep ≡ '\t' ∨ sep ≡ '\n') return pstas_pre_trailing_comment(pstate, offset);
  }
  else {
    if (sep ≡ ',') return pstas_argument(pstate, offset + 1);
  }
  return pstas_invalid(pstate, offset);
}
```

220. **error_code** *pstas_pre_trailing_comment*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  half i;
  i ← offset;
next_byte:
  if (i ≡ pstate→end) {
  finish_line: pstate→consume ← i;
    return LERR_NONE;
  }
  switch (pstas_source_byte(pstate, i)) {
  case '␣': case '\t': i++;
    goto next_byte;
  case '\n': i++;
    goto finish_line;
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) return pstas_real_comment(pstate, offset);
    else return pstas_invalid(pstate, offset);
  }
}
```

221. **error_code** *pstas_real_comment*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  cell tmp;
  half i, j;
  error_code reason;
  i ← offset;
next_byte:
  if (i ≡ pstate→end) {
  finish_comment: j ← i;
    while (ascii_space_p(pstas_source_byte(pstate, j − 1))) j−−;
    orreturn (cons(fix(j), NIL, &tmp));
    orreturn (cons(fix(j − offset), tmp, &tmp));
    orreturn (cons(pstate→source, tmp, &tmp));
    orreturn (statement_append_comment_m(pstate→partial, tmp));
    pstate→consume ← i;
    return LERR_NONE;
  }
  switch (pstas_source_byte(pstate, i)) {
  default:
    if (¬ascii_printable_p(pstas_source_byte(pstate, i))) return pstas_invalid(pstate, i);
  case '␣': case '\t': i++;
    goto next_byte;
  case '\n': i++;
    goto finish_comment;
  }
}
```

222. Incorporates *pstas_argument_parse_register*.

```
error_code pstas_argument_register(statement_parser *pstate, half offset)
{
  bool first, pop_p;
  byte b;
  byte label[24];
  half i, out;
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  pop_p ← (pstas_source_byte(pstate, offset) ≡ '=');
  i ← offset + pop_p;
  label[0] ← 'V';
  label[1] ← 'M';
  label[2] ← ':';
  out ← 3;
  first ← true;
next_byte:
  if (out ≡ 24 ∨ i ≡ pstate→end) return pstas_argument_encode_register(pstate, pop_p, label, out, i);
  switch ((b ← pstas_source_byte(pstate, i))) {
  case ',': case '␣': case '\t': case '\n':
    return pstas_argument_encode_register(pstate, pop_p, label, out, i);
  case '-': case '_': label[out++] ← '-';
    first ← true;
    i++;
    goto next_byte;
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      label[out++] ← first ? ascii_upcase(pstas_source_byte(pstate,
          i)) : ascii_downcase(pstas_source_byte(pstate, i));
      first ← false;
      i++;
      goto next_byte;
    }
    else return pstas_invalid(pstate, i);
  }
}
```

223.

```
error_code pstas_argument_encode_register(statement_parser *pstate, bool pop_p, byte
    *label, half length, half offset)
{
  cell sreg, rreg;
  error_code reason;
  orreturn (new_symbol_buffer(label, length, Λ, &sreg));
  reason ← env_search(Environment, sreg, &rreg);
  if (reason ≡ LERR_MISSING) return pstas_invalid(pstate, offset);
  if (failure_p(reason)) return reason;
  if (¬register_p(rreg)) return pstas_invalid(pstate, offset);
  orassert(statement_set_argument_m(pstate→partial, pstate→argument,
      pop_p ? ARGUMENT_REGISTER_POPPING : ARGUMENT_REGISTER, rreg));
  return pstas_pre_next_argument(pstate, offset);
}
```

224. **error_code** *pstas_argument_error*(**statement_parser** ∗*pstate*, **half** *offset*)

```
{
  if (offset ≡ pstate→end ∨ pstas_source_byte(pstate, offset) ≠ '\'')
    return pstas_invalid(pstate, offset);
  return pstas_any_symbol(pstate, pstas_argument_encode_error, offset + 1);
}
```

225. **error_code** *pstas_argument_encode_error*(**statement_parser** ∗*pstate*, **half** *length_offset*, **half** *offset*)

```
{
  cell lerr, serr;
  error_code reason;
  orreturn (new_symbol_segment(pstate→source, pstate→start + offset, length_offset − offset, &serr));
  reason ← env_search(Environment, serr, &lerr);
  if (reason ≡ LERR_MISSING) return pstas_invalid(pstate, length_offset);
  if (failure_p(reason)) return reason;
  if (¬error_p(lerr)) return pstas_invalid(pstate, length_offset);
  orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_ERROR, lerr));
  return pstas_pre_next_argument(pstate, length_offset);
}
```

226. **error_code** *pstas_any_symbol*(**statement_parser** ∗*pstate*, **error_code**(∗*then*)(**statement_parser** ∗, **half**, **half**), **half** *offset*)

```
{
  half i;
  i ← offset;
next_byte:
  if (i ≡ pstate→end) return then(pstate, i, offset);
  switch (pstas_source_byte(pstate, i)) {
  case ',': case '␣': case '\t': case '\n': return then(pstate, i, offset);
  default:
    if (ascii_printable_p(pstas_source_byte(pstate, i))) {
      i++;
      goto next_byte;
    }
    else return pstas_invalid(pstate, i);
  }
}
```

227. **error_code** *pstas_argument_object*(**statement_parser** ∗*pstate*, **bool** *full*, **half** *offset*)

```
{
  error_code reason;
  switch (pstas_source_byte(pstate, offset)) {
  case '-': return pstas_argument_signed_number(pstate, true, full, offset + 1);
  case '+': return pstas_argument_signed_number(pstate, false, full, offset + 1);
  case '#': return pstas_argument_special(pstate, full, offset + 1);
  case '(':
    if (offset + 1 ≡ pstate→end ∨ pstas_source_byte(pstate, offset + 1) ≠ ')')
      return pstas_invalid(pstate, offset + 1);
    orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, NIL));
    return pstas_pre_next_argument(pstate, offset + 2);
  case '\'':
    if (full) return pstas_any_symbol(pstate, pstas_argument_encode_symbol, offset + 1);
    else return pstas_invalid(pstate, offset);
  default:
    if (ascii_digit_p(pstas_source_byte(pstate, offset)))
      return pstas_argument_signed_number(pstate, false, full, offset);
    else return pstas_argument_register(pstate, offset);
  }
}
```

228. **error_code** *pstas_argument_encode_symbol*(**statement_parser** ∗*pstate*, **half** *length_offset*, **half** *offset*)

```
{
  cell sym;
  error_code reason;
  orreturn (new_symbol_segment(pstate→source, pstate→start + offset, length_offset − offset, &sym));
  orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, sym));
  return pstas_pre_next_argument(pstate, length_offset);
}
```

229. **error_code** *pstas_argument_special*(**statement_parser** ∗*pstate*, **bool** *full*, **half** *offset*)

```
{
  error_code reason;
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  switch (pstas_source_byte(pstate, offset)) {
  case 'b': case 'B': return pstas_argument_number(pstate, false, 2, full, offset + 1);
  case 'o': case 'O': return pstas_argument_number(pstate, false, 8, full, offset + 1);
  case 'd': case 'D': return pstas_argument_number(pstate, false, 10, full, offset + 1);
  case 'x': case 'X': return pstas_argument_number(pstate, false, 16, full, offset + 1);
  case 'f': case 'F':
    orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, LFALSE));
    return pstas_pre_next_argument(pstate, offset + 1);
  case 't': case 'T':
    orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, LTRUE));
    return pstas_pre_next_argument(pstate, offset + 1);
  case 'U':
    if (offset + 8 < pstate→end ∧ pstas_source_byte(pstate,
          offset + 1) ≡ 'N' ∧ pstas_source_byte(pstate, offset + 2) ≡ 'D' ∧ pstas_source_byte(pstate,
          offset + 3) ≡ 'E' ∧ pstas_source_byte(pstate, offset + 4) ≡ 'F' ∧ pstas_source_byte(pstate,
          offset + 5) ≡ 'I' ∧ pstas_source_byte(pstate, offset + 5) ≡ 'N' ∧ pstas_source_byte(pstate,
          offset + 6) ≡ 'E' ∧ pstas_source_byte(pstate, offset + 8) ≡ 'D') {
      orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT,
          UNDEFINED));
      return pstas_pre_next_argument(pstate, offset + 9);
    }
    else return pstas_invalid(pstate, offset);
  case 'V':
    if (offset + 3 < pstate→end ∧ pstas_source_byte(pstate, offset + 1) ≡ 'O' ∧ pstas_source_byte(pstate,
          offset + 2) ≡ 'I' ∧ pstas_source_byte(pstate, offset + 3) ≡ 'D') {
      orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, VOID));
      return pstas_pre_next_argument(pstate, offset + 4);
    }
    else return pstas_invalid(pstate, offset);
  default: return pstas_invalid(pstate, offset);
  }
}
```

230. **error_code** *pstas_argument_signed_number*(**statement_parser** ∗*pstate*, **bool** *negate*, **bool** *full*, **half** *offset*)

```
{
  if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
  if (pstas_source_byte(pstate, offset) ≡ '#') {
    offset++;
    if (offset ≡ pstate→end) return pstas_invalid(pstate, offset);
    switch (pstas_source_byte(pstate, offset)) {
    case 'b': case 'B': return pstas_argument_number(pstate, negate, 2, full, offset);
    case 'o': case 'O': return pstas_argument_number(pstate, negate, 8, full, offset);
    case 'd': case 'D': return pstas_argument_number(pstate, negate, 10, full, offset);
    case 'x': case 'X': return pstas_argument_number(pstate, negate, 16, full, offset);
    default: return pstas_invalid(pstate, offset);
    }
  }
  else if (ascii_digit_p(pstas_source_byte(pstate, offset)))
    return pstas_argument_number(pstate, negate, 10, full, offset);
  else return pstas_invalid(pstate, offset);
}
```

231. **error_code** *pstas_argument_number*(**statement_parser** ∗*pstate*, **bool** *negate*, **int** *base*, **bool** *full*, **half** *offset*)

```
{
  byte b;
  cell lsum;
  int add, sum, max, min, shift, width;
  half i, j;
  error_code reason;
  sum ← 0;
  if (full) {
    min ← −32768;
    max ← 32767;
  }
  else {
    min ← −64;
    max ← 63;
  }
  i ← offset;
  switch (base) {
  case 2: width ← 16;
    shift ← 1;
    break;
  case 8: width ← 6;
    shift ← 3;
    break;
  case 10: width ← 5;
    goto next_base10;
  case 16: width ← shift ← 4;
    break;
  }
next_digit:
  if (i ≡ pstate→end) goto finish_number;
  b ← pstas_source_byte(pstate, i);
  if (ascii_space_p(b) ∨ b ≡ ',') goto finish_number;
  else if (ascii_digit_p(b) ∨ ascii_hex_p(b)) {
    if (i − offset ≡ width) return pstas_invalid(pstate, offset);
    if (b ≤ '9') add ← b − '0';
    else add ← 10 + (b & 0xdf) − 'A';
    if (add ≥ base) return pstas_invalid(pstate, offset);
    sum ← (sum ≪ shift) | add;
    i++;
    goto next_digit;
  }
  else return pstas_invalid(pstate, offset);
next_base10:
  if (i ≡ pstate→end) goto finish_base10;
  b ← pstas_source_byte(pstate, i);
  if (ascii_space_p(b) ∨ b ≡ ',') goto finish_base10;
  else if (ascii_digit_p(b)) {
    if (i − offset ≡ width) return pstas_invalid(pstate, offset);
    i++;
    goto next_base10;
  }
  else return pstas_invalid(pstate, offset);
finish_base10:
  for (j ← offset; j < i; j++) {
```

```
    sum *= 10;
    sum += pstas_source_byte(pstate, j) - '0';
  }
finish_number:
  if (negate) sum ← -sum;
  if (sum < min ∨ sum > max) return pstas_invalid(pstate, i);
  orreturn (new_int_c(sum, &lsum));
  orassert(statement_set_argument_m(pstate→partial, pstate→argument, ARGUMENT_OBJECT, lsum));
  return pstas_pre_next_argument(pstate, i);
}
```

232. Assembler.

```
#define ASSEMBLY_STATUS 0
#define ASSEMBLY_STATUS_IN_PROGRESS 0
#define ASSEMBLY_STATUS_READY 1
#define ASSEMBLY_STATUS_INSTALLED 2
#define ASSEMBLY_BODY 1
#define ASSEMBLY_LENGTH 2
#define ASSEMBLY_PROGRESS_BODY 1
#define ASSEMBLY_PROGRESS_NEXT_ADDRESS 2
#define ASSEMBLY_PROGRESS_FAR_ADDRESS 3
#define ASSEMBLY_PROGRESS_FAR_ARGUMENT 4
#define ASSEMBLY_PROGRESS_PENDING_LABEL 5
#define ASSEMBLY_PROGRESS_BACKWARD_ADDRESS 6
#define ASSEMBLY_PROGRESS_FORWARD_ARGUMENT 7
#define ASSEMBLY_PROGRESS_OBJECTDB 8
#define ASSEMBLY_PROGRESS_COMMENT_STATEMENT 9
#define ASSEMBLY_PROGRESS_PENDING_COMMENT 10
#define ASSEMBLY_PROGRESS_COMMENTARY 11
#define ASSEMBLY_PROGRESS_BLOB 12
#define ASSEMBLY_PROGRESS_LENGTH 13
#define ASSEMBLY_READY_BODY 1
#define ASSEMBLY_READY_EXPORT 2
#define ASSEMBLY_READY_REQUIRE 3
#define ASSEMBLY_READY_OBJECTDB 5
#define ASSEMBLY_READY_COMMENTARY 6
#define ASSEMBLY_READY_BLOB 7
#define ASSEMBLY_READY_LENGTH 8
```

⟨ Function declarations 19 ⟩ +≡

```
error_code new_assembly_progress(cell *);
error_code new_assembly_buffer(byte *, word, cell *);
error_code new_assembly_segment(cell, cell *);
error_code assembly_append_comment_separator_m(cell);
error_code assembly_append_far_argument_m(cell, cell, cell, int);
error_code assembly_append_forward_argument_m(cell, cell, cell, int);
error_code assembly_append_pending_comment_m(cell, cell);
error_code assembly_clear_forward_argument_list_m(cell, cell);
error_code assembly_comment_statement(cell, cell *);
error_code assembly_commentary(cell, cell, cell *);
error_code assembly_far_address(cell, cell, cell *);
error_code assembly_far_argument_list(cell, cell, cell *);
error_code assembly_forward_argument(cell, cell, cell *);
bool assembly_has_far_address_p(cell, cell);
bool assembly_has_pending_label_p(cell);
error_code assembly_install_object_m(cell, cell, half *);
error_code assembly_next_address(cell, cell *);
error_code assembly_object_table(cell o, cell *);
error_code assembly_pending_comment(cell, cell *);
error_code assembly_pending_label(cell, cell *);
error_code assembly_set_backward_address_m(cell, cell, cell);
error_code assembly_set_comment_statement_m(cell, cell);
error_code assembly_set_commentary_m(cell, cell, cell);
error_code assembly_set_far_address_m(cell, cell, cell);
error_code assembly_set_next_address_m(cell, cell);
error_code assembly_set_pending_comment_m(cell, cell);
error_code assembly_set_pending_label_m(cell, cell);
error_code assembly_set_statement_m(cell, word, cell);
```

error_code *assembly_append_line_m*(**cell**, **cell**);
error_code *assembly_append_statement_m*(**cell**, **cell**, **cell** ∗);
error_code *assembly_encode_ALOT*(**int**, **cell**, **instruction** ∗);
error_code *assembly_encode_AREG*(**int**, **cell**, **instruction** ∗);
error_code *assembly_finish_m*(**cell**, **cell** ∗);
error_code *assembly_fix_forward_links_m*(**cell**, **cell**, **cell**);
error_code *assembly_install_m*(**cell**, **cell** ∗);
bool *assembly_validate_integer*(**int**, **bool**, **cell**, **word** ∗);

233.

#define *assembly_in_progress_p*(*O*)
(*assembly_p*(*O*) ∧ *array_base*(*O*)[ASSEMBLY_STATUS] ≡ *fix*(ASSEMBLY_STATUS_IN_PROGRESS))
#define *assembly_ready_p*(*O*)
(*assembly_p*(*O*) ∧ *array_base*(*O*)[ASSEMBLY_STATUS] ≡ *fix*(ASSEMBLY_STATUS_READY))
#define *assembly_installed_p*(*O*)
(*assembly_p*(*O*) ∧ *array_base*(*O*)[ASSEMBLY_STATUS] ≡ *fix*(ASSEMBLY_STATUS_INSTALLED))

error_code *new_assembly_progress*(**cell** ∗*ret*)
{
cell *body*, *blob*, *far_address*, *far_argument*;
cell *backward*, *forward*, *commentary*, *objectdb*;
error_code *reason*;
orreturn (*new_array*(100, *fix*(0), &*body*));
orreturn (*new_hashtable*(0, &*far_address*));
orreturn (*new_hashtable*(0, &*far_argument*));
orreturn (*new_array*(10, *fix*(0), &*backward*));
orreturn (*new_array*(10, *fix*(0), &*forward*));
orreturn (*new_array*(0, *fix*(0), &*objectdb*));
orreturn (*new_hashtable*(0, &*commentary*));
orreturn (*new_segment*(0, 0, &*blob*));
orreturn (*new_array_imp*(ASSEMBLY_PROGRESS_LENGTH, *fix*(0), NIL, FORM_ASSEMBLY, *ret*));
array_base(∗*ret*)[ASSEMBLY_STATUS] ← *fix*(ASSEMBLY_STATUS_IN_PROGRESS);
array_base(∗*ret*)[ASSEMBLY_PROGRESS_BODY] ← *body*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_NEXT_ADDRESS] ← *fix*(0);
array_base(∗*ret*)[ASSEMBLY_PROGRESS_FAR_ADDRESS] ← *far_address*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_FAR_ARGUMENT] ← *far_argument*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_BACKWARD_ADDRESS] ← *backward*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_FORWARD_ARGUMENT] ← *forward*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_OBJECTDB] ← *objectdb*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_COMMENTARY] ← *commentary*;
array_base(∗*ret*)[ASSEMBLY_PROGRESS_BLOB] ← *blob*;
return LERR_NONE;
}

234. **error_code** *assembly_statement*(**cell** *o*, **word** *lineno*, **cell** ∗*ret*)

