1. Introduction. `LossLess` is a lisp-like programming language. It is destined to not rely on the C library but for the time being uses *malloc* for some of the heavy lifting of memory management. Some other C features are taken advantage of or worked around while C the language of this implementation but these are not integral to the character of `LossLess`.

TODO: Remove headers which are no longer necessary.

```
⟨ System headers 1 ⟩ ≡
#include <assert.h>
#include <ctype.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
```

This code is used in sections 2 and 3.

2. The main product of this part of `LossLess` is a library for dynamic or static linking. Untitled sections are concatenated and included in this library, starting with this one.

```
  ⟨ System headers 1 ⟩
  ⟨ Repair the system headers 322 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Type definitions 6 ⟩
  ⟨ Function declarations 21 ⟩
  ⟨ Global variables 15 ⟩
```

3. Consumers of the library include this header file of the same type and function declarations, and global variables with an **extern** qualifier.

```
⟨ lossless.h 3 ⟩ ≡
  ⟨ System headers 1 ⟩
  ⟨ Repair the system headers 322 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Type definitions 6 ⟩
  ⟨ Function declarations 21 ⟩
  ⟨ External symbols 16 ⟩
```

4. Access to `LossLess`' parts from other languages requires some C macros to be made available through symbols, additional validation on functions, etc. These optional are placed in the file `ffi.c` and linked into `liblossless.so`/`lossless.dll`.

5. Library users may need symbols for C macros.

```
⟨ ffi.c 5 ⟩ ≡
#include "lossless.h"
```

See also section 326.

6. LossLess (and lisp) data all descend from the **cell**; two cells make an *atom*, called **wide**. A pointer to an atom is equivalent to a machine-specific memory address, ie. a normal C pointer or "**char** $*$".

These (untested) macros are intended to allow LossLess to compile on a wider variety of hardware than the latest 2^{n+1}-bit wintel fruit. Also defined here is **fixed** representing numbers small enough to fit in the value of a pointer with trickery and the **digit** which is used to represent unbounded integers.

⟨ Type definitions 6 ⟩ ≡
```
  typedef intptr_t cell;
  typedef uintptr_t digit;
#if SIZE_MAX ≤ 0xffffUL
  ⟨ Define a 16-bit addressing environment 10 ⟩
#elif SIZE_MAX ≤ 0xffffffffUL
  ⟨ Define a 32-bit addressing environment 9 ⟩
#elif SIZE_MAX ≤ 0xffffffffffffffffUL
  ⟨ Define a 64-bit addressing environment 8 ⟩
#else
#error Tiny computer.
#endif
```

See also sections 7, 13, 14, 25, 31, 44, 53, 84, 97, 125, 126, 134, 141, 178, 179, 186, and 190.
This code is used in sections 2 and 3.

7. Values common to all architectures. If a datum is smaller than INTERN_BYTES it may be directly "interned" rather than pointed at.

⟨ Type definitions 6 ⟩ +≡
```
#define CELL_MASK (∼(WIDE_BYTES − 1))
#define BYTE_BITS 8
#define BYTE_BYTES 1
#define BYTE 0xff
#define TAG_BYTES 1
#define DIGIT_MAX UINTPTR_MAX
#define HALF_MIN (INTPTR_MIN/2)
#define HALF_MAX (INTPTR_MAX/2)
#define INTERN_BYTES (WIDE_BYTES − 1)
  typedef intptr_t cell;
  typedef uintptr_t digit;
  typedef struct {
    char buffer[INTERN_BYTES];
    unsigned char length;
  } Ointern;
```

8. Fixed-integer values are up to 32 bits (signed) on a 64-bit architecture. This choice was arbitrary — the data encoding scheme at present in fact makes 60 bits available — but it's easier to envision dealing with half a word than 15 16^{th}s of a word.

⟨ Define a 64-bit addressing environment 8 ⟩ ≡

```
#define CELL_BITS 64
#define CELL_BYTES 8
#define CELL_SHIFT 4
#define WIDE_BITS 128
#define WIDE_BYTES 16
#define FIX_MIN (−0x7fffffffLL − 1)    /* 32 bits */
#define FIX_MAX 0x7fffffffLL
#define FIX_MASK 0xffffffffLL
#define FIX_SHIFT 32
#define FIX_BASE2 32
#define FIX_BASE8 11
#define FIX_BASE10 10
#define FIX_BASE16 8
  typedef union {
    struct {
      cell low;
      cell high;
    };
    struct {
      char b[16];
    } value;
  } wide;
  typedef int32_t fixed;
  typedef int32_t half;
```

This code is used in section 6.

9. 16 bit numbers would probably be too small to be of practical use to fixed integers in a 32-bit envionment use 24 (of the presently-available 28) bits to occupy 3 quarters of a word.

⟨ Define a 32-bit addressing environment 9 ⟩ ≡

```
#define CELL_BITS 32
#define CELL_BYTES 4
#define CELL_SHIFT 3
#define WIDE_BITS 64
#define WIDE_BYTES 8
#define FIX_MIN (−0x7fffffL − 1)    /* 24 bits */
#define FIX_MAX 0x7fffffL
#define FIX_MASK 0xffffffL
#define FIX_SHIFT 8
#define FIX_BASE2 24
#define FIX_BASE8 8
#define FIX_BASE10 8
#define FIX_BASE16 6
  typedef union {
    struct {
      cell low;
      cell high;
    };
    int64_t value;
  } wide;
  typedef int32_t fixed;    /* 32 − 24 = 8 unused bits */
  typedef int16_t half;
```

This code is used in section 6.

10. This section is here because it was easy to write. It remains to be seen how practical it is.

⟨ Define a 16-bit addressing environment 10 ⟩ ≡

```
#define CELL_BITS 16
#define CELL_BYTES 2
#define CELL_SHIFT 2
#define WIDE_BITS 32
#define WIDE_BYTES 4
#define FIX_MIN (−0x7f − 1)    /* 8 bits */
#define FIX_MAX 0x7f
#define FIX_MASK 0xff
#define FIX_SHIFT 8
#define FIX_BASE2 8
#define FIX_BASE8 3
#define FIX_BASE10 3
#define FIX_BASE16 2
  typedef union {
    struct {
      cell low;
      cell high;
    };
    int64_t value;
  } wide;
  typedef int8_t fixed;
  typedef int8_t half;
```

This code is used in section 6.

11. Something like these (unused) macros could merge the heap's tag with the cell data if the machine's pointers are large enough.

```
#define PTR_TAG_SHIFT 56
#define PTR_ADDRESS(p) ((intptr_t(p)) & ((1ULL ≪ PTR_TAG_SHIFT) − 1))
#define PTR_TAG_MASK(p) ((intptr_t(p)) & ∼((1ULL ≪ PTR_TAG_SHIFT) − 1))
#define PTR_TAG(p) (PTR_TAG_MASK(p) ≫ PTR_TAG_SHIFT)
#define PTR_SET_TAG(p, s) ((p) ← (((p) & PTR_ADDRESS(p)) | ((s) ≪ PTR_TAG_SHIFT)))
```

12. LossLess is entirely a single threaded program but includes the foundations necessary to support multiple threads. Each global or static variable can be **shared** between all threads or **unique** to each thread.

```
#define shared
#define unique __thread
```

13. Code comes with errors and programming is a way of finding them (and turning them into bugs).

Blocks of code which may fail while holding resources set up a "long-jump site" using `sigsetjmp(3)`, a pointer to which is passed to every fallible function that gets called. In general the pointer to a long-jump site is called *failure* and a locally established long-jump site is called *cleanup*. The error code is returned in *reason*.

```
#define failure_p(O) ((O) ≠ LERR_NONE)
⟨Type definitions 6⟩ +≡
  typedef enum {
    LERR_NONE,
    LERR_AMBIGUOUS,      /* Constant etc. incorrectly terminated. */
    LERR_DOUBLE_TAIL,    /* Two . operators. */
    LERR_EMPTY_TAIL,     /* A . without a tail expression. */
    LERR_EOF,            /* End of file or stream. */
    LERR_EXISTS,         /* New binding conflicts. */
    LERR_HEAVY_TAIL,     /* A . with more than one tail expression. */
    LERR_IMPROPER,       /* A list operation encountered an improper list. */
    LERR_INCOMPATIBLE,   /* Operation on incompatible operand. */
    LERR_INTERNAL,       /* Bug in LossLess. */
    LERR_INTERRUPT,      /* An operation was interrupted. */
    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 keytable or environment lookup failed. */
    LERR_NONCHARACTER,   /* Scanning UTF-8 encoding failed. */
    LERR_OOM,            /* Out of memory. */
    LERR_OVERFLOW,       /* Attempt to access past the end of a buffer. */
    LERR_SYNTAX,         /* Unrecognisable syntax (insufficient alone). */
    LERR_UNCLOSED_OPEN,  /* Missing ), ] or }. */
    LERR_UNCOMBINABLE,   /* Attempted to combine a non-program. */
    LERR_UNDERFLOW,      /* A stack was popped too far. */
    LERR_UNIMPLEMENTED,  /* A feature is not implemented. */
    LERR_UNOPENED_CLOSE, /* Premature ), ] or }. */
    LERR_UNPRINTABLE,    /* Failed serialisation attempt. */
    LERR_UNSCANNABLE,    /* Parser encountered LEXICAT_INVALID. */
    LERR_LENGTH
  } Verror;
```

14. The numeric codes are suitable for internal use but lisp is a language of processing symbols so at run-time the errors are given names (and, potentially, other metadata).

```
⟨Type definitions 6⟩ +≡
  typedef struct {
    char *message;
  } Oerror;
```

15. This list is sorted by the "column" on the right.

⟨ Global variables 15 ⟩ ≡

```
Oerror Ierror[LERR_LENGTH] ← {
    [LERR_AMBIGUOUS] ← {"ambiguous-syntax"},
    [LERR_EXISTS] ← {"binding-conflict"},
    [LERR_DOUBLE_TAIL] ← {"double-tail"},
    [LERR_EOF] ← {"end-of-file"},
    [LERR_IMPROPER] ← {"improper-list"},
    [LERR_INCOMPATIBLE] ← {"incompatible-operand"},
    [LERR_INTERRUPT] ← {"interrupted"},
    [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_OOM] ← {"out-of-memory"},
    [LERR_OVERFLOW] ← {"overflow"},
    [LERR_LIMIT] ← {"software-limit"},
    [LERR_SYNTAX] ← {"syntax-error"},
    [LERR_HEAVY_TAIL] ← {"tail-mid-list"},
    [LERR_UNCLOSED_OPEN] ← {"unclosed-brackets"},
    [LERR_UNCOMBINABLE] ← {"uncombinable"},
    [LERR_UNDERFLOW] ← {"underflow"},
    [LERR_UNIMPLEMENTED] ← {"unimplemented"},
    [LERR_UNOPENED_CLOSE] ← {"unopened-brackets"},
    [LERR_UNPRINTABLE] ← {"unprintable"},
    [LERR_UNSCANNABLE] ← {"unscannable-lexeme"},
    [LERR_EMPTY_TAIL] ← {"unterminated-tail"},
};
```

See also sections 19, 32, 40, 45, 54, 75, 94, 124, 148, 158, 187, and 246.

This code is used in section 2.

16. ⟨ External symbols 16 ⟩ ≡

```
extern Oerror Ierror[];
```

See also sections 20, 33, 76, 149, 247, 325, and 327.

This code is used in section 3.

17. When `LossLess` cannot proceed it aborts its current action and jumps back to the most recently established long-jump site with one of the error codes. If, as is usually the case, the error condition cannot be dealt with by that caller it will clean up its own resources and jump back to the long-jump site given it (passing on the same error code), which it had saved before creating its own.

The outline of a function which protects its resources in this way is shown here. A lot of pieces of code in this implementation have this basic outline so it will be helpful to understand it well enough to be able to ignore it because C is verbose enough.

Nearly all the time the resource in question is S items on top of the run-time stack. In some cases this is also or instead the register *Tmp_ier* (T). The *unwind* macro cleans up both, relying on the C compiler to remove the branches of unreachable code, and performs the long jump J with error code E.

In particular, allocated memory generally does *not* need to be freed explicitly if an error occurs while it's in use — as will be seen shortly, allocated memory is immediately tied into a list which is then scanned during garbage collection for unused allocations that can be released.

```
#define unwind(J, E, T, S) do
        {
          assert((E) ≠ LERR_NONE);
          if (T) Tmp_ier ← NIL;
          if (S) stack_clear(S);
          siglongjmp (*(J), (E));
        }
        while (0)
#if 0
  void example(sigjmp_buf *failure)
  {
    sigjmp_buf cleanup;    /* Allocated on C's stack, there is no need to clean this up. */
    Verror reason ← LERR_NONE;
    obtain_resources(failure);    /* Usually stack_protect or stack_reserve. */
    if (failure_p(reason ← sigsetjmp(cleanup, 1)))    /* Establish a new long-jump site. */
      unwind(failure, reason, false, n);    /* Clean up and abort. */
    use_resources(&cleanup);    /* ... */
    free_resources();    /* Usually stack_clear — only reached if there was no error. */
  }
#endif
```

18. An error which is handled throughout this code-base but cannot actually occur is external interruption. When signal handling is written and an interrupt occurs this *Interrupt* flag will be set to *true* and any potentially-unbounded or otherwise long-running operation will halt.

19. ⟨ Global variables 15 ⟩ +≡
unique bool *Interrupt* ← *false*;

20. ⟨ External symbols 16 ⟩ +≡
extern unique bool *Interrupt*;

21. Memory. It is possible to compute without recourse to core memory but it isn't very interesting. Memory resources are divided by `LossLess` into two categories, fixed-size *atoms* and dynamic *segments*. Segments are allocated in co-ordination with the kernel (if there is one) at arbitrary locations by a memory manager which at this time is C's *malloc*.

The very first segment allocated during initialisation is the "heap" from which atoms are allocated. When all the atoms in a heap have been taken an attempt to allocate another will cause garbage collection to reclaim any discarded atoms first (this is also when unused segments are returned to the master allocator).

⟨ Function declarations 21 ⟩ ≡

```
void *mem_alloc(void *, size_t, size_t, sigjmp_buf *);
```

See also sections 29, 34, 46, 56, 78, 85, 88, 96, 107, 111, 114, 121, 127, 132, 135, 142, 147, 161, 184, 191, 228, 248, 281, 295, 320, 331, and 335.

This code is used in sections 2 and 3.

22.

```
void *mem_alloc(void *old, size_t length, size_t align, sigjmp_buf *failure)
{
  void *r;
  if (¬align) r ← realloc(old, length);
  else {
    assert(old ≡ Λ);
    if (((align − 1) & align) ≠ 0 ∨ align ≡ 0) siglongjmp (*failure, LERR_INCOMPATIBLE);
    if ((length & (align − 1)) ≠ 0) siglongjmp (*failure, LERR_INCOMPATIBLE);
    r ← aligned_alloc(align, length);
  }
  if (r ≡ Λ) siglongjmp (*failure, LERR_OOM);
  return r;
}
```

23. There is a lot of blank space here for future versions of `LossLess` to fill in.

24. Atoms. Although allocated within a heap which is itself a segment, atoms are (nearly) the most fundamental datum out of which nearly everything else in `LossLess` is constructed, and so they are described here first. An atom consists of two cells (so it's name has already a mistake) the size of a data pointer[1] and "is tagged".

An atom's tag is located in such a way that it has no bearing on the location or size of the atom itself. On machines with a large address space the tag can be squeezed into the actual address of the atom's or as is currently the case in `LossLess` at an offset within the heap relative to the atom.

The tag is used to identify what the data is in each half of the atom. In particular the garbage collector must be able to identify cells which hold a pointer to another atom or set of atoms, so that they will be marked or copied during collection and not reclaimed.

Because atoms are identified by a real memory address and the size of each atom is itself two full memory addresses, each atom is always allocated on a 4, 8 or 16-byte boundary (for 16, 32 and 64-bit addressing), thus the lower 2, 3 or 4 bits of the address are known to always be zero. `LossLess` takes advantage of this by using these bits to identify values which do not in fact require heap storage.

If the lowest bit of a pointer (**cell**) is set it identifies one of these *special* variables. Five global constants with the odd values 1, 3, 5, 7 & 13 (9 & 11 are unused). The value 15 (0xf or 0*b1111*) is even more special: If the lowest 4 bits are exactly this then the rest of the pointer is not all zeros but a fixed-width (signed) integer value.

Finally `NIL` comes about and makes nothing even more special still. `NIL` is a **cell** with the literal value zero and so it looks like, and in fact is, a real address. It even looks like and *usually* is, C's Λ (`NULL`) pointer. However Λ is a C pointer and need not be zero even though zero *from a literal value in the* C *source code* is Λ, the "null pointer". Clear?

This answer to a related question on Stack Overflow clarifies the situation (emphasis added): The "... 'question about how one would assign 0 address to a pointer' formally has no answer. You simply **can't assign a specific address to a pointer in** C/C++. However, in the realm of implementation-defined features, the explicit integer-to-pointer conversion is intended to have that effect."

To keep these almost-but-not-quite-exactly-unlike-zero values straight transforming between a **cell** and a C **pointer** is always explicitly cast even though they occupy the "same" storage.

#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 `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*) ((**intptr_t**)(*O*) ≡ 0)
#define *special_p*(*O*) (*null_p*(*O*) ∨ ((**intptr_t**)(*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 *fix_p*(*O*) (((*O*) & `FIXED`) ≡ `FIXED`)
#define *defined_p*(*O*) (¬*undefined_p*(*O*))

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

[1] An instruction pointer does not necessarily need to be related in any way to a data pointer although on the most common architectures they are the same in practice.

25. The tag of an atom is used by garbage collection to keep track of atoms which are in use and/or have been partially scanned. The remainder of the tag identifies the contents of each of the atom's cells. This is known as the atom's *format*. In total the tag is 8 bits wide — 2 for garbage collector state (LTAG_LIVE & LTAG_DONE) and 6 for the format.

In practice of course everything in a modern computer has an inherent order but neither half of an atom is considered greater or lesser than the other and so they are referred to with the unfamiliar Latin words for left and right, *sinister* and *dexter*[1] to confound the reader's (and writer's) inherent mental bias while still establishing an order with which each concern can be handled in this source code, the intuitive "sindex".

The one format in which the order of each half does "matter" is the *pair* which for histerical raisins labels the sinister half the *car* and and the dexter half the *cdr*. The otherwise fruitless distinction is made between sin/dex and car/cdr to make it clear when an atom is a real pair and when an atom is to be treated as opaque.

The atomic formats are broadly categorised into four groups based on whether each half is or isn't a pointer to an atom and the first 2 of the 6 bits (LTAG_DSIN & LTAG_DDEX) hold this information. The remaining 4 bits are (mostly) arbitrary.

ATOM_TO_TAG finds an atom's tag and is defined below after the heap storage (in which it's located) is introduced.

```
#define LTAG_LIVE 0x80   /* Atom is referenced from a register. */
#define LTAG_DONE 0x40   /* Atom has been partially scanned. */
#define LTAG_DSIN 0x20   /* Atom's sin half points to an atom. */
#define LTAG_DDEX 0x10   /* Atom's dex half points to an atom. */
#define LTAG_BOTH (LTAG_DSIN | LTAG_DDEX)
#define LTAG_FORM (LTAG_BOTH | 0x0f)
#define LTAG_TDEX 0x02   /* A tree is threadable in its dex halves. */
#define LTAG_TSIN 0x01   /* A tree is threadable in its sin halves. */
#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_DONE)
#define ATOM_CLEAR_MORE_M(O) (TAG_SET_M((O), TAG(O) & ~LTAG_DONE))
#define ATOM_SET_MORE_M(O) (TAG_SET_M((O), TAG(O) | LTAG_DONE))
#define ATOM_FORM(O) (TAG(O) & LTAG_FORM)
#define ATOM_SIN_DATUM_P(O) (TAG(O) & LTAG_DSIN)
#define ATOM_DEX_DATUM_P(O) (TAG(O) & LTAG_DDEX)
#define ATOM_SIN_THREADABLE_P(O) (TAG(O) & LTAG_TSIN)
#define ATOM_DEX_THREADABLE_P(O) (TAG(O) & LTAG_TDEX)
```

⟨ Type definitions 6 ⟩ +≡

```
typedef unsigned char Otag;
typedef struct {
  cell sin, dex;
} Oatom;
```

[1] It also helps that it's a pair whose acronyms are both three runes long, which was not really the primary concern.

26. These are the formats known to `LossLess`. The numeric value of each format is relevent except (sort of) `FORM_NONE` which is zero and the rope and tree formats, which are carefully chosen so that atoms with these formats are *polymorphic*, which means they are distinct but implemented with (mostly) the same API and (mostly) the same implementation to do (mostly) the same thing.

Unallocated atoms' tags are initialised to `FORM_NONE`. Allocated atoms' tags will be one of the other values here.

```
#define FORM_NONE (LTAG_NONE | 0x00)
#define FORM_ARRAY (LTAG_NONE | 0x01)
#define FORM_COLLECTED (LTAG_NONE | 0x02)
#define FORM_FIX (LTAG_NONE | 0x03)
#define FORM_HEAP (LTAG_NONE | 0x04)
#define FORM_KEYTABLE (LTAG_NONE | 0x05)
#define FORM_RECORD (LTAG_NONE | 0x06)
#define FORM_RUNE (LTAG_NONE | 0x07)
#define FORM_SEGMENT_INTERN (LTAG_NONE | 0x08)
#define FORM_SYMBOL (LTAG_NONE | 0x09)
#define FORM_SYMBOL_INTERN (LTAG_NONE | 0x0a)

#define FORM_PRIMITIVE (LTAG_DDEX | 0x00)
#define FORM_SEGMENT (LTAG_DDEX | 0x01)

#define FORM_PAIR (LTAG_BOTH | 0x00)
#define FORM_APPLICATIVE (LTAG_BOTH | 0x01)
#define FORM_ENVIRONMENT (LTAG_BOTH | 0x02)
#define FORM_NOTE (LTAG_BOTH | 0x03)
#define FORM_OPERATIVE (LTAG_BOTH | 0x04)

#define FORM_ROPE (LTAG_BOTH | 0x08)
#define FORM_TROPE_SIN (LTAG_BOTH | 0x09)
#define FORM_TROPE_DEX (LTAG_BOTH | 0x0a)
#define FORM_TROPE_BOTH (LTAG_BOTH | 0x0b)
#define FORM_TREE (LTAG_BOTH | 0x0c)
#define FORM_TTREE_SIN (LTAG_BOTH | 0x0d)
#define FORM_TTREE_DEX (LTAG_BOTH | 0x0e)
#define FORM_TTREE_BOTH (LTAG_BOTH | 0x0f)
```

27. Predicates. Generally objects have a simple, even 1:1 mapping between them and a single format. Notable exceptions are symbols and segments which may be "interned", and trees which can masquerade as ropes or as doubly-linked lists.

Segments also form the basis for some other objects including arrays and records. When a segment is allocated its address is stored in one half of an atom and so the other half is available for use by some of the objects which are built on top of segments.

`LossLess` data are always held in C in storage or variables with type **cell** and so C's limited type validation is bypassed. To somewhat compensate for this the format of any **cell** arguments to C functions are indicated by initially calling *assert* with a (possibly qualified) predicate.

#define $\mathit{form}(O)$ $(\mathtt{TAG}(O)\,\&\,\mathtt{LTAG\_FORM})$
#define $\mathit{form\_p}(O,F)$ $(\neg \mathit{special\_p}(O) \land \mathit{form}(O) \equiv \mathtt{FORM\_}\#\#F)$
#define $\mathit{pair\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{PAIR}))$
#define $\mathit{array\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{ARRAY}))$
#define $\mathit{null\_array\_p}(O)$ $((O) \equiv \mathit{Null\_Array})$
#define $\mathit{collected\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{COLLECTED}))$
#define $\mathit{environment\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{ENVIRONMENT}))$
#define $\mathit{keytable\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{KEYTABLE}) \lor \mathit{null\_array\_p}(O))$
#define $\mathit{note\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{NOTE}))$
#define $\mathit{record\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RECORD}))$
#define $\mathit{rune\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RUNE}))$
#define $\mathit{segment\_intern\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{SEGMENT\_INTERN}))$
#define $\mathit{segment\_stored\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{SEGMENT}))$
#define $\mathit{segment\_p}(O)$ $(\mathit{segment\_intern\_p}(O) \lor \mathit{segment\_stored\_p}(O))$
#define $\mathit{symbol\_intern\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{SYMBOL\_INTERN}))$
#define $\mathit{symbol\_stored\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{SYMBOL}))$
#define $\mathit{symbol\_p}(O)$ $(\mathit{symbol\_intern\_p}(O) \lor \mathit{symbol\_stored\_p}(O))$
#define $\mathit{character\_p}(O)$ $(\mathit{eof\_p}(O) \lor \mathit{rune\_p}(O))$
#define $\mathit{arraylike\_p}(O)$ $(\mathit{array\_p}(O) \lor \mathit{keytable\_p}(O) \lor \mathit{record\_p}(O))$
#define $\mathit{pointer\_p}(O)$ $(\mathit{segment\_stored\_p}(O) \lor \mathit{arraylike\_p}(O) \lor \mathit{form\_p}((O),\mathtt{HEAP}))$
#define $\mathit{primitive\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{PRIMITIVE}))$
#define $\mathit{closure\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{APPLICATIVE}) \lor \mathit{form\_p}((O),\mathtt{OPERATIVE}))$
#define $\mathit{program\_p}(O)$ $(\mathit{closure\_p}(O) \lor \mathit{primitive\_p}(O))$
#define $\mathit{applicative\_p}(O)$ $((\mathit{closure\_p}(O) \land \mathit{form\_p}((O),\mathtt{APPLICATIVE})) \lor \mathit{primitive\_applicative\_p}(O))$
#define $\mathit{operative\_p}(O)$ $((\mathit{closure\_p}(O) \land \mathit{form\_p}((O),\mathtt{OPERATIVE})) \lor \mathit{primitive\_operative\_p}(O))$

28. A record is an array of named cells and optionally a segment. Records for internal use are identified by a (negative) integer, user records are not made available yet but will likely be identified by a symbol or closure.

TODO: Move *fix* into the macro.

#define `RECORD_ROPE_ITERATOR` -1
#define `RECORD_ENVIRONMENT_ITERATOR` -2
#define `RECORD_LEXEME` -3
#define `RECORD_LEXAR` -4
#define `RECORD_SYNTAX` -5
#define $\mathit{rope\_iter\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RECORD}) \land \mathit{record\_id}(O) \equiv \mathit{fix}(\mathtt{RECORD\_ROPE\_ITERATOR}))$
#define $\mathit{lexeme\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RECORD}) \land \mathit{record\_id}(O) \equiv \mathit{fix}(\mathtt{RECORD\_LEXEME}))$
#define $\mathit{lexar\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RECORD}) \land \mathit{record\_id}(O) \equiv \mathit{fix}(\mathtt{RECORD\_LEXAR}))$
#define $\mathit{syntax\_p}(O)$ $(\mathit{form\_p}((O),\mathtt{RECORD}) \land \mathit{record\_id}(O) \equiv \mathit{fix}(\mathtt{RECORD\_SYNTAX}))$

29. Fixed-size Integers. No operators are yet made available to work with any sort of numbers, however the fixed-size small integers are minimally defined here for the use of the few objects which do need them internally.