```
{
  cell body;
  assert(assembly_p(o));
  assert(ASSEMBLY_BODY ≡ ASSEMBLY_PROGRESS_BODY);
  body ← array_base(o)[ASSEMBLY_BODY];
  assert(lineno ≥ 0 ∧ lineno < array_length_c(body));
  ∗ret ← array_base(body)[lineno];
  assert(statement_p(∗ret));
  return LERR_NONE;
}
error_code assembly_set_statement_m(cell o, word lineno, cell statement)
{
  cell body;
  word grow;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(lineno ≥ 0);
  assert(statement_p(statement));
  body ← array_base(o)[ASSEMBLY_PROGRESS_BODY];
  grow ← (lineno + 0x80) & ∼0x7f;      /∗ Round to next multiple of 128. ∗/
  if (grow > array_length_c(body)) orreturn (array_resize_m(body, grow, NIL));
  array_base(body)[lineno] ← statement;
  return LERR_NONE;
}
```

235. **error_code** *assembly_next_address*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(assembly_in_progress_p(o));
  ∗ret ← array_base(o)[ASSEMBLY_PROGRESS_NEXT_ADDRESS];
  assert(integer_p(∗ret));
  return LERR_NONE;
}
error_code assembly_set_next_address_m(cell o, cell addr)
{
  assert(assembly_in_progress_p(o));
  assert(integer_p(addr));      /∗ ∧≥ 0 but maybe not < length. ∗/
  array_base(o)[ASSEMBLY_PROGRESS_NEXT_ADDRESS] ← addr;
  return LERR_NONE;
}
```

236. **bool** *assembly_has_far_address_p*(**cell** *o*, **cell** *label*)

```
{
  cell found;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(symbol_p(label));
  orreturn (hashtable_search(array_base(o)[ASSEMBLY_PROGRESS_FAR_ADDRESS], label, &found));
  return defined_p(found);
}
```

237. **error_code** *assembly_far_address*(**cell** *o*, **cell** *label*, **cell** ∗*ret*)

```
{
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(symbol_p(label));
  orreturn (hashtable_search(array_base(o)[ASSEMBLY_PROGRESS_FAR_ADDRESS], label, ret));
  assert(pair_p(∗ret));
  assert(integer_p(A(∗ret)¬dex));
  return LERR_NONE;
}
```

error_code *assembly_set_far_address_m*(**cell** *o*, **cell** *label*, **cell** *lineno*)

```
{
  cell tuple;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(symbol_p(label));
  assert(integer_p(lineno));
  orreturn (cons(label, lineno, &tuple));
  orreturn (hashtable_save_m(array_base(o)[ASSEMBLY_PROGRESS_FAR_ADDRESS], tuple, false));
  return LERR_NONE;
}
```

238. Hashtable of label : list of (line . argid) tuples.

```
error_code assembly_far_argument_list(cell o, cell label, cell *ret)
{
  cell found;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(symbol_p(label));
  orreturn (hashtable_search(array_base(o)[ASSEMBLY_PROGRESS_FAR_ARGUMENT], label, &found));
  if (pair_p(found)) *ret ← A(found)→dex;
  else *ret ← NIL;
  for (found ← *ret; ¬null_p(found); found ← A(found)→dex) {
    assert(pair_p(found));
    assert(pair_p(A(found)→sin));
    assert(integer_p(A(A(found)→sin)→sin));
    assert(integer_p(A(A(found)→sin)→dex));
  }
  return LERR_NONE;
}
error_code assembly_append_far_argument_m(cell o, cell label, cell lineno, int argid)
{
  bool replace;
  cell found, tuple;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(symbol_p(label));
  assert(integer_p(lineno));
  assert(argid ≤ 1);
  orreturn (cons(lineno, fix(argid), &tuple));
  orreturn (hashtable_search(array_base(o)[ASSEMBLY_PROGRESS_FAR_ARGUMENT], label, &found));
  replace ← defined_p(found);
  if (replace) {
    assert(pair_p(found) ∧ A(found)→sin ≡ label);
    orreturn (cons(tuple, A(found)→dex, &tuple));
  }
  else orreturn (cons(tuple, NIL, &tuple));
  orreturn (cons(label, tuple, &tuple));
  return hashtable_save_m(array_base(o)[ASSEMBLY_PROGRESS_FAR_ARGUMENT], tuple, replace);
}
```

239. **bool** *assembly_has_pending_label_p*(**cell** *o*)

```
{
  assert(assembly_in_progress_p(o));
  return ¬null_p(array_base(o)[ASSEMBLY_PROGRESS_PENDING_LABEL]);
}
error_code assembly_pending_label(cell o, cell *ret)
{
  assert(assembly_in_progress_p(o));
  *ret ← array_base(o)[ASSEMBLY_PROGRESS_PENDING_LABEL];
  return LERR_NONE;
}
error_code assembly_set_pending_label_m(cell o, cell label)
{
  assert(assembly_in_progress_p(o));
  if (¬null_p(label)) {
    assert((fixed_p(label) ∧ fixed_value(label) ≥ 0 ∧ fixed_value(label) ≤ 9));
    assert(null_p(array_base(o)[ASSEMBLY_PROGRESS_PENDING_LABEL]));
  }
  array_base(o)[ASSEMBLY_PROGRESS_PENDING_LABEL] ← label;
  return LERR_NONE;
}
```

240. Backward address. Array of 10 integers or NIL. TODO: flatten back/fore arrays.

```
error_code assembly_backward_address(cell o, cell link, cell *ret)
{
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link) ∧ fixed_value(link) ≥ 0 ∧ fixed_value(link) ≤ 9);
  o ← array_base(o)[ASSEMBLY_PROGRESS_BACKWARD_ADDRESS];
  *ret ← array_base(o)[fixed_value(link)];
  return LERR_NONE;
}
error_code assembly_set_backward_address_m(cell o, cell link, cell lineno)
{
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link) ∧ fixed_value(link) ≥ 0 ∧ fixed_value(link) ≤ 9);
  assert(integer_p(lineno));
  o ← array_base(o)[ASSEMBLY_PROGRESS_BACKWARD_ADDRESS];
  array_base(o)[fixed_value(link)] ← lineno;
  return LERR_NONE;
}
```

241. Forward argument. Array of 10 lists of (lineno . argid) tuples.

```
error_code assembly_forward_argument(cell o, cell link, cell *ret)
{
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link) ∧ fixed_value(link) ≥ 0 ∧ fixed_value(link) ≤ 9);
  o ← array_base(o)[ASSEMBLY_PROGRESS_FORWARD_ARGUMENT];
  *ret ← array_base(o)[fixed_value(link)];
  return LERR_NONE;
}
error_code assembly_append_forward_argument_m(cell o, cell link, cell lineno, int argid)
{
  cell linklist, tuple;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link) ∧ fixed_value(link) ≥ 0 ∧ fixed_value(link) ≤ 9);
  assert(integer_p(lineno));
  assert(argid ≥ 0 ∧ argid ≤ 2);
  orreturn (cons(lineno, fix(argid), &tuple));
  linklist ← array_base(o)[ASSEMBLY_PROGRESS_FORWARD_ARGUMENT];
  return cons(tuple, array_base(linklist)[fixed_value(link)], &array_base(linklist)[fixed_value(link)]);
}
error_code assembly_clear_forward_argument_list_m(cell o, cell link)
{
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link) ∧ fixed_value(link) ≥ 0 ∧ fixed_value(link) ≤ 9);
  o ← array_base(o)[ASSEMBLY_PROGRESS_FORWARD_ARGUMENT];
  array_base(o)[fixed_value(link)] ← NIL;
  return LERR_NONE;
}
```

242.

```
error_code assembly_object_table(cell o, cell *ret)
{
  assert(assembly_in_progress_p(o));
  *ret ← array_base(o)[ASSEMBLY_PROGRESS_OBJECTDB];
  return LERR_NONE;
}
```

243. **error_code** *assembly_install_object_m*(**cell** *o*, **cell** *object*, **half** ∗*ret*)

```
{
  cell objectdb;
  half i, length;
  assert(assembly_in_progress_p(o));
  assert(¬special_p(o));
  objectdb ← array_base(o)[ASSEMBLY_PROGRESS_OBJECTDB];
  length ← array_length_c(objectdb);
  for (i ← 0; i < length; i++) {
    if (cmpis_p(object, array_base(objectdb)[i])) {
      ∗ret ← i;
      return LERR_NONE;
    }
  }
  array_resize_m(objectdb, length + 1, UNDEFINED);
  array_base(objectdb)[length] ← object;
  ∗ret ← length;
  return LERR_NONE;
}
```

244. **error_code** *assembly_comment_statement*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(assembly_in_progress_p(o));
  ∗ret ← array_base(o)[ASSEMBLY_PROGRESS_COMMENT_STATEMENT];
  return LERR_NONE;
}
```

error_code *assembly_set_comment_statement_m*(**cell** *o*, **cell** *s*)

```
{
  assert(assembly_in_progress_p(o));
  assert(null_p(s) ∨ statement_p(s));
  array_base(o)[ASSEMBLY_PROGRESS_COMMENT_STATEMENT] ← s;
  return LERR_NONE;
}
```

245. **error_code** *assembly_pending_comment*(**cell** *o*, **cell** ∗*ret*)

```
{
  assert(assembly_in_progress_p(o));
  *ret ← array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT];
  return LERR_NONE;
}
error_code assembly_set_pending_comment_m(cell o, cell commentary)
{
  assert(assembly_in_progress_p(o));
  assert(null_p(commentary) ∨ pair_p(commentary));
  array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT] ← commentary;
  return LERR_NONE;
}
error_code assembly_append_pending_comment_m(cell o, cell line)
{
  cell tuple;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(defined_p(line));
  orreturn (cons(line, array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT], &tuple));
  array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT] ← tuple;
  return LERR_NONE;
}
error_code assembly_append_comment_separator_m(cell o)
{
  cell tuple;
  error_code reason;
  assert(assembly_in_progress_p(o));
  orreturn (cons(NIL, array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT], &tuple));
  array_base(o)[ASSEMBLY_PROGRESS_PENDING_COMMENT] ← tuple;
  return LERR_NONE;
}
```

246. **error_code** *assembly_commentary*(**cell** *o*, **cell** *lineno*, **cell** ∗*ret*)

```
{
  cell table;
  error_code reason;
  assert(assembly_in_progress_p(o));
  if (integer_p(lineno)) orreturn (int_to_symbol(lineno, &lineno));
  else assert(symbol_p(lineno));
  table ← array_base(o)[ASSEMBLY_PROGRESS_COMMENTARY];
  orreturn (hashtable_search(table, lineno, ret));
  if (undefined_p(∗ret)) ∗ret ← NIL;
  return LERR_NONE;
}
```

error_code *assembly_set_commentary_m*(**cell** *o*, **cell** *lineno*, **cell** *comment*)

```
{
  cell table;
  error_code reason;
  assert(assembly_in_progress_p(o));
  if (integer_p(lineno)) orreturn (int_to_symbol(lineno, &lineno));
  else assert(symbol_p(lineno));
  table ← array_base(o)[ASSEMBLY_PROGRESS_COMMENTARY];
  orreturn (cons(lineno, comment, &comment));
  return hashtable_save_m(table, comment, false);
}
```

247. TODO: Blob accessors.

248. **error_code** *assembly_append_line_m*(**cell** *o*, **cell** *line*)

```
{
  cell comment, label, lineno, previous;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(statement_p(line));
  assembly_next_address(o, &lineno);
  statement_far_label(line, &label);
  if (¬null_p(label)) {
    if (assembly_has_far_address_p(o, label)) return LERR_UNIMPLEMENTED;     /* in use */
    assembly_set_comment_statement_m(o, NIL);
    assembly_set_far_address_m(o, label, lineno);
  }
  statement_local_label(line, &label);
  if (¬null_p(label)) {
    assert(fixed_p(label) ∧ fixed_value(label) ≥ 0 ∧ fixed_value(label) ≤ 9);
    assembly_set_comment_statement_m(o, NIL);
    assembly_set_pending_label_m(o, label);
  }
  if (statement_has_instruction_p(line)) {
    assembly_pending_comment(o, &comment);
    if (¬null_p(comment)) {
      assert(pair_p(comment));
      if (null_p(A(comment)→sin)) comment ← A(comment)→dex;
      orreturn (assembly_set_commentary_m(o, lineno, comment));
      assembly_set_pending_comment_m(o, NIL);
    }
    orreturn (assembly_append_statement_m(o, line, &line));
    assembly_set_comment_statement_m(o, line);
  }
  else if (statement_has_comment_p(line)) {
    statement_comment(line, &comment);
    assert(pair_p(comment));
    if (¬null_p(A(comment)→sin)) comment ← A(comment)→sin;
    assembly_comment_statement(o, &previous);
    if (null_p(previous)) orreturn (assembly_append_pending_comment_m(o, comment));
    else orreturn (statement_append_comment_m(previous, comment));
  }
  else {
    statement_comment(line, &comment);
    if (¬null_p(comment) ∧ ¬null_p(A(comment)→sin))
      orreturn (assembly_append_comment_separator_m(o));
  }
  return LERR_NONE;
}
```

249. In practice line numbers all fit within the space of a fixed integer except on 16 bit machines where a 24 bit address space doesn't make much sense anyway. 16 bit support is not being considered especially at this time.

```
error_code assembly_append_statement_m(cell o, cell statement, cell *ret)
{
  cell argument, delta, label, lineat, lineno, link, op;
  int i;
  word cdelta, clineno, ignored;
  half tablerow;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(statement_p(statement));
  assembly_next_address(o, &lineno);
  if (assembly_has_pending_label_p(o)) {
    assembly_pending_label(o, &label);
    assert(fixed_p(label));
    orreturn (assembly_fix_forward_links_m(o, label, lineno));
  }
  else label ← NIL;
  for (i ← 0; i ≤ 2; i++) {      /* Argument 2 can never have an address. */
    statement_argument(statement, fix(i), &argument);
    if (null_p(argument)) continue;
    assert(argument_p(argument));
    switch (fixed_value(A(argument)→sin)) {
    case ARGUMENT_FAR_ADDRESS:
      orreturn (assembly_append_far_argument_m(o, A(argument)→dex, lineno, i));
      break;
    case ARGUMENT_FORWARD_ADDRESS:
      orreturn (assembly_append_forward_argument_m(o, A(argument)→dex, lineno, i));
      break;
    case ARGUMENT_BACKWARD_ADDRESS: link ← A(argument)→dex;
      assembly_backward_address(o, link, &lineat);
      if (null_p(lineat)) return LERR_UNIMPLEMENTED;
      assert(fixed_p(lineat) ∧ fixed_p(lineno));
      cdelta ← fixed_value(lineat) − fixed_value(lineno);
      orreturn (new_int_c(cdelta, &delta));
      orreturn (statement_set_argument_m(statement, i, ARGUMENT_RELATIVE_ADDRESS, delta));
      break;
    case ARGUMENT_REGISTER: case ARGUMENT_REGISTER_POPPING:
      assert(register_p(A(argument)→dex));
      break;
    case ARGUMENT_OBJECT: orassert(statement_instruction(statement, &op));
      if (integer_p(A(argument)→dex) ∧ assembly_validate_integer(16, true, A(argument)→dex,
          &ignored)) break;
      if (special_p(A(argument)→dex) ∧ ¬fixed_p(A(argument)→dex)) break;
      orreturn (assembly_install_object_m(o, A(argument)→dex, &tablerow));
      orreturn (new_int_c(tablerow, &link));
      orreturn (statement_set_argument_m(statement, i, ARGUMENT_TABLE, link));
      break;
    }
  }
  assert(fixed_p(lineno));
  clineno ← fixed_value(lineno);
  assert(clineno ≥ 0);
  assembly_set_statement_m(o, clineno, statement);
  if (¬null_p(label)) {
```

```
    assembly_set_backward_address_m(o, label, lineno);
    assembly_set_pending_label_m(o, NIL);
  }
  orreturn (new_int_c(clineno + 1, &lineno));
  assembly_set_next_address_m(o, lineno);
  *ret ← statement;
  return LERR_NONE;
}
```

250. **error_code** *assembly_fix_forward_links_m*(**cell** *o*, **cell** *link*, **cell** *lineto*)