TODO: Figure out negatives vs. logical/arithmetic shift vs. complements.

#define *fix_value*(O) ((**fixed**)((O) $\gg$ `FIX_SHIFT`))

⟨ Function declarations 21 ⟩ +≡
  cell *fix*(**intmax_t**);

30. **cell** *fix*(**intmax_t** *val*)
  {
    cell *r*;
    assert(*val* $\geq$ `FIX_MIN` $\wedge$ *val* $\leq$ `FIX_MAX`);
    r ← `FIXED`;
    r |= *val* $\ll$ `FIX_SHIFT`;
    return *r*;
  }

31. Heap. A heap is stored as a (singly-linked) list of pages. Each page has a **Oheap** header (as well as the transparent **Osegment** header because every heap page is allocated as a segment) followed immediately by the tags, possibly a gap and then the atoms of that heap extending all the way to the end of the page. An allocation request considers each page in turn until the first page is found with an atom available.

Initially there is a single heap consisting of a single page. When there are no more unused atoms left to allocate in any page garbage collection is performed which reports how many unused atoms were reclaimed. If there are no spare atoms available then another page is allocated and attached to the heap at the back of the list.

Pages within a heap are allocated automatically. At present they will never be detached from the list and reclaimed, and there is no way to initialise another heap. These abilities will be added when `LossLess` grows threads.

If a compacting garbage collector is being used (again, there is no way to actually convert a heap to compacting or create one yet but the ability will be necessary to support threads) then heap pages are allocated two at a time in pairs[1] which are pointed to each other in addition to their list link. Moreover each second page-half is linked in a list *backwards* for a trivial optimisation later.

Allocation in a compacting garbage collector is extremely fast at the expense of having half of each heap pair empty: the pointer to the next free atom is simply returned and incremented (unless the page is full). When the heap is full any atoms which are in use are re-allocated in the same linear fashion from the dormant half and the heap is effectively turned up-side down making the dormant halves active and the active halves dormant (and empty).

```
#define HEAP_CHUNK 0x1000
#define HEAP_MASK 0x0fff
#define HEAP_BOOKEND (sizeof(Osegment) + sizeof(Oheap))
#define HEAP_LEFTOVER ((HEAP_CHUNK − HEAP_BOOKEND)/(TAG_BYTES + WIDE_BYTES))
#define HEAP_LENGTH ((int) HEAP_LEFTOVER)
#define HEAP_HEADER ((HEAP_CHUNK/WIDE_BYTES) − HEAP_LENGTH)
#define ATOM_TO_ATOM(O) ((Oatom *)(O))
#define ATOM_TO_HEAP(O) (SEGMENT_TO_HEAP(ATOM_TO_SEGMENT(O)))
#define ATOM_TO_INDEX(O) (((((intptr_t)(O)) & HEAP_MASK) ≫ CELL_SHIFT) − HEAP_HEADER)
#define ATOM_TO_SEGMENT(O) ((Osegment *)(((intptr_t)(O)) & ∼HEAP_MASK))
#define HEAP_TO_SEGMENT(O) (ATOM_TO_SEGMENT(O))
#define SEGMENT_TO_HEAP(O) ((Oheap *)(O)→address)
#define HEAP_TO_LAST(O) ((Oatom *)(((intptr_t) HEAP_TO_SEGMENT(O)) + HEAP_CHUNK))
#define ATOM_TO_TAG(O) (ATOM_TO_HEAP(O)→tag[ATOM_TO_INDEX(O)])
```

⟨ Type definitions 6 ⟩ +≡

```
struct Oheap {
    Oatom *free;
    struct Oheap *next, *pair;
    Otag tag[];
};
typedef struct Oheap Oheap;
```

32. There are (conceptually) exactly two heaps, one for the current thread (*Theap*) and one shared between all threads (*Sheap*). New atoms are allocated in the current thread's heap. Upon encountering an atom in another thread's heap that atom will be copied (by the owning thread) into the shared heap so that the requesting thread can gain access to it.

⟨ Global variables 15 ⟩ +≡

```
shared Oheap *Sheap ← Λ;   /* Process-wide shared heap. */
unique Oheap *Theap ← Λ;   /* Per-thread private heap. */
```

[1] Not to be confused with pair *objects*.

33. ⟨ External symbols 16 ⟩ +≡
extern shared Oheap ∗*Sheap*;
extern unique Oheap ∗*Theap*;

34. The accessors here should probably be renamed to *lsin* and *ldex* (TODO).

⟨ Function declarations 21 ⟩ +≡
cell *lcar*(**cell**);
cell *lcdr*(**cell**);
Otag *ltag*(**cell**);
void *lcar_set_m*(**cell**, **cell**);
void *lcdr_set_m*(**cell**, **cell**);
void *heap_init_sweeping*(**Oheap** ∗, **Oheap** ∗);
void *heap_init_compacting*(**Oheap** ∗, **Oheap** ∗, **Oheap** ∗);
Oheap ∗*heap_enlarge*(**Oheap** ∗, **sigjmp_buf** ∗);
cell *heap_alloc*(**Oheap** ∗, **sigjmp_buf** ∗);
cell *atom*(**Oheap** ∗, **cell**, **cell**, **Otag**, **sigjmp_buf** ∗);

35. The thread heap is initialised when `LossLess` is starting but the shared heap remains unallocated until it's required. For book-keeping purposes a heap is allocated in the form of a segment and as a segment is pointed to by an object on a heap the first atom allocated within a new heap page will be that object.

Note that there is currently no way to convert this sweeping heap into a compacting heap (and so it follows that the compacting code is largely untested).

⟨ Initialise storage 35 ⟩ ≡
Theap ← `SEGMENT_TO_HEAP`(*segment_alloc*(−1, `HEAP_CHUNK`, 1, `HEAP_CHUNK`, *failure*));
heap_init_sweeping(*Theap*, Λ);
segment_init(`HEAP_TO_SEGMENT`(*Theap*), *heap_alloc*(*Theap*, *failure*));
Sheap ← Λ;

See also sections 77, 95, and 150.
This code is used in section 333.

36. If a heap will use a compacting garbage collector then each of its pages is allocated alongside another page and atoms are copied between them.

On of each pair is linked into a list in one direction and the other pair in the other direction (and each half-page pair links to the other half in its *pair* pointer). In this way after the garbage collector has walked over the entire heap the heap's head pointer can be changed to the head at the other end, effectively turning the heap upside down and beginning allocation from within the newly-emptied half.

Each half of a compacting heap is initialised identically: every atom's tag is initialised to `FORM_NONE` and its free pointer is set to the first available atom above the header.

```
void heap_init_compacting(Oheap *heap, Oheap *prev, Oheap *pair)
{
  int i;
  heap→pair ← pair;
  heap→free ← (Oatom *)(((intptr_t) HEAP_TO_SEGMENT(heap)) + HEAP_CHUNK);
  pair→free ← (Oatom *)(((intptr_t) HEAP_TO_SEGMENT(pair)) + HEAP_CHUNK);
  for (i ← 0; i < HEAP_LENGTH; i++) {
    heap→free−−;
    ATOM_TO_TAG(heap→free) ← FORM_NONE;
    pair→free−−;
    ATOM_TO_TAG(pair→free) ← FORM_NONE;
    heap→free→sin ← heap→free→dex ← NIL;
    pair→free→sin ← pair→free→dex ← NIL;
  }
  heap→pair ← pair;
  pair→pair ← heap;
  if (prev ≡ Λ) {
    heap→next ← pair→next ← Λ;
  }
  else {
    if ((heap→next ← prev→next) ≠ Λ) {
      assert(heap→next→pair→next ≡ prev→pair);
      heap→next→pair→next ← heap→pair;
    }
    pair→next ← prev→pair;
    prev→next ← heap;
  }
}
```

37. A heap which is to be garbage collected with a non-moving mark and sweep algorithm uses less space and has no complicated linkage however collection and allocation are slower than on a compacting heap. The heap is likewise initialised by setting each available atom's tag to `FORM_NONE` but instead of a pointer that gets incremented free atoms are linked through their dex cell in a list that ends in `NIL` at the top of the (now full) heap.

```
void heap_init_sweeping(Oheap *heap, Oheap *prev)
{
  int i;

  heap→pair ← Λ;
  heap→free ← HEAP_TO_LAST(heap) − 1;
  ATOM_TO_TAG(heap→free) ← FORM_NONE;
  heap→free→sin ← heap→free→dex ← NIL;
  for (i ← 1; i < HEAP_LENGTH; i++) {
    heap→free −−;
    ATOM_TO_TAG(heap→free) ← FORM_NONE;
    heap→free→sin ← NIL;
    heap→free→dex ← (cell)(heap→free + 1);
  }
  if (prev ≡ Λ) heap→next ← Λ;
  else {
    heap→next ← prev→next;
    prev→next ← heap;
  }
}
```

38. Enlarging a heap is practically the same either way. When allocating two heaps there is no need to clean up the first in case the second allocation fails — garbage collection will eventually release the unused segment (if the out of memory condition doesn't abort the whole process first).

```
Oheap *heap_enlarge(Oheap *heap, sigjmp_buf *failure)
{
  Oheap *new, *pair;
  Osegment *snew, *spair;
  cell owner;

  if (heap→pair ≡ Λ) {
    snew ← segment_alloc(−1, HEAP_CHUNK, 1, HEAP_CHUNK, failure);
    new ← SEGMENT_TO_HEAP(snew);
    heap_init_sweeping(new, heap);
    owner ← heap_alloc(new, failure);
    ATOM_TO_TAG(owner) ← FORM_HEAP;
    pointer_set_m(owner, snew);
    segment_set_owner_m(owner, owner);
  }
  else {
    snew ← segment_alloc(−1, HEAP_CHUNK, 1, HEAP_CHUNK, failure);
    spair ← segment_alloc(−1, HEAP_CHUNK, 1, HEAP_CHUNK, failure);
    new ← SEGMENT_TO_HEAP(snew);
    pair ← SEGMENT_TO_HEAP(spair);
    heap_init_compacting(new, heap, pair);
    owner ← heap_alloc(new, failure);
    ATOM_TO_TAG(owner) ← FORM_HEAP;
    pointer_set_m(owner, snew);
    segment_set_owner_m(owner, owner);
    owner ← heap_alloc(new, failure);
    ATOM_TO_TAG(owner) ← FORM_HEAP;
    pointer_set_m(owner, spair);
    segment_set_owner_m(owner, owner);
  }
  return new;
}
```

39. Allocating from either type of heap is broadly similar too. In each case the list of heap pages is iterated over until the first one is found with a free atom, which is where they differ. In either case if no atom is found then the appropriate type of garbage collection is performed and, if necessary, the heap is enlarged by another page (or pair of pages).

The garbage collector will actually call back into *heap_alloc* to move atoms and upon completion but not to allocate *new* atoms, only those which already were on the heap and are being moved into a recently-freed location.

```
cell heap_alloc(Oheap *heap, sigjmp_buf *failure)
{
  Oheap *h, *next;
  cell r;
  next ← heap;
  if (heap→pair ≠ Λ) {
  allocate_incrementing:
    while (next ≠ Λ) {
      h ← next;
      if (ATOM_TO_HEAP(h→free) ≡ heap) return (cell) h→free++;
      next ← h→next;
    }
    assert(failure ≠ Λ);    /* UNREACHABLE during collection. */
    if (gc_compacting(heap, true) > 0) next ← heap;
    else next ← heap_enlarge(h, failure);    /* Will succeed or goto failure. */
    goto allocate_incrementing;
  }
  else {
  allocate_listwise:
    while (next ≠ Λ) {
      h ← next;
      if (¬null_p(h→free)) {
        r ← (cell) h→free;
        h→free ← (Oatom *) h→free→dex;
        ((Oatom *) r)→dex ← NIL;
        return r;
      }
      next ← h→next;
    }
    assert(failure ≠ Λ);    /* UNREACHABLE during collection. */
    if (gc_sweeping(heap, true) > 0) next ← heap;
    else next ← heap_enlarge(h, failure);    /* Will succeed or goto failure. */
    goto allocate_listwise;
  }
}
```

40. A pointer to a new atom returned from *heap_alloc* must be formatted before it can be used by *atom* or *cons*. If the new sin or dex value is an atom (as indicated by the new tag) then it may not yet be referenced by a live object and would be discarded if allocation performs garbage collection.

To avoid this there are two registers *Tmp_SIN* and *Tmp_DEX* which the value is saved into immediately prior to allocation. *Tmp_ier* which has already be mentioned is defined here and serves a similar purpose for arrays and segments.

```
⟨ Global variables 15 ⟩ +≡
  unique cell Tmp_SIN ← NIL;    /* Allocator's storage for SIN/CAR pointer. */
  unique cell Tmp_DEX ← NIL;    /* Allocator's storage for DEX/CDR pointer. */
  unique cell Tmp_ier ← NIL;    /* Other temporary safe storage. */
```

41. This function is one of the few with an unusual error handling mechanism to clear its two registers.

```
#define cons(A, D, F) (atom(Theap, (A), (D), FORM_PAIR, (F)))
cell atom(Oheap *heap, cell nsin, cell ndex, Otag ntag, sigjmp_buf *failure)
{
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(ntag ≠ FORM_NONE);
  if (ntag & LTAG_DSIN) Tmp_SIN ← nsin;
  if (ntag & LTAG_DDEX) Tmp_DEX ← ndex;
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) goto fail;
  r ← heap_alloc(heap, &cleanup);
  TAG_SET_M(r, ntag);
  ((Oatom *) r)→sin ← (ntag & LTAG_DSIN) ? Tmp_SIN : nsin;
  ((Oatom *) r)→dex ← (ntag & LTAG_DDEX) ? Tmp_DEX : ndex;
  Tmp_SIN ← Tmp_DEX ← NIL;
  return r;
fail: Tmp_SIN ← Tmp_DEX ← NIL;
  siglongjmp (*failure, reason);
}
```

42. These accessors are not macros in part because the assertions have proven helpful while debugging LossLess and will likely continue to do so but also to be able patch in the ability to trap access from one thread to another's heap or to the shared heap (which will need a read lock).

```
Otag ltag(cell o)
{
  assert(¬special_p(o));
  return TAG(o);
}
cell lcar(cell o)
{
  assert(¬special_p(o));
  return ((Oatom *) o)→sin;
}
cell lcdr(cell o)
{
  assert(¬special_p(o));
  return ((Oatom *) o)→dex;
}
void lcar_set_m(cell o, cell datum)
{
  assert(¬special_p(o));
  assert(¬ATOM_SIN_DATUM_P(o) ∨ defined_p(datum));
  ((Oatom *) o)→sin ← datum;
}
void lcdr_set_m(cell o, cell datum)
{
  assert(¬special_p(o));
  assert(¬ATOM_DEX_DATUM_P(o) ∨ defined_p(datum));
  ((Oatom *) o)→dex ← datum;
}
```

43. Few of these shorthand accessors are actually used and none of them is exposed at run-time but there is a whole page here before the next major section starts and their existence adds no run-time cost.

#define $lcaar(O)$ $(lcar(lcar(O)))$
#define $lcadr(O)$ $(lcar(lcdr(O)))$
#define $lcdar(O)$ $(lcdr(lcar(O)))$
#define $lcddr(O)$ $(lcdr(lcdr(O)))$
#define $lcaaar(O)$ $(lcar(lcar(lcar(O))))$
#define $lcaadr(O)$ $(lcar(lcar(lcdr(O))))$
#define $lcadar(O)$ $(lcar(lcdr(lcar(O))))$
#define $lcaddr(O)$ $(lcar(lcdr(lcdr(O))))$
#define $lcdaar(O)$ $(lcdr(lcar(lcar(O))))$
#define $lcdadr(O)$ $(lcdr(lcar(lcdr(O))))$
#define $lcddar(O)$ $(lcdr(lcdr(lcar(O))))$
#define $lcdddr(O)$ $(lcdr(lcdr(lcdr(O))))$
#define $lcaaaar(O)$ $(lcar(lcar(lcar(lcar(O)))))$
#define $lcaaadr(O)$ $(lcar(lcar(lcar(lcdr(O)))))$
#define $lcaadar(O)$ $(lcar(lcar(lcdr(lcar(O)))))$
#define $lcaaddr(O)$ $(lcar(lcar(lcdr(lcdr(O)))))$
#define $lcadaar(O)$ $(lcar(lcdr(lcar(lcar(O)))))$
#define $lcadadr(O)$ $(lcar(lcdr(lcar(lcdr(O)))))$
#define $lcaddar(O)$ $(lcar(lcdr(lcdr(lcar(O)))))$
#define $lcadddr(O)$ $(lcar(lcdr(lcdr(lcdr(O)))))$
#define $lcdaaar(O)$ $(lcdr(lcar(lcar(lcar(O)))))$
#define $lcdaadr(O)$ $(lcdr(lcar(lcar(lcdr(O)))))$
#define $lcdadar(O)$ $(lcdr(lcar(lcdr(lcar(O)))))$
#define $lcdaddr(O)$ $(lcdr(lcar(lcdr(lcdr(O)))))$
#define $lcddaar(O)$ $(lcdr(lcdr(lcar(lcar(O)))))$
#define $lcddadr(O)$ $(lcdr(lcdr(lcar(lcdr(O)))))$
#define $lcdddar(O)$ $(lcdr(lcdr(lcdr(lcar(O)))))$
#define $lcddddr(O)$ $(lcdr(lcdr(lcdr(lcdr(O)))))$

44. Segments. Memory can be allocated in any size (plus overhead) in a segment. A segment is allocated in two parts, the allocated memory itself and an atom on the heap to refer to it.

If the allocation is small enough and the object it's being allocated for permits it (eg. it has no particular memory alignment requirements) then the storage for the segment will be within the atom itself. This is used especially for short symbols and text.

The address of the allocation is referenced by a pointer object. This object is the atom on the heap that contains the allocation's address[1] and owing to the nature of the heap only half of the atom is needed — the other half is a "spare" datum that segment-like objects can use if necessary or helpful.

```
#define pointer(O) ((void *) lcar(O))
#define pointer_datum(O) (lcdr(O))
#define pointer_erase_m(O) (lcar_set_m((O), (cell) Λ))
#define pointer_set_datum_m(O, D) (lcdr_set_m((O), (cell)(D)))
#define pointer_set_m(O, D) (lcar_set_m((O), (cell)(D)))

#define segint_p(O) (segment_intern_p(O))
#define segint_address(O) (segint_base(O)→buffer)
#define segint_base(O) ((Ointern *)(O))
#define segint_header(O) ((long) 0)
#define segint_length(O) ((long) segint_base(O)→length)
#define segint_set_length_m(O, V) (segint_base(O)→length ← (V))
#define segint_owner(O) (O)
#define segint_stride(O) ((long) 1)

#define segbuf_base(O) ((Osegment *) pointer(O))
#define segbuf_address(O) (segbuf_base(O)→address)
#define segbuf_header(O) (segbuf_base(O)→header)
#define segbuf_length(O) (segbuf_base(O)→length)
#define segbuf_next(O) (segbuf_base(O)→next)
#define segbuf_owner(O) (segbuf_base(O)→owner)    /* ≡ O */
#define segbuf_prev(O) (segbuf_base(O)→prev)
#define segbuf_stride(O) (segbuf_base(O)→stride ? segbuf_base(O)→stride : 1)

#define segment_address(O) (segint_p(O) ? segint_address(O) : segbuf_address(O))
#define segment_base(O) (segint_p(O) ? segint_base(O) : segbuf_base(O))
#define segment_header(O) (segint_p(O) ? segint_header(O) : segbuf_header(O))
#define segment_length(O) (segint_p(O) ? segint_length(O) : segbuf_length(O))
#define segment_owner(O) (segint_p(O) ? segint_owner(O) : segbuf_owner(O))
#define segment_stride(O) (segint_p(O) ? segint_stride(O) : segbuf_stride(O))

#define segment_set_owner_m(O, N) do
        {
          assert(pointer_p(O));
          segbuf_owner(O) ← (N);
        }
        while (0)
```

⟨ Type definitions 6 ⟩ +≡

```
  struct Osegment {    /* Must remain pointer-aligned. */
    struct Osegment *next, *prev;    /* Linked list of all allocated segments. */
    half length, stride;    /* Notably absent: header size & alignment. */
    cell owner;    /* The referencing atom; cleared and re-set during garbage collection. */
    char address[];    /* Base address of the available space (occupies no header space). */
  };
  typedef struct Osegment Osegment;
```

[1] If LossLess ever needs plain pointers the same atomic structure with a new format will be used.

45. The process-wide global list of every allocated segment. As soon as memory is successfully allocated it's added to this list. When scanning for live objects during garbage collection fails to re-set the owner (which have all been set to `NIL` prior to scanning) this list can be scanned for unused allocations to release.

⟨ Global variables 15 ⟩ +≡
 shared Osegment ∗*Allocations* ← Λ;

46. ⟨ Function declarations 21 ⟩ +≡
 Osegment ∗*segment_alloc_imp*(**Osegment** ∗, **long**, **long**, **long**, **long**, **sigjmp_buf** ∗);
 cell *segment_init*(**Osegment** ∗, **cell**);
 cell *segment_new_imp*(**Oheap** ∗, **long**, **long**, **long**, **long**, **Otag**, **sigjmp_buf** ∗);
 void *segment_release_imp*(**Osegment** ∗, **bool**);
 void *segment_release_m*(**cell**, **bool**);
 cell *segment_resize_m*(**cell**, **long**, **long**, **sigjmp_buf** ∗);

47. The memory underlying an allocated segment is obtained through *segment_alloc* or its imp. The header size is -1 if the length and stride together define the size to be allocated *including* the segment's own header (this allows heap pages to (easily) be exactly one operating system page).

If the stride value is zero then the real stride to calculate with is one but zero is stored to indicate that if the segment is subsequently reduced in size enough it can be interned.

The rather unpleasant looking test before aborting with LERR_OOM takes advantage of *flagged arithmetic* if the C compiler allows for it. Flagged arithmetic uses the flags variously raised after actually performing CPU arithmetic to detect overflow and carry rather than defensively checking that the operands are within range prior to each operation. After removing the noise the function performed appears as $size \leftarrow \mathbf{sizeof}(\mathbf{Osegment}) + header + (length * stride)$.

If the allocation is new then it's inserted at the end of the *Allocations* list.

TODO: These arguments should be **size_t** type.

```
#define segment_alloc(H, L, S, A, F) segment_alloc_imp(Λ, (H), (L), (S), (A), (F))

Osegment *segment_alloc_imp(Osegment *old, long header, long length, long stride,
        long align, sigjmp_buf *failure)
{
  long cstride;
  size_t size;
  Osegment *r;

  assert(header ≥ −1 ∧ length ≥ 0 ∧ stride ≥ 0);
  if (header ≡ −1) header ← −sizeof(Osegment);
  assert(old ≡ Λ ∨ stride ≡ old→stride);
  assert(align ≡ 0 ∨ old ≡ Λ);
  cstride ← stride ? stride : 1;
  if (length > HALF_MAX ∨ stride > HALF_MAX ∨
        ckd_mul(&size, length, cstride) ∨
        ckd_add(&size, size, header) ∨ ckd_add(&size, size, sizeof(Osegment)))
    siglongjmp (*failure, LERR_OOM);
  r ← mem_alloc(old, size, align, failure);
  r→length ← length;
  if (old ≡ Λ) {
    r→stride ← stride;
    r→owner ← NIL;
  }
  if (Allocations ≡ Λ) Allocations ← r→next ← r→prev ← r;
  else {
    r→next ← Allocations;
    r→prev ← Allocations→prev;
    Allocations→prev→next ← r;
    Allocations→prev ← r;
  }
  return r;
}
```

48. In most cases segment allocations are made by this function which includes the support for creating an interned segment or it allocates the memory and returns a new atom pointing to it.

TODO: These arguments should (probably) *not* be **size_t** but they should probably not be **long** either.

```
#define segment_new(H, L, S, A, F) segment_new_imp(Theap, (H), (L), (S), (A), FORM_SEGMENT, (F))
  cell segment_new_imp(Oheap *heap, long header, long length, long stride, long align, Otag ntag,
          sigjmp_buf *failure)
  {
    cell r;
    long total;
    Osegment *s;
    assert(stride ≥ 0);
    assert(ntag ≠ FORM_NONE);
    if (ckd_add(&total, header, length)) siglongjmp (*failure, LERR_LIMIT);
    if (stride ≡ 0 ∧ total ≤ INTERN_BYTES) {
      assert(ntag ≡ FORM_SEGMENT);
      r ← atom(heap, NIL, NIL, FORM_SEGMENT_INTERN, failure);
      segint_set_length_m(r, length);
      return r;
    }
    Tmp_ier ← atom(heap, NIL, NIL, FORM_PAIR, failure);
    s ← segment_alloc(header, length, stride, align, failure);
    TAG_SET_M(Tmp_ier, ntag);
    ATOM_TO_ATOM(Tmp_ier)→sin ← (cell) s;
    s→owner ← Tmp_ier;
    Tmp_ier ← NIL;
    return s→owner;
  }
```

49. When a segment is going to be used as a heap it may not be able to allocate the pointer to it on an existing heap so the memory and atom are allocated directly and *segment_init* sets up their attributes correctly.

```
  cell segment_init(Osegment *seg, cell container)
  {
    assert(¬special_p(container));
    seg→owner ← container;
    ATOM_TO_TAG(container) ← FORM_HEAP;
    ATOM_TO_ATOM(container)→sin ← (cell) seg;
    ATOM_TO_ATOM(container)→dex ← NIL;
    return container;
  }
```

50. A segment can be resized by passing it as an argument to *segment_alloc_imp*. If the segment was and remains interned then the stored length is simply updated or if the segment was and remains allocated then the allocation is resized.