```
{
  cell argid, delta, linefrom, statement, tuple, pending;
  word cdelta;
  error_code reason;
  assert(assembly_in_progress_p(o));
  assert(fixed_p(link));
  assert(integer_p(lineto));
  assembly_forward_argument(o, link, &pending);
  for ( ; ¬null_p(pending); pending ← A(pending)¬dex) {
    assert(pair_p(pending) ∧ pair_p(A(pending)¬sin));
    tuple ← A(pending)¬sin;
    linefrom ← A(tuple)¬sin;
    argid ← A(tuple)¬dex;
    assert(fixed_p(argid));
    assert(fixed_p(linefrom) ∧ fixed_p(lineto));
    cdelta ← fixed_value(lineto) − fixed_value(linefrom);
    orreturn (new_int_c(cdelta, &delta));
    assembly_statement(o, fixed_value(linefrom), &statement);
    assert(statement_p(statement));
    orreturn (statement_set_argument_m(statement, fixed_value(argid),
        ARGUMENT_RELATIVE_ADDRESS, delta));
  }
  return assembly_clear_forward_argument_list_m(o, link);
}
```

251. Check:

pending-label is `NIL`

forward-argument-table links are all `NIL`.

Change:

Add any pending comment to the table at epilogue.

Filter far addresses into exported and dependencies.

For all far-address If begins !, add to export Otherwise if there is no list of statements add it to requires Update matching statements to relative and remove from far-argument

If there are statements left in far-argument they are requirements.

252. **error_code** *assembly_finish_m*(**cell** *o*, **cell** ∗*ret*)

```
{
  cell far_address, far_argument, fromlist, exportdb, require;
  cell argid, body, comment, lineto, linefrom, next, statement, sym;
  cell table, tuple;
  word delta, i;
  error_code reason;
  assert(assembly_in_progress_p(o));
  if (assembly_has_pending_label_p(o)) return LERR_UNIMPLEMENTED;
  table ← array_base(o)[ASSEMBLY_PROGRESS_FORWARD_ARGUMENT];
  for (i ← 0; i ≤ 9; i++)
    if (¬null_p(array_base(table)[i])) return LERR_UNIMPLEMENTED;
  ⟨ Collect pending comments and reduce the code array 253 ⟩
  ⟨ Update in-page far links to relative 254 ⟩
  ⟨ Record remaining required far links 255 ⟩
  orreturn (new_array_imp(ASSEMBLY_READY_LENGTH, fix(0), NIL, FORM_ASSEMBLY, ret));
  array_base(∗ret)[ASSEMBLY_STATUS] ← fix(ASSEMBLY_STATUS_READY);
  array_base(∗ret)[ASSEMBLY_READY_BODY] ← body;
  array_base(∗ret)[ASSEMBLY_READY_EXPORT] ← exportdb;
  array_base(∗ret)[ASSEMBLY_READY_REQUIRE] ← require;
  array_base(∗ret)[ASSEMBLY_READY_OBJECTDB] ← array_base(o)[ASSEMBLY_PROGRESS_OBJECTDB];
  array_base(∗ret)[ASSEMBLY_READY_COMMENTARY] ← array_base(o)[ASSEMBLY_PROGRESS_COMMENTARY];
  array_base(∗ret)[ASSEMBLY_READY_BLOB] ← array_base(o)[ASSEMBLY_PROGRESS_BLOB];
  return LERR_NONE;
}
```

253. ⟨ Collect pending comments and reduce the code array 253 ⟩ ≡

```
orreturn (assembly_pending_comment(o, &comment));
if (¬null_p(comment)) {
  orreturn (new_symbol_const("epilogue", &sym));
  assembly_set_commentary_m(o, sym, comment);
}
lineto ← array_base(o)[ASSEMBLY_PROGRESS_NEXT_ADDRESS];
assert(fixed_p(lineto));
body ← array_base(o)[ASSEMBLY_PROGRESS_BODY];
orreturn (array_resize_m(body, fixed_value(lineto), NIL));
```

This code is used in section 252.

254. ⟨ Update in-page far links to relative 254 ⟩ ≡

```
far_address ← array_base(o)[ASSEMBLY_PROGRESS_FAR_ADDRESS];
far_argument ← array_base(o)[ASSEMBLY_PROGRESS_FAR_ARGUMENT];
orreturn (new_hashtable(0, &exportdb));
for (i ← 0; i < hashtable_length_c(far_address); i++) {
  next ← hashtable_base(far_address)[i];
  if (null_p(next) ∨ ¬defined_p(next)) continue;
  lineto ← A(next)→dex;
  assert(fixed_p(lineto));
  if (symbol_buffer_c(A(next)→sin)[0] ≡ '!') {
    orreturn (cons(A(next)→sin, lineto, &tuple));
    orreturn (hashtable_save_m(exportdb, tuple, false));
  }
  orreturn (hashtable_search(far_argument, A(next)→sin, &fromlist));
  if (undefined_p(fromlist)) continue;
  assert(pair_p(fromlist) ∧ A(fromlist)→sin ≡ A(next)→sin);
  fromlist ← A(fromlist)→dex;
  for ( ; ¬null_p(fromlist); fromlist ← A(fromlist)→dex) {
    assert(pair_p(A(fromlist)→sin));
    tuple ← A(fromlist)→sin;
    linefrom ← A(tuple)→sin;
    assert(fixed_p(linefrom));
    argid ← A(tuple)→dex;
    assert(fixed_p(argid));
    delta ← fixed_value(lineto) − fixed_value(linefrom);
    assembly_statement(o, fixed_value(linefrom), &statement);
    statement_set_argument_m(statement, fixed_value(argid), ARGUMENT_RELATIVE_ADDRESS,
        fix(delta));
  }
  hashtable_erase_m(far_argument, A(next)→sin, false);
}
```

This code is used in section 252.

255. ⟨ Record remaining required far links 255 ⟩ ≡

```
orreturn (new_hashtable(0, &require));
for (i ← 0; i < hashtable_length_c(far_argument); i++) {
  next ← hashtable_base(far_argument)[i];
  if (null_p(next) ∨ ¬defined_p(next)) continue;
  orreturn (cons(A(next)→sin, A(next)→dex, &tuple));
  orreturn (hashtable_save_m(require, tuple, false));
}
```

This code is used in section 252.

256. **error_code** *assembly_install_m*(**cell** *o*, **cell** ∗*ret*)

```
{
  address avalue, boffset, page, real;
  cell arg, blob, body, found, label, link, lins, objectdb, op, tmp;
  cell new_table, new_program, statement_halt;
  half i, ioffset, new_objectdb_length, next_export;
  instruction ins, ivalue;
  word ito, wvalue;
  opcode_table ∗opb;
  error_code reason;
  assert(assembly_ready_p(o));
  ⟨ Prepare a new code page 257 ⟩
  pthread_mutex_lock(&Program_Lock);
  ⟨ Copy constant objects into Program_ObjectDB 258 ⟩
  ⟨ Look for required address symbols 259 ⟩
  ⟨ Add exported address symbols to a copy of Program_Export_Table 260 ⟩
  ⟨ Install instructions as bytecode and commentary 261 ⟩
  Program_Export_Table ← new_table;
  Program_Export_Free ← next_export;
  Program_ObjectDB_Free ← new_objectdb_length;
  ∗ret ← new_program;
  pthread_mutex_unlock(&Program_Lock);
  return LERR_NONE;
Trap:
  while (new_objectdb_length > Program_ObjectDB_Free)
    array_base(Program_ObjectDB)[−−new_objectdb_length] ← NIL;
  pthread_mutex_unlock(&Program_Lock);
  free_mem((void ∗) page);
  return reason;
}
```

257. ⟨ Prepare a new code page 257 ⟩ ≡

```
orreturn (alloc_mem(Λ, CODE_PAGE_LENGTH, CODE_PAGE_LENGTH, (void **) &page));
reason ← new_array_imp(ASSEMBLY_PROGRESS_LENGTH, fix(0), NIL, FORM_ASSEMBLY, &new_program);
if (failure_p(reason)) {
  free_mem((void ∗) page);
  return reason;
}
array_base(new_program)[ASSEMBLY_STATUS] ← fix(ASSEMBLY_STATUS_INSTALLED);
∗((cell ∗) page) ← new_program;      /∗ Point to the program atom at offset 0. ∗/
blob ← array_base(o)[ASSEMBLY_READY_BLOB];
  /∗ Copy binary data to the beginning of the page. ∗/
memmove((void ∗)(page + sizeof(cell)), segment_base(blob), segment_length_c(blob));
ioffset ← segment_length_c(blob)/sizeof(instruction);
if (segment_length_c(blob) % sizeof(instruction)) ioffset++;
ioffset += sizeof(cell)/sizeof(instruction);
boffset ← ioffset ∗ sizeof(instruction);
```

This code is used in section 256.

258. Does not increment *Program_ObjectDB_Free* until the code page is completed.
TODO: memmove occasionally segfaults.

```
⟨ Copy constant objects into Program_ObjectDB 258 ⟩ ≡
  objectdb ← array_base(o)[ASSEMBLY_READY_OBJECTDB];
  new_objectdb_length ← Program_ObjectDB_Free;
  i ← new_objectdb_length + array_length_c(objectdb);
  if (i ≥ OBJECTDB_LENGTH) {
    reason ← LERR_LIMIT;
    goto Trap;
  }
  if (i ≥ array_length_c(Program_ObjectDB)) ortrap (array_resize_m(Program_ObjectDB, i, NIL));
  new_objectdb_length ← i;
  memmove(array_base(Program_ObjectDB) + (Program_ObjectDB_Free * sizeof(cell)),
      array_base(objectdb), array_length_c(objectdb) * sizeof(cell));
```

This code is used in section 256.

259. Table of (label . list-of-statements). It might be marginally more efficient to perform this scan (and the subsequent one) during the process of building each instruction.

```
⟨ Look for required address symbols 259 ⟩ ≡
  for (i ← 0; i < hashtable_length_c(array_base(o)[ASSEMBLY_READY_REQUIRE]); i++) {
    label ← hashtable_base(array_base(o)[ASSEMBLY_READY_REQUIRE])[i];
    if (null_p(label) ∨ ¬defined_p(label)) continue;
    ortrap (hashtable_search(Program_Export_Table, A(label)¬sin, &found));
    if (undefined_p(found)) {
      reason ← LERR_MISSING;
      goto Trap;
    }
  }
```

This code is used in section 256.

260. Table of (label . address). No need to save/copy *Program_Export_Base*, only the table and *Program_Export_Free*.

Links to the address' location in *Program_Export_Base* is saved in the next free slot in *page*, beginning above the blob (and back-link pointer that shouldn't be there): *boffset*/*ioffset*.

Page layout is:

Pointer to program object

Blob with padding to instruction-size boundary

Program code, instruction 0 is at 0 + *ioffset*.

The real address (ie. (0 + ioffset OR page)) is put in the next free slot in *Program_Export_Base* and the hashtable is adjusted to point to that offset in place of the bytecode offset (ne address).

Does not increment *Program_Export_Free* until the code page is ready.

```
⟨ Add exported address symbols to a copy of Program_Export_Table 260 ⟩ ≡
  ortrap (copy_hashtable(Program_Export_Table, &new_table));
  next_export ← Program_Export_Free;
  for (i ← 0; i < hashtable_length_c(array_base(o)[ASSEMBLY_READY_EXPORT]); i++) {
    label ← hashtable_base(array_base(o)[ASSEMBLY_READY_EXPORT])[i];
    if (null_p(label) ∨ undefined_p(label)) continue;
    assert(pair_p(label));
    assert(fixed_p(A(label)→dex));      /* destination offset from start of program code */
    ito ← fixed_value(A(label)→dex);
    real ← ito * sizeof(instruction);
    real += boffset;
    real |= page;
    Program_Export_Base[next_export] ← real;      /* Real location. */
    ortrap (new_int_c(next_export, &link));      /* Link offset in PEB. */
    ortrap (cons(A(label)→sin, link, &link));
    ortrap (hashtable_save_m(new_table, link, false));
      /* Will conflict with LERR_EXISTS if necessary. */
    next_export++;
  }
```

This code is used in section 256.

261.

```
⟨ Install instructions as bytecode and commentary 261 ⟩ ≡
  body ← array_base(o)[ASSEMBLY_READY_BODY];
  ortrap (new_statement_imp(Op[OP_HALT].owner, &statement_halt));
  for (i ← 0; i < array_length_c(body); i++) {
    lins ← array_base(body)[i];
    if (null_p(lins)) lins ← statement_halt;
    assert(statement_p(lins));
    statement_instruction(lins, &op);
    opb ← opcode_object(op);
    ins ← htobe32((fixed_value(opcode_id_c(op)) & 0xff) ≪ 24);
    if (opb→arg0 ≡ NARG) {
      assert(opb→arg1 ≡ NARG ∧ opb→arg2 ≡ NARG);
      statement_argument(lins, fix(0), &arg);
      if (¬null_p(arg)) goto incompatible;
      goto finish_arguments;
    }
    ⟨ Encode the first argument 262 ⟩
    ⟨ Encode the second argument 267 ⟩
    ⟨ Encode the third argument 271 ⟩
  finish_arguments: ((instruction *)(page | boffset))[i] ← ins;
  }
```

This code is used in section 256.

262. ⟨ Encode the first argument 262 ⟩ ≡
```
statement_argument(lins, fix(0), &arg);
if (null_p(arg)) goto incompatible;
switch (opb→arg0) { case AADD: ⟨ Encode a single address argument and break 263 ⟩case ARGH:
⟨ Encode an error identifier argument and break 264 ⟩case ALOT: ⟨ Encode the first object argument
    and break 265 ⟩case AREG: ⟨ Encode the first register argument and break 266 ⟩
default: reason ← LERR_INTERNAL;
goto Trap; }
```
This code is used in section 261.

263. ie. 24 bits.

⟨ Encode a single address argument and break 263 ⟩ ≡
```
assert(opb→arg1 ≡ NARG ∧ opb→arg2 ≡ NARG);
statement_argument(lins, fix(1), &tmp);
if (¬null_p(tmp)) goto incompatible;
switch (fixed_value(A(arg)→sin)) {
case ARGUMENT_RELATIVE_ADDRESS:
  if (¬assembly_validate_integer(22, true, A(arg)→dex,
        &wvalue) ∨ wvalue ≡ 0 ∨ wvalue < −i ∨ wvalue > array_length_c(body) − i) {
    reason ← LERR_OUT_OF_BOUNDS;
    goto Trap;
  }
  wvalue ∗= sizeof(instruction);
  ins |= htobe32((wvalue & 0x3fffff) | LBC_ADDRESS_RELATIVE);
  break;
case ARGUMENT_FAR_ADDRESS: ortrap (vm_locate_entry(A(arg)→dex, &wvalue, &avalue));
    /∗ Table index ∗/
  assert(instruction_page(avalue) ≠ page);
  ins |= htobe32((wvalue & 0x3fffff) | LBC_ADDRESS_INDIRECT);
  break;
case ARGUMENT_REGISTER: case ARGUMENT_REGISTER_POPPING:
  ortrap (assembly_encode_AREG(0, arg, &ivalue));
  ins |= ivalue;
  ins |= htobe32(LBC_ADDRESS_REGISTER);      /∗ In fact this is 0. ∗/
  break;
default: incompatible:      /∗ This exit point is common; here is as good a place as any. ∗/
  reason ← LERR_INCOMPATIBLE;
  goto Trap;
}
break;
```
This code is used in section 262.

264. These can be combined.

⟨ Encode an error identifier argument and break 264 ⟩ ≡
```
assert(opb→arg1 ≡ NARG ∧ opb→arg2 ≡ NARG);
if (A(arg)→sin ≠ fix(ARGUMENT_ERROR) ∨ ¬error_p(A(arg)→dex)) goto incompatible;
statement_argument(lins, fix(1), &tmp);
if (¬null_p(tmp)) goto incompatible;
if (¬assembly_validate_integer(8, true, error_id_c(A(arg)→dex), &wvalue)) goto incompatible;
    /∗ Overkill; guaranteed by the error object. ∗/
ortrap (assembly_encode_ALOT(0, arg, &ivalue));
ins |= ivalue;
break;
```
This code is used in section 262.

265. ⟨ Encode the first object argument and **break** 265 ⟩ ≡
```
ortrap (assembly_encode_ALOT(0, arg, &ivalue));
ins |= ivalue;
break;
```
This code is used in section 262.

266. ⟨ Encode the first register argument and **break** 266 ⟩ ≡
```
ortrap (assembly_encode_AREG(0, arg, &ivalue));
ins |= ivalue;
break;
```
This code is used in section 262.

267. Second argument.

⟨ Encode the second argument 267 ⟩ ≡
```
statement_argument(lins, fix(1), &arg);
if (opb¬arg1 ≡ NARG) {
  if (null_p(arg)) goto finish_arguments;
  else {
    reason ← LERR_INCOMPATIBLE;
    goto Trap;
  }
}
else if (null_p(arg)) {
  reason ← LERR_INCOMPATIBLE;
  goto Trap;
}
switch (opb¬arg1) { case AADD: ⟨ Encode a 16-bit address and break 268 ⟩case ALOB:
⟨ Encode a large object and break 269 ⟩case ALOT: ⟨ Encode the middle ALOT and break 270 ⟩
default: reason ← LERR_INTERNAL;
goto Trap; }
```
This code is used in section 261.

268. 16-bit relative only. No 16 bit indirect jumps (yet?).

⟨ Encode a 16-bit address and **break** 268 ⟩ ≡
```
assert(opb¬arg2 ≡ NARG);
switch (fixed_value(A(arg)¬sin)) {
case ARGUMENT_RELATIVE_ADDRESS:
  if (¬assembly_validate_integer(16, true, A(arg)¬dex,
        &wvalue) ∨ wvalue ≡ 0 ∨ wvalue < −i ∨ wvalue > array_length_c(body) − i) {
    reason ← LERR_OUT_OF_BOUNDS;
    goto Trap;
  }
  wvalue ∗= sizeof(instruction);
  ins |= htobe32((wvalue & 0xffff) | LBC_ADDRESS_RELATIVE);
  break;
case ARGUMENT_REGISTER: case ARGUMENT_REGISTER_POPPING:
  ortrap (assembly_encode_AREG(1, arg, &ivalue));
  ins |= ivalue;
  ins |= htobe32(LBC_ADDRESS_REGISTER);      /∗ In fact this is 0. ∗/
  break;
default: goto incompatible;
}
break;
```
This code is used in section 267.

269. ALOB; anything: int, const, pair:reg/bool, pair:table/index

```
⟨ Encode a large object and break 269 ⟩ ≡
  assert(opb→arg2 ≡ NARG);
  switch (fixed_value(A(arg)→sin)) {
  case ARGUMENT_TABLE: assert(integer_p(A(arg)→dex));      /* assembly table offset */
    if (¬assembly_validate_integer(16, false, A(arg)→dex, &wvalue)) goto incompatible;
    else if ((wvalue += Program_ObjectDB_Free) ≥ OBJECTDB_LENGTH) {
      reason ← LERR_OUT_OF_BOUNDS;
      goto Trap;
    }
    ins |= htobe32((wvalue & 0xffff) | LBC_OBJECT_TABLE);
    break;
  case ARGUMENT_OBJECT:
    if (integer_p(A(arg)→dex)) {
      if (¬assembly_validate_integer(16, true, A(arg)→dex, &wvalue)) goto incompatible;
      ins |= htobe32((wvalue & 0xffff) | LBC_OBJECT_INTEGER);
    }
    else {
      if (¬special_p(A(arg)→dex)) goto incompatible;
    case ARGUMENT_REGISTER: case ARGUMENT_REGISTER_POPPING:
      ortrap (assembly_encode_ALOT(1, arg, &ivalue));
      ins |= ivalue;
    }
    break;
  default: goto incompatible;
  }
  break;
```

This code is used in section 267.

270. ⟨ Encode the middle ALOT and **break** 270 ⟩ ≡

```
  assert(opb→arg2 ≡ ALOT);
  ortrap (assembly_encode_ALOT(1, arg, &ivalue));
  ins |= ivalue;
  break;
```

This code is used in section 267.

271. ⟨ Encode the third argument 271 ⟩ ≡

```
  if (opb→arg2 ≠ NARG) {
    statement_argument(lins, fix(2), &arg);
    if (null_p(arg)) goto incompatible;
    ortrap (assembly_encode_ALOT(2, arg, &ivalue));
    ins |= ivalue;
  }
```

This code is used in section 261.

272. 24 bits are used to encode relative address offsets and the error number to `OP_TRAP`. 16 bits encode relative offsets and any 16 bit integer.

```
bool assembly_validate_integer(int width, bool signed_p, cell lvalue, word *ret)
{
  word min, max, cvalue;
  error_code reason;
  assert(integer_p(lvalue));
  assert(signed_p ≡ 0 ∨ signed_p ≡ 1);
  switch (width | signed_p) {
  default: case 8: abort( );
  case 9: min ← −0x80;
    max ← 0x7f;
    break;
  case 16: min ← 0;
    max ← 0xffff;
    break;
  case 17: min ← −0x8000;
    max ← 0x7fff;
    break;
  case 22: min ← 0;
    max ← 0x3fffff;
    break;
  case 23: min ← −0x200000;
    max ← 0x1fffff;
    break;
  }
  reason ← int_value(lvalue, &cvalue);
  if (reason ≡ LERR_LIMIT) return false;
  else assert(¬failure_p(reason));
  if (cvalue < min ∨ cvalue > max) return false;
  *ret ← cvalue;
  return true;
}
```

273. **error_code** *assembly_encode_ALOT*(**int** *argc*, **cell** *argv*, **instruction** ∗*ret*)

```
{
  static int mask[3] ← {0, htobe32(LBC_FIRST_INTEGER), htobe32(LBC_SECOND_INTEGER)};
    /* part of opcode */
  assert(argc ≥ 0 ∧ argc ≤ 2);
  assert(argument_p(argv));
  if (A(argv)→sin ≡ fix(ARGUMENT_ERROR)) {
    argv ← fixed_value(error_id_c(A(argv)→dex));
    goto integer;
  }
  if (A(argv)→sin ≠ fix(ARGUMENT_OBJECT)) return assembly_encode_AREG(argc, argv, ret);
  assert(argc ≠ 0);
  assert(special_p(A(argv)→dex));
  if (fixed_p(A(argv)→dex)) {
    if (¬assembly_validate_integer(8, true, A(argv)→dex, &argv)) return LERR_INCOMPATIBLE;
  integer: argv ← argv & 0x ff;
    ∗ret ← mask[argc];
  }
  else {
    argv ← (((A(argv)→dex + 1)/2) & 0x 7f) | 0x 80;
    ∗ret ← 0;
  }
  ∗ret |= htobe32(argv ≪ ((2 − argc) ∗ 8));
  return LERR_NONE;
}
```

274. **error_code** *assembly_encode_AREG*(**int** *argc*, **cell** *argv*, **instruction** ∗*ret*)

```
{
  bool popping;
  assert(argc ≥ 0 ∧ argc ≤ 2);
  assert(argument_p(argv));
  popping ← A(argv)→sin ≡ fix(ARGUMENT_REGISTER_POPPING);
  assert(popping ∨ A(argv)→sin ≡ fix(ARGUMENT_REGISTER));
  assert(register_p(A(argv)→dex));
  argv ← fixed_value(register_id_c(A(argv)→dex)) | (popping ≪ 5);
  argv ≪= ((2 − argc) ∗ 8);
  ∗ret ← htobe32(argv);
  return LERR_NONE;
}
```

275. **error_code** *new_assembly_buffer*(**byte** ∗*source*, **word** *length*, **cell** ∗*ret*)

```
{
  cell dupe;
  error_code reason;
  assert(length ≤ HALF_MAX);
  orreturn (new_segment(length, 0, &dupe));
  memmove(segment_base(dupe), source, length);
  return new_assembly_segment(dupe, ret);
}
```

276. **error_code** *new_assembly_segment*(**cell** *source*, **cell** ∗*ret*)

```
{
  cell ass, statement;
  cell lconsume, llength, loffset;
  error_code reason;
  assert(segment_p(source));
  orreturn (new_int_c(segment_length_c(source), &llength));
  if (¬fixed_p(llength)) return LERR_LIMIT;
  orreturn (new_assembly_progress(&ass));
  loffset ← fix(0);
  int l ← 1;
  while (fixed_value(loffset) < segment_length_c(source)) {
    orreturn (parse_segment_to_statement(source, loffset, llength, &lconsume, &statement));
    assert(fixed_p(lconsume) ∧ lconsume ≠ fix(0));
    orreturn (assembly_append_line_m(ass, statement));
#if 0
    printf("␣␣␣␣␣␣␣%5d␣", l);
    for (int i ← fixed_value(loffset); i < fixed_value(loffset) + fixed_value(lconsume); i++)
      if (segment_base(source)[i] ≠ '\n') putchar(segment_base(source)[i]);
    printf("\r0x%4x\n",
        (void ∗)((fixed_value(array_base(ass)[ASSEMBLY_PROGRESS_NEXT_ADDRESS]) − 1) ∗ 4 + 8));
#endif
    llength ← fix(fixed_value(llength) − fixed_value(lconsume));
    loffset ← fix(fixed_value(loffset) + fixed_value(lconsume));
    l++;
  }
  return assembly_finish_m(ass, ret);
}
```

277. Evaluator.

⟨ Global variables 13 ⟩ +≡
shared cell *Evaluate_Program*;

278. ⟨ Data for initialisation 12 ⟩ +≡
#**include** `"evaluate.c"`

279. These symbols are defined in `evaluate.c`, which embeds the contents of `evaluate.la` in a C variable.

⟨ External C symbols 14 ⟩ +≡
extern shared cell *Evaluate_Program*;
extern shared char *Evaluate_Source*[ ];
extern shared long *Evaluate_Source_Length*;

280. ⟨ Initialise evaluator and other bytecode 280 ⟩ ≡
orreturn (*new_assembly_buffer*((**byte** ∗) *Evaluate_Source*, *Evaluate_Source_Length*, &*ltmp*));
orreturn (*assembly_install_m*(*ltmp*, &*ltmp*));
Evaluate_Program ← *ltmp*;
This code is used in section 126.

281. ⟨ Data for initialisation 12 ⟩ +≡
#**include** `"barbaroi.c"`

282. ⟨ External C symbols 14 ⟩ +≡
extern shared char *Barbaroi_Source*[ ];
extern shared long *Barbaroi_Source_Length*;

283. ⟨ Initialise `Lossless` procedures 283 ⟩ ≡
orreturn (*new_segment*(*Barbaroi_Source_Length*, 0, &*ltmp*));
memmove(*segment_base*(*ltmp*), *Barbaroi_Source*, *Barbaroi_Source_Length*);
General[0] ← *ltmp*; /∗ Buffer ∗/
General[1] ← *fix*(0); /∗ Starting offset ∗/
orreturn (*new_symbol_const*(`PROGRAM_EXIT`, &*ltmp*)); /∗ Locate return address. ∗/
orreturn (*vm_locate_entry*(*ltmp*, Λ, &*atmp*));
orreturn (*new_pointer*(*atmp*, &*jexit*));
orreturn (*stack_array_push*(&*Control_Link*, *jexit*));
orreturn (*new_symbol_const*(`PROGRAM_READ`, &*ltmp*)); /∗ Locate read primitive. ∗/
orreturn (*vm_locate_entry*(*ltmp*, Λ, &*Ip*));
orreturn (*interpret*( )); /∗ Read `barbaroi.ll`. ∗/
Expression ← *Accumulator*;
orreturn (*stack_array_push*(&*Control_Link*, *jexit*));
orreturn (*new_symbol_const*(`PROGRAM_EVALUATE`, &*ltmp*));
orreturn (*vm_locate_entry*(*ltmp*, Λ, &*Ip*));
orreturn (*interpret*( )); /∗ Evaluate `barbaroi.ll`. ∗/
General[0] ← *General*[1] ← *Expression* ← *Accumulator* ← `NIL`;
This code is used in section 126.

284. Threads.

285. ⟨ Type definitions 11 ⟩ +≡
```
struct osthread {
  struct osthread *next, *prev;
  heap_pun *root;
  cell ret;
  cell owner;      /* A segment (scow). */
  pthread_t tobj;
  address ip;
  bool pending;
};
typedef struct osthread osthread;
```

286. Why *Threads* and *Thread_DB*? I can't remember.

⟨ Global variables 13 ⟩ +≡
```
shared osthread *Threads ← Λ;
shared cell *Thread_DB ← Λ;
shared int Thread_DB_Length ← 0;
shared pthread_mutex_t Thread_DB_Lock;
shared pthread_barrier_t Thready;
```

287. ⟨ External C symbols 14 ⟩ +≡
```
extern shared osthread *Threads;
extern shared cell *Thread_DB;
extern shared int Thread_DB_Length;
extern shared pthread_mutex_t Thread_DB_Lock;
extern shared pthread_barrier_t Thready;
```

288. ⟨ Function declarations 19 ⟩ +≡
```
error_code init_osthread(void);
error_code init_osthread_mutex(pthread_mutex_t *, bool, bool);
```

289. ⟨ Initialise threading 289 ⟩ ≡
```
orabort(init_osthread_mutex(&Thread_DB_Lock, false, true));
if (pthread_barrier_init(&Thready, Λ, 2))
  just_abort(LERR_INTERNAL, "failed to intialise Thread Ready barrier");
orabort(alloc_mem(Λ, Thread_DB_Length * sizeof(cell), 0, (void **) &Thread_DB));
SCOW_Attributes[LSCOW_PTHREAD_T].length ← sizeof(osthread);
  /* Wrapper around pthread_t. */
SCOW_Attributes[LSCOW_PTHREAD_T].align ← sizeof(void *);
```
This code is used in section 22.

290.
```
error_code init_osthread(void)
{
  int i;
  Thread_Ready ← false;
  ⟨ (Re-)Initialise thread register pointers 131 ⟩
  Thread_Ready ← true;
  return LERR_NONE;
}
```

291. Errors: `EINVAL`, attributes are invalid, or `ENOMEM`.

```
error_code init_osthread_mutex(pthread_mutex_t *mx, bool recursive, bool robust unused)
{
  pthread_mutexattr_t mutex_attr;
  pthread_mutexattr_init(&mutex_attr);
  pthread_mutexattr_settype(&mutex_attr,
      recursive ? PTHREAD_MUTEX_RECURSIVE : PTHREAD_MUTEX_ERRORCHECK);
#ifdef pthread_mutexattr_setrobust
  if (robust) pthread_mutexattr_setrobust(&mutex_attr, PTHREAD_MUTEX_ROBUST);
#endif      /* pthread_mutexattr_setrobust */
  if (pthread_mutex_init(mx, &mutex_attr) != 0) return LERR_OOM;
  return LERR_NONE;
}
```

292. Testing.

Each unit test consists of five phases:

1. Prepare the environment.
2. Run the procedure.
3. Offer a prayer of hope to your god.
4. Validate the result.
5. Exit the test.

The test system is mostly self-contained but uses the C library for I/O in these functions: *llt_usage* & *llt_print_test* (CLI), *llt_sprintf* which builds message strings (*vsnprintf*), and the TAP API *tap_plan* & *tap_out*.

```
⟨ testless.h 292 ⟩ ≡
#ifndef LL_TESTLESS_H
#define LL_TESTLESS_H
  ⟨ Test fixture header 293 ⟩      /* Order matters. */
  ⟨ Test definitions 295 ⟩
  ⟨ Test functions 294 ⟩
#endif
```

293. The test fixture is kept in its own section to avoid visual confusion within the previous one. Every test unit in every test suite begins with this header.

Because the functions in `testless.c` don't know the final size of the fixture until after they have been linked with the test script this *llt_fixture_fetch* macro advances the test suite pointer forward by the correct size.

```
  format llt_forward llt_thunk      /* A C type-mangling hack. */
#define llt_fixture_fetch(O, I) ((llt_header *)(((char *)(O)) + Test_Fixture_Size * (I)))
#define llt_fixture_grow(O, L, D)
        (alloc_mem((O), Test_Fixture_Size * ((L) + (D)), 0, (void **) &(O)))
```

```
⟨ Test fixture header 293 ⟩ ≡
#define LLT_FIXTURE_HEADER
        char *name;      /* Name of this test unit. */
        int id;      /* Numeric identifier for listing. */
        int total;      /* Total number ot units in the suite. */
        void **leaks;      /* Array of allocated memory. */
        llt_forward prepare;      /* Preparation function. */
        llt_forward run;      /* Carrying out the test. */
        llt_forward validate;      /* Verifying the result. */
        llt_forward clean;      /* Cleaning up after. */
        int progress;      /* The unit's progress through the harness. */
        int tap;      /* The ID of the "current" test tap. */
        int taps;      /* The number of taps in this unit. */
        int tap_start;      /* The tap ID of this unit's first tap. */
        bool perform;      /* Whether to carry out this unit. */
        error_code expect;      /* The error expected to occur. */
        error_code reason;      /* The error that did occur. */
        cell res;      /* The result if there was no error. */
        void *resp;      /* The same if a C pointer is expected. */
        cell meta;      /* Misc. data saved by the validator. */
        int ok;      /* The final result of this unit. */
```

This code is used in section 292.

294. ⟨ Test functions 294 ⟩ ≡

```
void llt_fixture__init_common(llt_header *, int, llt_thunk, llt_thunk, llt_thunk, llt_thunk);
error_code llt_list_suite(llt_header *);
error_code llt_load_tests(bool, llt_header **);
error_code llt_main(int, char **, bool);
int llt_perform_test(int *, llt_header *);
void llt_print_test(llt_header *);
error_code llt_run_suite(llt_header *);
error_code llt_skip_test(int *, llt_header *, char *);
error_code llt_usage(char *, bool);
```

See also sections 310, 321, and 323.

This code is used in section 292.

295. The header contains four **llt_forward** objects which should actually be **llt_thunk** to avoid problems caused by the order of these definitions.

⟨ Test definitions 295 ⟩ ≡

```
struct llt_header;
typedef error_code (*llt_forward)(void *);
typedef struct {
  LLT_FIXTURE_HEADER
} llt_header;
typedef error_code (*llt_initialise)(llt_header *, int *, bool, llt_header **);
typedef int (*llt_thunk)(llt_header *);
```

See also sections 309 and 312.

This code is used in section 292.

296. ⟨ Test common preamble 296 ⟩ ≡

```
#include <assert.h>
#include <errno.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "lossless.h"
#include "testless.h"
```

This code is used in sections 297, 328, 337, 345, 353, and 369.

297. `testless.c` compiles to an archive containing all of the shared `llt_` and `tap_` functions.

⟨ `testless.c` 297 ⟩ ≡

```
#include <err.h>
#include <getopt.h>
extern int optind;
⟨ Test common preamble 296 ⟩
extern llt_initialise Test_Suite[];
extern int Test_Fixture_Size;
```

See also sections 298, 301, 302, 303, 304, 305, 306, 307, 308, 311, 316, 317, 318, 319, 320, 322, 324, 325, and 327.

298. The shared *llt_main* function presents each test script with common `-h` (help) & `-l` (list) options and a simple way of specifying specific test units to run.