If a segment changes to/from being interned then a new allocation is requested and the relevant data copied.

```
cell segment_resize_m(cell o, long header, long delta, sigjmp_buf *failure)
{
  Osegment *new, *old;
  long i, nlength, nstride;
  cell r ← NIL;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(segment_p(o) ∨ arraylike_p(o));
  assert(delta ≥ −segment_length(o));
  if (ckd_add(&nlength, segment_length(o), delta)) siglongjmp (*failure, LERR_OOM);
  if (segment_intern_p(o) ∧ nlength ≤ INTERN_BYTES) {
    segint_set_length_m(o, nlength);
    return o;
  }
  else if ((arraylike_p(o) ∨ segment_stored_p(o)) ∧ (segbuf_base(o)→stride ∨ nlength > INTERN_BYTES))
    {
    old ← segbuf_base(o);
    nstride ← segment_stride(o);
    segment_release_m(o, false);
    new ← segment_alloc_imp(old, header, nlength, nstride, 0, failure);
    pointer_set_m(o, new);
    return o;
  }
  assert(header ≡ 0);
  assert(segment_stride(o) ≡ 0);
  assert(null_p(Tmp_ier));
  Tmp_ier ← o;
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, true, 0);
  r ← segment_new(0, nlength, 0, 0, &cleanup);
  if (segment_length(Tmp_ier) < nlength) nlength ← segment_length(Tmp_ier);
  for (i ← 0; i < nlength; i++) segment_address(r)[i] ← segment_address(Tmp_ier)[i];
  Tmp_ier ← NIL;
  return r;
}
```

51. When a segment is released explicitly the pointer to it which is in its heap atom is erased to avoid the possibility of using deallocated memory.

```
void segment_release_m(cell o, bool reclaim)
{
  assert(pointer_p(o));     /* Useful objects piggy-back on segments. */
  segment_release_imp(pointer(o), reclaim);
  pointer_erase_m(o);     /* For safety. */
}
```

52. When a segment is released by the garbage collector its heap atom has already been lost so it calls the release imp directly to remove the segment from the *Allocations* list and reclaim the underlying storage.

```
void segment_release_imp(Osegment *o, bool reclaim)
{
  if (o ≡ Allocations) Allocations ← o→next;
  if (o→next ≡ o) Allocations ← Λ;
  else o→prev→next ← o→next, o→next→prev ← o→prev;
  o→next ← o→prev ← o;       /* For safety. */
  if (reclaim) free(o);
}
```

53. Registers. To collect unused memory the garbage collector recursively scans the descendents of every live atom. An atom is considered live if it is referenced by an atom held in a *register*. `LossLess` defines a number of registers for various purposes which are explained as they are introduced.

⟨ Type definitions 6 ⟩ +≡
```
enum {
  LGCR_TMPSIN, LGCR_TMPDEX, LGCR_TMPIER,
  LGCR_NULL,
  LGCR_SYMBUFFER, LGCR_SYMTABLE,
  LGCR_STACK,
  LGCR_PROTECT_0, LGCR_PROTECT_1, LGCR_PROTECT_2, LGCR_PROTECT_3,
  LGCR_EXPRESSION, LGCR_ENVIRONMENT, LGCR_ACCUMULATOR, LGCR_ARGUMENTS, LGCR_CLINK,
  LGCR_OPERATORS,
  LGCR_USER, LGCR_COUNT
};
```

54. One register which is defined here is the user register which is not used by `LossLess` for any purpose but is made available by the `LossLess` library.

⟨ Global variables 15 ⟩ +≡
```
unique cell *Registers[LGCR_COUNT];
shared cell User_Register ← NIL;     /* Unused by LossLess — for library users. */
```

55. This list of pointers to the registers is re-initialised once per thread. Probably. The garbage collector updates the pointer if the atom moves.

⟨ Save register locations 55 ⟩ ≡
```
Registers[LGCR_TMPSIN] ← &Tmp_SIN;
Registers[LGCR_TMPDEX] ← &Tmp_DEX;
Registers[LGCR_TMPIER] ← &Tmp_ier;
Registers[LGCR_NULL] ← &Null_Array;
Registers[LGCR_SYMBUFFER] ← &Symbol_Buffer;
Registers[LGCR_SYMTABLE] ← &Symbol_Table;
Registers[LGCR_STACK] ← &Stack;
Registers[LGCR_PROTECT_0] ← Protect + 0;
Registers[LGCR_PROTECT_1] ← Protect + 1;
Registers[LGCR_PROTECT_2] ← Protect + 2;
Registers[LGCR_PROTECT_3] ← Protect + 3;
Registers[LGCR_EXPRESSION] ← &Expression;
Registers[LGCR_ENVIRONMENT] ← &Environment;
Registers[LGCR_ACCUMULATOR] ← &Accumulator;
Registers[LGCR_ARGUMENTS] ← &Arguments;
Registers[LGCR_CLINK] ← &Control_Link;
Registers[LGCR_OPERATORS] ← &Root;
Registers[LGCR_USER] ← &User_Register;
```
This code is used in section 333.

56. Garbage Collection. `LossLess` includes two garbage collectors. A small, simple but moderately slow mark-and-sweep collector and a larger and more complicated but faster compacting collector.

When the compacting collector moves a live atom it's replaced with a *collected* atom, sometimes called a tombstone, so that references to it can be correctly updated. This object does not show up outside the garbage collector.

```
#define collected_datum(O) (lcar(O))
#define collected_set_datum_m(O, V) (lcar_set_m((O), (V)))
```

⟨ Function declarations 21 ⟩ +≡

```
size_t gc_compacting(Oheap *, bool);
void gc_disown_segments(Oheap *);
cell gc_mark(Oheap *, cell, bool, cell *, size_t *);
size_t gc_reclaim_heap(Oheap *);
void gc_release_segments(Oheap *);
size_t gc_sweeping(Oheap *, bool);
```

57. To collect a heap by sweeping up the unused atoms

```
size_t gc_sweeping(Oheap *heap, bool segments)
{
  size_t count, remain;
  int i;
  Oatom *a;
  Oheap *p;
  assert((heap ≡ Theap) ∨ ((Sheap ≠ NIL) ∧ (heap ≡ Sheap)));
  count ← remain ← 0;
  p ← heap;
  while (p ≠ Λ) {
    remain += HEAP_LENGTH;
    p ← p→next;
  }
  if (segments) gc_disown_segments(heap);
  for (i ← 0; i < LGCR_COUNT; i++)
    if (¬special_p(*Registers[i])) *Registers[i] ← gc_mark(heap, *Registers[i], false, Λ, &count);
  p ← heap;
  while (p ≠ Λ) {
    a ← HEAP_TO_LAST(p);
    p→free ← (Oatom *) NIL;
    for (i ← 0; i < HEAP_LENGTH; i++) {
      a−−;
      if (ATOM_LIVE_P(a)) ATOM_CLEAR_LIVE_M(a);
      else {
        ATOM_TO_TAG(a) ← FORM_NONE;
        a→sin ← NIL;
        a→dex ← (cell) p→free;
        p→free ← a;
      }
    }
    p ← p→next;
  }
  count += gc_reclaim_heap(heap);
  if (segments) gc_release_segments(heap);
  return remain − count;
}
```

58.

```
size_t gc_compacting(Oheap *heap, bool segments)
{
  size_t count, remain;
  int i;
  Oheap *last, *p;
  assert((heap ≡ Theap) ∨ ((Sheap ≠ NIL) ∧ (heap ≡ Sheap)));
  count ← remain ← 0;
  p ← heap;
  while (p ≠ Λ) {
    last ← p;
    remain += HEAP_LENGTH;
    p→pair→pair ← Λ;
    p ← p→next;
  }
  if (segments) gc_disown_segments(heap);
  for (i ← 0; i < LGCR_COUNT; i++)
    if (¬special_p(Registers[i])) *Registers[i] ← gc_mark(last, *Registers[i], true, Λ, &count);
  count += gc_reclaim_heap(heap);
  if (segments) gc_release_segments(heap);
  p ← last;
  while (p ≠ Λ) {
    p→pair→free ← HEAP_TO_LAST(p→pair);
    for (i ← 0; i < HEAP_LENGTH; i−−) {
      ATOM_TO_TAG(−−p→pair→free) ← FORM_NONE;
        /* warning: operation on p→pair→free may be undefined */
      p→pair→free→sin ← p→pair→free→dex ← NIL;
    }
    p→pair→pair ← p;
    heap ← p;
    p ← p→next;
  }
  if (heap ≡ Theap) Theap ← last;
  else Sheap ← last;
  return remain − count;
}
```

59.

```
size_t gc_reclaim_heap(Oheap *heap)
{
  size_t count ← 0;
  Oheap *h;
  do {
    h ← heap;
    while (1) {
      segment_init(HEAP_TO_SEGMENT(h), heap_alloc(heap, Λ));
      count++;
      if (h→next ≡ Λ) break;
      h ← h→next;
    }
  } while (h→pair ≠ Λ ∧ (h ≡ Theap ∨ h ≡ Sheap));
  return count;
}
```

60. TODO: Use *heap* to determine whether this segment might be owned by another heap.

```
void gc_disown_segments(Oheap *heap Lunused)
{
  Osegment *s;
  s ← Allocations;
  while (1) {
    if (¬null_p(s→owner) ∧ (ATOM_TO_HEAP(s→owner)→pair ≡ Λ ∨ ATOM_TO_HEAP(s→owner)→pair→pair ≡
          Λ)) s→owner ← NIL;
    if ((s ← s→next) ≡ Allocations) break;
  }
}
```

61. TODO: Split in two? Rename?

```
void gc_release_segments(Oheap *heap Lunused)
{
  Osegment *s, *n;
  s ← Allocations;
  while (1) {
    n ← s→next;
    if (null_p(s→owner)) segment_release_imp(s, true);
    if (n ≡ Allocations) break;
    s ← n;
  }
}
```

62. **#define** $atom\_saved\_p(O)$ $(\texttt{ATOM\_TO\_HEAP}(O) \rightarrow pair \equiv \Lambda)$

```
cell gc_mark(Oheap *heap, cell next, bool compacting, cell *cycles, size_t *remain)
{
  bool remember;
  cell copied, parent, tmp;
  long i;

  remember ← (cycles ≠ Λ ∧ ¬null_p(*cycles));
  parent ← tmp ← NIL;
  while (1) {
    if (¬special_p(next) ∧ ¬ATOM_LIVE_P(next)) {
      (*remain)++;
      ATOM_SET_LIVE_M(next);
      if (¬compacting) copied ← next;
      else {⟨Move the atom to a new heap 63⟩}
      if (pointer_p(next)) segment_set_owner_m(next, copied);
      else if (primitive_p(next)) Iprimitive[primitive(next)].box ← next;
      next ← copied;
      if (ATOM_SIN_DATUM_P(next) ∧
          ATOM_DEX_DATUM_P(next)) {⟨Mark the car of a pair-like atom 68⟩}
      else if (ATOM_SIN_DATUM_P(next)) {⟨Begin marking a sin-ward atom 64⟩}
      else if (ATOM_DEX_DATUM_P(next)) {⟨Begin marking a dex-ward atom 66⟩}
      else if (arraylike_p(next)) {⟨Begin marking an array 71⟩}
    }
    else if (special_p(parent)) break;
    else if (ATOM_SIN_DATUM_P(parent) ∧ ATOM_DEX_DATUM_P(parent)) {
      if (ATOM_MORE_P(parent)) {⟨Continue marking a pair-like atom 69⟩}
      else {⟨Finish marking a pair-like atom 70⟩}
    }
    else if (ATOM_SIN_DATUM_P(parent)) {⟨Finish marking a sin-ward atom 65⟩}
    else if (ATOM_DEX_DATUM_P(parent)) {⟨Finish marking a dex-ward atom 67⟩}
    else if (arraylike_p(parent)) {
      i ← array_progress(parent);
      if (i < array_length(parent) − 1) {⟨Continue marking an array 72⟩}
      else {⟨Finish marking an array 73⟩}
    }
    else next ← parent;
  }
  if (collected_p(next)) next ← collected_datum(next);
  return next;
}
```

63. ⟨Move the atom to a new heap 63⟩ ≡

```
copied ← heap_alloc(heap, Λ);   /* TODO: Move last as the tail fills up? */
*ATOM_TO_ATOM(copied) ← *ATOM_TO_ATOM(next);
TAG_SET_M(copied, form(next));   /* Without GC flags. */
collected_set_datum_m(next, copied);
TAG_SET_M(next, FORM_COLLECTED);
```

This code is used in section 62.

64. ⟨ Begin marking a sin-ward atom 64 ⟩ ≡
$tmp \leftarrow$ `ATOM_TO_ATOM`($next$)→sin; /* Unlive or recursive. */
if ($remember \land \neg special\_p(tmp) \land$ `ATOM_LIVE_P`(tmp)) $gc\_serial(*cycles, tmp)$;
`ATOM_TO_ATOM`($next$)→$sin \leftarrow parent$;
$parent \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

65. ⟨ Finish marking a sin-ward atom 65 ⟩ ≡
$tmp \leftarrow parent$;
$parent \leftarrow$ `ATOM_TO_ATOM`(tmp)→sin;
if ($collected\_p(next)$) $next \leftarrow collected\_datum(next)$;
`ATOM_TO_ATOM`(tmp)→$sin \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

66. ⟨ Begin marking a dex-ward atom 66 ⟩ ≡
$tmp \leftarrow$ `ATOM_TO_ATOM`($next$)→dex; /* Unlive or recursive. */
if ($remember \land \neg special\_p(tmp) \land$ `ATOM_LIVE_P`(tmp)) $gc\_serial(*cycles, tmp)$;
`ATOM_TO_ATOM`($next$)→$dex \leftarrow parent$;
$parent \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

67. ⟨ Finish marking a dex-ward atom 67 ⟩ ≡
$tmp \leftarrow parent$;
$parent \leftarrow$ `ATOM_TO_ATOM`(tmp)→dex;
if ($collected\_p(next)$) $next \leftarrow collected\_datum(next)$;
`ATOM_TO_ATOM`(tmp)→$dex \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

68. ⟨ Mark the car of a pair-like atom 68 ⟩ ≡
`ATOM_SET_MORE_M`($next$);
$tmp \leftarrow$ `ATOM_TO_ATOM`($next$)→sin; /* Unlive or recursive. */
if ($remember \land \neg special\_p(tmp) \land$ `ATOM_LIVE_P`(tmp)) $gc\_serial(*cycles, tmp)$;
`ATOM_TO_ATOM`($next$)→$sin \leftarrow parent$;
$parent \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

69. Leave *parent* alone so we come back to this object after completing *dex*.

⟨ Continue marking a pair-like atom 69 ⟩ ≡
`ATOM_CLEAR_MORE_M`($parent$);
$tmp \leftarrow$ `ATOM_TO_ATOM`($parent$)→dex; /* Unlive or recursive. */
if ($remember \land \neg special\_p(tmp) \land$ `ATOM_LIVE_P`(tmp)) $gc\_serial(*cycles, tmp)$;
`ATOM_TO_ATOM`($parent$)→$dex \leftarrow$ `ATOM_TO_ATOM`($parent$)→sin;
if ($collected\_p(next)$) $next \leftarrow collected\_datum(next)$;
`ATOM_TO_ATOM`($parent$)→$sin \leftarrow next$;
$next \leftarrow tmp$;
This code is used in section 62.

70. ⟨ Finish marking a pair-like atom 70 ⟩ ≡

```
tmp ← ATOM_TO_ATOM(parent)→dex;
if (collected_p(next)) next ← collected_datum(next);
ATOM_TO_ATOM(parent)→dex ← next;
next ← parent;
parent ← tmp;
```

This code is used in section 62.

71. ⟨ Begin marking an array 71 ⟩ ≡

```
i ← 0;
if (array_length(next) > 0) {
  ATOM_SET_MORE_M(next);
  array_set_progress_m(next, i);
  tmp ← array_ref(next, i);    /* Unlive or recursive. */
  if (remember ∧ ¬special_p(tmp) ∧ ATOM_LIVE_P(tmp)) gc_serial(*cycles, tmp);
  array_set_m(next, i, parent);
  parent ← next;
  next ← tmp;
}
```

This code is used in section 62.

72. ⟨ Continue marking an array 72 ⟩ ≡

```
assert(ATOM_MORE_P(parent));    /* Not actually useful for arrays. */
i++;
tmp ← array_ref(parent, i);    /* Unlive or recursive. */
if (remember ∧ ¬special_p(tmp) ∧ ATOM_LIVE_P(tmp)) gc_serial(*cycles, tmp);
array_set_m(parent, i, array_ref(parent, i − 1));
if (collected_p(next)) next ← collected_datum(next);
array_set_m(parent, i − 1, next);
next ← tmp;
array_set_progress_m(parent, i);
```

This code is used in section 62.

73. ⟨ Finish marking an array 73 ⟩ ≡

```
ATOM_CLEAR_MORE_M(parent);
tmp ← array_ref(parent, i);
if (collected_p(next)) next ← collected_datum(next);
array_set_m(parent, i, next);
next ← parent;
parent ← tmp;
```

This code is used in section 62.

74. Structural Data. Two arrangements of memory available to `LossLess` have been introduced, fixed-size atoms and dynamically-sized segments. An atom without further specialisation is a standard pair for which a complete API is already available. This section describe formats which combine these into various useful structures.

Owing to its ubiquitous presence and aided by being at the top of the alphabet the first of these will be arrays.

75. Arrays.

⟨ Global variables 15 ⟩ +≡
```
shared cell Null_Array ← NIL;
```

76. ⟨ External symbols 16 ⟩ +≡
```
extern shared cell Null_Array;
```

77. ⟨ Initialise storage 35 ⟩ +≡
```
Null_Array ← array_new_imp(0, NIL, FORM_ARRAY, failure);
```

78.

```
#define array_progress(O) (fix_value(pointer_datum(O)))
#define array_set_progress_m(O, V) (pointer_set_datum_m((O), fix(V)))
#define array_length(O) (segment_length(O))
#define array_address(O) ((cell *) segment_address(O))
```

⟨ Function declarations 21 ⟩ +≡
```
cell array_new_imp(long, cell, Otag, sigjmp_buf *);
cell array_grow(cell, long, cell, sigjmp_buf *);
cell array_grow_m(cell, long, cell, sigjmp_buf *);
cell array_ref(cell, long);
void array_set_m(cell, long, cell);
```

79.
```
#define array_new(L, F) ((L) ≡ 0 ? Null_Array : array_new_imp((L), NIL, FORM_ARRAY, (F)))

cell array_new_imp(long length, cell fill, Otag ntag, sigjmp_buf *failure)
{
  cell r;
  long i;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(length ≥ 0);
  assert(null_p(Tmp_ier));
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, true, 0);
  Tmp_ier ← fill;      /* Safe because segment_new_imp won't use Tmp_ier. */
  r ← segment_new_imp(Theap, 0, length, sizeof(cell), sizeof(cell), ntag, &cleanup);
  if (defined_p(Tmp_ier) ∧ length > 0) {
    array_set_m(r, 0, Tmp_ier);
    for (i ← 1; i < length; i++) array_address(r)[i] ← array_address(r)[0];
  }
  Tmp_ier ← NIL;
  return r;
}
```

80. I think the interface between this and *segment_resize_m* kept growing bugs which is why they are currently independent.

```
cell array_grow(cell o, long delta, cell fill, sigjmp_buf *failure)
{
  static int Sobject ← 1, Sfill ← 0;
  long i, j, nlength, slength;
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(arraylike_p(o) ∧ o ≠ Stack);
  assert(delta ∧ delta ≥ −array_length(o));
  if (ckd_add(&nlength, array_length(o), delta)) siglongjmp (*failure, LERR_OOM);
  stack_protect(1, fill, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
  r ← array_new_imp(nlength, UNDEFINED, FORM_ARRAY, &cleanup);
  slength ← delta ≥ 0 ? array_length(SO(Sobject)) : nlength;
  for (i ← 0; i < slength; i++) array_set_m(r, i, array_ref(SO(Sobject), i));
  if (defined_p(fill)) {
    if (i < nlength) array_set_m(r, i, SO(Sfill));
    for (j ← i + 1; j < nlength; j++) array_address(r)[j] ← array_address(r)[i];
  }
  stack_clear(1);
  return r;
}
```

81.

```
cell array_grow_m(cell o, long delta, cell fill, sigjmp_buf *failure)
{
  static int Sfill ← 0;     /* Also Tmp_ier in case the stack is resizing. */
  bool isstack;
  cell r;
  long i, slength;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(arraylike_p(o));
  assert(delta ∧ delta ≥ −array_length(o));
  assert(null_p(Tmp_ier));
  Tmp_ier ← o;     /* Safe because Tmp_ier will never be an interned segment. */
  if (¬(isstack ← (o ≡ Stack))) stack_protect(1, fill, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, true, isstack ? 1 : 0);
  slength ← array_length(Tmp_ier);
  r ← segment_resize_m(Tmp_ier, sizeof(cell), delta, &cleanup);
  if (¬isstack ∧ delta > 0 ∧ defined_p(SO(Sfill))) {
    for (i ← slength; i < array_length(r); i++)
      if (i ≡ slength) array_set_m(r, i, SO(Sfill));
      else array_address(r)[i] ← array_address(r)[slength];
  }
  Tmp_ier ← NIL;
  if (¬isstack) stack_clear(1);
  return r;
}
```

82. **cell** *array_ref* (**cell** o, **long** *idx*)
{
  assert (*arraylike_p* (o));
  assert ($idx \geq 0 \land idx < array\_length(o)$);
  return *array_address* (o)[*idx*];
}

83. **void** *array_set_m*(**cell** o, **long** *idx*, **cell** d)
{
  assert (*arraylike_p* (o));
  assert ($idx \geq 0 \land idx < array\_length(o)$);
  assert (*defined_p* (d));
  $array\_address(o)[idx] \leftarrow d$;
}

84. Key-Based Lookup Table. K&R hash function as adapted by pdksh.

⟨ Type definitions 6 ⟩ +≡
  typedef uint32_t Vhash;

85. ⟨ Function declarations 21 ⟩ +≡
  Vhash *hash_cstr*(**char** ∗, **long** ∗, **sigjmp_buf** ∗);
  Vhash *hash_buffer*(**char** ∗, **long**, **sigjmp_buf** ∗);

86. **Vhash** *hash_cstr*(**char** ∗*buf*, **long** ∗*length*, **sigjmp_buf** ∗*failure*)

```
{
  Vhash r ← 0;
  char *p ← buf;
  while (*p ≠ '\0')
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    else r ← 33 * r + (unsigned char)(*p++);
  *length ← p − buf;
  return r;
}
```

87. Interned symbols call this with NULL but are small so it's safe to ignore interrupts.

```
Vhash hash_buffer(char *buf, long length, sigjmp_buf *failure)
{
  Vhash r ← 0;
  long i;
  assert(length ≥ 0);
  for (i ← 0; i < length; i++)
    if (Interrupt ∧ failure ≠ Λ) siglongjmp (*failure, LERR_INTERRUPT);
    else r ← 33 * r + (unsigned char)(*buf++);
  return r;
}
```

88. Maybe store meta-tuple's length in segment's other "unused" field, *stride*.

```
#define KEYTABLE_MINLENGTH 8
#define KEYTABLE_MAXLENGTH (INT_MAX ≫ 1)
#define keytable_free(O) (null_array_p(O) ? 0 : fix_value(array_ref((O), array_length(O) − 1)))
#define keytable_free_p(O) (keytable_free(O) > 0)
#define keytable_length(O) (null_array_p(O) ? (long) 0 : array_length(O) − 1)
#define keytable_ref(O, I) (array_ref((O), (I)))
#define keytable_set_free_m(O, V) (array_set_m((O), array_length(O) − 1, fix(V)))
```

⟨ Function declarations 21 ⟩ +≡
  cell *keytable_new*(**long**, **sigjmp_buf** ∗);
  cell *keytable_enlarge_m*(**cell**, **Vhash**(∗)(**cell**, **sigjmp_buf** ∗), **sigjmp_buf** ∗);
  void *keytable_remove_m*(**cell**, **long**);
  void *keytable_save_m*(**cell**, **long**, **cell**);
  int *keytable_search*(**cell**, **Vhash**, **int**(∗)(**cell**, **void** ∗, **sigjmp_buf** ∗), **void** ∗, **sigjmp_buf** ∗);

89. **cell** *keytable_new*(**long** *length*, **sigjmp_buf** **failure*)

```
{
  cell r;
  long f;
  assert(length ≥ 0);
  if (length ≥ KEYTABLE_MAXLENGTH) siglongjmp (*failure, LERR_LIMIT);
  else if (length ≡ 0) f ← 0;
  else if (length ≤ KEYTABLE_MINLENGTH) f ← (length ← KEYTABLE_MINLENGTH) − 1;
  else f ← (7 * (length ← 1 ≪ (high_bit(length) − 1)))/10;
  if (length ≡ 0) return Null_Array;
  r ← array_new_imp(length + 1, NIL, FORM_KEYTABLE, failure);
  keytable_set_free_m(r, f);
  return r;
}
```

90. **cell** *keytable_enlarge_m*(**cell** *o*, **Vhash**(**hashfn*)(**cell**, **sigjmp_buf** *), **sigjmp_buf** **failure*)

```
{
  static int Sobject ← 1, Sret ← 0;
  cell r ← NIL, s;
  long i, j, nlength;
  long nfree;
  Vhash nhash;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(keytable_p(o));
  nlength ← keytable_length(o);
  if (nlength ≥ (INT_MAX ≫ 2)) siglongjmp (*failure, LERR_LIMIT);
  if (nlength ≡ 0) nlength ← KEYTABLE_MINLENGTH;
  else nlength ≪= 1;
  stack_protect(2, o, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  SS(Sret, r ← keytable_new(nlength, &cleanup));
  nfree ← (7 * nlength)/10;    /* ⌊70%⌋ */
  for (i ← 0; i < keytable_length(SO(Sobject)); i++)
    if (Interrupt) siglongjmp (cleanup, LERR_INTERRUPT);
    else if (¬null_p((s ← array_ref(SO(Sobject), i)))) {
      nhash ← hashfn(s, &cleanup);
      for (j ← nhash % nlength; ¬null_p(keytable_ref(r, j)); j−−)
        if (j ≡ 0) j ← nlength − 1;
      array_set_m(r, j, s);
      nfree−−;
    }
  r ← SO(Sret);
  keytable_set_free_m(r, nfree);
  stack_clear(2);
  return r;
}
```

91. **void** *keytable_save_m*(**cell** *o*, **long** *idx*, **cell** *datum*)

```
{
  assert(keytable_p(o));
  assert(idx ≥ 0 ∧ idx < keytable_length(o));
  assert(keytable_free_p(o));
  assert(null_p(keytable_ref(o, idx)));
  array_set_m(o, idx, datum);
  keytable_set_free_m(o, keytable_free(o) − 1);
}
```

92. **void** *keytable_remove_m*(**cell** *o*, **long** *idx*)

```
{
  int i, j;
  assert(keytable_p(o));
  assert(idx ≥ 0 ∧ idx < keytable_length(o));
  assert(¬null_p(keytable_ref(o, idx)));
  i ← idx;
  while (1) {
    j ← i − 1;
    if (j ≡ 0) j ← keytable_length(o) − 1;
    array_set_m(o, i, array_ref(o, j));
    if (null_p(keytable_ref(o, i))) break;
    i ← j;
  }
  keytable_set_free_m(o, keytable_free(o) + 1);
}
```

93. Return value must not be used if $*failure \equiv$ `LERR_MISSING` $\wedge$ *keytable_full_p*(*o*).

```
int keytable_search(cell o, Vhash hash, int(*match)(cell, void *, sigjmp_buf *), void
      *ctx, sigjmp_buf *failure)
{
  int p, r;
  assert(keytable_p(o));
  if (null_array_p(o)) return FAIL;
  for (r ← hash % keytable_length(o); ¬null_p(keytable_ref(o, r)); r−−) {
    p ← match(keytable_ref(o, r), ctx, failure);
    if (p ≡ 0) return r;
    if (r ≡ 0) r ← keytable_length(o) − 1;
  }
  return r;
}
```

94. Symbols. *Symbol_Table* looks up symbols by hash. *symbol_buffer* stores TAG_SYMBOL content, atom has length/buffer-offset. TAG_SYMBOL_HERE has length in first byte of car, content in rest of car/cdr pair.

```
#define SYMBOL_CHUNK 0x1000
#define SYMBOL_MAX INT_MAX
#define SYMBOL_BUFFER_MAX LONG_MAX
#define Symbol_Buffer_Length (segment_length(Symbol_Buffer))
#define Symbol_Buffer_Base ((char *) segment_address(Symbol_Buffer))
#define Symbol_Table_Length (keytable_length(Symbol_Table))
#define Symbol_Table_ref(i) (keytable_ref(Symbol_Table, (i)))
```

⟨ Global variables 15 ⟩ +≡

```
shared cell Symbol_Buffer ← NIL;
shared cell Symbol_Table ← NIL;
shared int Symbol_Buffer_Free ← 0, Symbol_Table_Free ← 0;
```

95. ⟨ Initialise storage 35 ⟩ +≡

```
Symbol_Buffer ← segment_new_imp(Theap, 0, SYMBOL_CHUNK, sizeof(char), 0, FORM_SEGMENT, failure);
memset(segment_address(Symbol_Buffer), '\0', SYMBOL_CHUNK);    /* off-by-many? */
Symbol_Table ← keytable_new(0, failure);
```

96. ⟨ Function declarations 21 ⟩ +≡

```
int symbol_table_cmp(cell, void *, sigjmp_buf *);
long symbol_table_search(Vhash, Osymbol_compare, sigjmp_buf *);
Vhash symbol_table_rehash(cell s, sigjmp_buf *);
cell symbol_new_buffer(char *, long, sigjmp_buf *);
cell symbol_new_imp(Vhash, char *, long, sigjmp_buf *);
```

97.

```
#define symint_p(O) (symbol_intern_p(O))
#define symint_length(O) (((Ointern *)(O))→length)
#define symint_buffer(O) (((Ointern *)(O))→buffer)
#define symint_hash(O) (hash_buffer(symint_buffer(O), symint_length(O), Λ))
#define symbuf_length(O) ((long) lcar(O))
#define symbuf_set_length_m(O, V) (lcar_set_m((O), (V)))
#define symbuf_offset(O) ((long) lcdr(O))
#define symbuf_set_offset_m(O, V) (lcdr_set_m((O), (V)))
#define symbuf_store(O) ((Osymbol *)(Symbol_Buffer_Base + symbuf_offset(O)))
#define symbuf_buffer(O) (symbuf_store(O)→buffer)
#define symbuf_hash(O) (symbuf_store(O)→hash)
#define symbol_length(O) (symint_p(O) ? symint_length(O) : symbuf_length(O))
#define symbol_buffer(O) (symint_p(O) ? symint_buffer(O) : symbuf_buffer(O))
#define symbol_hash(O) (symint_p(O) ? symint_hash(O) : symbuf_hash(O))
```

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {
  Vhash hash;
  char buffer[];
} Osymbol;
typedef struct {
  char *buf;
  int length;
} Osymbol_compare;
```

98. Might need to be made interruptable.