```
#define LLT_DO_TESTS 0
#define LLT_LIST_TESTS 1
⟨ testless.c 297 ⟩ +≡
  error_code llt_main(int argc, char **argv, bool init)
  {
    int act, i, opt;
    char *tail;
    unsigned long value;
    llt_header *suite;
    error_code reason;
    assert(argc ≥ 1);
    act ← LLT_DO_TESTS;
    if (argc > 1) {⟨ Parse command line options 299 ⟩}
    orreturn (init_mem());
    if (init) orreturn (init_vm());
    orreturn (llt_load_tests(act ≡ LLT_DO_TESTS, &suite));
    if (argc ≠ 1) {⟨ Parse a test run specification from the command line 300 ⟩}
    if (act ≡ LLT_DO_TESTS) return llt_run_suite(suite);
    else return llt_list_suite(suite);
  }
```

299. Long options are also supported because why not?

```
⟨ Parse command line options 299 ⟩ ≡
  static struct option llt_common_options[] ← {
      {"help", no_argument, Λ, 'h'},
      {"list", no_argument, Λ, 'l'},
      {Λ, 0, Λ, 0}      /* What's all this then? */
  };
  while ((opt ← getopt_long(argc, argv, "lh", llt_common_options, Λ)) ≠ −1) {
    switch (opt) {
    case 'l': act ← LLT_LIST_TESTS; break;
    case 'h': return llt_usage(argv[0], true);
    default: return llt_usage(argv[0], false);
    }
  }
  argc −= optind − 1;
```

This code is used in section 298.

300. The script with no arguments runs all test units. The script can be restricted to specific units can by identifying them on the command line. The identifier is obtained with the `-l|--list` option.

```
⟨ Parse a test run specification from the command line 300 ⟩ ≡
  for (i ← 0; i < suite→total; i++) llt_fixture_fetch(suite, i)→perform ← false;
  for (i ← 1; i < argc; i++) {
    if (argv[(optind − 1) + i][0] < '1' ∨ argv[(optind − 1) + i][0] > '9') {
  invalid_id:
      errc(1, EINVAL, "Invalid test id '%s'; maximum is %d", argv[(optind − 1) + i], suite→total);
    }
    else {
      errno ← 0;
      value ← strtoul(argv[(optind − 1) + i], &tail, 10);
      if (*tail ≠ '\0' ∨ errno ≡ ERANGE ∨ value > INT_MAX) goto invalid_id;
      if ((int) value > suite→total) goto invalid_id;
      if (llt_fixture_fetch(suite, value − 1)→perform) warn("Duplicate test id %lud", value);
      llt_fixture_fetch(suite, value − 1)→perform ← true;
    }
  }
```

This code is used in section 298.

301. TODO: Adjust tabbing for the length of *name*.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_usage(char *name, bool ok)
  {
    printf("Usage:\n");
    printf("\t%s\t\t\tRun all tests.\n", name);
    printf("\t%s -l | --list\tList all test cases as an s-expression.\n", name);
    printf("\t%s id...\t\tRun the specified tests.\n", name);
    printf("\t%s -h | --help\tDisplay this help and exit.\n", name);
    return ok ? LERR_NONE : LERR_USER;
  }
```

302. When the harness first starts it initialises the fixtures for all the test cases by calling each function mentioned in *Test_Cases*. Those which have an expensive initialisation routine can skip it if the script is only listing the test unit titles.

Each test unit initialiser will enlarge the buffer holding the suite as much as it needse and increment the *num_tests* variable by the number of individual test units which were added. The number of taps in each unit is then counted and a running total of each unit's starting tap and the total number of taps is kept.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_load_tests(bool full, llt_header **ret)
  {
    llt_header *r;
    llt_initialise *tc;
    int before, i, num_tests, tap;
    error_code reason;
    orreturn (alloc_mem(Λ, Test_Fixture_Size, 0, (void **) &r));
    num_tests ← 0;
    tap ← 1;
    tc ← Test_Suite;
    while (*tc ≠ Λ) {
      before ← num_tests;
      orreturn ((*tc)(r, &num_tests, full, &r));
      for (i ← before; i < num_tests; i++) {
        llt_fixture_fetch(r, i)→tap_start ← tap;
        tap += llt_fixture_fetch(r, i)→taps;
      }
      tc++;
    }
    for (i ← 0; i < num_tests; i++) llt_fixture_fetch(r, i)→total ← num_tests;
    *ret ← r;
    return LERR_NONE;
  }
```

303. Listing the test units. Specifying the tests to list is probably pointless but easier than complaining about an erroneous specification.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_list_suite(llt_header *suite)
  {
    int i;
    for (i ← 0; i < suite→total; i++)
      if (llt_fixture_fetch(suite, i)→perform) llt_print_test(llt_fixture_fetch(suite, i));
    return LERR_NONE;
  }
```

304.

```
⟨ testless.c 297 ⟩ +≡
  void llt_print_test(llt_header *o)
  {
    char *p;
    printf("(%d␣|", o→id + 1);
    for (p ← o→name; *p; p++) {
      if (*p ≡ '|' ∨ *p ≡ '#') putchar('#');
      putchar(*p);
    }
    printf("|)\n");
  }
```

305. `#define LLT_PROGRESS_INIT 0`
`#define LLT_PROGRESS_PREPARE 1`
`#define LLT_PROGRESS_RUN 2`
`#define LLT_PROGRESS_VALIDATE 3`
`#define LLT_PROGRESS_CLEAN 4`
`#define LLT_PROGRESS_SKIP 5`

⟨ testless.c 297 ⟩ +≡

```
void llt_fixture__init_common(llt_header *fixture, int id, llt_thunk prepare, llt_thunk
        run, llt_thunk validate, llt_thunk clean)
{
  fixture→name ← "";
  fixture→id ← id;
  fixture→total ← −1;
  fixture→leaks ← Λ;
  fixture→perform ← true;
  fixture→prepare ← (llt_forward) prepare;
  fixture→run ← (llt_forward) run;
  fixture→validate ← (llt_forward) validate;
  fixture→clean ← (llt_forward) clean;
  fixture→progress ← LLT_PROGRESS_INIT;
  fixture→taps ← 1;
  fixture→tap ← fixture→tap_start ← 0;
  fixture→ok ← false;
  fixture→res ← NIL;
  fixture→resp ← Λ;
  fixture→reason ← fixture→expect ← LERR_NONE;
}
```

306. To run a suite each unit in turn is passed through *llt_perform_test* or *llt_skip_test* if it wasn't included in a command line specification. If the number of taps a unit claimed and the number it performed differ then a warning will be emitted and the tap stream that gets output will be broken (test IDs will clash).

TODO: Should I not reset the tap start ID each time and instead vigorously enforce incrementing the output tap id?

```
#define LLT_RUN_ABORT -1
#define LLT_RUN_FAIL 1
#define LLT_RUN_CONTINUE 0
#define LLT_RUN_PANIC 2
#define orfail(E)
        if (failure_p(E)) return LLT_RUN_FAIL;
⟨ testless.c 297 ⟩ +≡
  error_code llt_run_suite(llt_header *suite)
  {
    int i, run, t;
    error_code reason;
    t ← 0;
    for (i ← 0; i < suite→total; i++) t += llt_fixture_fetch(suite, i)→taps;
    tap_plan(t);
    run ← 0;
    t ← 1;
    for (i ← 0; run ≠ LLT_RUN_ABORT ∧ i < suite→total; i++) {
      if (llt_fixture_fetch(suite, i)→perform) {
        t ← llt_fixture_fetch(suite, i)→tap_start;
        if (run ≡ LLT_RUN_CONTINUE) run ← llt_perform_test(&t, llt_fixture_fetch(suite, i));
        if (run ≠ LLT_RUN_CONTINUE) warnx("Unknown test failure");
        else if (t ≠ llt_fixture_fetch(suite, i)→tap_start + llt_fixture_fetch(suite, i)→taps)
          warnx("Test tap mismatch: %d != %d", t, llt_fixture_fetch(suite,
              i)→tap_start + llt_fixture_fetch(suite, i)→taps);
      }
      else {
        orreturn (llt_skip_test(&t, llt_fixture_fetch(suite, i), "command line"));
      }
    }
    return LERR_NONE;
  }
```

307. If the test is being skipped then the appropriate number of taps are printed in place of running the unit.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_skip_test(int *tap, llt_header *testcase, char *excuse)
  {
    int i;
    char *msg;
    error_code reason;
    orreturn (llt_sprintf(testcase, &msg, "--- # SKIP %s", excuse));
    testcase→tap ← *tap;
    testcase→progress ← LLT_PROGRESS_SKIP;
    for (i ← 0; i < testcase→taps; i++) tap_ok(testcase, msg, true, NIL);
    *tap ← testcase→tap;
    return LERR_NONE;
  }
```

308. Actually perform a test. Call the preparatation routine if there is one then carry out the desiered action and validate the result. The protection around *Test_Memory* is necessary but the paranoia behind saving and restoring the prior value is probably not — if *Test_Memory→active* was previously true then the clean up routine should certainly set it stright back to false again.

Some care is taken to protect against errors in the four test stages themselves but really they shouldn't be necessary if the test complexity is kept under control.

There is no need to cast the function pointers to a real **llt_thunk** because the only difference from **llt_forward** is that the latter uses **void** ∗ for its pointer argument types.

```
⟨ testless.c 297 ⟩ +≡
  int llt_perform_test(int *tap, llt_header *testcase)
  {
    bool allocating;
    int n, r;
    if (testcase→progress ≠ LLT_PROGRESS_INIT) return LLT_RUN_ABORT;
    n ← testcase→prepare ≡ Λ ? LLT_RUN_CONTINUE : testcase→prepare(testcase);
    if (n ≠ LLT_RUN_CONTINUE) return n;
    testcase→progress ← LLT_PROGRESS_PREPARE;
    if (testcase→run ≡ Λ) return LLT_RUN_ABORT;
    n ← testcase→run(testcase);
    if (n ≠ LLT_RUN_CONTINUE) return n;
    testcase→progress ← LLT_PROGRESS_RUN;
    if (testcase→validate ≡ Λ) r ← LLT_RUN_ABORT;
    else {
      if (Test_Memory ≠ Λ) {
        allocating ← Test_Memory→active;
        Test_Memory→active ← false;
      }
      testcase→tap ← *tap;
      r ← testcase→validate(testcase);
      *tap ← testcase→tap;
      if (Test_Memory ≠ Λ) Test_Memory→active ← allocating;
    }
    if (r < LLT_RUN_PANIC) n ← testcase→clean ≡ Λ ? LLT_RUN_CONTINUE : testcase→clean(testcase);
    if (r ≠ LLT_RUN_CONTINUE) return r;
    else return n;
  }
```

309. Testing memory allocation. Those tests which need to mock the core memory allocator point *Test_Memory* to an instance of this object (eg. created in *main* before calling *llt_main*) with pointers to alternative allocation and release functions.

```
⟨ Test definitions 295 ⟩ +≡
  typedef struct {
    bool active;      /* Whether alloc_mem should revert to these. */
    bool available;     /* Whether the false allocation should succeed. */
    error_code(*alloc)(void *, size_t, size_t, void **);
    error_code(*free)(void *);
  } llt_allocation;
```

310.
```
⟨ Test functions 294 ⟩ +≡
  error_code llt_appendf(llt_header *, char *, char **, char *, ...);
  void llt_free(llt_header *);
  error_code llt_leak(llt_header *, size_t, void **);
  error_code llt_sprintf(llt_header *, char **, char *, ...);
  error_code llt_vsprintf(llt_header *, int, char **, char *, va_list);
```

311.
```
⟨ testless.c 297 ⟩ +≡
  shared llt_allocation *Test_Memory ← Λ;
```

312.
```
⟨ Test definitions 295 ⟩ +≡
  extern shared llt_allocation *Test_Memory;
```

313. These sections are responsible for diverting allocation and deallocation to the alternatives. The code is proteced by preprocessor macros so it will not be included in a non-test binary.

```
⟨ Testing memory allocator 313 ⟩ ≡
  if (Test_Memory ≠ Λ ∧ Test_Memory→active) return Test_Memory→alloc(old, length, align, ret);
```

This code is used in section 20.

314.
```
⟨ Testing memory deallocator 314 ⟩ ≡
  if (Test_Memory ≠ Λ ∧ Test_Memory→active) return Test_Memory→free(o);
```

This code is used in section 21.

315. Allocating memory while testing.

316. Ordinarily test scripts are expected to be run once and immediately quit and so expect to be able to allocate memory with wild abandon without caring to clean it up. In case there is a desire to have scripts remain in memory the allocations made by/for each fixture are kept in an array of pointers in the *leak* attribute, itself made from such an allocation the first time one is requested.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_leak(llt_header *fixture, size_t length, void **ret)
  {
    int lid;
    error_code reason;
    if (fixture→leaks ≡ Λ) lid ← 1;
    else lid ← (word) fixture→leaks[0];
    orreturn (alloc_mem(fixture→leaks, sizeof(void *) * (lid + 1), 0, (void **) &fixture→leaks));
    fixture→leaks[0] ← (void *)(intptr_t) lid;
    fixture→leaks[lid] ← Λ;
    orreturn (alloc_mem(Λ, length, 0, ret));
    fixture→leaks[lid] ← *ret;
    fixture→leaks[0] ← (void *)(intptr_t)(lid + 1);
    return LERR_NONE;
  }
```

317. Although nothing uses it the *llt_free* function will clean up a fixture's memory allocations. Ignores error returns but *free* doesn't fail anyway.

```
⟨ testless.c 297 ⟩ +≡
  void llt_free(llt_header *fixture)
  {
    int i;
    if (fixture→leaks ≠ Λ) {
      for (i ← 0; i < (long) fixture→leaks[0]; i++)
        if (fixture→leaks[i] ≠ Λ) free_mem(fixture→leaks[i]);
      free_mem(fixture→leaks);
    }
    fixture→leaks ← Λ;
  }
```

318. The main consumer of *llt_leak* is this wrapper around `printf` and its two users *llt_sprintf* and *llt_appendf*.

```
⟨ testless.c 297 ⟩ +≡
  error_code llt_vsprintf(llt_header *fixture, int length, char **ret, char *fmt, va_list args)
  {
    char *buf;
    error_code reason;
    orreturn (llt_leak(fixture, length + 1, (void **) &buf));
    if (vsnprintf(buf, length + 1, fmt, args) < 0) return LERR_INTERNAL;
    *ret ← buf;
    return LERR_NONE;
  }
```

319. ⟨ `testless.c` 297 ⟩ +≡

```
error_code llt_sprintf(llt_header *fixture, char **ret, char *fmt, ...)
{
  int length;
  va_list args;
  error_code reason;
  va_start(args, fmt);
  length ← vsnprintf(Λ, 0, fmt, args);
  va_end(args);
  va_start(args, fmt);
  reason ← llt_vsprintf(fixture, length, ret, fmt, args);
  va_end(args);
  return reason;
}
```

320. ⟨ `testless.c` 297 ⟩ +≡

```
error_code llt_appendf(llt_header *fixture, char *prior, char **ret, char *fmt, ...)
{
  char *append;
  int length;
  va_list args;
  error_code reason;
  va_start(args, fmt);
  length ← vsnprintf(Λ, 0, fmt, args);
  va_end(args);
  va_start(args, fmt);
  reason ← llt_vsprintf(fixture, length, &append, fmt, args);
  va_end(args);
  if (failure_p(reason)) return reason;
  return llt_sprintf(fixture, ret, "%s%s", prior, append);
}
```

321. Objects Under Test.

⟨ Test functions 294 ⟩ +≡
```
bool llt_out_match_p(cell, cell);
```

322. ⟨ testless.c 297 ⟩ +≡
```
bool llt_out_match_p(cell got, cell want)
{
  if (special_p(want) ∨ symbol_p(want)) return got ≡ want;
  if (special_p(got) ∨ symbol_p(got)) return false;
  if (T(got) ≠ T(want)) return false;
  if (pair_p(want)) return pair_p(got) ∧ llt_out_match_p(A(got)↣sin,
      A(want)↣sin) ∧ llt_out_match_p(A(got)↣dex, A(want)↣dex);
  if (pointer_p(want)) {
    if (¬pointer_p(got)) return false;
    if (pointer(got) ≠ pointer(want)) return false;
    return llt_out_match_p(pointer_datum(got), pointer_datum(want));
  }
  if (closure_p(want)) {
    if (¬closure_p(got)) return false;
    if (¬llt_out_match_p(array_base(got)[CLOSURE_ADDRESS], array_base(want)[CLOSURE_ADDRESS]))
      return false;
    if (¬llt_out_match_p(array_base(got)[CLOSURE_ENVIRONMENT],
        array_base(want)[CLOSURE_ENVIRONMENT])) return false;
    if (¬llt_out_match_p(array_base(got)[CLOSURE_SIGNATURE],
        array_base(want)[CLOSURE_SIGNATURE])) return false;
    return llt_out_match_p(array_base(got)[CLOSURE_BODY], array_base(want)[CLOSURE_BODY]);
  }
  if (environment_p(want)) {
    return got ≡ want;      /* TBD */
  }
  assert(¬"unsupported␣match␣type");
}
```

323. TAP. Tap routines to implement a rudimentary stream of test results in the *Test Anything Protocol*[1]

⟨ Test functions 294 ⟩ +≡
```
bool tap_ok(llt_header *, char *, bool, cell);
void tap_out(char *, ...);
void tap_plan(int);
```

324. ⟨ testless.c 297 ⟩ +≡
```
void tap_plan(int length)
{
  assert(length ≥ 1);
  printf("1..%d\n", length);
}
```

325. #define *tap_fail*(*C*, *T*, *M*) *tap_ok*((*C*), (*T*), *false*, (*M*))
#define *tap_pass*(*C*, *T*, *M*) *tap_ok*((*C*), (*T*), *true*, (*M*))

⟨ testless.c 297 ⟩ +≡
```
bool tap_ok(llt_header *testcase, char *title, bool result, cell meta)
{
  assert(testcase→progress ≡ LLT_PROGRESS_RUN ∨ testcase→progress ≡ LLT_PROGRESS_SKIP);
  testcase→meta ← meta;
  testcase→ok ← result ? LLT_RUN_CONTINUE : LLT_RUN_FAIL;
  if (result) tap_out("ok");
  else tap_out("not␣ok");
  tap_out("␣%d␣-␣", testcase→tap++);
  if (testcase→name ≠ Λ) tap_out("%s:␣", testcase→name);
  tap_out("%s\n", title);
  return result;
}
```

326. Like *tap_ok* but the unit result so far must be a success.

#define *tap_and*(*C*, *T*, *R*, *M*)
((*th*→*ok* ≡ LLT_RUN_CONTINUE) ? *tap_ok*((*C*), (*T*), (*R*), (*M*)) : *tap_fail*((*C*), (*T*), (*M*)))

327. ⟨ testless.c 297 ⟩ +≡
```
void tap_out(char *fmt, ...)
{
  va_list args;
  va_start(args, fmt);
  vprintf(fmt, args);
  va_end(args);
}
```

[1] http://testanything.org/

328. Test suite tests.

```
⟨ t/insanity.c 328 ⟩ ≡
  ⟨ Test common preamble 296 ⟩
  int main(int argc, char **argv)
  { return llt_main(argc, argv, true); }
```

See also sections 329, 330, 331, 332, 333, 334, 335, and 336.