```
int symbol_table_cmp(cell sym, void *ctx, sigjmp_buf *failure)
{
  Osymbol_compare *scmp ← ctx;
  int i;
  assert(symbol_p(sym));
  if (symbol_length(sym) > scmp→length) return 1;
  if (symbol_length(sym) < scmp→length) return 1;
  for (i ← 0; i < scmp→length; i++) {
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    if (symbol_buffer(sym)[i] > scmp→buf[i]) return 1;
    if (symbol_buffer(sym)[i] < scmp→buf[i]) return 1;
  }
  return 0;
}
```

99.
```
long symbol_table_search(Vhash hash, Osymbol_compare scmp, sigjmp_buf *failure)
{
  return keytable_search(Symbol_Table, hash, symbol_table_cmp, &scmp, failure);
}
```

100.
```
Vhash symbol_table_rehash(cell s, sigjmp_buf *failure Lunused)
{
  Vhash r;
  assert(symbol_p(s));
  r ← symbol_hash(s);
  return r;
}
```

101.
```
#define symbol_new_segment(O, F)
        (symbol_new_buffer(segment_address(O), segment_length(O), (F)))
#define symbol_new_const(O) (symbol_new_buffer((O), 0, Λ))
cell symbol_new_buffer(char *buf, long length, sigjmp_buf *failure)
{
  Vhash hash;
  assert(length ≥ 0);
  if (length ≡ 0) hash ← hash_cstr(buf, &length, failure);
  else hash ← hash_buffer(buf, length, failure);
  if (length > SYMBOL_MAX) siglongjmp (*failure, LERR_LIMIT);
  return symbol_new_imp(hash, buf, length, failure);
}
```

102.

```
cell symbol_new_imp(Vhash hash, char *buf, long length, sigjmp_buf *failure)
{
  static int Sret ← 0;
  cell new, r ← NIL;
  int boff, i, size, idx;
  Osymbol_compare scmp ← {buf, length};
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(length ≥ 0);
search: idx ← symbol_table_search(hash, scmp, failure);
  if (idx ≡ FAIL ∨ (null_p(keytable_ref(Symbol_Table, idx)) ∧ ¬keytable_free_p(Symbol_Table))) {
    new ← keytable_enlarge_m(Symbol_Table, symbol_table_rehash, failure);
    Symbol_Table ← new;
    goto search;
  }
  if (¬null_p(keytable_ref(Symbol_Table, idx))) return keytable_ref(Symbol_Table, idx);
  stack_reserve(1, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
  if (length ≤ WIDE_BYTES − 1) {⟨Create an interned symbol 103⟩}
  else {⟨Create a buffered symbol 104⟩}
  keytable_save_m(Symbol_Table, idx, r);
  stack_clear(1);
  return r;
}
```

103. ⟨Create an interned symbol 103⟩ ≡

```
SS(Sret, r ← atom(Theap, NIL, NIL, FORM_SYMBOL_INTERN, &cleanup));
symint_length(r) ← length;
for (i ← 0; i < length; i++) symbol_buffer(r)[i] ← buf[i];
```

This code is used in section 102.

104. ⟨Create a buffered symbol 104⟩ ≡

```
if (ckd_add(&size, sizeof(Osymbol), length)) siglongjmp (*failure, LERR_LIMIT);
while (size > Symbol_Buffer_Length ∨ Symbol_Buffer_Length − size < Symbol_Buffer_Free) {
  if (Symbol_Buffer_Length ≥ SYMBOL_BUFFER_MAX) siglongjmp (*failure, LERR_LIMIT);
  new ← segment_resize_m(Symbol_Buffer, 0, SYMBOL_CHUNK, &cleanup);
  Symbol_Buffer ← new;
}
boff ← Symbol_Buffer_Free;
Symbol_Buffer_Free += sizeof(Osymbol) + length;
SS(Sret, r ← atom(Theap, NIL, NIL, FORM_SYMBOL, &cleanup));
symbuf_set_offset_m(r, boff);
symbuf_set_length_m(r, length);
symbuf_hash(r) ← hash;
for (i ← 0; i < length; i++) symbol_buffer(r)[i] ← buf[i];
```

This code is used in section 102.

105. Trees & Double-Linked Lists.

Basic structure: An `LTAG_BOTH` atom with a datum in sin and link pair in dex. Link pair is another `LTAG_BOTH` atom who's contents are `NIL` or an object with the same form (ie. same threading attributes).

The actual format of the link pair is irrelevant to structure provided it's `LTAG_BOTH` so its a `FORM_TREE` with the *lower* sin and dex bits (which do not affect the garbage collecter) raised with the link in question is a thread.

A doubly-linked list looks identical and is distinguished by having its link pair atom be formatted as a `FORM_PAIR`.

The three formats tree, rope and doubly-linked list are identified by *tree_p*, *rope_p* and *dlist_p* respectively. Threading can be identified by *sin_threadable* or *dex_threadable* inserted into the predicate name as for example *tree_sin_threadable_p* (doubly-linked lists cannot be threaded). The shared accessors *treeish_p* (et al.) and *dryadic_p* identify an object as either a tree or a rope, or as one of all three.

Accessors which work on any of these objects use the *dryad* namespace while specific accessors use *tree* or *rope*.

NOT TRUE: Accessors to the threads themselves are within the *tree_thread* namespace (except for the predicates). After an atom has been identified as a rope or a tree the way they process their threads is the same so there is no need for a *rope_thread* namespace.

A rope's link pair atom is always a tree variant even though *dlist_p* will treat a rope who's link pair is a plain `FORM_PAIR` pair as a doubly-linked list. This arrangement is unintentional and not used.

#define $dryad\_datum(O)$ $(lcar(O))$
#define $dryad\_link(O)$ $(lcdr(O))$
#define $dryad\_sin(O)$ $(lcadr(O))$
#define $dryad\_dex(O)$ $(lcddr(O))$
#define $dryad\_sin\_p(O)$ $(\neg null\_p(dryad\_sin(O)))$
#define $dryad\_dex\_p(O)$ $(\neg null\_p(dryad\_dex(O)))$
#define $dryad\_set\_sin\_m(O,V)$ $(lcar\_set\_m(lcdr(O),(V)))$
#define $dryad\_set\_dex\_m(O,V)$ $(lcdr\_set\_m(lcdr(O),(V)))$

#define $dryadic\_p(O)$ $(\neg special\_p(O) \land (form(O)\ \&\ \texttt{FORM\_ROPE}) \equiv \texttt{FORM\_ROPE})$
#define $dlist\_p(O)$ $(dryadic\_p(O) \land pair\_p(dryad\_link(O)))$
#define $treeish\_p(O)$ $(dryadic\_p(O) \land \neg dlist\_p(O))$ /* Any tree or rope. */
#define $tree\_p(O)$ $(treeish\_p(O) \land (form(O)\ \&\ \texttt{FORM\_TREE}) \equiv \texttt{FORM\_TREE})$ /* Any tree. */
#define $plain\_tree\_p(O)$ $(treeish\_p(O) \land form\_p((O),\texttt{TREE}))$
#define $rope\_p(O)$ $(treeish\_p(O) \land (form(O)\ \&\ \texttt{FORM\_TREE}) \equiv \texttt{FORM\_ROPE})$ /* Any rope. */
#define $plain\_rope\_p(O)$ $(treeish\_p(O) \land form\_p((O),\texttt{ROPE}))$

106. The link of a threaded tree which is not NIL may be a descendent link or a thread to a sinward or dexward peer. The format of the link atom indicates whether a non-NIL link is real or a thread.

#define *treeish_sin_threadable_p*(O) (*treeish_p*(O) ∧ (*form*(O) & LTAG_TSIN))
#define *treeish_dex_threadable_p*(O) (*treeish_p*(O) ∧ (*form*(O) & LTAG_TDEX))
#define *tree_sin_threadable_p*(O) (*tree_p*(O) ∧ *treeish_sin_threadable_p*(O))
#define *tree_dex_threadable_p*(O) (*tree_p*(O) ∧ *treeish_dex_threadable_p*(O))
#define *tree_threadable_p*(O) (*tree_sin_threadable_p*(O) ∧ *tree_dex_threadable_p*(O))
#define *rope_sin_threadable_p*(O) (*rope_p*(O) ∧ *treeish_sin_threadable_p*(O))
#define *rope_dex_threadable_p*(O) (*rope_p*(O) ∧ *treeish_dex_threadable_p*(O))
#define *rope_threadable_p*(O) (*rope_sin_threadable_p*(O) ∧ *rope_dex_threadable_p*(O))

#define *treeish_sin_has_thread_p*(O) (*treeish_sin_threadable_p*(O) ∧ *dryad_sin_p*(O) ∧ (*form*(*dryad_link*(O)) & LTAG_TSIN))
#define *treeish_dex_has_thread_p*(O) (*treeish_dex_threadable_p*(O) ∧ *dryad_dex_p*(O) ∧ (*form*(*dryad_link*(O)) & LTAG_TDEX))
#define *tree_sin_has_thread_p*(O) (*tree_p*(O) ∧ *treeish_sin_has_thread_p*(O))
#define *tree_dex_has_thread_p*(O) (*tree_p*(O) ∧ *treeish_dex_has_thread_p*(O))
#define *rope_sin_has_thread_p*(O) (*rope_p*(O) ∧ *treeish_sin_has_thread_p*(O))
#define *rope_dex_has_thread_p*(O) (*rope_p*(O) ∧ *treeish_dex_has_thread_p*(O))

#define *tree_thread_set_sin_thread_m*(O) (TAG_SET_M(*dryad_link*(O), *form*(*dryad_link*(O) | LTAG_TSIN)))
#define *tree_thread_set_sin_live_m*(O) (TAG_SET_M(*dryad_link*(O), *form*(*dryad_link*(O) & ∼LTAG_TSIN)))
#define *tree_thread_set_dex_thread_m*(O) (TAG_SET_M(*dryad_link*(O), *form*(*dryad_link*(O) | LTAG_TDEX)))
#define *tree_thread_set_dex_live_m*(O) (TAG_SET_M(*dryad_link*(O), *form*(*dryad_link*(O) & ∼LTAG_TDEX)))

#define *tree_thread_live_sin*(O) (*treeish_sin_has_thread_p*(O) ? NIL : *dryad_sin*(O))
#define *tree_thread_live_dex*(O) (*treeish_dex_has_thread_p*(O) ? NIL : *dryad_dex*(O))
#define *tree_thread_next_sin*(O, F) (*anytree_next_sin*((O), (F)))
#define *tree_thread_next_dex*(O, F) (*anytree_next_dex*((O), (F)))

107. ⟨ Function declarations 21 ⟩ +≡
cell *anytree_next_sin*(**cell**, **sigjmp_buf** ∗);
cell *anytree_next_dex*(**cell**, **sigjmp_buf** ∗);
cell *dryad_node_new*(**bool**, **bool**, **bool**, **cell**, **cell**, **cell**, **sigjmp_buf** ∗);
cell *treeish_clone*(**cell**, **sigjmp_buf** ∗*failure*);
cell *treeish_edge_imp*(**cell**, **bool**, **sigjmp_buf** ∗);

108. Construction is the same process for all variants of dryad.

```
cell dryad_node_new(bool tree, bool sinward, bool dexward, cell datum, cell nsin, cell
        ndex, sigjmp_buf *failure)
{
  static int Sdatum ← 2, Snsin ← 1, Sndex ← 0;
  Otag ntag;
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  ntag ← tree ? FORM_TREE : FORM_ROPE;
  if (sinward) ntag |= LTAG_TSIN;
  if (dexward) ntag |= LTAG_TDEX;
  assert(null_p(nsin) ∨ form(nsin) ≡ ntag);
  assert(null_p(ndex) ∨ form(ndex) ≡ ntag);
  stack_protect(3, datum, nsin, ndex, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 2))) unwind(failure, reason, false, 3);
  r ← atom(Theap, SO(Snsin), SO(Sndex), FORM_TREE, &cleanup);
  r ← atom(Theap, SO(Sdatum), r, ntag, &cleanup);
  stack_clear(3);
  return r;
}
```

109. Finding the edge of a tree is the same regardless of threading: walk down a tree's links until the next node in the indicated direction is NIL.

```
#define treeish_sinmost(O, F) treeish_edge_imp((O), true, (F))
#define treeish_dexmost(O, F) treeish_edge_imp((O), false, (F))
  cell treeish_edge_imp(cell o, bool sinward, sigjmp_buf *failure)
  {
    cell r;
    assert(treeish_p(o));
    r ← o;
    while (sinward ? dryad_sin_p(r) : dryad_dex_p(r))
      if (Interrupt) siglongjmp(*failure, LERR_INTERRUPT);
      else if (null_p((o ← sinward ? dryad_sin(r) : dryad_dex(r)))) return r;
      else r ← o;
    return r;
  }
```

110.

```
#define anytree_next_imp(IN, OTHER)
        cell anytree_next_##IN(cell o, sigjmp_buf *failure)
        {
          cell r;
          assert(dryadic_p(o));
          r ← dryad_##IN(o);
          if (¬treeish_##IN##_threadable_p(o) ∨ ¬treeish_##IN##_has_thread_p(o)) return r;
          return treeish_##OTHER##most(r, failure);
        }
  anytree_next_imp(sin, dex)
  anytree_next_imp(dex, sin)
```

111. ⟨ Function declarations 21 ⟩ +≡

```
  void treeish_rethread_imp(cell, cell, Otag, cell);
  cell treeish_rethread_m(cell, bool, bool, sigjmp_buf *);
```

112.
```
cell treeish_rethread_m(cell o, bool sinward, bool dexward, sigjmp_buf *failure)
{
    cell head, next, prev, remember;
    Otag ntag;

    assert(treeish_p(o));
    ntag ← form(o) & FORM_TREE;
    head ← dryad_node_new(ntag ≡ FORM_TREE, treeish_sin_threadable_p(o), treeish_dex_threadable_p(o),
            NIL, o, NIL, failure);
    if (sinward) ntag |= LTAG_TSIN;
    if (dexward) ntag |= LTAG_TDEX;
    next ← head;
    remember ← NIL;
    while (1) {
        if (null_p(next)) break;
        prev ← tree_thread_live_sin(next);
        if (¬null_p(prev)) {
            while (¬(prev ≡ remember ∨ null_p(tree_thread_live_dex(prev))))
                prev ← tree_thread_live_dex(prev);
            if (prev ≠ remember) {    /* Insert or remove stack */
                dryad_set_dex_m(prev, next);
                tree_thread_set_dex_live_m(prev);
                next ← tree_thread_live_sin(next);    /* Go to left */
                continue;
            }
            else {
                dryad_set_dex_m(prev, NIL);
                tree_thread_set_dex_live_m(prev);
            }
        }
        if (treeish_sin_has_thread_p(next)) {
            dryad_set_sin_m(next, NIL);
            tree_thread_set_sin_live_m(next);
        }
        treeish_rethread_imp(next, prev, ntag, head);
        remember ← next;    /* Go to the right or up */
        next ← tree_thread_live_dex(next);
    }
    return dryad_sin(head);
}
```

113. **void** *treeish_rethread_imp*(**cell** *current*, **cell** *previous*, **Otag** *ntag*, **cell** *head*)

```
{
  TAG_SET_M(current, ntag);
  if (null_p(previous)) {
    dryad_set_sin_m(current, NIL);
    tree_thread_set_sin_live_m(current);
  }
  else if (current ≡ head) {
    dryad_set_dex_m(previous, NIL);
    tree_thread_set_dex_live_m(previous);
  }
  else {
    if (ntag & LTAG_TSIN ∧ ¬dryad_sin_p(current)) {
      dryad_set_sin_m(current, previous);
      tree_thread_set_sin_thread_m(current);
    }
    if (ntag & LTAG_TDEX ∧ ¬dryad_dex_p(previous)) {
      dryad_set_dex_m(previous, current);
      tree_thread_set_dex_thread_m(previous);
    }
  }
}
```

114. Doubly-linked lists. These piggy-pack on top of plain unthreaded trees. The list loops around on itself so the link nodes will never be `NIL` and care is taken to ensure a loop is not inserted “into” itself.

```
#define dlist_datum(o) (dryad_datum(o))
#define dlist_prev(o) (dryad_sin(o))
#define dlist_next(o) (dryad_dex(o))
```

⟨ Function declarations 21 ⟩ +≡

```
cell dlist_new(cell, sigjmp_buf *);
cell dlist_append_datum_m(cell, cell, sigjmp_buf *);
cell dlist_append_m(cell, cell);
cell dlist_clone(cell, sigjmp_buf *);
cell dlist_insert_datum_imp(cell, cell, bool, sigjmp_buf *);
cell dlist_insert_imp(cell, cell, bool, sigjmp_buf *);
cell dlist_remove_m(cell);
void dlist_set_next_m(cell, cell);
void dlist_set_prev_m(cell, cell);
```

115.

```
cell dlist_new(cell datum, sigjmp_buf *failure)
{
    cell r;
    r ← dryad_node_new(true, false, false, datum, NIL, NIL, failure);
    TAG_SET_M(dryad_link(r), FORM_PAIR);
    dryad_set_sin_m(r, r);
    dryad_set_dex_m(r, r);
    return r;
}
```

116.

```
void dlist_set_m(cell o, cell datum)
{
    assert(dlist_p(o));
    lcar_set_m(o, datum);
}
```

117.

```
#define dlist_set(DIRECTION, YIN, YANG)
        void dlist_set_##DIRECTION##_m(cell hither, cell yon)
        {
            assert(dlist_p(hither));
            assert(dlist_p(yon));
            YIN##_set_m(dryad_link(hither), yon);
            YANG##_set_m(dryad_link(yon), hither);
        }
dlist_set(next, lcdr, lcar)
dlist_set(prev, lcar, lcdr)
```

118. **#define** *dlist_prepend_m*(*O*, *L*) *dlist_append_m*(*dlist_prev*(*O*), (*L*))

```
cell dlist_append_m(cell o, cell l)
{
  cell after, before;
  assert(dlist_p(o));
  assert(dlist_p(l));
  after ← dlist_next(o);
  before ← dlist_prev(l);    /* Usually l. */
  dlist_set_next_m(o, l);
  dlist_set_prev_m(after, before);
  return l;
}
```

119. **#define** *dlist_prepend_datum_m*(*O*, *D*, *F*) *dlist_append_datum_m*(*dlist_prev*(*O*), (*D*), (*F*))

```
cell dlist_append_datum_m(cell o, cell d, sigjmp_buf *failure)
{
  static int Sobject ← 1, Sdatum ← 0;
  cell r ← NIL;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(dlist_p(o));
  assert(defined_p(d));
  stack_protect(2, o, d, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  r ← dlist_new(SO(Sdatum), &cleanup);
  r ← dlist_append_m(SO(Sobject), r);
  stack_clear(2);
  return r;
}
```

120. **cell** *dlist_remove_m*(**cell** *o*)

```
{
  cell next, prev;
  assert(dlist_p(o));
  prev ← dlist_prev(o);
  next ← dlist_next(o);
  if (prev ≡ next) return NIL;
  dlist_set_next_m(prev, next);
  return next;
}
```

121. Records. Not (yet) exposed to userspace, records use a key table to associate a name with an index into an array.

Ordinarily a key-based table is identified by being an array with a fixed integer in its spare slot indicating how many table entries are free. However if the free slot is arraylike then the (first) array is in fact the key-based table describing a record. Losing the number of free entries in the table is safe because a record is defined once and will never grow although care must be taken that searching a key-based table will not reference this value.

A record is defined using a key-based table to relate attribute names to an index value. An instance of a record is an array holding those values who's spare slot points to the key-based table used for lookup. That table is identified by pointing to itself.

In addition the records used for internally-defined objects (which use a slightly different name-to-index lookup mechanism) may optionally point to an arbitrary segment in the first array slot and this segment itself has a spare slot which is not wasted (it uses the pseudo-index -1).

```
#define RECORD_MAXLENGTH (INT_MAX ≫ 1)
#define record_next(O) (array_ref((O), 0))
#define record_next_p(O) (segment_p(record_next(O)))    /* TODO: inadequate test! */
#define record_id(O) (record_next_p(O) ? pointer_datum(record_next(O)) : record_next(O))
#define record_base(O) (record_next_p(O) ? segment_address(record_next(O)) : (char *) Λ)
#define record_offset(O) (record_next_p(O) ? 1 : 0)
#define record_cell(O, I) (array_ref((O), (I) + 1))
#define record_set_cell_m(O, I, D) (array_set_m((O), (I) + 1, (D)))
#define record_object(T, O, A) (((T) record_base(O))→A)
#define record_set_object_m(T, O, A, D) (record_object((T), (O), (A)) ← (D))
```

⟨ Function declarations 21 ⟩ +≡

```
  cell record_new(cell, int, int, sigjmp_buf *);
```

122.

```
cell record_new(cell record_form,    /* The record form. */
  int array_length,    /* The width of each object. */
  int segment_length,    /* The length of an option segment. */
  sigjmp_buf *failure)
  {
    static int Sform ← 1, Sret ← 0;
    cell r ← NIL, tmp;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    assert(symbol_p(record_form) ∨ fix_p(record_form));
    assert(array_length ≥ 0);
    if (array_length ≥ RECORD_MAXLENGTH) siglongjmp (*failure, LERR_LIMIT);
    stack_protect(2, record_form, NIL, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
    if (segment_length > 0) array_length++;
    r ← array_new_imp(array_length + 1, NIL, FORM_RECORD, &cleanup);
    if (segment_length > 0) {
      SS(Sret, r);
      tmp ← segment_new(0, segment_length, 1, 0, &cleanup);
      pointer_set_datum_m(tmp, SO(Sform));
      r ← SO(Sret);
      array_set_m(r, 0, tmp);
    }
    else array_set_m(r, 0, SO(Sform));
    stack_clear(2);
    return r;
  }
```

123. Valuable Data.

124. Characters (Runes). To avoid constantly mistyping the shift key when referring to UTF-8 the suffix 'o' is used instead creating "utfo" for "UTF-Octal" (although utfio means "UTF I/O"). To continue the theme *utfh* is UTF-Hexadecimal or UTF-16 and *utft* is UTF-Trigintadyodecimal.

```
#define UCP_MAX 0x10ffff
⟨ Global variables 15 ⟩ +≡
  struct {
    char size;        /* How many bits are supplied by this byte. */
    uint8_t data;     /* Mask: Encoded bits. */
    uint8_t lead;     /* Mask: Leading bits which will be set. */
    int32_t max;      /* Maximum code-point value this many bytes can encode. */
  } UTFIO[] ← {
      {6, 0x3f, 0x80, 0x000000},     /* Continuation byte. */
      {7, 0x7f, 0x00, 0x00007f},     /* Single ASCII byte. */
      {5, 0x1f, 0xc0, 0x0007ff},     /* Start of 2-byte encoding. */
      {4, 0x0f, 0xe0, 0x00ffff},     /* Start of 3-byte encoding. */
      {3, 0x07, 0xf0, 0x10ffff},     /* Start of 4-byte encoding. */
  };
```

125. Going into the parser from `UTFIO_COMPLETE` (ie. 0) will exit in any one of these states.

```
⟨ Type definitions 6 ⟩ +≡
  typedef enum {
    UTFIO_COMPLETE,      /* A code point is complete (or unstarted). */
    UTFIO_INVALID,       /* Encountered an invalid byte/encoding. */
    UTFIO_BAD_CONTINUATION,     /* Continuation byte when expecting a starter byte. */
    UTFIO_BAD_STARTER,     /* Starter byte when expecting a continuation byte. */
    UTFIO_OVERLONG,      /* Overlong encoding was used. */
    UTFIO_SURROGATE,      /* Surrogate pair half was encoded. */
    UTFIO_PROGRESS,
      /* Byte is valid but more are required (a final error if EOF was premature). */
    UTFIO_EOF     /* EOF encountered prematurely. */
  } Vutfio_parse;
```

126. The last two code points of each plane are noncharacters: U+...FFFE and U+...FFFF (recall that the byte-order-mark has value U+FEFF) for a total of 34 code points in 17 planes. In addition, there is a contiguous range of another 32 noncharacter code points in the BMP (plane 0): U+FDD0–U+FDEF.

```
#define UCP_SURROGATE_MIN 0xd800    /* Values ≥ 2^16 in UTF-16 encoded text. */
#define UCP_SURROGATE_MAX 0xe000
#define UCP_NONBMP_MIN 0xfdd0    /* Contained within the "Arabic Presentation */
#define UCP_NONBMP_MAX 0xfdef    /* ... Forms-A block" by a historic accident. */
#define UCP_REPLACEMENT 0xfffd
#define UCP_REPLACEMENT_LENGTH 3
#define UCP_NONCHAR_MASK 0xfffe    /* The lowest bit doesn't matter. */
#define utfio_noncharacter_p(C) (utfio_nonplane_p(C) ∨ utfio_nonrange_p(C))
#define utfio_nonplane_p(C) (((C)→value & UCP_NONCHAR_MASK) ≡ UCP_NONCHAR_MASK)
#define utfio_nonrange_p(C) ((C)→value ≥ UCP_NONBMP_MIN ∧ (C)→value ≤ UCP_NONBMP_MAX)
#define utfio_overlong_p(C) ((C)→value ≤ UTFIO[(C)→offset − 1].max)
#define utfio_surrogate_p(C) ((C)→value ≥ UCP_SURROGATE_MIN ∧ (C)→value ≤ UCP_SURROGATE_MAX)
#define utfio_too_large_p(C) ((C)→value > UCP_MAX)
```

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {
    int32_t value;
    char offset, remaining;
    char buf[4];
    Vutfio_parse status;
} Outfio;
```

127. ⟨ Function declarations 21 ⟩ +≡

```
Vutfio_parse utfio_read(Outfio *, char);
Vutfio_parse utfio_reread(Outfio *, char);
Outfio utfio_scan_start(void);
Outfio utfio_write(int32_t);
```

128.

```
Outfio utfio_scan_start(void)
{
    Outfio r ← {0};
    return r;
}
```

129. **Vutfio_parse** *utfio_read*(**Outfio** *ctx, **char** *byte*)

```
{
  int32_t vbyte ← byte;
  int i;
  for (i ← 0; i < 4; i++) {
    if ((byte & ~UTFIO[i].data) ≠ UTFIO[i].lead) continue;
    else if (i ≡ 0) {
      if (ctx→remaining) ctx→remaining−−;
      else return ctx→status ← UTFIO_BAD_CONTINUATION;
    }
    else {
      if (ctx→remaining) return ctx→status ← UTFIO_BAD_STARTER;
      else ctx→remaining ← i − 1;
    }
    ctx→buf[(int) ctx→offset++] ← byte;
    ctx→value |= (vbyte & UTFIO[i].data) ≪ (6 * ctx→remaining);
    if (ctx→remaining) return ctx→status ← UTFIO_PROGRESS;
    else if (utfio_too_large_p(ctx) ∨ utfio_noncharacter_p(ctx))
      return ctx→status ← UTFIO_INVALID;
    else if (utfio_surrogate_p(ctx)) return ctx→status ← UTFIO_SURROGATE;
    else if (utfio_overlong_p(ctx)) return ctx→status ← UTFIO_OVERLONG;
    else return ctx→status ← UTFIO_COMPLETE;
  }
  return ctx→status ← UTFIO_INVALID;
}
```

130. Again but without error checking — only for use on known-valid UTF-8.

```
Vutfio_parse utfio_reread(Outfio *ctx, char byte)
{
  int32_t vbyte ← byte;
  int i;
  for (i ← 0; i < 4; i++)
    if ((byte & ~UTFIO[i].data) ≠ UTFIO[i].lead) continue;
    else {
      if (ctx→remaining) ctx→remaining−−;
      else ctx→remaining ← i − 1;
      ctx→buf[(int) ctx→offset++] ← byte;
      ctx→value ← (vbyte & UTFIO[i].data) ≪ (6 * ctx→remaining);
      ctx→status ← ctx→remaining ? UTFIO_PROGRESS : UTFIO_COMPLETE;
      return ctx→status;
    }
  abort();   /* UNREACHABLE */
}
```

131. **Outfio** *utfio_write*(**int32_t** *c*)

```
{
  int i;
  int32_t mask, next;
  Outfio r ← {0};
  assert(c ≥ 0 ∧ c ≤ UCP_MAX);
  r.status ← UTFIO_PROGRESS;
  r.value ← c;
  if (utfio_surrogate_p(&r)) r.status ← UTFIO_SURROGATE;
  else if (r.value < 0 ∨ utfio_noncharacter_p(&r)) r.status ← UTFIO_INVALID;
  else if (utfio_too_large_p(&r)) r.status ← UTFIO_OVERLONG;
  if (r.status ≡ UTFIO_PROGRESS) r.status ← UTFIO_COMPLETE;
  else c ← UCP_REPLACEMENT;
  if (c ≤ UTFIO[1].max) r.buf[(int) r.offset++] ← c;
  else {
    if (c ≤ UTFIO[2].max) r.remaining ← i ← 2;
    else if (c ≤ UTFIO[3].max) r.remaining ← i ← 3;
    else r.remaining ← i ← 4;
    for ( ; r.remaining; r.remaining−−) {
      mask ← UTFIO[r.remaining ≡ i ? i : 0].data;
      mask ≪= 6 ∗ (r.remaining − 1);
      next ← c & mask;
      next ≫= 6 ∗ (r.remaining − 1);
      next |= UTFIO[r.remaining ≡ i ? i : 0].lead;
      r.buf[(int) r.offset++] ← next;
    }
  }
  return r;
}
```

132. Of the 11 spare bits, 3 are used to store the length of the UTF-8 encoding and the other 8 a failure code. Lots of magic masks here.

Runes have come through one of the utfio functions above as a **Outfio** or are a **int32_t** to put through *utfio_write* for validation.

⟨ Function declarations 21 ⟩ +≡

```
cell rune_new_utfio(Outfio, sigjmp_buf ∗);
```

133. Assumes *ctx.status* is correct so only valid encodings must be `UTFIO_COMPLETE` (0). Note that any (invalid) character may have status `UTFIO_EOF` if `EOF` was encountered in the middle of a multi-byte code point.

Status `UTFIO_PROGRESS` indicates an otherwise acceptable code point which is split between two rope nodes.

```
#define UCPVAL(V) (((V) & 0x001fffff)) /* Only code point bits. */
#define UCPLEN(V) (((V) & 0x00e00000) >> 21)
#define UCPFAIL(V) (((V) & 0xff000000) >> 24) /* Only bottom 3 used. */
#define rune_raw(O) ((CELL_BITS >= 32) ? (int32_t) lcar(O) :
        ((((int32_t) lcar(O) & 0xffff) << 16) | ((int32_t) lcdr(O) & 0xffff)))
#define rune_failure_p(O) (!!UCPFAIL(rune_raw(O)))
#define rune_failure(O) (UCPFAIL(rune_raw(O)))
#define rune_parsed(O) (UCPLEN(rune_raw(O)))
#define rune(O) (rune_failure_p(O) ? UCP_REPLACEMENT : UCPVAL(rune_raw(O)))
#define rune_new_value(V, F) (rune_new_utfio(utfio_write(V), (F)))

  cell rune_new_utfio(Outfio ctx, sigjmp_buf *failure)
  {
    Oatom packed;

    ctx.value |= (ctx.offset << 21);
    ctx.value |= (ctx.status << 24);
    if (CELL_BITS >= 32) {
      packed.sin = (cell) ctx.value;
      packed.dex = NIL;
    }
    else {
      packed.sin = (cell)((ctx.value & 0xffff0000LL) >> 16);
      packed.dex = (cell)((ctx.value & 0x0000ffffLL) >> 0);
    }
    return atom(Theap, packed.sin, packed.dex, FORM_RUNE, failure);
  }
```

134. Ropes. If the rope contains only correctly-encoded valid code points the length is recorded (and the fact noted) in cplength. Likewise glength counts the number of whole glyphs. Neither of these features is implemented and so is liable to change.

```
#define rope_segment(O) (dryad_datum(O))
#define rope_base(O) ((Orope *) segment_address(rope_segment(O)))
#define rope_blength(O) ((long) segment_length(rope_segment(O)) − 1)
#define rope_cplength(O) (rope_base(O)→cplength)
#define rope_glength(O) (rope_base(O)→glength)
#define rope_buffer(O) (rope_base(O)→buffer)
#define rope_first(O, F) (treeish_sinmost((O), (F)))
#define rope_last(O, F) (treeish_dexmost((O), (F)))
#define rope_next(O, F) (anytree_next_dex((O), (F)))
#define rope_prev(O, F) (anytree_next_sin((O), (F)))
#define rope_byte(O, B) (rope_buffer(O)[(B)])
```

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {
  long cplength;
  long glength;
  char buffer[];
} Orope;
```

135. ⟨ Function declarations 21 ⟩ +≡

```
cell rope_node_new_clone(bool, bool, cell, cell, cell, sigjmp_buf *);
cell rope_node_new_length(bool, bool, long, cell, cell, sigjmp_buf *);
cell rope_new_ascii(bool, bool, char *, long, sigjmp_buf *);
cell rope_new_buffer(bool, bool, const char *, long, sigjmp_buf *);
cell rope_new_utfo(bool, bool, char *, long, sigjmp_buf *);
```

136. Always allocates one more byte than requested to be a Λ-terminator in case the rope's buffer ever leaks into something expecting a C-string. This should never happen but the byte is there anyway as a safety-valve.

```
#define rope_node_new_empty(S, D, F) rope_node_new_length((S), (D), 0, NIL, NIL, (F))

cell rope_node_new_length(bool sinward, bool dexward, long length, cell nsin, cell ndex, sigjmp_buf
        *failure)
{
  static int Snsin ← 1, Sndex ← 0;
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(null_p(nsin) ∨ rope_p(nsin));    /* Threading checked by */
  assert(null_p(nsin) ∨ rope_p(ndex));    /* dryad_node_new. */
  if (ckd_add(&length, length, 1)) siglongjmp(*failure, LERR_LIMIT);
  stack_protect(2, nsin, ndex, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  r ← segment_new(sizeof(Orope), length, 0, 0, &cleanup);
  r ← dryad_node_new(false, sinward, dexward, r, SO(Snsin), SO(Sndex), &cleanup);
  rope_cplength(r) ← rope_glength(r) ← −1;
  rope_buffer(r)[length − 1] ← '\0';
  stack_clear(2);
  return r;
}
```

137. **cell** *rope_node_new_clone*(**bool** *sinward*, **bool** *dexward*, **cell** *o*, **cell** *nsin*, **cell** *ndex*, **sigjmp_buf** **failure*)

```
{
  static int Sobject ← 2, Snsin ← 1, Sndex ← 0;
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(rope_p(o));
  assert(null_p(nsin) ∨ rope_p(nsin));      /* Threading checked by */
  assert(null_p(nsin) ∨ rope_p(ndex));      /* dryad_node_new. */
  stack_protect(3, o, nsin, ndex, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 3);
  r ← dryad_node_new(false, sinward, dexward, rope_segment(SO(Sobject)), SO(Snsin), SO(Sndex),
      &cleanup);
  rope_buffer(r)[segment_length(rope_segment(SO(Sobject)))] ← '\0';
  stack_clear(3);
  return r;
}
```

138. Some internal helpers: A rope can be created by copying the contents of a buffer.

#define *rope_new_length*(*S*, *D*, *L*, *F*) *rope_node_new_length*((*S*), (*D*), (*L*), NIL, NIL, (*F*))

cell *rope_new_buffer*(**bool** *thread_sin*, **bool** *thread_dex*, **const char** **buffer*, **long** *length*, **sigjmp_buf** **failure*)

```
{
  char *dst;
  cell r;
  int i;
  r ← rope_new_length(thread_sin, thread_dex, length, failure);
  dst ← rope_buffer(r);
  for (i ← 0; i < length; i++) dst[i] ← buffer[i];
  return r;
}
```

139. TODO: check that no byte is ≥ 0x80?

cell *rope_new_ascii*(**bool** *thread_sin*, **bool** *thread_dex*, **char** **buffer*, **long** *length*, **sigjmp_buf** **failure*)

```
{
  cell r;
  r ← rope_new_buffer(thread_sin, thread_dex, buffer, length, failure);
  rope_cplength(r) ← length;
  return r;
}
```

140. **cell** *rope_new_utfo*(**bool** *thread_sin* **Lunused**, **bool** *thread_dex* **Lunused**, **char** **buffer* **Lunused**, **long** *length* **Lunused**, **sigjmp_buf** **failure* **Lunused**)

```
{
  siglongjmp (*failure, LERR_UNIMPLEMENTED);
}
```

141. Rope Iterator.

```
#define ROPE_ITER_TWINE 0
#define ROPE_ITER_LENGTH 1
#define rope_iter(O) ((Orope_iter *) record_base(O))
#define rope_iter_twine(O) (record_cell((O), ROPE_ITER_TWINE))
#define rope_iter_set_twine_m(O, D) (record_set_cell_m((O), ROPE_ITER_TWINE, (D)))
```

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {
  int bvalue;      /* Value of the last-read byte. */
  long tboffset;   /* Byte offset into twine of next read. */
  long boffset;    /* — „ — the entire rope. */
  long cpoffset;   /* Code-point offset into the rope. */
  Outfio cp;       /* Code-point parser's working area. */
} Orope_iter;
```

142. ⟨ Function declarations 21 ⟩ +≡

```
cell rope_iterate_start(cell, long, sigjmp_buf *);
int rope_iterate_next_byte(cell, sigjmp_buf *);
cell rope_iterate_next_utfo(cell, sigjmp_buf *);
```

143.

```
cell rope_iterate_start(cell o, long begin, sigjmp_buf *failure)
{
  static int Sobject ← 0;
  cell r ← NIL, twine;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(rope_p(o));
  stack_protect(1, o, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
  r ← record_new(fix(RECORD_ROPE_ITERATOR), ROPE_ITER_LENGTH, sizeof(Orope_iter), &cleanup);
  twine ← SO(Sobject);
  if (begin < 0) {
    twine ← rope_first(twine, &cleanup);
    begin ← 0;
  }
  rope_iter_set_twine_m(r, twine);
  rope_iter(r)→bvalue ← rope_iter(r)→cpoffset ← 0;
  rope_iter(r)→tboffset ← rope_iter(r)→boffset ← begin;
  stack_clear(1);
  return r;
}
```

144.
```
int rope_iterate_next_byte(cell o, sigjmp_buf *failure)
{
  cell twine;

  assert(rope_iter_p(o));
  twine ← rope_iter_twine(o);
  if (null_p(twine)) siglongjmp (*failure, LERR_EOF);
  while (rope_iter(o)→tboffset ≡ rope_blength(twine)) {
    twine ← rope_next(twine, failure);
    rope_iter_set_twine_m(o, twine);
    rope_iter(o)→tboffset ← 0;
    if (null_p(twine)) return EOF;
  }
  rope_iter(o)→bvalue ← rope_buffer(twine)[rope_iter(o)→tboffset++];
  rope_iter(o)→boffset++;
  return rope_iter(o)→bvalue;
}
```

145. Asserts, buggily, that a code point will never be split between two rope twine (although any coarser unit may be and the rope may be validly not utf-8 encoded).

```
cell rope_iterate_next_utfo(cell o, sigjmp_buf *failure)
{
  int c;
  cell r, start, twine;
  Vutfio_parse res, (*readchar)(Outfio *, char);

  assert(rope_iter_p(o));
  twine ← rope_iter_twine(o);
  if (rope_p(twine) ∧ rope_cplength(twine) ≥ 0) readchar ← utfio_reread;
  else readchar ← utfio_read;
  start ← twine;
  rope_iter(o)→cp ← utfio_scan_start();
  r ← VOID;     /* NIL is a possible value. */
  while (void_p(r)) {
    c ← rope_iter(o)→bvalue ← rope_iterate_next_byte(o, failure);
    if (c ≡ EOF) {
      rope_iter(o)→cp.status ← UTFIO_EOF;
      if (¬rope_iter(o)→cp.remaining) return LEOF;
      break;
    }
    res ← readchar(&rope_iter(o)→cp, c);
    if (res ≡ UTFIO_COMPLETE ∨ res ≠ UTFIO_PROGRESS) break;
  }
  if (rope_iter_twine(o) ≠ start) rope_iter(o)→cp.status ← UTFIO_PROGRESS;
  rope_iter(o)→cpoffset++;
  return rune_new_utfio(rope_iter(o)→cp, failure);
}
```

146. Operational Data.

147. Stack.

#define `STACK_CHUNK` $0x100$

⟨ Function declarations 21 ⟩ +≡
void *stack_push*(**cell**, **sigjmp_buf** ∗);
cell *stack_pop*(**long**, **sigjmp_buf** ∗);
cell *stack_ref*(**long**, **sigjmp_buf** ∗);
void *stack_set_m*(**long**, **cell**, **sigjmp_buf** ∗);
cell *stack_ref_abs*(**long**, **sigjmp_buf** ∗);
void *stack_reserve*(**int**, **sigjmp_buf** ∗);
void *stack_protect*(**int**, . . .);

148. ⟨ Global variables 15 ⟩ +≡
unique cell *Stack* ← `NIL`;
unique long *StackP* ← −1;
unique cell *Stack_Tmp* ← `NIL`;

149. ⟨ External symbols 16 ⟩ +≡
extern unique long *StackP*;

150. ⟨ Initialise storage 35 ⟩ +≡
Stack ← *array_new*(`STACK_CHUNK`, *failure*);

151. **void** *stack_push*(**cell** *o*, **sigjmp_buf** ∗*failure*)
{
  if (*StackP* ≡ *array_length*(*Stack*)) {
    Stack_Tmp ← *o*;
    Stack ← *array_grow_m*(*Stack*, `STACK_CHUNK`, `NIL`, *failure*);
    o ← *Stack_Tmp*;
    Stack_Tmp ← `NIL`;
  }
  array_set_m(*Stack*, ++*StackP*, *o*);
}

152. **#define** *stack_clear*(*O*) *stack_pop*((*O*), Λ)

cell *stack_pop*(**long** *num*, **sigjmp_buf** ∗*failure*)
{
  cell *r*;
  assert(*num* ≥ 1);
  r ← *stack_ref*(*num* − 1, *failure*);
  StackP −= *num*;
  return *r*;
}

153. **#define** `SO`(*O*) *stack_ref*((*O*), Λ)

cell *stack_ref*(**long** *offset*, **sigjmp_buf** ∗*failure*)
{
  cell *r* ← `NIL`;
  assert(*offset* ≥ 0);
  assert(*failure* ≠ Λ ∨ *StackP* ≥ *offset*);
  if (*StackP* < *offset*) **siglongjmp** (∗*failure*, `LERR_UNDERFLOW`);
  else *r* ← *array_ref*(*Stack*, *StackP* − *offset*);
  return *r*;
}

154. #**define** SS(O, D) $stack\_set\_m((O), (D), \Lambda)$

```
void stack_set_m(long offset, cell datum, sigjmp_buf *failure)
{
  assert(offset ≥ 0);
  assert(failure ≠ Λ ∨ StackP ≥ offset);
  if (StackP < offset) siglongjmp (*failure, LERR_UNDERFLOW);
  else array_set_m(Stack, StackP − offset, datum);
}
```

155.

```
cell stack_ref_abs(long offset, sigjmp_buf *failure)
{
  cell r ← NIL;
  assert(offset ≥ 0);
  assert(failure ≠ Λ ∨ StackP ≥ offset);
  if (StackP < offset) siglongjmp (*failure, LERR_UNDERFLOW);
  else r ← array_ref(Stack, offset);
  return r;
}
```

156.

```
void stack_set_abs_m(long offset, cell datum, sigjmp_buf *failure)
{
  assert(offset ≥ 0);
  assert(failure ≠ Λ ∨ StackP ≥ offset);
  if (StackP < offset) siglongjmp (*failure, LERR_UNDERFLOW);
  else array_set_m(Stack, offset, datum);
}
```

157.

```
void stack_reserve(int delta, sigjmp_buf *failure)
{
  int i;
  while (array_length(Stack) − (StackP + 1) < delta)
    Stack ← array_grow_m(Stack, STACK_CHUNK, NIL, failure);
  StackP += delta;
  for (i ← 0; i < delta; i++) array_set_m(Stack, StackP − i, NIL);
}
```

158. ⟨ Global variables 15 ⟩ +≡

```
unique cell Protect[4];
```

159. **void** *stack_protect*(**int** *num*, ...)

```
{
  va_list ap;
  int i;
  sigjmp_buf cleanup, *failure;
  Verror reason ← LERR_NONE;
  assert(num > 0 ∧ num ≤ 4);
  va_start(ap, num);
  for (i ← 0; i < num; i++) Protect[i] ← va_arg(ap, cell);
  failure ← va_arg(ap, sigjmp_buf *);
  va_end(ap);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) goto fail;
  stack_reserve(num, &cleanup);
  for (i ← 0; i < num; i++) {
    SS(num − (i + 1), Protect[i]);
    Protect[i] ← NIL;
  }
  return;
fail:
  for (i ← 0; i < num; i++) Protect[i] ← NIL;
  siglongjmp (*failure, reason);
}
```

160. Environments.

161.
```
#define env_layer(O) (lcdr(O))
#define env_previous(O) (lcar(O))
#define env_replace_layer_m(O, E) (lcdr_set_m((O), (E)))
#define env_root_p(O) (environment_p(O) ∧ null_p(env_previous(O)))
```
⟨ Function declarations 21 ⟩ +≡
```
  Vhash env_rehash(cell, sigjmp_buf *);
  void env_clear(cell, cell, sigjmp_buf *);
  cell env_define(cell, cell, cell, sigjmp_buf *);
  cell env_extend(cell, sigjmp_buf *);
  cell env_here(cell, cell, sigjmp_buf *);
  int env_match(cell, void *, sigjmp_buf *);
  cell env_new_imp(cell, sigjmp_buf *);
  cell env_search(cell, cell, bool, sigjmp_buf *);
  cell env_set(cell, cell, cell, sigjmp_buf *);
  cell env_set_imp(cell, cell, cell, bool, sigjmp_buf *);
  cell env_unset(cell, cell, sigjmp_buf *);
```

162.
```
#define env_empty(F) (env_new_imp(NIL, (F)))

  cell env_new_imp(cell o, sigjmp_buf *failure)
  {
    static int Sobject ← 0;
    cell r ← NIL;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    stack_protect(1, o, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
    r ← keytable_new(0, &cleanup);
    r ← atom(Theap, SO(Sobject), r, FORM_ENVIRONMENT, &cleanup);
    stack_clear(1);
    return r;
  }
```

163.
```
cell env_extend(cell o, sigjmp_buf *failure)
  {
    assert(environment_p(o));
    return env_new_imp(o, failure);
  }
```

164.
```
Vhash env_rehash(cell o, sigjmp_buf *failure Lunused)
  {
    assert(pair_p(o) ∧ symbol_p(lcar(o)));
    return symbol_hash(lcar(o));
  }
```

165.
```
int env_match(cell binding, void *ctx, sigjmp_buf *failure Lunused)
  {
    cell maybe ← (cell) ctx;

    assert(symbol_p(maybe));
    assert(pair_p(binding));
    assert(symbol_p(lcar(binding)));
    return lcar(binding) ≡ maybe ? 0 : −1;
  }
```

166. **cell** *env_set_imp*(**cell** *where*, **cell** *label*, **cell** *datum*, **bool** *new_p*, **sigjmp_buf** ∗*failure*)

```
{
  static int Swhere ← 3, Slabel ← 2, Sdatum ← 1, Stable ← 0;
  cell table, r ← NIL;
  Vhash hash;
  long idx;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(environment_p(where));
  assert(symbol_p(label));
    /* datum validated by caller — in particular it could be UNDEFINED. */
  stack_protect(4, where, label, datum, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
  hash ← symbol_hash(SO(Slabel));
again: SS(Stable, (table ← env_layer(SO(Swhere))));
  idx ← keytable_search(table, hash, env_match, (void *) SO(Slabel), &cleanup);
  if (idx ≡ FAIL ∨ null_p(keytable_ref(table, idx))) {
    if (¬new_p) siglongjmp (*failure, LERR_MISSING);
    if (¬keytable_free_p(table)) {
      table ← keytable_enlarge_m(table, env_rehash, &cleanup);
      env_replace_layer_m(SO(Swhere), table);
      goto again;
    }
  }
  else if (new_p) siglongjmp (*failure, LERR_EXISTS);
  r ← cons(SO(Slabel), SO(Sdatum), &cleanup);
  keytable_save_m(SO(Stable), idx, r);
  r ← keytable_ref(SO(Stable), idx);
  stack_clear(4);
  return r;
}
```

167. **cell** *env_define*(**cell** *where*, **cell** *label*, **cell** *datum*, **sigjmp_buf** ∗*failure*)

```
{
  assert(defined_p(datum));
  return env_set_imp(where, label, datum, true, failure);
}
```

168. **cell** *env_set*(**cell** *where*, **cell** *label*, **cell** *datum*, **sigjmp_buf** ∗*failure*)

```
{
  assert(defined_p(datum));
  return env_set_imp(where, label, datum, false, failure);
}
```

169. **cell** *env_unset*(**cell** *where*, **cell** *label*, **sigjmp_buf** ∗*failure*)

```
{
  return env_set_imp(where, label, UNDEFINED, false, failure);
}
```

170. **void** *env_clear*(**cell** *where*, **cell** *label*, **sigjmp_buf** **failure*)

```
{
  cell table;
  Vhash hash;
  long idx;

  assert(environment_p(where));
  assert(symbol_p(label));
  hash ← symbol_hash(label);
  table ← env_layer(where);
  idx ← keytable_search(table, hash, env_match, (void *) label, failure);
  if (idx ≠ FAIL ∧ ¬null_p(keytable_ref(table, idx))) keytable_remove_m(table, idx);
}
```

171. **cell** *env_here*(**cell** *haystack*, **cell** *needle*, **sigjmp_buf** **failure*)

```
{
  cell r ← NIL;
  long idx;

  assert(environment_p(haystack));
  assert(symbol_p(needle));
  idx ← keytable_search(env_layer(haystack), symbol_hash(needle), env_match, (void *)
      needle, failure);
  if (idx ≡ FAIL) return NIL;
  r ← keytable_ref(env_layer(haystack), idx);
  if (null_p(r) ∨ undefined_p(lcdr(r))) return NIL;
  else return r;
}
```

172. **#define** *env_look*(*H*, *N*, *F*) *env_search*((*H*), (*N*), *false*, (*F*))

```
cell env_search(cell haystack, cell needle, bool ascend, sigjmp_buf *failure)
{
  cell r;

  assert(environment_p(haystack));
  assert(symbol_p(needle));
  for ( ; ¬null_p(haystack); haystack ← env_previous(haystack)) {
    r ← env_here(haystack, needle, failure);
    if (¬null_p(r)) return lcdr(r);
    else if (¬ascend) break;
  }
  return UNDEFINED;
}
```

173. Lexemes. Non-numeric lexemes on the whole are sufficient on their own without flags. The exception is blank space which can be horizontal or vertical.

```
#define LLF_NONE 0x00
#define LLF_HORIZONTAL 0x01
#define LLF_VERTICAL 0x02
```

174. Numeric lexemes on the other hand pick up bits explaining which base the scanner detected, the presence of a sign, decimal point, etc. This information is squeezed into 8 bits, mostly because I could.