329. Most test scripts will use a customised **llt_fixture** object (which needn't be called that). The full size must be put in *Test_Fixture_Size*.

```
⟨ t/insanity.c 328 ⟩ +≡
  typedef struct {
    LLT_FIXTURE_HEADER
    instruction *program;
  } llt_fixture;
  int Test_Fixture_Size ← sizeof(llt_fixture);
```

330. This suite consists of a single unit initialised by *llt_Sanity__Nothing*.

```
⟨ t/insanity.c 328 ⟩ +≡
  error_code llt_Sanity__Nothing(llt_header *, int *, bool, llt_header **);
  error_code llt_Sanity__Halt(llt_header *, int *, bool, llt_header **);
  int llt_Sanity__prepare(llt_header *);
  int llt_Sanity__noop(llt_header *);
  int llt_Sanity__interpret(llt_header *);
  int llt_Sanity__validate(llt_header *);
  llt_initialise Test_Suite[] ← {llt_Sanity__Nothing, llt_Sanity__Halt, Λ};
```

331. The Nothing test unit has a single test case in it which requires no preparation or cleanup.

```
⟨ t/insanity.c 328 ⟩ +≡
  error_code llt_Sanity__Nothing(llt_header *suite, int *count, bool full unused, llt_header **ret)
  {
    llt_fixture *tc;
    error_code reason;
    orreturn (llt_fixture_grow(suite, *count, 1));
    tc ← (llt_fixture *) suite;
    llt_fixture__init_common((llt_header *)(tc + *count), *count, Λ, llt_Sanity__noop,
        llt_Sanity__validate, Λ);
    tc[*count].name ← "do nothing";
    tc[*count].program ← Λ;
    tc[*count].taps ← 2;
    (*count)++;
    *ret ← suite;
    return LERR_NONE;
  }
```

332. ⟨ t/insanity.c 328 ⟩ +≡

```
error_code llt_Sanity__Halt(llt_header *suite, int *count, bool full unused, llt_header **ret)
{
  llt_fixture *tc;
  error_code reason;
  instruction *ins;
  orreturn (llt_fixture_grow(suite, *count, 1));
  tc ← (llt_fixture *) suite;
  llt_fixture__init_common((llt_header *)(tc + *count), *count, llt_Sanity__prepare,
      llt_Sanity__interpret, llt_Sanity__validate, Λ);
  tc[*count].name ← "HALT";
  orreturn (llt_leak(suite, sizeof(instruction), (void **) &ins));
  ins[0] ← htobe32(OP_HALT ≪ 24);
  tc[*count].program ← ins;
  tc[*count].taps ← 2;
  (*count)++;
  *ret ← suite;
  return LERR_NONE;
}
```

333. ⟨ t/insanity.c 328 ⟩ +≡

```
int llt_Sanity__prepare(llt_header *testcase_ptr unused)
{
  llt_fixture *tc ← (llt_fixture *) testcase_ptr;
  Ip ← (address) tc→program;
  return LLT_RUN_CONTINUE;
}
```

334. Nothing in `Lossless` is tested by this test although the parts used by the test harness are exercised.

⟨ t/insanity.c 328 ⟩ +≡

```
int llt_Sanity__noop(llt_header *testcase_ptr unused)
{
  return LLT_RUN_CONTINUE;
}
```

335. ⟨ t/insanity.c 328 ⟩ +≡

```
int llt_Sanity__interpret(llt_header *testcase_ptr unused)
{
  interpret( );
  return LLT_RUN_CONTINUE;
}
```

336. ⟨ t/insanity.c 328 ⟩ +≡

```
int llt_Sanity__validate(llt_header *testcase_ptr)
{
  tap_ok(testcase_ptr, "done", true, NIL);
  tap_ok(testcase_ptr, "VPU␣is␣not␣trapped", ¬failure_p(Trapped), NIL);
  return testcase_ptr→ok;
}
```

337. Hashtable tests.

```
#define LLT_HASHTABLE_SEED (HASHTABLE_TINY + 1)
#define LLT_HASHTABLE_FACTOR 24
```

⟨ t/hashtable.c 337 ⟩ ≡

```
⟨ Test common preamble 296 ⟩
int main(int argc, char **argv)
{ return llt_main(argc, argv, false); }
typedef struct {
  LLT_FIXTURE_HEADER
  word new_length;      /* How long a hashtable to create. */
  word pre_length;      /* The pre-test size actually created. */
  cell test_table;      /* Prepared hashtable to run a test against. */
  cell test_datum;      /* Combined key/value to test with. */
  bool test_replace;    /* Replace flag to hashtable_save_m. */
  cell seed[LLT_HASHTABLE_SEED];      /* To insert prior to testing. */
} llt_fixture;
int Test_Fixture_Size ← sizeof(llt_fixture);
error_code llt_Hashtable__New(llt_header *, int *, bool, llt_header **);
error_code llt_Hashtable__Save(llt_header *, int *, bool, llt_header **);
llt_initialise Test_Suite[] ← {llt_Hashtable__New, llt_Hashtable__Save, Λ};
```

See also sections 338, 339, 340, 341, 342, 343, and 344.

338. To test hashtable creation a new hashtable of various sizes is created and probed.
TODO: Remove boilerplate from `*__New` functions.

```
⟨ t/hashtable.c 337 ⟩ +≡
  int llt_Hashtable__New_run(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    tc→reason ← new_hashtable(tc→new_length, &tc→res);
    return LLT_RUN_CONTINUE;
  }
  int llt_Hashtable__New_validate(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    char *title;
    error_code reason;
    tap_ok(th, "success", ¬failure_p(tc→reason), NIL);
    tap_and(th, "hashtable?", hashtable_p(tc→res), NIL);
    if (tc→new_length ≡ 0)
      tap_and(th, "length␣is␣0", hashtable_length_c(tc→res) ≡ 0, NIL);
    else
      tap_and(th, "length␣is␣a␣power␣of␣2",
          hashtable_length_c(tc→res) ≡ (1 ≪ high_bit(hashtable_length_c(tc→res))), NIL);
    orassert(llt_sprintf(th, &title, "available␣slots␣>=␣%d", tc→new_length));
    tap_and(th, title, hashtable_free_c(tc→res) ≥ tc→new_length, NIL);
    return LLT_RUN_CONTINUE;
  }
  error_code llt_Hashtable__New(llt_header *suite, int *count, bool fullunused, llt_header **ret)
  {
    llt_fixture *tc;
    llt_header *th;
    error_code reason;
    int i;
    orreturn (llt_fixture_grow(suite, *count, HASHTABLE_TINY * 2 + 1));
    tc ← (llt_fixture *) suite;
    for (i ← 0; i < HASHTABLE_TINY * 2 + 1; i++) {
      tc ← ((llt_fixture *) suite) + *count + i;
      th ← (llt_header *) tc;
      llt_fixture__init_common(th, *count + i, Λ, llt_Hashtable__New_run, llt_Hashtable__New_validate, Λ);
      orreturn (llt_sprintf(th, &tc→name, "new␣hashtable,␣length␣%d", i));
      tc→taps ← 4;
      tc→new_length ← i;
    }
    (*count) += HASHTABLE_TINY * 2 + 1;
    *ret ← suite;
    return LERR_NONE;
  }
```

339. ⟨ t/hashtable.c 337 ⟩ +≡

```
  error_code llt_Hashtable__datumfn(half idx, cell seed, cell *ret)
  {
    cell label;
    error_code reason;
    orreturn (int_to_symbol(fix(idx), &label));
    return cons(label, seed, ret);
  }
```

340. Prepare tests which save into a hashtable by creating a hashtable and preseeding it.

```
⟨ t/hashtable.c 337 ⟩ +≡
  int llt_Hashtable__Save_prepare(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    cell tmp;
    int i, j;
    orfail(new_hashtable(tc→new_length, &tc→test_table));
    tc→pre_length ← hashtable_length_c(tc→test_table);
    for (i ← 0; i < LLT_HASHTABLE_SEED; i++) {
      if (defined_p(tc→seed[i])) {
        j ← i * LLT_HASHTABLE_FACTOR;
        orfail(llt_Hashtable__datumfn(j, tc→seed[i], &tmp));
        orfail(hashtable_save_m(tc→test_table, tmp, false));
      }
    }
    return LLT_RUN_CONTINUE;
  }
```

341.
```
⟨ t/hashtable.c 337 ⟩ +≡
  int llt_Hashtable__Save_run(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    tc→reason ← hashtable_save_m(tc→test_table, tc→test_datum, tc→test_replace);
    return LLT_RUN_CONTINUE;
  }
```

342. When this routine was first written the key was passed to the hashtable routines rather than being discovered within them which allowed a simple scanning mechanism; the variable *hack* allows the hashtable to be scanned in a similar manner.

```
⟨ t/hashtable.c 337 ⟩ +≡
  int llt_Hashtable__Save_validate_success(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    bool scanned;
    cell hack, found;
    int i, j;
    tap_ok(th, "success", ¬failure_p(tc→reason), fix(tc→reason));
    orfail(hashtable_search(tc→test_table, A(tc→test_datum)→sin, &found));
    tap_ok(th, "key␣is␣found", defined_p(found), found);
    tap_and(th, "saved␣datum␣is␣returned", found ≡ tc→test_datum, found);
    scanned ← true;
    for (i ← 0; scanned ∧ i < LLT_HASHTABLE_SEED; i++) {
      if (defined_p(tc→seed[i])) {
        j ← i * LLT_HASHTABLE_FACTOR;
        if (failure_p(llt_Hashtable__datumfn(j, NIL, &hack))) return LLT_RUN_ABORT;
        if (failure_p(hashtable_search(tc→test_table, A(hack)→sin, &found))) scanned ← false;
        else if (¬tc→test_replace) scanned ← pair_p(found) ∧ A(found)→dex ≡ tc→seed[i];
      }
    }
    tap_ok(th, "other␣entries␣remain␣unchanged", scanned, NIL);
    return LLT_RUN_CONTINUE;
  }
```

343. ⟨ t/hashtable.c 337 ⟩ +≡

```
int llt_Hashtable__Save_validate_failure(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  cell found;
  tap_ok(th, "fails", tc→reason ≡ tc→expect, NIL);
  orfail(hashtable_search(tc→test_table, A(tc→test_datum)→sin, &found));
  tap_ok(th, "nothing␣is␣saved", found ≠ tc→test_datum, found);
  return LLT_RUN_CONTINUE;
}
```

344. ⟨ t/hashtable.c 337 ⟩ +≡

```
error_code llt_Hashtable__Save(llt_header *suite, int *count, bool full unused, llt_header **ret)
{
  llt_fixture *tc;
  llt_header *th;
  error_code reason;
  int i, j;
  orreturn (llt_fixture_grow(suite, *count, 8));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < 8; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count + i, llt_Hashtable__Save_prepare, llt_Hashtable__Save_run,
        llt_Hashtable__Save_validate_success, Λ);
    tc→taps ← 4;
    tc→new_length ← HASHTABLE_TINY − 1;
    tc→test_replace ← false;
    for (j ← 0; j < LLT_HASHTABLE_SEED; j++) tc→seed[j] ← UNDEFINED;
    orfail(llt_Hashtable__datumfn(2 * LLT_HASHTABLE_FACTOR, fix(−1), &tc→test_datum));
  }
  tc ← ((llt_fixture *) suite) + *count;
  tc[0].name ← "save␣into␣hashtable,␣length␣0";
  tc[0].new_length ← 0;
  tc[1].name ← "save␣into␣hashtable,␣empty";
  tc[2].name ← "save␣into␣hashtable,␣seeded";
  for (i ← 1; i < LLT_HASHTABLE_SEED; i += 2) tc[2].seed[i] ← fix(i * 2);
  tc[3].name ← "save␣into␣hashtable,␣replacing";
  for (i ← 0; i < LLT_HASHTABLE_SEED; i += 2) tc[3].seed[i] ← fix(i * 2);
  tc[3].test_replace ← true;
  tc[4].name ← "insert␣in␣full␣hashtable";
  tc[5].name ← "replace␣in␣full␣hashtable";
  assert(LLT_HASHTABLE_SEED > HASHTABLE_TINY);
  tc[4].seed[0] ← tc[5].seed[0] ← fix(0);
  tc[4].seed[1] ← tc[5].seed[1] ← fix(2);
  for (i ← 3; i < HASHTABLE_TINY − 1; i++) tc[4].seed[i] ← tc[5].seed[i] ← fix(i * 2);
  tc[4].seed[i] ← fix(i * 2);
  tc[5].seed[2] ← fix(4);
  tc[5].test_replace ← true;
  tc[6].name ← "insert␣in␣hashtable,␣conflicting";
  tc[6].seed[2] ← fix(4);
  tc[6].validate ← (llt_forward) llt_Hashtable__Save_validate_failure, tc[6].expect ← LERR_EXISTS;
  tc[7].name ← "replace␣in␣hashtable,␣missing";
  tc[7].validate ← (llt_forward) llt_Hashtable__Save_validate_failure, tc[7].expect ← LERR_MISSING;
  tc[7].test_replace ← true;
  tc[6].taps ← tc[7].taps ← 2;
  (*count) += 8;
  *ret ← suite;
  return LERR_NONE;
}
```

345. Evaluator tests.

```
⟨ t/evaluator.c 345 ⟩ ≡
  ⟨ Test common preamble 296 ⟩
  int main(int argc, char **argv)
  { return llt_main(argc, argv, true); }
  typedef struct {
    LLT_FIXTURE_HEADER
    cell expression;
    cell want;
    cell save_environment;
    half interpret_limit;
  } llt_fixture;
  int Test_Fixture_Size ← sizeof(llt_fixture);
  error_code llt_Evaluator__Immediate(llt_header *, int *, bool, llt_header **);
  error_code llt_Evaluator__Simple(llt_header *, int *, bool, llt_header **);
  int llt_Evaluator__prepare(llt_header *);
  int llt_Evaluator__run(llt_header *);
  int llt_Evaluator__validate(llt_header *);
  llt_initialise Test_Suite[] ← {llt_Evaluator__Immediate, llt_Evaluator__Simple, Λ};
```

See also sections 346, 347, 348, 349, 350, 351, and 352.