To achieve this the base, 2, 8, 10 or 16 is instead encoded as 2, 8, 0 and 10 which use only the bits 2^1 (2) & 2^3 (8). This algorithm uses bit twiddling to turn the base stuffed in the lexeme's flags into its numeric equivalent.

```
#define LLF_BASE2 0x02
#define LLF_BASE8 0x08
#define LLF_BASE16 0x0a
#define LLF_BASE(O) flag2base(O)
  int flag2base(int f)
  {                                               /* 0  2  8 10 */
    f |= 5;                                       /* 5  7 13 15 */
    f++;                                          /* 6  8 14 16 */
    f ← (f & ~2) | (((f & 8) ≫ 2) ⊕ (f & 2));     /* 6 10 12 16 */
    f ← (f & ~8) | ((f & 4) ≪ 1);                 /*14  2 12 16 */
    return f & 26;                                /*10  2  8 16 */
  }
```

175. For comparison here's the reverse algorithm using tests and branching.

```
  int base2flag(int b)
  {
    if (b ≡ 10) return 0;
    else if (b ≡ 16) return 10;
    else return b;
  }
```

176. Complex or imaginary components of a number are indicated with an `i`, `j` or `k` suffix. The lexical analyser detects and flags these using the bits at 2^0 and 2^2 in order to fit around the bits which encode the base. In this case no bits set indicates a real number. Macros `LLF_COMPLEXITY` and `LLF_IMAGINATE` transform the flag value into a number $0 - 3$ and vice versa.

```
#define LLF_COMPLEXI 0x01
#define LLF_COMPLEXJ 0x04
#define LLF_COMPLEXK 0x05
#define LLF_COMPLEX_P(O) ((O) & 0x05)
#define LLF_COMPLEXITY(O) (((O) & 1) | (LLF_COMPLEX_P(O) ≫ 1))   /* [␣IJK] → [0123] */
#define LLF_IMAGINATE(O) (((O) & ~2) | (((O) & 2) ≪ 1))          /* [0123] → [␣IJK] */
```

177. With the 4 lower bits in use the remaining 4 indicate whether a sign is present and which, and likewise a decimal point or slash.

```
#define LLF_NEGATIVE 0x10
#define LLF_POSITIVE 0x20
#define LLF_SIGN 0x30

#define LLF_DOT 0x40
#define LLF_SLASH 0x80
#define LLF_RATIO 0xc0
```

178. Alphabetical order. TODO: Short in-line description and run-time reverse mapping.

```
⟨ Type definitions 6 ⟩ +≡
  typedef enum {
    LEXICAT_NONE,    /* 0 */
    LEXICAT_CLOSE, LEXICAT_CONSTANT, LEXICAT_CURIOUS, LEXICAT_DELIMITER,    /* ... 4 */
    LEXICAT_DOT, LEXICAT_END, LEXICAT_ESCAPED_STRING, LEXICAT_ESCAPED_SYMBOL,    /* ... 8 */
    LEXICAT_NUMBER, LEXICAT_OPEN, LEXICAT_RAW_STRING, LEXICAT_RAW_SYMBOL,    /* ... 12 */
    LEXICAT_RECURSE_HERE, LEXICAT_RECURSE_IS, LEXICAT_SPACE, LEXICAT_SYMBOL,    /* ... 16 */
    LEXICAT_INVALID
  } Vlexicat;
```

179. A lexeme records where in a rope it began and how many bytes and runes it occupies.

```
#define LEXEME_TWINE 0    /* The rope twine in which a lexeme started. */
#define LEXEME_LENGTH 1
#define lexeme(O) ((Olexeme *) record_base(O))
#define lexeme_twine(O) (record_cell((O), LEXEME_TWINE))
#define lexeme_set_twine_m(O, D) (record_set_cell_m((O), LEXEME_TWINE, (D)))
#define lexeme_byte(O, I) (rope_byte(lexeme_twine(O), lexeme(O)→tboffset + (I)))
⟨ Type definitions 6 ⟩ +≡
  typedef struct {
    long tboffset;    /* Byte in the twine where the lexeme began. */
    long cpstart;    /* Lexeme's starting point in the rope, in runes. */
    long blength, cplength;    /* Lexeme's length in bytes & runes. */
    char flags;    /* Extra detail as described previously. */
    Vlexicat cat;    /* The lexeme's category as above. */
  } Olexeme;
```

180.

```
cell lexeme_new(Vlexicat cat, char flags, cell twine, long tboffset, long blength, long
        cpstart, long cplength, sigjmp_buf *failure)
  {
    static int Stwine ← 0;
    cell r;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;
    assert(null_p(twine) ∨ rope_p(twine));
    stack_protect(1, twine, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
    r ← record_new(fix(RECORD_LEXEME), LEXEME_LENGTH, sizeof(Olexeme), &cleanup);
    lexeme_set_twine_m(r, SO(Stwine));
    lexeme(r)→cat ← cat;
    lexeme(r)→flags ← flags;
    lexeme(r)→tboffset ← tboffset;
    lexeme(r)→blength ← blength;
    lexeme(r)→cpstart ← cpstart;
    lexeme(r)→cplength ← cplength;
    stack_clear(1);
    return r;
  }
```

181. Syntax Parser. The completed result of the lexical analyser is given to the syntax parser to transform it from a list of tokens into a tree of operations. Each node in this tree which with some irony doesn't use any of the built-in trees is a syntax object — a record (of only cells) holding the datum which was parsed from the source and a record of where in the stream of lexemes it was found.

```
#define SYNTAX_DATUM 0   /* The parsed datum. */
#define SYNTAX_NOTE 1   /* (Unused) a note for the future use of the evaluator. */
#define SYNTAX_START 2   /* The lexeme which began this datum. */
#define SYNTAX_END 3   /* The lexeme which ended this datum (inclusive). */
#define SYNTAX_VALID 4   /* Whether the source is valid and can be evaluated. */
#define SYNTAX_LENGTH 5
#define syntax_datum(O) (record_cell((O), SYNTAX_DATUM))
#define syntax_end(O) (record_cell((O), SYNTAX_END))
#define syntax_note(O) (record_cell((O), SYNTAX_NOTE))
#define syntax_start(O) (record_cell((O), SYNTAX_START))
#define syntax_valid(O) (record_cell((O), SYNTAX_VALID))
#define syntax_new(D, S, E, F) syntax_new_imp((D), NIL, (S), (E), true, (F))
#define syntax_invalid(D, S, E, F) syntax_new_imp((D), NIL, (S), (E), false, (F))

  cell syntax_new_imp(cell datum, cell note, cell start, cell end, bool valid, sigjmp_buf *failure)
  {
    static int Sdatum ← 3, Snote ← 2, Sstart ← 1, Send ← 0;
    cell r ← NIL;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    assert(defined_p(datum));
    assert(null_p(note) ∨ symbol_p(note));
    assert(dlist_p(start) ∧ lexeme_p(dlist_datum(start)));
    assert(dlist_p(end) ∧ lexeme_p(dlist_datum(end)));   /* ∧ start→...→next ≡ end */
    stack_protect(4, datum, note, start, end, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
    r ← record_new(fix(RECORD_SYNTAX), SYNTAX_LENGTH, 0, &cleanup);
    record_set_cell_m(r, SYNTAX_VALID, predicate(valid));
    record_set_cell_m(r, SYNTAX_NOTE, SO(Snote));
    record_set_cell_m(r, SYNTAX_DATUM, SO(Sdatum));
    record_set_cell_m(r, SYNTAX_START, SO(Sstart));
    record_set_cell_m(r, SYNTAX_END, SO(Send));
    stack_clear(4);
    return r;
  }
```

182. Annotated pairs. These are used by the evaluator (below) to keep track of its partial work. The evaluator should probably be refactored to use syntax nodes instead. At least they should use a numeric identifier rather than a `LossLess` symbol to avoid this horrific API:

```
#define Sym_APPLICATIVE (symbol_new_const("APPLICATIVE"))
#define Sym_COMBINE_APPLY (symbol_new_const("COMBINE-APPLY"))
#define Sym_COMBINE_BUILD (symbol_new_const("COMBINE-BUILD"))
#define Sym_COMBINE_DISPATCH (symbol_new_const("COMBINE-DISPATCH"))
#define Sym_COMBINE_FINISH (symbol_new_const("COMBINE-FINISH"))
#define Sym_COMBINE_OPERATE (symbol_new_const("COMBINE-OPERATE"))
#define Sym_CONDITIONAL (symbol_new_const("CONDITIONAL"))
#define Sym_DEFINITION (symbol_new_const("DEFINITION"))
#define Sym_EVALUATE_DISPATCH (symbol_new_const("EVALUATE-DISPATCH"))
#define Sym_SAVE_AND_EVALUATE (symbol_new_const("SAVE-ENVIRONMENT-AND-EVALUATE"))
#define Sym_ENVIRONMENT_P (symbol_new_const("ENVIRONMENT?"))
#define Sym_ENVIRONMENT_M (symbol_new_const("ENVIRONMENT!"))
#define Sym_EVALUATE (symbol_new_const("EVALUATE"))
#define Sym_OPERATIVE (symbol_new_const("OPERATIVE"))
```

⟨ Prepare constants & symbols 182 ⟩ ≡

```
  (void) Sym_APPLICATIVE;
  (void) Sym_COMBINE_APPLY;
  (void) Sym_COMBINE_BUILD;
  (void) Sym_COMBINE_DISPATCH;
  (void) Sym_COMBINE_FINISH;
  (void) Sym_COMBINE_OPERATE;
  (void) Sym_CONDITIONAL;
  (void) Sym_DEFINITION;
  (void) Sym_EVALUATE;
  (void) Sym_ENVIRONMENT_P;
  (void) Sym_OPERATIVE;
```

This code is used in section 333.

183.

```
#define note(O) (lcar(O))
#define note_pair(O) (lcdr(O))
#define note_car(O) (lcar(note_pair(O)))
#define note_cdr(O) (lcdr(note_pair(O)))
#define note_set_car_m(O, V) (lcar_set_m(note_pair(O), (V)))
#define note_set_cdr_m(O, V) (lcdr_set_m(note_pair(O), (V)))

  cell note_new(cell label, cell ncar, cell ncdr, sigjmp_buf *failure)
  {
    static int Slabel ← 3, Sncar ← 2, Sncdr ← 1, Stmp ← 0;
    cell r;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    assert(symbol_p(label));
    assert(defined_p(ncar) ∧ defined_p(ncdr));
    stack_protect(4, label, ncar, ncdr, NIL, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
    SS(Stmp, cons(SO(Sncar), SO(Sncdr), &cleanup));
    r ← atom(Theap, SO(Slabel), SO(Stmp), FORM_NOTE, &cleanup);
    stack_clear(4);
    return r;
  }
```

184. Programs (Closures). Programs in `LossLess` are divided into two categories: *operative* and *applicative*. Programs are also and more formally known as *combiners* when they are the first expression in a list which is being evaluated, which is by *combining* the multiple expressions (*combiner* & *arguments*) into a single expression (return value).

A combiner is a program which condenses zero or more arguments into a single expression. Internally a combiner is further distinguished by whether it has been provided by the implementation or defined at run-time. A *closure* is a combiner which includes the environment that was in place when it was defined and it re-established when the closure program is evaluated.

Given the astonishing compute capabilities which closures enable they have comically simple storage requirements. A list records the run-time environment they were created in with the expression to evaluate and its arguments (*formals*). Applicative and operative closures are identified by their tag.

```
#define closure_formals(O) (lcar(O))
#define closure_environment(O) (lcadr(O))
#define closure_body(O) (lcaddr(O))
```

⟨ Function declarations 21 ⟩ +≡

```
cell closure_new(bool, cell, cell, cell, sigjmp_buf *);
```

185. Usually (always?) these arguments are actually in registers and the stack dancing is unnecessary.

```
cell closure_new(bool is_applicative,
        cell formals,    /* From register: Accumulator, */
        cell environment,    /* ... Environment, */
        cell body,    /* ... Expression. */
        sigjmp_buf *failure)
{
    static int Sformals ← 3, Senv ← 2, Sbody ← 1, Sret ← 0;
    cell r;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    stack_protect(4, formals, environment, body, NIL, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
    SS(Sret, cons(SO(Sbody), SO(Sret), &cleanup));
    SS(Sret, cons(SO(Senv), SO(Sret), &cleanup));
    SS(Sret, atom(Theap, SO(Sformals), SO(Sret),
            is_applicative ? FORM_APPLICATIVE : FORM_OPERATIVE, &cleanup));
    r ← SO(Sret);
    stack_clear(4);
    return r;
}
```

186. Despite being an incredibly powerful abstraction tool closures cannot actually *do* anything on their own. Closures are built from closures built out of closures which, eventually, must use *primitive* tasks to perform actions — closures are really little more than an elaborate means of structuring memory.

The definition of primitives here is overly simplistic and has a number of deficiencies both in terms of time and space, which will be especially felt on the older and/or smaller architectures which are also being targetted. For now (2022) this is a "temporary solution"[1] until the evaluation process (below) is ready to be considered some form of "complete".

A primitive is a block of C code identified by a **Vprimitive**, an integer offset into an array of **Oprimitive** objects, *Iprimitive*.

```
#define primitive(O) (fix_value(lcar(O)))
#define primitive_label(O) (lcdr(O))
#define primitive_base(O) (&Iprimitive[primitive(O)])
#define primitive_applicative_p(O) (primitive_p(O) ∧ primitive_base(O)→applicative)
#define primitive_operative_p(O) (primitive_p(O) ∧ ¬primitive_base(O)→applicative)
```

⟨ Type definitions 6 ⟩ +≡
```
  typedef enum { ⟨ Symbolic primitive identifiers 274 ⟩ PRIMITIVE_LENGTH } Vprimitive;
  typedef struct {
    char *label;      /* LossLess binding. */
    cell box;      /* Heap storage. */
    bool applicative;      /* Or operative? */
    char min, max;      /* Minimum and/or maximum required arguments. */
  } Oprimitive;
#if 0      /* Something like this to share segments between raw string and rope/symbol storage? */
  [PRIMITIVE_FOO] ← { &Sym_FOO, NIL , ... } ,
  shared Osegment Sym_FOO ← {.address ← "foo"};
#endif
```

187. The *Iprimitive* array associates the internal numeric identifier with the `LossLess` symbol representing the primitive. During initialisation each primitive is bound to its symbol in a pair stored in *Root*, which is the initial environment the run-time is in prior to establishing any closures and is what's returned by `(root-environment)`.

⟨ Global variables 15 ⟩ +≡
```
  Oprimitive Iprimitive[] ← {⟨ Primitive definitions 275 ⟩};
  shared cell Root ← NIL;
```

188. ⟨ Register primitive operators 188 ⟩ ≡
```
  Root ← env_empty(failure);
  for (i ← 0; i < PRIMITIVE_LENGTH; i++) {
    x ← symbol_new_const(Iprimitive[i].label);
    x ← Iprimitive[i].box ← atom(Theap, fix(i), x, FORM_PRIMITIVE, failure);
    env_define(Root, primitive_label(x), x, failure);
  }
```
This code is used in section 333.

[1] It's important only that primitives work and speed is not of the essence.

189. Compute.

190. Lexical Analysis. Source code is first scanned to categorise each byte or contiguous sequence of bytes and create a lexeme object out of them, concatenated together in a doubly-linked list.

The lexical analyser (*LEXAR*) is a state machine in the form of a record holding at each moment where it is in the source code and what it has most recently seen. Scanning proceeds until the source rope is entirely consumed.

This lexical analyser occasionally needs to examine thext rune to determine what lexeme is being represented. To achieve this it's possible to return a rune into the state machine so that subsequent iteration will return it instead of iterating over the backing rope. This ordinarily simple task is complicated by the need to track where in the rope the lexeme islocated, both by byte/rune position and taking note of the rope twine, which may have changed between runes.

Achieving this uses two pairs of cells in the analyser state machine, each pair consisting of a rune and the twine in which it was encountered.

The first pair is the "peeked" pair. Every returned twine/rune combination is put here before being analysed and until the rune is accepted further peeking will return that rune. If the rune is put back then the combination is moved to the second — "backput" — pair and will not be considered when the analysis state is used to create a complete lexical token but *will* be returned the next time a rune is peeked (after being moved back to the "peeked" pair of attributes).

```
#define LEXAR_STARTER 0        /* Lexeme's starting twine. */
#define LEXAR_ITERATOR 1       /* Current position in rope. */
#define LEXAR_PEEKED_TWINE 2   /* Twine containing the rune under consideration. */
#define LEXAR_PEEKED_RUNE 3    /* The rune under consideration. */
#define LEXAR_BACKPUT_TWINE 4  /* The twine containing an unwanted rune. */
#define LEXAR_BACKPUT_RUNE 5   /* The unwanted rune. */
#define LEXAR_LENGTH 6
#define lexar(O) ((Olexical_analyser *) record_base(O))
#define lexar_starter(O) (record_cell((O), LEXAR_STARTER))
#define lexar_iterator(O) (record_cell((O), LEXAR_ITERATOR))
#define lexar_peeked_rune(O) (record_cell((O), LEXAR_PEEKED_RUNE))
#define lexar_peeked_twine(O) (record_cell((O), LEXAR_PEEKED_TWINE))
#define lexar_backput_rune(O) (record_cell((O), LEXAR_BACKPUT_RUNE))
#define lexar_backput_twine(O) (record_cell((O), LEXAR_BACKPUT_TWINE))
#define lexar_set_starter_m(O, D) (record_set_cell_m((O), LEXAR_STARTER, (D)))
#define lexar_set_iterator_m(O, D) (record_set_cell_m((O), LEXAR_ITERATOR, (D)))
#define lexar_set_peeked_rune_m(O, D) (record_set_cell_m((O), LEXAR_PEEKED_RUNE, (D)))
#define lexar_set_peeked_twine_m(O, D) (record_set_cell_m((O), LEXAR_PEEKED_TWINE, (D)))
#define lexar_set_backput_rune_m(O, D) (record_set_cell_m((O), LEXAR_BACKPUT_RUNE, (D)))
#define lexar_set_backput_twine_m(O, D) (record_set_cell_m((O), LEXAR_BACKPUT_TWINE, (D)))
```

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {    /* For brevity's sake accessors are not defined for these attributes. */
    long tbstart;    /* Byte offset where this lexeme began. */
    long blength;    /* Number of bytes scanned so far. */
    long cpstart;    /* — „ — runes (into the whole rope). */
    long cplength;   /* — „ — scanned. */
} Olexical_analyser;
```

191. ⟨ Function declarations 21 ⟩ +≡

```
cell lexar_append(int, int, Vlexicat, int, sigjmp_buf *);
cell lexar_clone(cell, sigjmp_buf *);
cell lexar_peek(int, sigjmp_buf *);
void lexar_putback(int);
cell lexar_start(cell, sigjmp_buf *);
cell lexar_take(int, sigjmp_buf *);
cell lex_rope(cell, sigjmp_buf *);
cell lexar_token(int, int, sigjmp_buf *);
```

192. To begin scanning *lexar_start* is called with the rope to scan and it returns a fresh lexical analyser state machine ready to iterate from the beginning of the rope.

```
cell lexar_start(cell o, sigjmp_buf *failure)
{
  static int Srope ← 1, Sret ← 0;
  cell r;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(rope_p(o));
  stack_protect(2, o, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  SS(Sret, record_new(fix(RECORD_LEXAR), LEXAR_LENGTH, sizeof(Olexical_analyser), &cleanup));
  lexar_set_iterator_m(SO(Sret), rope_iterate_start(SO(Srope), −1, &cleanup));
  lexar_set_starter_m(SO(Sret), NIL);
  lexar_set_peeked_twine_m(SO(Sret), VOID);
  lexar_set_backput_twine_m(SO(Sret), VOID);
  r ← SO(Sret);
  lexar(r)→tbstart ← lexar(r)→blength ← lexar(r)→cpstart ← lexar(r)→cplength ← 0;
  stack_clear(2);
  return r;
}
```

193. Raw strings and symbols scan three lexemes at a time and when each begins the state of the analyser is cloned so that they can be created correctly later.

```
cell lexar_clone(cell o, sigjmp_buf *failure)
{
  static int Slexar ← 1, Sret ← 0;
  cell r, tmp;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(lexar_p(o));
  stack_protect(2, o, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  SS(Sret, r ← record_new(fix(RECORD_LEXAR), LEXAR_LENGTH, sizeof(Olexical_analyser),
      &cleanup));
  tmp ← lexar_starter(SO(Slexar));
  lexar_set_starter_m(r, tmp);
  tmp ← rope_iterate_start(tmp, lexar(SO(Slexar))→tbstart, &cleanup);
  r ← SO(Sret);
  lexar_set_iterator_m(r, tmp);
  lexar_set_peeked_twine_m(r, VOID);
  lexar_set_backput_twine_m(r, VOID);
  *lexar(r) ← *lexar(SO(Slexar));
  stack_clear(2);
  return r;
}
```

194. When a rune is obtained, from the put-back buffer or by iteration over the backing rope, or when a rune is put back into the put-back buffer, its position in the source is updated. If the "rune" is in fact `LEOF` then this takes up no space so care must be taken when it gets put back.

After a lexeme is created and appended the starting location is reset by setting the starter twine to `NIL`. In most cases the analyser immediately returns the lexeme to the caller but those which do not use *lexar_reset* directly to indicate the beginning of a new lexeme without peeking.

```
#define lexar_reset(L, R) do
        {
          lexar_set_starter_m((L), rope_iter_twine((R)));
          lexar(L)→tbstart ← rope_iter(R)→tboffset;
          lexar(L)→cpstart ← rope_iter(R)→cpoffset;
        }
        while (0)

  cell lexar_peek(int Silex, sigjmp_buf *failure)
  {
    cell irope, ilex, r, tmp;
    assert(lexar_p(SO(Silex)));
    ilex ← SO(Silex);
    irope ← lexar_iterator(ilex);
    if (null_p(lexar_starter(ilex))) lexar_reset(ilex, irope);
    if (¬void_p(lexar_backput_twine(ilex))) {      /* Something has been put back. */
      assert(void_p(lexar_peeked_twine(ilex)));
      if (¬eof_p(lexar_backput_rune(ilex))) {      /* Start at the re-included rune. */
        lexar(ilex)→tbstart −= rune_parsed(lexar_backput_rune(ilex));
        lexar(ilex)→cpstart −= 1;
      }
      lexar_set_peeked_twine_m(ilex, lexar_backput_twine(ilex));
      lexar_set_peeked_rune_m(ilex, lexar_backput_rune(ilex));
      lexar_set_backput_twine_m(ilex, VOID);
    }
    else if (void_p(lexar_peeked_twine(ilex))) {      /* Nothing is pending. */
      tmp ← rope_iterate_next_utfo(irope, failure);
      lexar_set_peeked_rune_m(SO(Silex), tmp);
      lexar_set_peeked_twine_m(SO(Silex), rope_iter_twine(irope));
    }
    else return lexar_peeked_rune(ilex);      /* A rune is already being examined. */
    r ← lexar_peeked_rune(ilex);
    if (¬eof_p(r)) {
      lexar(ilex)→cplength++;
      lexar(ilex)→blength += rune_parsed(r);
    }
    return r;
  }
```

195. If a rune is accepted by the scanner *lexar_take* clears the peeked-at rune from the analyser state.

```
  cell lexar_take(int Silex, sigjmp_buf *failure)
  {
    cell r;
    assert(lexar_p(SO(Silex)));
    assert(void_p(lexar_backput_twine(SO(Silex))));
    r ← lexar_peek(Silex, failure);
    lexar_set_peeked_twine_m(SO(Silex), VOID);
    return r;
  }
```

196. A rune can only be put back if the “backput” buffer is empty. If so then the twine/rune pair is moved from the peeked buffer and the scanned length count reduced (unless the “rune” was really `LEOF`).

```
void lexar_putback(int Silex)
{
  cell tmp;
  assert(lexar_p(SO(Silex)));
  assert(¬void_p(lexar_peeked_twine(SO(Silex))));
  assert(void_p(lexar_backput_twine(SO(Silex))));
  tmp ← lexar_peeked_rune(SO(Silex));
  if (¬eof_p(tmp)) {
    lexar(SO(Silex))→cplength −−;
    lexar(SO(Silex))→blength −= rune_parsed(tmp);
  }
  lexar_set_backput_rune_m(SO(Silex), tmp);
  lexar_set_backput_twine_m(SO(Silex), lexar_peeked_twine(SO(Silex)));
  lexar_set_peeked_twine_m(SO(Silex), VOID);
}
```

197. When a lexical token is complete *lexar_append* uses the current state of the analyser to create a lexeme and resets the state sufficient that the next scan will begin a new token by setting the starter twine to `NIL`.

If the token being appended is `LEXICAT_SPACE` or `LEXICAT_INVALID` and the previous token matches it then that token is extended rather than appending another one.

```
cell lexar_append(int Silex, int Sret, Vlexicat cat, int flags, sigjmp_buf *failure)
{
  cell r ← NIL, tmp;
  Olexical_analyser *l;
  assert(lexar_p(SO(Silex)));
  if ((cat ≡ LEXICAT_SPACE ∨ cat ≡ LEXICAT_INVALID) ∧ (tmp ← dlist_datum(SO(Sret)),
        lexeme_p(tmp)) ∧ lexeme(tmp)→cat ≡ cat ∧ lexeme(tmp)→flags ≡ flags) {
    lexeme(tmp)→blength += lexar(SO(Silex))→blength;
    lexeme(tmp)→cplength += lexar(SO(Silex))→cplength;
    r ← SO(Sret);
  }
  else {
    l ← lexar(SO(Silex));
    tmp ← lexeme_new(cat, flags, lexar_starter(SO(Silex)), l→tbstart, l→blength, l→cpstart, l→cplength,
          failure);
    r ← dlist_append_datum_m(SO(Sret), tmp, failure);
    SS(Sret, r);
  }
  l ← lexar(SO(Silex));
  if (¬void_p(lexar_peeked_twine(SO(Silex)))) lexar_take(Silex, failure);
  l→blength ← l→cplength ← 0;
  lexar_set_starter_m(SO(Silex), NIL);
  return dlist_datum(r);
}
```

198. The analysis state machine itself is implemented in *lexar_token* which consumes bytes sufficient to emit a single token, possibly leaving a rune in the buffer for the next token.

On top of the state held in the *lexar* object each token uses the following variables.