346.

```
⟨ t/evaluator.c 345 ⟩ +≡
  int llt_Evaluator__prepare(llt_header *th)
  {
    llt_fixture *tc ← (llt_fixture *) th;
    address fin;
    cell extended, label, tmp;
    error_code reason;
    ortrap (new_symbol_const(LLT_LOOKUP_MISSING, &label));
    reason ← env_search(Environment, label, &tmp);
    if (failure_p(reason) ∧ reason ≠ LERR_MISSING) {
      printf("#␣"LLT_LOOKUP_MISSING"␣is␣bound.\n");
      goto Trap;
    }
    tc→save_environment ← Environment;
    ortrap (new_env(Environment, &extended));
    ortrap (new_symbol_const(LLT_LOOKUP_PRESENT, &label));
    ortrap (new_symbol_const(LLT_LOOKUP_CORRECT, &tmp));
    ortrap (env_save_m(extended, label, tmp, false));
    Interpret_Count ← 0;
    Interpret_Limit ← tc→interpret_limit;
    ortrap (new_symbol_const(PROGRAM_EVALUATE, &label));
    ortrap (vm_locate_entry(label, Λ, &Ip));
    ortrap (new_symbol_const(PROGRAM_EXIT, &label));
    ortrap (vm_locate_entry(label, Λ, &fin));
    ortrap (new_pointer(fin, &label));
    ortrap (stack_array_push(&Control_Link, label));
    Environment ← extended;
    Expression ← tc→expression;
    return LLT_RUN_CONTINUE;
  Trap: return LLT_RUN_FAIL;
  }
```

347. ⟨ t/evaluator.c 345 ⟩ +≡

```
int llt_Evaluator__run(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tc→reason ← interpret( );
  return LLT_RUN_CONTINUE;
}
```

348. ⟨ t/evaluator.c 345 ⟩ +≡

```
int llt_Evaluator__validate(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tap_ok(th, "success", ¬failure_p(tc→reason), fix(tc→reason));
  tap_ok(th, "correct␣result", Accumulator ≡ tc→want, Accumulator);
  return LLT_RUN_CONTINUE;
}
```

349. ⟨ t/evaluator.c 345 ⟩ +≡

```
int llt_Evaluator__failure(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tap_ok(th, "fails", tc→reason ≡ tc→expect, fix(tc→reason));
  return LLT_RUN_CONTINUE;
}
```

350. ⟨ t/evaluator.c 345 ⟩ +≡

```
int llt_Evaluator__clean(llt_header *th unused)
{
  Environment ← ((llt_fixture *) th)→save_environment;
  return LLT_RUN_CONTINUE;
}
```

351. ⟨ t/evaluator.c 345 ⟩ +≡

```
error_code llt_Evaluator__Immediate(llt_header *suite, int *count, bool full, llt_header **ret)
{
  llt_fixture *tc;
  llt_header *th;
  error_code reason;
  int i;
  orreturn (llt_fixture_grow(suite, *count, 4));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < 4; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count + i, llt_Evaluator__prepare, llt_Evaluator__run,
        llt_Evaluator__validate, llt_Evaluator__clean);
    tc→taps ← 2;
    tc→expression ← tc→want ← NIL;
    tc→interpret_limit ← 42;
  }
  tc ← ((llt_fixture *) suite) + *count;
  tc[0].name ← "evaluate␣nil";
  tc[0].want ← tc[0].expression ← NIL;
  tc[1].name ← "evaluate␣constant";
  tc[1].want ← tc[1].expression ← fix(42);
  tc[2].name ← "evaluate␣symbol,␣present";
  if (full) {
    orreturn (new_symbol_const(LLT_LOOKUP_CORRECT, &tc[2].want));
    orreturn (new_symbol_const(LLT_LOOKUP_PRESENT, &tc[2].expression));
  }
  tc[3].name ← "evaluate␣symbol,␣missing";
  tc[3].want ← UNDEFINED;
  if (full) orreturn (new_symbol_const(LLT_LOOKUP_MISSING, &tc[3].expression));
  tc[3].expect ← LERR_MISSING;
  tc[3].validate ← (llt_forward) llt_Evaluator__failure;
  tc[3].taps ← 1;
  (*count) += 4;
  *ret ← suite;
  return LERR_NONE;
}
```

352. ⟨ t/evaluator.c 345 ⟩ +≡

```
error_code llt_Evaluator__Simple(llt_header *suite, int *count, bool full, llt_header **ret)
{
  llt_fixture *tc;
  llt_header *th;
  error_code reason;
  int i;
  orreturn (llt_fixture_grow(suite, *count, 1));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < 1; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count + i, llt_Evaluator__prepare, llt_Evaluator__run,
        llt_Evaluator__validate, llt_Evaluator__clean);
    tc→taps ← 2;
    tc→expression ← tc→want ← NIL;
    tc→interpret_limit ← 128;
  }
  tc ← ((llt_fixture *) suite) + *count;
  tc[0].name ← "evaluate␣(root-environment)";
  tc[0].want ← Environment;
  if (full) {
    orreturn (new_symbol_const("root-environment", &tc[0].expression));
    orreturn (cons(tc[0].expression, NIL, &tc[0].expression));
  }
  (*count) += 1;
  *ret ← suite;
  return LERR_NONE;
}
```

353. Reader tests.

⟨ t/reader.c 353 ⟩ ≡
```
#include <string.h>
  ⟨ Test common preamble 296 ⟩
  int main(int argc, char **argv)
  { return llt_main(argc, argv, true); }
  typedef struct {
    LLT_FIXTURE_HEADER
    half interpret_limit;
    cell source;
    cell start_offset;
    cell want;
  } llt_fixture;
  int Test_Fixture_Size ← sizeof(llt_fixture);
  error_code llt_Reader__Simple(llt_header *, int *, bool, llt_header **);
  int llt_Reader__prepare(llt_header *);
  int llt_Reader__run(llt_header *);
  int llt_Reader__validate(llt_header *);
  ⟨ Object constructors for reader tests 359 ⟩
```
See also sections 354, 355, 356, 357, and 358.

354. ⟨ t/reader.c 353 ⟩ +≡
```
  char LLT_Glyph_Tab[] ← {0xe2, 0xad, 0xbe, 0x00};      /* #u2b7e — horizontal tab key */
  char LLT_Glyph_Newline[] ← {0xe2, 0x90, 0xa4, 0x00};      /* #u2424 — symbol for newline */
  struct {
    char *source;
    error_code (*build)(cell *);
  } LLT_Reader_Rules[] ← {
      {"42", llt_Reader__build_integer_42},
      {"(42)", llt_Reader__build_list_42},
      {"(42 )", llt_Reader__build_list_42},
      {"()", llt_Reader__build_NIL},
      {"( )", llt_Reader__build_NIL},
      {"(\t)", llt_Reader__build_NIL},
      {"(\n\t)", llt_Reader__build_NIL},
      {"#f", llt_Reader__build_FALSE},
      {"#t", llt_Reader__build_TRUE},
      {"(())", llt_Reader__build_pair_NIL_NIL},
      {"( ())", llt_Reader__build_pair_NIL_NIL},
      {"(() . ())", llt_Reader__build_pair_NIL_NIL},
      {"symbol", llt_Reader__build_symbol},
      {"long-symbol", llt_Reader__build_long_symbol},
      {"(list)", llt_Reader__build_list},
      {"(+)", llt_Reader__build_list_tiny},
      {"(x y z)", llt_Reader__build_xyz},
      {"(lambda ()x y z)", llt_Reader__build_application},
      {"((x y z) x y z)", llt_Reader__build_list_nested},
      {"((x y z) . (x y z))", llt_Reader__build_list_nested},
      {Λ, Λ}};
  llt_initialise Test_Suite[] ← {llt_Reader__Simple, Λ};
```

355. ⟨ t/reader.c 353 ⟩ +≡

```
int llt_Reader__prepare(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  address fin;
  cell label;
  error_code reason;
  Interpret_Count ← 0;
  Interpret_Limit ← tc→interpret_limit;
  ortrap (new_symbol_const(PROGRAM_READ, &label));
  ortrap (vm_locate_entry(label, Λ, &Ip));
  ortrap (new_symbol_const(PROGRAM_EXIT, &label));
  ortrap (vm_locate_entry(label, Λ, &fin));
  ortrap (new_pointer(fin, &label));
  ortrap (stack_array_push(&Control_Link, label));
  General[0] ← tc→source;
  General[1] ← tc→start_offset;
  return LLT_RUN_CONTINUE;
Trap: return LLT_RUN_FAIL;
}
```

356. ⟨ t/reader.c 353 ⟩ +≡

```
int llt_Reader__run(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tc→reason ← interpret( );
  return LLT_RUN_CONTINUE;
}
```

357. ⟨ t/reader.c 353 ⟩ +≡

```
int llt_Reader__validate(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tap_ok(th, "success", ¬failure_p(tc→reason), fix(tc→reason));
  tap_ok(th, "correct␣result", llt_out_match_p(Accumulator, tc→want), Accumulator);
  return LLT_RUN_CONTINUE;
}
```

358. ⟨ t/reader.c 353 ⟩ +≡

```
error_code llt_Reader__Simple(llt_header *suite, int *count, bool full, llt_header **ret)
{
  llt_fixture *tc;
  llt_header *th;
  char *name;
  int i, length, rules;
  error_code reason;
  for (rules ← 0; LLT_Reader_Rules[rules].source; rules++) ;
  orreturn (llt_fixture_grow(suite, *count, rules));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < rules; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count+i, llt_Reader__prepare, llt_Reader__run, llt_Reader__validate, Λ);
    tc→taps ← 2;
    tc→want ← NIL;
    tc→interpret_limit ← 2048;
    orreturn (llt_sprintf(th, &tc→name, "read␣‘"));
    name ← LLT_Reader_Rules[i].source;
    while (*name) {      /* This is horrifically inefficient. */
      switch (*name) {
      case '\n': orreturn (llt_appendf(th, tc→name, &tc→name, "%s", LLT_Glyph_Newline));
        break;
      case '\t': orreturn (llt_appendf(th, tc→name, &tc→name, "%s", LLT_Glyph_Tab));
        break;
      default: orreturn (llt_appendf(th, tc→name, &tc→name, "%c", *name));
        break;
      }
      name++;
    }
    orreturn (llt_appendf(th, tc→name, &tc→name, "’"));
  }
  if (full)
    for (i ← 0; i < rules; i++) {
      tc ← ((llt_fixture *) suite) + *count + i;
      th ← (llt_header *) tc;
      length ← strlen(LLT_Reader_Rules[i].source);
      orreturn (new_segment(length, 0, &tc→source));
      memmove(segment_base(tc→source), LLT_Reader_Rules[i].source, length);
      tc→start_offset ← fix(0);
      orreturn (LLT_Reader_Rules[i].build(&tc→want));
    }
  (*count) += rules;
  *ret ← suite;
  return LERR_NONE;
}
```

359. ⟨ Object constructors for reader tests 359 ⟩ ≡

```
error_code llt_Reader__build_integer_42(cell *ret)
{
  *ret ← fix(42);
  return LERR_NONE;
}
error_code llt_Reader__build_NIL(cell *ret)
{
  *ret ← NIL;
  return LERR_NONE;
}
error_code llt_Reader__build_FALSE(cell *ret)
{
  *ret ← LFALSE;
  return LERR_NONE;
}
error_code llt_Reader__build_TRUE(cell *ret)
{
  *ret ← LTRUE;
  return LERR_NONE;
}
```

See also sections 360, 361, 362, 363, 364, 365, 366, 367, and 368.

This code is used in section 353.

360. ⟨ Object constructors for reader tests 359 ⟩ +≡

```
error_code llt_Reader__build_pair_NIL_NIL(cell *ret)
{
  return cons(NIL, NIL, ret);
}
```

361. ⟨ Object constructors for reader tests 359 ⟩ +≡

```
error_code llt_Reader__build_symbol(cell *ret)
{
  return new_symbol_cstr("symbol", ret);
}
```

362. ⟨ Object constructors for reader tests 359 ⟩ +≡

```
error_code llt_Reader__build_long_symbol(cell *ret)
{
  return new_symbol_cstr("long-symbol", ret);
}
```

363. ⟨ Object constructors for reader tests 359 ⟩ +≡

```
error_code llt_Reader__build_list(cell *ret)
{
  cell tmp;
  error_code reason;
  orreturn (new_symbol_cstr("list", &tmp));
  return cons(tmp, NIL, ret);
}
```

364. ⟨ Object constructors for reader tests 359 ⟩ +≡
```
error_code llt_Reader__build_list_42(cell *ret)
{
  return cons(fix(42), NIL, ret);
}
```

365. ⟨ Object constructors for reader tests 359 ⟩ +≡
```
error_code llt_Reader__build_list_tiny(cell *ret)
{
  cell tmp;
  error_code reason;
  orreturn (new_symbol_cstr("+", &tmp));
  return cons(tmp, NIL, ret);
}
```

366. ⟨ Object constructors for reader tests 359 ⟩ +≡
```
error_code llt_Reader__build_xyz(cell *ret)
{
  cell tmp, x, y, z;
  error_code reason;
  orreturn (new_symbol_cstr("x", &x));
  orreturn (new_symbol_cstr("y", &y));
  orreturn (new_symbol_cstr("z", &z));
  orreturn (cons(z, NIL, &tmp));
  orreturn (cons(y, tmp, &tmp));
  return cons(x, tmp, ret);
}
```

367. ⟨ Object constructors for reader tests 359 ⟩ +≡
```
error_code llt_Reader__build_list_nested(cell *ret)
{
  cell head, tail;
  error_code reason;
  orreturn (llt_Reader__build_xyz(&head));
  orreturn (llt_Reader__build_xyz(&tail));
  return cons(head, tail, ret);
}
```

368. ⟨ Object constructors for reader tests 359 ⟩ +≡
```
error_code llt_Reader__build_application(cell *ret)
{
  cell label, tmp;
  error_code reason;
  orreturn (llt_Reader__build_xyz(&tmp));
  orreturn (new_symbol_cstr("lambda", &label));
  orreturn (cons(NIL, tmp, &tmp));
  return cons(label, tmp, ret);
}
```

369. Closure tests.

⟨ t/closure.c 369 ⟩ ≡
```
#include <string.h>
  ⟨ Test common preamble 296 ⟩
  int main(int argc, char **argv)
  { return llt_main(argc, argv, true); }
  typedef struct {
    LLT_FIXTURE_HEADER
    error_code(*build)(cell *);
    half interpret_limit;
    char *csource;
    cell lsource, ssource;
    cell want;
  } llt_fixture;
  int Test_Fixture_Size ← sizeof(llt_fixture);
  error_code llt_Closure__Evaluate(llt_header *, int *, bool, llt_header **);
  error_code llt_Closure__Sequence(llt_header *, int *, bool, llt_header **);
  error_code llt_Closure__Simple(llt_header *, int *, bool, llt_header **);
  int llt_Closure__prepare(llt_header *);
  int llt_Closure__run(llt_header *);
  int llt_Closure__validate(llt_header *);
  ⟨ Object constructors for closure tests 377 ⟩
```
See also sections 370, 371, 372, 373, 374, 375, and 376.

370. ⟨ t/closure.c 369 ⟩ +≡

```
struct {
  char *source;
  error_code (*build)(cell *);
} LLT_Closure_Sequence_Rules[] ← {
    {"(do)", llt_Closure__out_sequence_empty},
    {"(do #f)", llt_Closure__out_sequence_false},
    {"(do #t)", llt_Closure__out_sequence_true},
    {"(do #f 42 #t)", llt_Closure__out_sequence_true},
    {Λ, Λ}};
struct {
  char *source;
  error_code (*build)(cell *);
} LLT_Closure_Simple_Rules[] ← {
    {"(lambda ())", llt_Closure__build_},
    {"(lambda x)", llt_Closure__build__x},
    {"(lambda (x))", llt_Closure__build_x},
    {"(lambda (x y))", llt_Closure__build_xy},
    {"(lambda (x y . z))", llt_Closure__build_xy_z},
    {"(vov ())", llt_Closure__build_},
    {"(vov ((a copy)))", llt_Closure__build_a},
    {"(vov ((e environment)))", llt_Closure__build_e},
    {"(vov ((a copy-list) (e environment)))", llt_Closure__build_ae},
    {Λ, Λ}};
struct {
  char *source;
  error_code (*build)(cell *);
} LLT_Closure_Evaluate_Rules[] ← {
    {"((lambda ()) )", llt_Closure__out_sequence_empty},
    {"((lambda x) )", llt_Closure__out_sequence_empty},
    {"((lambda x x) )", llt_Closure__out_sequence_NIL},
    {"((lambda x x) 1 2 3)", llt_Closure__out_sequence_123},
    {"((lambda (x y z) (cons x (cons y (cons z ()))))) 1 2 3)",
    llt_Closure__out_sequence_123},
    {"((vov ()) )", llt_Closure__out_sequence_empty},
    {"((vov ((x copy-list)) x) marco polo!)", llt_Closure__out_operative_arguments},
    {"((vov ((x environment)) x))", llt_Closure__out_operative_environment},
    {Λ, Λ}};
llt_initialise Test_Suite[] ← {llt_Closure__Sequence, llt_Closure__Simple, llt_Closure__Evaluate, Λ};
```

371. ⟨ t/closure.c 369 ⟩ +≡

```
int llt_Closure__prepare(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  address fin;
  cell label, jexit;
  int length;
  error_code reason;
  ortrap (tc→build(&tc→want));
  length ← strlen(tc→csource);
  orreturn (new_segment(length, 0, &tc→ssource));
  memmove(segment_base(tc→ssource), tc→csource, length);
  ortrap (new_symbol_const(PROGRAM_EXIT, &label));
  ortrap (vm_locate_entry(label, Λ, &fin));
  ortrap (new_pointer(fin, &jexit));
  ortrap (stack_array_push(&Control_Link, jexit));
  General[0] ← tc→ssource;
  General[1] ← fix(0);
  ortrap (new_symbol_const(PROGRAM_READ, &label));
  ortrap (vm_locate_entry(label, Λ, &Ip));
  Interpret_Count ← Interpret_Limit ← 0;
  ortrap (interpret( ));
  General[0] ← General[1] ← NIL;
  tc→lsource ← Accumulator;
  Interpret_Count ← 0;
  Interpret_Limit ← tc→interpret_limit;
  ortrap (stack_array_push(&Control_Link, jexit));
  ortrap (new_symbol_const(PROGRAM_EVALUATE, &label));
  ortrap (vm_locate_entry(label, Λ, &Ip));
  Expression ← tc→lsource;
  return LLT_RUN_CONTINUE;
Trap: return LLT_RUN_FAIL;
}
```

372. ⟨ t/closure.c 369 ⟩ +≡

```
int llt_Closure__run(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tc→reason ← interpret( );
  return LLT_RUN_CONTINUE;
}
```