```
#define RIS(O, V) (rune(O) ≡ (V))    /* Rune is ... */
#define CIS(O, V) (lexeme(O)→cat ≡ (V))    /* Category is ... */
#define lexar_space_p(O) (¬rune_failure_p(O) ∧
        (RIS((O), '␣') ∨ RIS((O), '\t') ∨ RIS((O), '\r') ∨ RIS((O), '\n')))
#define lexar_opening_p(O) (¬rune_failure_p(O) ∧ (RIS((O), '(') ∨ RIS((O), '{') ∨ RIS((O), '[')))
#define lexar_closing_p(O) (¬rune_failure_p(O) ∧ (RIS((O), ')') ∨ RIS((O), '}') ∨ RIS((O), ']')))
#define lexar_terminator_p(O) (eof_p(O) ∨
        lexar_space_p(O) ∨ lexar_opening_p(O) ∨ lexar_closing_p(O))
#define lexeme_terminator_p(O) (CIS((O), LEXICAT_END) ∨
        CIS((O), LEXICAT_OPEN) ∨ CIS((O), LEXICAT_CLOSE) ∨
        CIS((O), LEXICAT_SPACE))

cell lexar_token(int Silex, int Sret, sigjmp_buf *failure)
{
    static int Ssdelim ← 3;    /* The opening-delimiter lexeme of a raw string/symbol. */
    static int Sedelim ← 2;    /* — „ — closing. */
    static int Srdelim ← 1;    /* Remember where a closing delimiter may have begun. */
    static int Sditer ← 0;    /* Rope iterator over an opening delimiter. */
    cell c;    /* Current rune. */
    int32_t d, v;    /* — „ — & opening delimiter's value. */
    int base ← 10;    /* The base of the number being scanned. */
    int has_imagination ← 0;    /* The complexity of a number. */
    int has_sign ← 0;    /* Whether a number began with a sign, and which. */
    int has_ratio ← 0;    /* Whether and how a number is rational. */
    int flags ← 0;    /* See LLF_*. */
    int want_digit ← MUST;    /* Whether a numeric digit is permitted. */
    Vlexicat cat ← LEXICAT_NONE;    /* The category that is discovered. */
    Orope_iter *irope, *idelim;    /* The rope and ending-delimiter iterators. */
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;
    cell r, tmp;
    int i;
    ⟨ Perform lexical analysis 199 ⟩
}
```

199. The regular *lexar_peek* / *lexar_putback* / *lexar_take* API is further augmented by *lexar_another* to check whether the source has terminated with LEOF, possibly prematurely and in many cases it's known that the rune, if there is one, will be immediately taken.

```
#define lexar_another(V, I, T, A, L, R, F) do
        {    /* Variable | Iterator | Take? | Allow-invalid? | Label | Return | Failure */
          (V) ← lexar_peek((I), (F));
          if (eof_p(V)) goto L;
          else if (¬(A) ∧ rune_failure_p(V))
            return lexar_append((I), (R), LEXICAT_INVALID, LLF_NONE, (F));
          else if (T) (V) ← lexar_take((I), (F));
        }
        while (0)
```

⟨ Perform lexical analysis 199 ⟩ ≡

```
  c ← lexar_peek(Silex, failure);
  if (eof_p(c)) return lexar_append(Silex, Sret, LEXICAT_END, LLF_NONE, failure);
  else if (rune_failure(c) ≡ UTFIO_EOF) goto LEXAR_premature_eof;
  else if (rune_failure_p(c)) return lexar_append(Silex, Sret, LEXICAT_INVALID, LLF_NONE, failure);
  else switch (rune(c)) {
      ⟨ Look for blank space 200 ⟩
      ⟨ Look for a bracketing token 201 ⟩
      ⟨ Look for a symbol 202 ⟩
      ⟨ Look for a string 203 ⟩
      ⟨ Look for a curious token 210 ⟩
      ⟨ Look for a number 211 ⟩
    }
  abort();    /* UNREACHABLE. */
LEXAR_raw_eof:    /* LEOF encountered while scanning a raw string or symbol. */
  stack_clear(4);
  Silex −= 4;
  Sret −= 4;
LEXAR_premature_eof:    /* LEOF encountered any where else it should not be. */
  lexar_append(Silex, Sret, LEXICAT_INVALID, LLF_NONE, failure);
  tmp ← rope_iter_twine(lexar_iterator(SO(Silex)));
  lexar_set_starter_m(SO(Silex), tmp);
  return lexar_append(Silex, Sret, LEXICAT_END, LLF_NONE, failure);
```

This code is used in section 198.

200. The range of space considered blank can be widened in future implementations of LossLess but these four will do for now.

⟨ Look for blank space 200 ⟩ ≡

```
case '␣': case '\t': return lexar_append(Silex, Sret, LEXICAT_SPACE, LLF_HORIZONTAL, failure);
case '\r': case '\n': return lexar_append(Silex, Sret, LEXICAT_SPACE, LLF_VERTICAL, failure);
```

This code is used in section 199.

201. From the lexical analyser's perspective all brackets look the same. It is the parser's job to consider the combination of an opening bracket, closing bracket and their contents.

⟨ Look for a bracketing token 201 ⟩ ≡

```
case '(':    /* List */
case '[':    /* Vector */
case '{':    /* Relation */
  return lexar_append(Silex, Sret, LEXICAT_OPEN, LLF_NONE, failure);
case '.':
  return lexar_append(Silex, Sret, LEXICAT_DOT, LLF_NONE, failure);
case ')': case ']': case '}':
  return lexar_append(Silex, Sret, LEXICAT_CLOSE, LLF_NONE, failure);
```

This code is used in section 199.

202. Strings and Symbols. Symbols are begun by any character which isn't matched by anything else; the syntactic runes above or those below which indicate some other token. Unlike most lisp or scheme implementations only space or brackets terminate a symbol — any other rune (in particular `.`) is considered part of the symbol token.

At this stage no effort is made to constrain non-printable-ASCII runes, including control characters, and the myriad unicode exceptions and confusables. Some certainly should.

⟨ Look for a symbol 202 ⟩ ≡
```
default:
  while (1) {
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    lexar_take(Silex, failure);
  symbol:     /* comefrom what might have been numbers but aren't. */
    tmp ← lexar_peek(Silex, failure);
    if (lexar_terminator_p(tmp)) {
      lexar_putback(Silex);
      return lexar_append(Silex, Sret, LEXICAT_SYMBOL, LLF_NONE, failure);
    }
  }
  break;
```
This code is used in section 199.

203. Strings are delimited by ⟪|⟫, special characters within them are escaped with ⟪#⟫. These were chosed to ease development of `LossLess` and are subject to change when `LossLess` is subject to less change.

Symbols with arbitrary labels can also be created using the same escaping rules as strings by preceeding the opening ⟪|⟫ delimiter with ⟪#⟫: ⟪#|...|⟫.

⟨ Look for a string 203 ⟩ ≡
```
case '\'': case '"': return lexar_append(Silex, Silex, LEXICAT_INVALID, LLF_NONE, failure);
case '|': cat ← LEXICAT_ESCAPED_STRING;
string:     /* or "comefrom "#|"" with cat ← LEXICAT_ESCAPED_SYMBOL. */
  lexar_take(Silex, failure);
  while (1) {
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    lexar_another(c, Silex, true, true, LEXAR_premature_eof, Sret, failure);
    if (rune(c) ≡ '|') return lexar_append(Silex, Sret, cat, LLF_NONE, failure);
    else if (rune(c) ≡ '#') {
      lexar_another(c, Silex, true, true, LEXAR_premature_eof, Sret, failure);
      switch (rune(c)) {⟨ Scan an escape sequence 204 ⟩}
    }
  }
  break;
```
See also section 205.

This code is used in section 199.

204. Few escape sequences are defined. Anything other than these following a `#` is an error (except that the scanner is case insensitive). The macros detect whether the appropriate byte(s) are there.

`#|`	→	Literal ⟪`|`⟫.
`##`	→	Literal ⟪`#`⟫.
`#x`xy	→	Byte of value $0x\,xy$.
`#o`xyz	→	Byte of value $^\circ xyz$.
`#u`$wxyz$	→	UTF-8 sequence encoding unicode code point of value $0x\,wxyz$.
`#(`$xyz...$`)`	→	(1–6 bytes) UTF-8 encoding of unicode code point with value $0x\,xyz\ldots$.

```
#define lexar_detect_octal(V, I, C, R, F)
        {
          lexar_another((V), (I), true, true, LEXAR_premature_eof, (R), (F));
          if (¬rune_failure_p(V) ∧ rune(V) ≥ '0' ∧ rune(V) ≤ '7') {
            lexar_take((I), (F));
            continue;
          }
          (C) ← LEXICAT_INVALID;
          break;
        }
#define lexar_detect_hexadecimal(V, I, C, R, F)
        {
          int32_t v;
          lexar_another((V), (I), true, true, LEXAR_premature_eof, (R), (F));
          v ← rune(V);
          if (¬rune_failure_p(V) ∧ ((v ≥ '0' ∧ v ≤ '9') ∨ (v ≥ 'a' ∧ v ≤ 'f') ∨ (v ≥ 'A' ∧ v ≤ 'F')))
            {
            lexar_take((I), (F));
            continue;
          }
          (C) ← LEXICAT_INVALID;
          break;
        }
```

⟨ Scan an escape sequence 204 ⟩ ≡

```
case '#': case '|': break;
case 'o': case 'O':    /* Byte in octal. */
  for (i ← 0; i < 3; i++) lexar_detect_octal(c, Silex, cat, Sret, failure);
  break;
case 'x': case 'X':    /* Byte in hex. */
  for (i ← 0; i < 2; i++) lexar_detect_hexadecimal(c, Silex, cat, Sret, failure);
  break;
case 'u': case 'U':    /* Unicode code point (rune) in 4 hex digits. */
  for (i ← 0; i < 4; i++) lexar_detect_hexadecimal(c, Silex, cat, Sret, failure);
  break;
case '(':    /* Variable length unicode code point (rune). */
  lexar_detect_hexadecimal(c, Silex, cat, Sret, failure);
  for (i ← 5; i; i−−) lexar_another(c, Silex, true, true, LEXAR_premature_eof, Sret, failure);
  if (rune(c) ≡ ')') break;
  else lexar_detect_hexadecimal(c, Silex, cat, Sret, failure);
  if (¬i ∧ cat ≠ LEXICAT_INVALID)
    lexar_another(c, Silex, true, true, LEXAR_premature_eof, Sret, failure);
  if (rune(c) ≠ ')') cat ← LEXICAT_INVALID;
  break;
default: cat ← LEXICAT_INVALID;
  break;
```

This code is used in section 203.

205. Strings of arbitrary bytes are delimited by arbitrary delimiters which are themselves delimited by a pair of `$`s. The delimiter is everything between a pair of ⟨⟨`$`⟩⟩ symbols — for the time being constrained to ASCII letters, numbers and ⟨⟨`-`⟩⟩ & ⟨⟨`_`⟩⟩ — and may be empty as in ⟨⟨`$$`⟩⟩. Successfully scanning a raw string appends three lexemes: "`LEXICAT_DELIMITER` | `LEXICAT_RAW_STRING` | `LEXICAT_DELIMITER`".

Arbitrary symbols can be defined with the same mechanism by preceeding the opening delimiter with ⟨⟨`#`⟩⟩, as with escaped strings, and the result then includes the `LEXICAT_RAW_SYMBOL` lexeme instead of `LEXICAT_RAW_STRING`.

```
⟨ Look for a string 203 ⟩ +≡
case '$': cat ← LEXICAT_RAW_STRING;
delimiter:   /* or "comefrom "#$"" with cat ← LEXICAT_RAW_SYMBOL. */
  ⟨ Scan a delimiter 206 ⟩
  stack_reserve(4, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
  Silex += 4;
  Sret += 4;
  tmp ← lexar_append(Silex, Sret, LEXICAT_DELIMITER, LLF_NONE, &cleanup);
  lexar_reset(SO(Silex), lexar_iterator(SO(Silex)));
  SS(Ssdelim, tmp);
  ⟨ Scan a raw string for its closing delimiter 207 ⟩
```

206. A raw string delimiter is "everything" from one ⟨⟨`$`⟩⟩ to another.

TODO: Broaden the range of permitted code-points from [`a-zA-Z0-9_-`].

```
⟨ Scan a delimiter 206 ⟩ ≡
  lexar_take(Silex, failure);
  while (1) {
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    lexar_another(c, Silex, true, false, LEXAR_premature_eof, Sret, failure);
    v ← rune(c);
    if (v ≡ '$') break;
    else if (¬((v ≥ 'a' ∧ v ≤ 'z') ∨ (v ≥ 'A' ∧ v ≤ 'Z') ∨ (v ≥ '0' ∧ v ≤ '9') ∨ v ≡ '_' ∨ v ≡ '-'))
      return lexar_append(Silex, Sret, LEXICAT_INVALID, LLF_NONE, failure);
  }
```

This code is used in section 205.

207. Scanning a raw string bypasses the usual lexar API to iterate over the rope by bytes instead of runes. Each byte is accepted and ignored unless it's ⟨⟨`$`⟩⟩ which begins scanning for the closing delimiter.

```
⟨ Scan a raw string for its closing delimiter 207 ⟩ ≡
  lexar(SO(Silex))→tbstart ← rope_iter(lexar_iterator(SO(Silex)))→tboffset;
  while (1) {
    if (Interrupt) siglongjmp (cleanup, LERR_INTERRUPT);
    tmp ← lexar_clone(SO(Silex), &cleanup);
      /* Remember where we are if the delimiter is about to start. */
    SS(Sedelim, tmp);
    v ← rope_iterate_next_byte(lexar_iterator(SO(Silex)), &cleanup);
    lexar(SO(Silex))→blength++;
    if (v ≡ EOF) goto LEXAR_raw_eof;
    else if (v ≠ '$') continue;
redelimiter:   /* Think "$abc$...$ab$ab$abc$". */
    ⟨ Scan a potential closing delimiter 208 ⟩
  }
delimited:
  ⟨ Finish and return a raw string/symbol combination 209 ⟩
```

This code is used in section 205.

208. A literal `$` may be the beginning of a closing delimiter or it may be part of the string, as may anything following the `$` up to the closing delimiter's closing `$`.

To deal with this irritating state of affairs whenever a `$` is first encountered its position has already been saved in the stack at *Sedlim*. While scanning for a closing delimiter the state which may have to be saved if this turns out not to be a closing delimiter but a `$` suggests another one might be starting is saved at *Srdelim*.

With that book-keeping out the way the rope underlying the opening delimiter is iterated along with the source rope until the closing delimiter is complete or a conflict between the two indicates that the string is not finished.

⟨ Scan a potential closing delimiter 208 ⟩ ≡

```
tmp ← rope_iterate_start(lexeme_twine(SO(Ssdelim)), lexeme(SO(Ssdelim))→tboffset, &cleanup);
SS(Sditer, tmp);
rope_iterate_next_byte(tmp, failure);      /* ⟨⟨$⟩⟩ — Will not fail (or be multi-byte). */
while (1) {
  if (Interrupt) siglongjmp (cleanup, LERR_INTERRUPT);
  SS(Srdelim, lexar_clone(SO(Silex), &cleanup));      /* Where the delimiter might re-start. */
  v ← rope_iterate_next_byte(lexar_iterator(SO(Silex)), failure);
  if (v ≡ EOF) goto LEXAR_raw_eof;
  else {
    lexar(SO(Silex))→cplength++;
    d ← rope_iterate_next_byte(SO(Sditer), &cleanup);      /* Will not fail. */
    if (v ≠ d) {
      if (v ≡ '$') {
        SS(Sedelim, SO(Srdelim));
        lexar(SO(Silex))→blength += lexar(SO(Silex))→cplength;
        lexar(SO(Silex))→cplength ← 0;
        goto redelimiter;
      }
      else break;
    }
    else if (v ≡ '$') goto delimited;
  }
}
lexar(SO(Silex))→blength += lexar(SO(Silex))→cplength;
lexar(SO(Silex))→cplength ← 0;
```

This code is used in section 207.

209. After a closing delimiter is successfully scanned the saved analyser state clones are used to create the three lexemes covering the string and its delimiters.

TODO: gcc warns that *irope* and *idelim* are set but not used — are they a holdover from earlier buggier days?

⟨ Finish and return a raw string/symbol combination 209 ⟩ ≡

```
irope ← rope_iter(lexar_iterator(SO(Silex)));
idelim ← rope_iter(lexar_iterator(SO(Sedelim)));
lexar(SO(Silex))→cplength++;
lexar(SO(Silex))→blength ← lexar(SO(Silex))→cplength;
lexar(SO(Silex))→tbstart += lexar(SO(Sedelim))→blength;
lexar(SO(Sedelim))→cplength ← 0;
lexar_append(Sedelim, Sret, cat, LLF_NONE, &cleanup);
r ← lexar_append(Silex, Sret, LEXICAT_DELIMITER, LLF_NONE, &cleanup);
stack_clear(4);
return r;
```

This code is used in section 207.

210. Curios. Apart from strings, symbols and numbers (to follow) there are curious tokens which are "all bets are side off" tokens that begin with the rune ⟪`#`⟫. That is to say that the exact rules of how to parse a token beginning with a `#` are custom to each curious token.

The curious tokens that `LossLess` understands, working in tandem with the syntax parser, are:

`#f`/`#F` or `#t`/`#T`	Boolean false or true.
`#=` ⟨*symbol*⟩	Denominate a recursive expression.
`##` ⟨*symbol*⟩	Refer to a recursive expression.
`#`[`bBoOdDxX`]...	Curious number — in base 2, 8, 10 or 16 respectively.
`#$`⟨*delimiter*⟩`$`...	Beginning of a delimited *symbol*.
`#|`...`|`	Symbol with escape characters.

Note that curiously signed numbers (±`#`[`bodx`]...) are not scanned for here. Any other rune following a `#` is considered an error for now.

```
⟨ Look for a curious token 210 ⟩ ≡
case '#':
  lexar_take(Silex, failure);
  lexar_another(c, Silex, true, false, LEXAR_premature_eof, Sret, failure);
  switch (rune(c)) {
    ⟨ Detect a curious number 214 ⟩
  case 'f': case 'F':
  case 't': case 'T': return lexar_append(Silex, Sret, LEXICAT_CONSTANT, LLF_NONE, failure);
  case '=': return lexar_append(Silex, Sret, LEXICAT_RECURSE_IS, LLF_NONE, failure);
  case '#': return lexar_append(Silex, Sret, LEXICAT_RECURSE_HERE, LLF_NONE, failure);
  case '$': cat ← LEXICAT_RAW_SYMBOL; goto delimiter;
  case '|': cat ← LEXICAT_ESCAPED_SYMBOL; goto string;
  default: return lexar_append(Silex, Sret, LEXICAT_INVALID, LLF_NONE, failure);
  }
  break;
```

This code is used in section 199.

211. Numbers. A number might be encountered bare by beginning with an ASCII-encoded Arabic digit `0`–`9` or it might be signed or be curiously encoded in a base other than 10.

An initial `-` or `+` may be the beginning of a number or a symbol. These succeeded by a digit `0`–`9` is a number, by `.` requires further testing and anything else was a symbol, which the syntax parser may reject.

```
#define UCP_INFINITY 0x211e
#define UCP_NAN_0 0x2116
#define UCP_NAN_1 0x20e0
```

⟨ Look for a number 211 ⟩ ≡

```
case '-': case '+':
  has_sign ← (c ≡ '-') ? LLF_NEGATIVE : LLF_POSITIVE;
  lexar_take(Silex, failure);
  lexar_another(c, Silex, false, false, false_symbol, Sret, failure);   /* See what's next. */
  switch (rune(c)) {⟨ Look for a signed number 212 ⟩}
```

See also sections 219, 220, and 221.

This code is used in section 199.

212. ± followed by is the start of a number.

⟨ Look for a signed number 212 ⟩ ≡

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
  want_digit ← CAN;
  goto number;
```

See also sections 213, 215, 216, and 217.

This code is used in section 211.

213. ⟪±#⟫ might be a signed curious number or it might be a tokenisation error. ⟪±.⟫ might be a signed vague number or a signed vague symbol.

⟨ Look for a signed number 212 ⟩ +≡

```
case '#': lexar_take(Silex, failure);
  lexar_another(c, Silex, true, false, LEXAR_premature_eof, Sret, failure);
  switch (rune(c)) {⟨ Detect a curious number 214 ⟩}   /* Now with has_sign set. */
  return lexar_append(Silex, Sret, LEXICAT_INVALID, LLF_NONE, failure);
case '.': lexar_take(Silex, failure);
  has_ratio ← LLF_DOT;
  want_digit ← MUST;
  lexar_another(c, Silex, false, false, noisy_false_symbol, Sret, failure);
  v ← rune(c);
  if (v ≥ '0' ∧ v ≤ '9') goto number;
  goto noisy_false_symbol;
  WARN();
```

214. A curious number, ie. one in any base, may or may not be signed. In both cases this same section of code is included so if the conclusion were to be reached with a standard C **goto** there would be two destinations in the *lexar_token* function with the same label. To get around this, this inelegant **case**/**if**/**else** adaptation of Duff's device[1] is used in place of a **goto** into the final block.

In effect the "**if** (1)" can be ignored if **else** is read as "**goto case** `'[xX]'` after it has set *base*".

⟨ Detect a curious number 214 ⟩ ≡

```
case 'b': case 'B': if (1) base ← 2; else
case 'd': case 'D': if (1) base ← 10; else
case 'o': case 'O': if (1) base ← 8; else
case 'x': case 'X': if (1) base ← 16;     /* comefrom the other cases to after this. */
  flags ← has_sign | base2flag(base);
  lexar_append(Silex, Sret, LEXICAT_CURIOUS, flags, failure); want_digit ← MUST;
  goto number;
```

This code is used in sections 210 and 213.

215. ⟨⟨±`i`⟩⟩ might begin the number ⟨⟨±`inf`⟩⟩ or it might indicate an ambiguous symbol. A literal infinity symbol is unambiguous.

⟨ Look for a signed number 212 ⟩ +≡

```
case 'i': case 'I': lexar_take(Silex, failure);
  lexar_another(c, Silex, false, false, noisy_false_symbol, Sret, failure);
  v ← rune(c);
  if (v ≡ 'n' ∨ v ≡ 'N') {
    lexar_take(Silex, failure);
    lexar_another(c, Silex, false, false, false_symbol, Sret, failure);
    v ← rune(c);
    if (v ≡ 'f' ∨ v ≡ 'F') goto infinity;
  }
  goto false_symbol;
case UCP_INFINITY:
  goto infinity;
```

216. Scanning ±`nan` is the same as ±`inf` apart from the letters. A literal not-a-number symbol is likewise unambiguous but consists of two the distinct runes `UCP_NAN_0` followed by `UCP_NAN_1`.

⟨ Look for a signed number 212 ⟩ +≡

```
case 'n': case 'N': lexar_take(Silex, failure);
  lexar_another(c, Silex, false, false, noisy_false_symbol, Sret, failure);
  v ← rune(c);
  if (v ≡ 'a' ∨ v ≡ 'A') {
    lexar_take(Silex, failure);
    lexar_another(c, Silex, false, false, false_symbol, Sret, failure);
    v ← rune(c);
    if (v ≡ 'n' ∨ v ≡ 'N') goto nan;
  }
  goto false_symbol;
case UCP_NAN_0: lexar_take(Silex, failure);
  lexar_another(c, Silex, false, false, noisy_false_symbol, Sret, failure);
  if (rune(c) ≡ UCP_NAN_1) goto nan;
  goto noisy_false_symbol;
```

[1] https://en.wikipedia.org/wiki/Duff%27s_device

217. If something that initially looked like a number turned out to be a symbol the analyser continues to scan as though it were always a symbol but emits a warning indicating that the scanner/parser might be confused.

This warning mechanism is poorly concieved (TODO).

⟨ Look for a signed number 212 ⟩ +≡
noisy_false_symbol: `WARN`( );
default:
false_symbol: *lexar_putback*(*Silex*); **goto** *symbol*;

218. When scanning a number the analyser, for readability's sake the analyer allows sequential digits to be separated by an underscore rune ⟪_⟫. Although readability's sake dictates that this would be used every third rune or so the reality is that `LossLess` must accept more. When a `_` rune is permissable *want_digit* is set to the false value `CAN` and when it is not to the truth `MUST`. When not only can a `_` be accepted but neither can a digit it is set to the other truth `CANNOT`.

#define `CANNOT` -1
#define `CAN` 0
#define `MUST` 1

219. Curious numbers aside, numbers might not be a number — the not-a-number symbol, or this block is jumped into by signed ±`nan` (see above). *has_ratio* is set to ensure that a `.` or `/` rune is no longer considered valid.

⟨ Look for a number 211 ⟩ +≡
case `UCP_NAN_0`: *lexar_take*(*Silex*, *failure*);
  lexar_another(*c*, *Silex*, *false*, *false*, *false_symbol*, *Sret*, *failure*);
  if (*rune*(*c*) ≠ `UCP_NAN_1`) **goto** *noisy_false_symbol*;
nan:
  lexar_take(*Silex*, *failure*);
  lexar_another(*c*, *Silex*, *false*, *false*, *LEXAR_premature_eof*, *Sret*, *failure*);
  if (*rune*(*c*) ≡ `'.'`) *has_ratio* ← `LLF_DOT`;   /∗ Not really a ratio. ∗/
  else *cat* ← `LEXICAT_INVALID`;
  want_digit ← `MUST`;
  goto *number*;

220. Similarly a number might be infinity.

⟨ Look for a number 211 ⟩ +≡
case `UCP_INFINITY`:
infinity:
  lexar_take(*Silex*, *failure*);
  has_ratio ← `LLF_RATIO`;   /∗ Not really a ratio. ∗/
  want_digit ← `CANNOT`;
  goto *number*;

221. Normal numbers, though, begin with one of the standard 10 digits. When scanning a number is finished a lexeme is finally emitted unless a(nother) digit is required.

⟨ Look for a number 211 ⟩ +≡

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
  want_digit ← CAN;
number:
  lexar_take(Silex, failure);
  while (1) {
    if (Interrupt) siglongjmp (*failure, LERR_INTERRUPT);
    lexar_another(c, Silex, false, false, finish, Sret, failure);
    switch ((v ← rune(c))) {⟨ Scan the body of a number 222 ⟩}
  }
finish:
  if (want_digit ≡ MUST) cat ← LEXICAT_INVALID;
  else if (cat ≡ LEXICAT_NONE) cat ← LEXICAT_NUMBER;
  flags ← has_sign | base2flag(base) | has_imagination;
  lexar_append(Silex, Sret, cat, flags, failure);
  if (eof_p(c)) {
    lexar_reset(SO(Silex), lexar_iterator(SO(Silex)));
    return lexar_append(Silex, Sret, LEXICAT_END, LLF_NONE, failure);
  }
  return dlist_datum(SO(Sret));
```

222. All types of scanning for number eventually end up here, which checks that a digit's representation fits within the current base or permits a lone _ to pass, toggling *want_digit*.

⟨ Scan the body of a number 222 ⟩ ≡

```
case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
  if (base < 16) cat ← LEXICAT_INVALID;
case '9': case '8':
  if (base < 10) cat ← LEXICAT_INVALID;
case '7': case '6': case '5':
case '4': case '3': case '2':
  if (base < 8) cat ← LEXICAT_INVALID;
case '1': case '0':
  lexar_take(Silex, failure);
  if (want_digit < CAN) cat ← LEXICAT_INVALID;
  want_digit ← CAN;
  break;
case '_': lexar_take(Silex, failure);
  if (want_digit ≠ CAN) cat ← LEXICAT_INVALID;
  want_digit ← MUST;
  break;
```

See also sections 223, 224, and 226.

This code is used in section 221.

223. A single number lexeme can include a lone `.` or `/`, followed by a number following the regular scanning rules, to represent a ratio.

⟨ Scan the body of a number 222 ⟩ +≡

```
case '/':
  if (has_ratio) cat ← LEXICAT_INVALID;
  else if (1) has_ratio ← LLF_SLASH;
  else
case '.':
  if (has_ratio) cat ← LEXICAT_INVALID;
  else if (1) has_ratio ← LLF_DOT;
  lexar_take(Silex, failure);
  want_digit ← MUST;
  break;
```

224. A normal rational, infinite or non-number can be succeeded by `i`, `j` or `k` which represents a number's imaginary or quaterniate component. At this stage of analysis such a rune terminates scanning this lexeme and it is left up to the parser to ensure that successive numeric components are valid (that is to say that at the moment such numbers are in `LossLess` wholly imaginary, or unimplemented).

⟨ Scan the body of a number 222 ⟩ +≡

```
case 'i': case 'j': case 'k': lexar_take(Silex, failure);
  has_imagination ← LLF_IMAGINATE(v − 'i');
  goto finish;
```

225. If a rune less numeric than a NaN is encountered while scanning a number then it's put back into the analyser and the number lexeme ends.

226. ⟨ Scan the body of a number 222 ⟩ +≡

```
default: lexar_putback(Silex);
  if (¬lexar_terminator_p(c)) cat ← LEXICAT_INVALID;
  goto finish;
```

227. The only practical way to use the lexical analyser so far is this *lex_rope* which expects an entire rope to have been read in already.