373. ⟨ t/closure.c 369 ⟩ +≡

```
int llt_Closure__validate(llt_header *th)
{
  llt_fixture *tc ← (llt_fixture *) th;
  tap_ok(th, "success", ¬failure_p(tc→reason), fix(tc→reason));
  tap_ok(th, "correct␣result", llt_out_match_p(Accumulator, tc→want), Accumulator);
  return LLT_RUN_CONTINUE;
}
```

374. ⟨ t/closure.c 369 ⟩ +≡

```
error_code llt_Closure__Simple(llt_header *suite, int *count, bool full unused, llt_header **ret)
{
  llt_fixture *tc;
  llt_header *th;
  error_code reason;
  int i, rules;
  for (rules ← 0; LLT_Closure_Simple_Rules[rules].source; rules++) ;
  orreturn (llt_fixture_grow(suite, *count, rules));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < rules; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count + i, llt_Closure__prepare, llt_Closure__run, llt_Closure__validate,
        Λ);
    orreturn (llt_sprintf(th, &tc→name, "construct␣closure␣‘%s’",
        LLT_Closure_Simple_Rules[i].source));
    tc→build ← LLT_Closure_Simple_Rules[i].build;
    tc→csource ← LLT_Closure_Simple_Rules[i].source;
    tc→taps ← 2;
    tc→want ← tc→lsource ← tc→ssource ← NIL;
    tc→interpret_limit ← 2048;
  }
  (*count) += rules;
  *ret ← suite;
  return LERR_NONE;
}
```

375. ⟨ t/closure.c 369 ⟩ +≡

```
error_code llt_Closure__Sequence(llt_header *suite, int *count, bool full unused, llt_header
        **ret)
{
  llt_fixture *tc;
  llt_header *th;
  error_code reason;
  int i, rules;
  for (rules ← 0; LLT_Closure_Sequence_Rules[rules].source; rules++) ;
  orreturn (llt_fixture_grow(suite, *count, rules));
  tc ← (llt_fixture *) suite;
  for (i ← 0; i < rules; i++) {
    tc ← ((llt_fixture *) suite) + *count + i;
    th ← (llt_header *) tc;
    llt_fixture__init_common(th, *count + i, llt_Closure__prepare, llt_Closure__run, llt_Closure__validate,
        Λ);
    orreturn (llt_sprintf(th, &tc→name, "perform␣sequence␣‘%s’",
        LLT_Closure_Sequence_Rules[i].source));
    tc→build ← LLT_Closure_Sequence_Rules[i].build;
    tc→csource ← LLT_Closure_Sequence_Rules[i].source;
    tc→taps ← 2;
    tc→want ← tc→lsource ← tc→ssource ← NIL;
    tc→interpret_limit ← 2048;
  }
  (*count) += rules;
  *ret ← suite;
  return LERR_NONE;
}
```

376. This is the third function that's practically identical...

```
⟨ t/closure.c 369 ⟩ +≡
  error_code llt_Closure__Evaluate(llt_header *suite, int *count, bool full unused, llt_header
          **ret)
  {
    llt_fixture *tc;
    llt_header *th;
    error_code reason;
    int i, rules;
    for (rules ← 0; LLT_Closure_Evaluate_Rules[rules].source; rules++) ;
    orreturn (llt_fixture_grow(suite, *count, rules));
    tc ← (llt_fixture *) suite;
    for (i ← 0; i < rules; i++) {
      tc ← ((llt_fixture *) suite) + *count + i;
      th ← (llt_header *) tc;
      llt_fixture__init_common(th, *count + i, llt_Closure__prepare, llt_Closure__run, llt_Closure__validate,
          Λ);
      orreturn (llt_sprintf(th, &tc→name, "evaluate␣closure␣‘%s’",
          LLT_Closure_Evaluate_Rules[i].source));
      tc→build ← LLT_Closure_Evaluate_Rules[i].build;
      tc→csource ← LLT_Closure_Evaluate_Rules[i].source;
      tc→taps ← 2;
      tc→want ← tc→lsource ← tc→ssource ← NIL;
      tc→interpret_limit ← 2048;
    }
    (*count) += rules;
    *ret ← suite;
    return LERR_NONE;
  }
```

377. These build a closure to compare with the one built by the test.

```
⟨ Object constructors for closure tests 377 ⟩ ≡
  error_code llt_Closure__build_(cell *ret)
  {    /* (lambda ()) remains (). */
    return new_closure(NIL, NIL, ret);
  }
```

See also sections 378, 379, 380, 381, 382, 383, 384, 385, 386, and 387.

This code is used in section 369.

378.

```
⟨ Object constructors for closure tests 377 ⟩ +≡
  error_code llt_Closure__build__x(cell *ret)
  {    /* (lambda x) becomes ((x eval-list)). */
    cell leval, sign, tmp, x;
    error_code reason;
    orreturn (new_symbol_const("eval-list", &leval));
    orreturn (new_symbol_const("x", &x));
    orreturn (cons(leval, NIL, &tmp));
    orreturn (cons(x, tmp, &tmp));
    orreturn (cons(tmp, NIL, &sign));
    return new_closure(sign, NIL, ret);
  }
```

379. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_a(cell *ret)
{    /* (vov ((a copy))) */
  cell a, copy, tmp, sign;
  error_code reason;
  orreturn (new_symbol_const("copy", &copy));
  orreturn (new_symbol_const("a", &a));
  orreturn (cons(copy, NIL, &tmp));
  orreturn (cons(a, tmp, &tmp));
  orreturn (cons(tmp, NIL, &sign));
  return new_closure(sign, NIL, ret);
}
```

380. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_ae(cell *ret)
{    /* (vov ((a copy-list) (e environment))) */
  cell a, atmp, e, etmp, env, lcopy, sign;
  error_code reason;
  orreturn (new_symbol_const("environment", &env));
  orreturn (new_symbol_const("copy-list", &lcopy));
  orreturn (new_symbol_const("a", &a));
  orreturn (new_symbol_const("e", &e));
  orreturn (cons(env, NIL, &etmp));
  orreturn (cons(e, etmp, &etmp));
  orreturn (cons(lcopy, NIL, &atmp));
  orreturn (cons(a, atmp, &atmp));
  orreturn (cons(etmp, NIL, &sign));
  orreturn (cons(atmp, sign, &sign));
  return new_closure(sign, NIL, ret);
}
```

381. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_e(cell *ret)
{    /* (vov ((a environment))) */
  cell env, tmp, sign, e;
  error_code reason;
  orreturn (new_symbol_const("environment", &env));
  orreturn (new_symbol_const("e", &e));
  orreturn (cons(env, NIL, &tmp));
  orreturn (cons(e, tmp, &tmp));
  orreturn (cons(tmp, NIL, &sign));
  return new_closure(sign, NIL, ret);
}
```

382. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_x(cell *ret)
{    /* (lambda (x)) becomes ((x eval)). */
  cell eval, tmp, sign, x;
  error_code reason;
  orreturn (new_symbol_const("eval",&eval));
  orreturn (new_symbol_const("x",&x));
  orreturn (cons(eval,NIL,&tmp));
  orreturn (cons(x,tmp,&tmp));
  orreturn (cons(tmp,NIL,&sign));
  return new_closure(sign,NIL,ret);
}
```

383. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_xy(cell *ret)
{    /* (lambda (x y)) becomes ((x eval) (y eval)). */
  cell eval, sign, x, xtmp, y, ytmp;
  error_code reason;
  orreturn (new_symbol_const("eval",&eval));
  orreturn (new_symbol_const("x",&x));
  orreturn (new_symbol_const("y",&y));
  orreturn (cons(eval,NIL,&eval));
  orreturn (cons(x,eval,&xtmp));
  orreturn (cons(y,eval,&ytmp));
  orreturn (cons(ytmp,NIL,&sign));
  orreturn (cons(xtmp,sign,&sign));
  return new_closure(sign,NIL,ret);
}
```

384. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__build_xy_z(cell *ret)
{    /* (lambda (x y . z)) becomes ((x eval) (y eval) (z eval-list)). */
  cell eval, leval, sign, x, xtmp, y, ytmp, z, ztmp;
  error_code reason;
  orreturn (new_symbol_const("eval",&eval));
  orreturn (new_symbol_const("eval-list",&leval));
  orreturn (new_symbol_const("x",&x));
  orreturn (new_symbol_const("y",&y));
  orreturn (new_symbol_const("z",&z));
  orreturn (cons(eval,NIL,&eval));
  orreturn (cons(leval,NIL,&leval));
  orreturn (cons(x,eval,&xtmp));
  orreturn (cons(y,eval,&ytmp));
  orreturn (cons(z,leval,&ztmp));
  orreturn (cons(ztmp,NIL,&sign));
  orreturn (cons(ytmp,sign,&sign));
  orreturn (cons(xtmp,sign,&sign));
  return new_closure(sign,NIL,ret);
}
```

385. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__out_sequence_empty(cell *ret)
{   /* (do) does nothing and 'returns' #VOID. */
  *ret ← VOID;
  return LERR_NONE;
}
error_code llt_Closure__out_sequence_NIL(cell *ret)
{   /* To test that a sequence returns (). */
  *ret ← NIL;
  return LERR_NONE;
}
error_code llt_Closure__out_sequence_false(cell *ret)
{   /* To test that a sequence returns #f. */
  *ret ← LFALSE;
  return LERR_NONE;
}
error_code llt_Closure__out_sequence_true(cell *ret)
{   /* To test that a sequence returns #t. */
  *ret ← LTRUE;
  return LERR_NONE;
}
error_code llt_Closure__out_sequence_123(cell *ret)
{   /* To test that a sequence returns (1 2 3). */
  cell tmp;
  error_code reason;
  orreturn (cons(fix(3), NIL, &tmp));
  orreturn (cons(fix(2), tmp, &tmp));
  return cons(fix(1), tmp, ret);
}
```

386. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__out_operative_arguments(cell *ret)
{   /* To test that a closure returns its arguments unevaluated. */
  cell label, list;
  error_code reason;
  orreturn (new_symbol_const("polo!", &label));
  orreturn (cons(label, NIL, &list));
  orreturn (new_symbol_const("marco", &label));
  return cons(label, list, ret);
}
```

387. ⟨ Object constructors for closure tests 377 ⟩ +≡

```
error_code llt_Closure__out_operative_environment(cell *ret)
{   /* To test that a closure returns its caller's environment. */
  *ret ← Environment;
  return LERR_NONE;
}
```

388. Index. And some remaining bits & pieces.

```
#define PROGRAM_EVALUATE "!Evaluate"
#define PROGRAM_EXIT "!Exit"
#define PROGRAM_READ "!Primitive/read-expression"
#define LLT_LOOKUP_MISSING "absent-binding"
#define LLT_LOOKUP_PRESENT "existing-binding"
#define LLT_LOOKUP_CORRECT "polo!"
#define LLT_LOOKUP_STALE "marco"
```

389. ⟨ Function declarations 19 ⟩ +≡

```
void dump(cell o);
```

390.

```
void dump(cell o){
  if (null_p(o)) printf("()");
  else if (false_p(o)) printf("#f");
  else if (true_p(o)) printf("#t");
  else if (integer_p(o)) printf("%ld",fixed_value(o));
  else if (void_p(o)) printf("#?");
  else if (symbol_p(o)) psym(o)
  else
    if (pair_p(o)) {
      printf("(");
      dump(A(o)→sin);
      printf("␣.␣");
      dump(A(o)→dex);
      printf(")");
    }
    else if (syntax_p(o)) {
      printf("{");
      dump(A(o)→sin);
      printf("=");
      dump(A(o)→dex);
      printf("}");
    }
    else printf("?%x?",TAG(o));
}
```

391. ⟨ Carry out an operation 172 ⟩ +≡

```
case OP_DUMP: ortrap (interpret_solo_argument(ins,&VM_Arg1));
  dump(VM_Arg1);
  printf("\n");
  break;
```

⟨ (Re-)Initialise thread register pointers 131 ⟩ Used in section 290.
⟨ Add exported address symbols to a copy of *Program_Export_Table* 260 ⟩ Used in section 256.
⟨ Allocate a segment for a previously interned segment 57 ⟩ Used in section 55.
⟨ Carry out an operation 172, 173, 174, 175, 176, 177, 178, 179, 187, 188, 189, 190, 191, 192, 193, 391 ⟩ Used in section 163.
⟨ Collect pending comments and reduce the code array 253 ⟩ Used in section 252.
⟨ Copy constant objects into *Program_ObjectDB* 258 ⟩ Used in section 256.
⟨ Data for initialisation 12, 132, 139, 145, 278, 281 ⟩ Used in section 8.
⟨ Define a 16-bit addressing environment 26 ⟩ Used in section 23.
⟨ Define a 32-bit addressing environment 25 ⟩ Used in section 23.
⟨ Define a 64-bit addressing environment 24 ⟩ Used in section 23.
⟨ Encode a 16-bit address and **break** 268 ⟩ Used in section 267.
⟨ Encode a large object and **break** 269 ⟩ Used in section 267.
⟨ Encode a single address argument and **break** 263 ⟩ Used in section 262.
⟨ Encode an error identifier argument and **break** 264 ⟩ Used in section 262.
⟨ Encode the first argument 262 ⟩ Used in section 261.
⟨ Encode the first object argument and **break** 265 ⟩ Used in section 262.
⟨ Encode the first register argument and **break** 266 ⟩ Used in section 262.
⟨ Encode the middle ALOT and **break** 270 ⟩ Used in section 267.
⟨ Encode the second argument 267 ⟩ Used in section 261.
⟨ Encode the third argument 271 ⟩ Used in section 261.
⟨ Essential types 23 ⟩ Used in section 5.
⟨ External C symbols 14, 18, 36, 48, 100, 107, 116, 123, 130, 138, 144, 147, 279, 282, 287 ⟩ Used in section 5.
⟨ Function declarations 19, 31, 37, 49, 62, 79, 84, 101, 108, 117, 124, 150, 162, 181, 195, 232, 288, 389 ⟩ Used in section 5.
⟨ Global variables 13, 17, 35, 47, 99, 106, 115, 122, 129, 137, 143, 146, 277, 286 ⟩ Used in section 7.
⟨ Hacks and warts 9, 63, 75 ⟩ Used in section 5.
⟨ Initialise `Lossless` primitives 148 ⟩ Used in section 126.
⟨ Initialise `Lossless` procedures 283 ⟩ Used in section 126.
⟨ Initialise assembler symbols 133, 134, 140 ⟩ Used in section 126.
⟨ Initialise error symbols 15 ⟩ Used in section 126.
⟨ Initialise evaluator and other bytecode 280 ⟩ Used in section 126.
⟨ Initialise foreign linkage 118 ⟩ Used in section 22.
⟨ Initialise heap 38 ⟩ Used in section 22.
⟨ Initialise memory allocator 50 ⟩ Used in section 22.
⟨ Initialise program linkage 125 ⟩ Used in section 22.
⟨ Initialise run-time environment 109, 151 ⟩ Used in section 22.
⟨ Initialise symbol table 102 ⟩ Used in section 22.
⟨ Initialise threading 289 ⟩ Used in section 22.
⟨ Install instructions as bytecode and commentary 261 ⟩ Used in section 256.
⟨ Intern a previously allocated segment 56 ⟩ Used in section 55.
⟨ Link `Lossless` primitives to installed bytecode 149 ⟩ Used in section 126.
⟨ Look for required address symbols 259 ⟩ Used in section 256.
⟨ Object constructors for closure tests 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387 ⟩ Used in section 369.
⟨ Object constructors for reader tests 359, 360, 361, 362, 363, 364, 365, 366, 367, 368 ⟩ Used in section 353.
⟨ Parse a test run specification from the command line 300 ⟩ Used in section 298.
⟨ Parse command line options 299 ⟩ Used in section 298.
⟨ Prepare a new code page 257 ⟩ Used in section 256.
⟨ Record remaining required far links 255 ⟩ Used in section 252.
⟨ Resize an allocated segment 58 ⟩ Used in section 55.
⟨ System headers 6 ⟩ Cited in section 4. Used in section 5.
⟨ Test common preamble 296 ⟩ Used in sections 297, 328, 337, 345, 353, and 369.
⟨ Test definitions 295, 309, 312 ⟩ Used in section 292.
⟨ Test fixture header 293 ⟩ Used in section 292.
⟨ Test functions 294, 310, 321, 323 ⟩ Used in section 292.
⟨ Testing memory allocator 313 ⟩ Used in section 20.

⟨ Testing memory deallocator 314 ⟩ Used in section 21.
⟨ Type definitions 11, 28, 34, 46, 82, 91, 98, 114, 121, 135, 136, 141, 142, 180, 203, 285 ⟩ Used in section 5.
⟨ Update in-page far links to relative 254 ⟩ Used in section 252.
⟨ `initialise.c` 8, 22, 126 ⟩
⟨ `lossless.h` 5 ⟩
⟨ `t/closure.c` 369, 370, 371, 372, 373, 374, 375, 376 ⟩
⟨ `t/evaluator.c` 345, 346, 347, 348, 349, 350, 351, 352 ⟩
⟨ `t/hashtable.c` 337, 338, 339, 340, 341, 342, 343, 344 ⟩
⟨ `t/insanity.c` 328, 329, 330, 331, 332, 333, 334, 335, 336 ⟩
⟨ `t/reader.c` 353, 354, 355, 356, 357, 358 ⟩
⟨ `testless.c` 297, 298, 301, 302, 303, 304, 305, 306, 307, 308, 311, 316, 317, 318, 319, 320, 322, 324, 325, 327 ⟩
⟨ `testless.h` 292 ⟩

LOSSLESS