```
cell lex_rope(cell src, sigjmp_buf *failure)
{
  static int Ssource ← 3, Siter ← 2, Snext ← 1, Sret ← 0;
  cell r ← NIL, tmp;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(rope_p(src));
  stack_protect(4, src, NIL, NIL, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 4);
  SS(Siter, tmp ← lexar_start(SO(Ssource), &cleanup));
  SS(Sret, tmp ← dlist_new(NIL, &cleanup));
  SS(Snext, tmp);
  while (1) {
    tmp ← lexar_token(Siter, Snext, &cleanup);
    assert(lexeme_p(tmp));
    if (lexeme(tmp)→cat ≡ LEXICAT_END) break;
  }
  r ← dlist_remove_m(SO(Sret));
  stack_clear(4);
  return r;
}
```

228. Syntax parser. The only entry point to the syntax parser is *parse* which accepts the list of lexemes produced by the lexical analyser and returns a syntax tree with, if any, the failures encountered while scanning it.

⟨ Function declarations 21 ⟩ +≡

cell *parse*(**cell**, **bool** ∗, **sigjmp_buf** ∗);
cell *transform_lexeme_segment*(**cell**, **long**, **long**, **bool**, **int**, **bool** ∗, **sigjmp_buf** ∗);
char *parse_ascii_hex*(**cell**, **sigjmp_buf** ∗);

229. A flat list of lexemes is transformed into a syntax tree in order. The parser keeps track of the lexeme at which translation began (*Sstart*) and that currently under consideration (*Sllex*). An (unused for now) empty environment (*Senv*) is created to hold the symbols used to build recursive structures; this environment is entirely separate from the run-time environments used by the evaluator (but does it need to be? TODO: discuss).

As parsing proceeds completed nodes are added to a stack (*Swork*) awaiting inclusion in a list. When an opening token (`(`, `[` or `{`) is encountered that lexeme is added instead and when a closing token (`)`, `]` or `}`) is encountered nodes are removed from this stack until the corresponding opening lexeme is reached.

If an invalid or unexpected lexeme is encountered or any other problem occurs its location is noted in a list of failures in *Sfail*, parsing continues and the caller is informed that the syntax is ultimately invalid.

```
#define parse_fail(S, E, L, F) do
        {
          cell _x ← cons(fix((E)), (L), (F));
          SS((S), cons(_x, SO(S), (F)));
          (L) ← lcdr(lcar(SO(S)));
        }
        while (0)

  cell parse(cell llex, bool *valid, sigjmp_buf *failure)
  {
    static int Sstart ← 6;    /* Where we started. */
    static int Sllex ← 5;    /* Where we are now. */
    static int Senv ← 4;    /* Namespace for syntactic recursion (#=/##). */
    static int Swork ← 3;    /* Stack of remaining work. */
    static int Sbuild ← 2;    /* Temporary workspace. */
    static int Stmp ← 1;    /* Temporarier workspace. */
    static int Sfail ← 0;    /* The litany of failure. */
    cell r, lex, x, y, z;    /* Work space so temporary it's ignored by the garbage collector. */
    Vlexicat cat ← LEXICAT_NONE;    /* The current lexeme's category. */
    Verror pfail ← LERR_NONE;    /* Any failure while constructing a list. */
    char *buf ← Λ;    /* Buffer space for parsing strings and symbols. */
    bool has_tail ← false, m;    /* Whether a . has been seen (and a temporary m). */
    long offset ← 0, a, b, c, i;    /* Short-lived temporary variables. */
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    assert(dlist_p(llex));    /* ... of lexemes. */
    lex ← dlist_datum(dlist_prev(llex));
    assert(lexeme_p(lex));
    assert(lexeme(lex)→cat ≡ LEXICAT_END);    /* Not saved in cat. */
    stack_protect(2, llex, llex, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
    stack_reserve(5, &cleanup);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 7);
    SS(Senv, env_empty(&cleanup));
    llex ← SO(Sllex);
    ⟨ Construct a syntax tree from a list of lexemes 230 ⟩
    r ← SO(Sbuild);
    if (¬*valid) r ← cons(r, SO(Sfail), &cleanup);
    stack_clear(7);
    return r;
  }
```

230. The main parser loop iterates along the lexeme list one at a time and ends when it reaches the terminating/starting `LEXICAT_END` lexeme. After this the parsed syntax tree will be left in *Sbuild* and *Swork* should be empty.

⟨ Construct a syntax tree from a list of lexemes 230 ⟩ ≡
```
while (cat ≠ LEXICAT_END) {
  if (Interrupt) siglongjmp (∗failure, LERR_INTERRUPT);
  lex ← dlist_datum(llex);
  assert(lexeme_p(lex));
  cat ← lexeme(lex)→cat;
  switch (cat) {⟨ Process the next lexeme 231 ⟩}
  SS(Sllex, llex ← dlist_next(SO(Sllex)));
}
assert(null_p(SO(Swork)));      /∗ I think... ∗/
if (¬null_p(SO(Swork))) {
  SS(Stmp, SO(Sbuild));
  SS(Sbuild, NIL);
  while (¬null_p(SO(Swork))) {
    SS(Sbuild, cons(lcar(SO(Swork)), SO(Sbuild), &cleanup));
    SS(Swork, lcdr(SO(Swork)));
  }
  x ← syntax_invalid(SO(Sbuild), SO(Sstart), SO(Sllex), &cleanup);
  parse_fail(Sfail, LERR_SYNTAX, x, &cleanup);
  SS(Sbuild, cons(x, SO(Stmp), &cleanup));
}
if (¬null_p(SO(Sfail))) ∗valid ← false;
```
This code is used in section 229.

231. The simplest lexemes to handle are spaces which are ignored and invalid lexemes which are also ignored but only recording the failure.

⟨ Process the next lexeme 231 ⟩ ≡
```
case LEXICAT_SPACE: break;      /∗ Space is meaningless[1]. ∗/
case LEXICAT_INVALID: default: x ← syntax_invalid(lex, llex, llex, &cleanup);
  if (cat ≡ LEXICAT_INVALID) parse_fail(Sfail, LERR_UNSCANNABLE, x, &cleanup);
  else parse_fail(Sfail, LERR_INTERNAL, x, &cleanup);
  SS(Swork, cons(x, SO(Swork), &cleanup));
  break;
```
See also sections 232, 233, 237, 238, and 245.

This code is used in section 230.

232. Almost as simple is the two (four) boolean constants `#f` & `#t` (and `#F` & `#T`) which are appended to the work queue.

⟨ Process the next lexeme 231 ⟩ +≡
```
case LEXICAT_CONSTANT: a ← lexeme_byte(lex, 1);
  x ← predicate(a ≡ 't' ∨ a ≡ 'T');
  y ← dlist_datum(dlist_next(llex));
  if (¬lexeme_terminator_p(y)) {
    z ← syntax_invalid(x, llex, llex, &cleanup);
    parse_fail(Sfail, LERR_AMBIGUOUS, z, &cleanup);
  }
  else z ← syntax_new(x, llex, llex, &cleanup);
  SS(Swork, cons(z, SO(Swork), &cleanup));
  break;
```

[1] There's *literally everything* in space.

233. Building a list involves multiple lexemes working in tandem. A `LEXICAT_OPEN` lexeme is appended to the list waiting for a `LEXICAT_CLOSE` lexeme to consume it (and everything after it). A `LEXICAT_DOT` lexeme is also appended to the working list as-is and is also consumed en route to the `LEXICAT_OPEN` which started the list.

`LEXICAT_END` works similarly to `LEXICAT_CLOSE` in the way it consumes list items except that encountering a pending `LEXICAT_OPEN` (or a `LEXICAT_DOT`) indicates an error and the beginning of the list is instead indicated by the working list being entirely consumed.

⟨Process the next lexeme 231⟩ +≡
```
case LEXICAT_OPEN: case LEXICAT_DOT: SS(Swork, cons(llex, SO(Swork), &cleanup));
  break;
case LEXICAT_END: assert(dlist_next(llex) ≡ SO(Sstart));
case LEXICAT_CLOSE: pfail ← LERR_NONE;
  has_tail ← false;
  SS(Sbuild, NIL);      /* Work in progress. */
  c ← 0;
  while (1) {⟨Finalise items stacked into Swork 234⟩}
  x ← SO(Sbuild);      /* Built object. */
  y ← SO(Stmp);      /* Starting lexeme. */
  z ← SO(Sllex);      /* Terminating lexeme. */
  if (pfail ≠ LERR_NONE) {
    SS(Sbuild, syntax_invalid(x, y, z, &cleanup));
    x ← SO(Sbuild);
    parse_fail(Sfail, LERR_SYNTAX, x, &cleanup);
  }
  else SS(Sbuild, syntax_new(x, y, z, &cleanup));
  if (cat ≠ LEXICAT_END)      /* Put it back on the head of Swork to carry on parsing. */
    SS(Swork, cons(SO(Sbuild), SO(Swork), &cleanup));
  break;
```

234. Looping until the working queue is consumed — as expected if the current lexeme is a (the) `LEXICAT_END` — the list is built item by item into *Sbuild*. If the queue of work is fully consumed and the current lexeme is *not* `LEXICAT_END` this indicates an attempt to close a list which was not opened.

⟨Finalise items stacked into *Swork* 234⟩ ≡
```
  c++;
  if (Interrupt) siglongjmp (cleanup, LERR_INTERRUPT);
  llex ← SO(Sllex);
  if (null_p(SO(Swork))) {
    if (cat ≠ LEXICAT_END) parse_fail(Sfail, pfail ← LERR_UNOPENED_CLOSE, llex, &cleanup);
    SS(Stmp, SO(Sstart));
    break;
  }
  x ← lcar(SO(Swork));      /* The next working item to copy or process. */
  if (¬syntax_p(x)) {⟨Finish building the list or fix its tail 235⟩}
  SS(Sbuild, cons(lcar(SO(Swork)), SO(Sbuild), &cleanup));
  SS(Swork, lcdr(SO(Swork)));
```
This code is used in section 233.

235. `LEXICAT_DOT` can only appear under constrained circumstances (or not at all). If `LEXICAT_OPEN` is found and the current lexeme *is* `LEXICAT_END` that indicates a list was begun and not ended.

⟨ Finish building the list or fix its tail 235 ⟩ ≡

```
assert(dlist_p(x) ∧ lexeme_p(dlist_datum(x)));
x ← dlist_datum(x);    /* Opener ... */
if (lexeme(x)→cat ≡ LEXICAT_DOT) {
  if (lexeme(lex)→cat ≠ LEXICAT_CLOSE ∨ lexeme_byte(lex, 0) ≠ ')')
    parse_fail(Sfail, pfail ← LERR_LISTLESS_TAIL, llex, &cleanup);
  else if (has_tail) parse_fail(Sfail, pfail ← LERR_DOUBLE_TAIL, llex, &cleanup);
  else {
    has_tail ← true;
    if (null_p(SO(Sbuild))) parse_fail(Sfail, pfail ← LERR_EMPTY_TAIL, llex, &cleanup);
    else if (¬null_p(lcdr(SO(Sbuild)))) parse_fail(Sfail, pfail ← LERR_HEAVY_TAIL, llex, &cleanup);
    else {
      SS(Sbuild, lcar(SO(Sbuild)));
      SS(Swork, lcdr(SO(Swork)));
      continue;
    }
  }
}
else {    /* Found the/a LEXICAT_OPEN. */
  if (cat ≡ LEXICAT_END) parse_fail(Sfail, pfail ← LERR_UNCLOSED_OPEN, llex, &cleanup);
  else {⟨ Complete parsing a list-like syntax 236 ⟩}
  SS(Stmp, lcar(SO(Swork)));
  SS(Swork, lcdr(SO(Swork)));
  break;
}
```

This code is used in section 234.

236. After collecting all the items in a list-like construction the opening and closing brackets are checked that they match and which bracket it is indicates what object to create. Paired parentheses ⟪`( ... )`⟫ surround a list and parsing is complete. Square brackes ⟪`[ ... ]`⟫ mark an array which the list is converted into and braces ⟪`{ ... }`⟫ produce a relation, which is unimplemented.

⟨ Complete parsing a list-like syntax 236 ⟩ ≡

```
lex ← dlist_datum(SO(Sllex));    /* ... Closer */
assert(rope_p(lexeme_twine(lex)));
a ← lexeme_byte(x, 0);    /* (, [ or {. */
a ← ((a + 1) & ∼1) + 1;    /* ASCII tricks → what we want. */
if (cat ≡ LEXICAT_END) b ← '\0';
else b ← lexeme_byte(lex, 0);    /* What we got. */
if (a ≠ b) parse_fail(Sfail, pfail ← LERR_MISMATCH, llex, &cleanup);
else if (a ≡ ']') {
  x ← array_new_imp(c, UNDEFINED, FORM_ARRAY, &cleanup);
  y ← SO(Sbuild);
  for (i ← 0; i < c; i++, y ← lcdr(y)) array_set_m(x, i, lcar(y));
  SS(Sbuild, x);
}
else if (a ≡ '}') parse_fail(Sfail, pfail ← LERR_UNIMPLEMENTED, llex, &cleanup);
```

This code is used in section 235.

237. A simple `LEXICAT_SYMBOL` can be read directly into a segment and converted into a symbol.

```
⟨ Process the next lexeme 231 ⟩ +≡
case LEXICAT_SYMBOL: y ← dlist_datum(dlist_next(SO(Sllex)));
  if (¬lexeme_terminator_p(y)) {
    z ← syntax_invalid(lex, llex, llex, &cleanup);
    parse_fail(Sfail, LERR_AMBIGUOUS, z, &cleanup);
  }
  else {
    a ← lexeme(lex)→blength;
    SS(Sbuild, x ← segment_new(0, a, 0, 0, &cleanup));
    buf ← segment_address(x);
    lex ← dlist_datum(SO(Sllex));
    x ← rope_iterate_start(lexeme_twine(lex), lexeme(lex)→tboffset, &cleanup);
    for (i ← 0; i < a; i++) buf[i] ← rope_iterate_next_byte(x, &cleanup);
    y ← symbol_new_buffer(buf, a, &cleanup);
    z ← syntax_new(y, SO(Sllex), SO(Sllex), &cleanup);
    buf ← Λ;
  }
  SS(Swork, cons(z, SO(Swork), &cleanup));
  break;
```

238. A similar process is used to read the body of symbols and strings which are included raw (delimited) or with embededded escape characters. In the first case a buffer of the appropriate size can be created, filled and used directly. In the latter it must be processed to convert any escape character combinations within it.

Unlike a plain symbol for convenience these four cases all use the more heavy-weight method outlined in *transform_lexeme_segment* to copy and optionally transform a lexeme into a segment.

```
⟨ Process the next lexeme 231 ⟩ +≡
case LEXICAT_ESCAPED_SYMBOL: offset++;    /* |...| */
case LEXICAT_ESCAPED_STRING: offset++;    /* #|...| */
  SS(Stmp, llex);
case LEXICAT_DELIMITER:
  if (cat ≡ LEXICAT_DELIMITER) {    /* [#]$xxx$...$xxx$ */
  ⟨ Validate the lexical triplet in a delimited string/symbol 239 ⟩}    /* Sets z if there was an error. */
  if (null_p(lex)) SS(Sbuild, z);
  else {
    m ← true;
    x ← transform_lexeme_segment(lex, offset, lexeme(lex)→blength, (offset ≠ 0), Sfail, &m, &cleanup);
    SS(Sbuild, x);
    if (¬m) SS(Sbuild, syntax_invalid(SO(Sbuild), SO(Sllex), SO(Stmp), &cleanup));
    else {
      if (cat ≡ LEXICAT_RAW_STRING ∨ cat ≡ LEXICAT_ESCAPED_STRING)
        y ← rope_new_buffer(true, true, segment_address(x), segment_length(x), &cleanup);
      else y ← symbol_new_segment(x, &cleanup);
      SS(Sbuild, syntax_new(y, SO(Sllex), SO(Stmp), &cleanup));
    }
  }
  SS(Swork, cons(SO(Sbuild), SO(Swork), &cleanup));
  if (cat ≠ LEXICAT_ESCAPED_SYMBOL ∧ cat ≠ LEXICAT_ESCAPED_STRING) SS(Sllex, SO(Stmp));
  offset ← 0;    /* Must always begin at zero. */
  break;
```

239. A delimited string or symbol consists of two `LEXICAT_DELIMITER`s with a `LEXICAT_RAW_STRING` or `LEXICAT_RAW_SYMBOL` in the middle. No other combination of these lexemes is valid and the contents of the delimiters must match exactly. These are not actually verified but is what is created by the lexical analyser.

Strictly speaking there's no need to enforce requiring a terminating lexeme after an escapable or delimited string or symbol but is done for consistency with plain symbols and also to catch some ambiguous potential mistakes such as ⟪`|foreshort 4|2`⟫.

⟨ Validate the lexical triplet in a delimited string/symbol 239 ⟩ ≡

```
SS(Sbuild, x ← dlist_next(llex));      /* String/symbol content. */
SS(Stmp, y ← dlist_next(x));      /* Closing delimiter. */
lex ← dlist_datum(x);
cat ← lexeme(lex)→cat;
if (cat ≡ LEXICAT_INVALID) {      /* Source ended without the closing delimiter. */
  z ← syntax_invalid(lex, SO(Sllex), SO(Sbuild), &cleanup);
  parse_fail(Sfail, LERR_UNSCANNABLE, z, &cleanup);
  SS(Stmp, SO(Sbuild));
  lex ← NIL;
}
else {
  assert(cat ≡ LEXICAT_RAW_STRING ∨ cat ≡ LEXICAT_RAW_SYMBOL);
  z ← dlist_next(y);
  assert(lexeme(dlist_datum(y))→cat ≡ LEXICAT_DELIMITER);
  if (¬lexeme_terminator_p(dlist_datum(z))) {
    z ← syntax_invalid(lex, lcar(SO(Swork)), SO(Stmp), &cleanup);
    parse_fail(Sfail, LERR_AMBIGUOUS, z, &cleanup);
    lex ← NIL;
  }
}
```

This code is used in section 238.

240. To create a string or symbol object from source the opening delimiter's *offset* bytes are skipped then *length* bytes are read by iterating over the source rope.

Note that *transform_lexeme_segment* iterates over *bytes* not runes to accomodate both raw data and regular data, which has already had its runes' underlying bytes validated by the lexical analyser.

This function has an awful name (TODO).

```
cell transform_lexeme_segment(cell o, long offset, long length, bool escape, int Sfail,
        bool *valid, sigjmp_buf *failure)
{
  static int Ssrc ← 2, Sdst ← 1, Siter ← 0;
  cell r ← NIL, tmp;
  char *buf, b;
  long i, j, k;
  int32_t cp;
  Outfio ucp;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(lexeme_p(o));
  assert(offset ≥ 0 ∧ offset < lexeme(o)→blength);
  assert(length ≥ 1 ∧ lexeme(o)→blength − offset ≤ length);
  stack_protect(3, o, NIL, NIL, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 3);
  Sfail += 3;
  SS(Siter, rope_iterate_start(lexeme_twine(SO(Ssrc)), lexeme(SO(Ssrc))→tboffset, &cleanup));
  if (offset) length −= offset + 1;
  while (offset−−) rope_iterate_next_byte(SO(Siter), &cleanup);
  SS(Sdst, segment_new(0, length, 0, 0, &cleanup));
  buf ← segment_address(SO(Sdst));
  ⟨Copy, transforming, length bytes after offset 241⟩
  r ← SO(Sdst);
  stack_clear(3);
  return r;
}
```

241. If the data will not include escape sequences the rope can be simply copied, otherwise each byte is examined and copied or transformed. The segment written to is reduced in size as necessary prior to being returned.

```
⟨Copy, transforming, length bytes after offset 241⟩ ≡
  if (¬escape)
    for (i ← 0; i < length; i++) buf[i] ← rope_iterate_next_byte(SO(Siter), &cleanup);
  else {
    j ← 0;
    for (i ← 0; i < length; i++) {
      b ← rope_iterate_next_byte(SO(Siter), &cleanup);
      if (b ≠ '#') buf[j++] ← b;
      else {
        i++;
        b ← rope_iterate_next_byte(SO(Siter), &cleanup);
        switch (b) {⟨Append an escaped byte sequence 243⟩}
      }
    }
    if (i ≠ j) SS(Sdst, segment_resize_m(SO(Sdst), 0, j − i, &cleanup));
  }
```

This code is used in section 240.

242. This macro and function converts one or two ASCII-encoded hex digits into their numeric value.

#define $hexscii\_to\_int(O)$ $(((O) \geq$ `'a'`$)$? $(O) -$ `'a'` : $((O) \geq$ `'A'`$)$? $(O) -$ `'A'` : $(O) -$ `'0'`$)$
#define $int\_to\_hexscii(O, C)$ $((O) < 10$? $(O) +$ `'0'` : $((C)$? $(O) +$ `'A'` : $(O) +$ `'a'`$))$

```
char parse_ascii_hex(cell o, sigjmp_buf *failure)
{
  int b, r;
  assert(rope_iter(o));
  b ← rope_iterate_next_byte(o, failure);
  r ← hexscii_to_int(b) ≪ 4;
  b ← rope_iterate_next_byte(o, failure);
  r |= hexscii_to_int(b);
  return (char) r;
}
```

243. The lexical analyser has ensured all escape sequences are validly encoded so these sequences need no special effort.

⟨ Append an escaped byte sequence 243 ⟩ ≡

```
case '#': case '|': buf[j++] ← b;
  break;
case 'o': case 'O':    /* A byte represented by 3 octal digits. */
  buf[j] ← (rope_iterate_next_byte(SO(Siter), &cleanup) − '0') ≪ 6;
  buf[j] |= (rope_iterate_next_byte(SO(Siter), &cleanup) − '0') ≪ 3;
  buf[j] |= (rope_iterate_next_byte(SO(Siter), &cleanup) − '0');
  i += 3;
  j++;
  break;
case 'x': case 'X':    /* A byte represented by 3 hex digits. */
  i += 2;
  buf[j++] ← parse_ascii_hex(SO(Siter), &cleanup);
  break;
```

See also section 244.

This code is used in section 241.

244. Although the source is well-encoded, the value representing an escaped rune may not be a valid unicode code point. Such non-characters are appended instead as `UCP_REPLACEMENT` (`U+FFFD`).

Note that a literal rune with value `U+FFFD` is *not* a parser error.

⟨ Append an escaped byte sequence 243 ⟩ +≡

```
case 'u': case 'U':    /* A UTF-8 encoded byte sequence represented by 4 hex digits. */
  cp ← 0;
  cp ← parse_ascii_hex(SO(Siter), &cleanup) ≪ 8;
  cp |= parse_ascii_hex(SO(Siter), &cleanup);
  i += 4;
  goto escaped_rune;
case '(':    /* A UTF-8 encoded byte sequence represented by 1-6 hex digits. */
  cp ← 0;
  rope_iterate_next_byte(SO(Siter), &cleanup);    /* ( */
  while ((b ← rope_iterate_next_byte(SO(Siter), &cleanup)) ≠ ')') {
    i++;
    cp ≪= 4;
    cp |= hexscii_to_int(b);
  }
  i += 2;
escaped_rune: ucp ← utfio_write(cp);
  tmp ← rune_new_utfio(ucp, &cleanup);
  if (rune_failure_p(tmp)) {
    *valid ← false;
    tmp ← fix(j);
    parse_fail(Sfail, LERR_NONCHARACTER, tmp, &cleanup);
    ucp ← utfio_write(UCP_REPLACEMENT);
    tmp ← rune_new_utfio(ucp, &cleanup);
  }
  for (k ← 0; k < rune_parsed(tmp); k++) buf[j++] ← ucp.buf[k];
```

245. This early implementation of `LossLess` doesn't support parsing numbers or syntactic recursion.

⟨ Process the next lexeme 231 ⟩ +≡

```
case LEXICAT_CURIOUS: case LEXICAT_NUMBER:
case LEXICAT_RECURSE_HERE: case LEXICAT_RECURSE_IS:
  z ← syntax_invalid(lex, llex, llex, &cleanup);
  parse_fail(Sfail, LERR_UNIMPLEMENTED, z, &cleanup);
  SS(Swork, cons(z, SO(Swork), &cleanup));
  break;
```

246. Evaluator. The evaluator is based distantly on that presented by Steele and Sussman in "Design of LISP-Based Processors'.

There are five registers used by the evaluator. The argument to *evaluate* — the expression which is to be computed — is saved in *Expression* (`EXPR`) and with *Arguments* (`ARGS`) they represent the state of the data being evaluated. Alongside those *Control_Link* (`CLINK`) then represents the state of the computation evaluating it in the form of a stack of partial work to later resume.

The run-time's current environment is in *Environment* (`ENV`) and the result of computation (or the partial result while computation is incomplete) in the *Accumulator* `ACC`.

As a general rule the shorter names are used within the evaluator to avoid being overwhelmed by verbosity.

#define `ACC` *Accumulator*
#define `ARGS` *Arguments*
#define `CLINK` *Control_Link*
#define `ENV` *Environment*
#define `EXPR` *Expression*

⟨ Global variables 15 ⟩ +≡
  unique cell *Accumulator* ← `NIL`;
  unique cell *Arguments* ← `NIL`;
  unique cell *Control_Link* ← `NIL`;
  unique cell *Environment* ← `NIL`;
  unique cell *Expression* ← `NIL`;

247. The accumulator (the answer) is the only part of the evaluator externally visible.

⟨ External symbols 16 ⟩ +≡
  extern unique cell *Accumulator*;
  extern unique cell *Environment*;

248. **#define** *evaluate_desyntax*(*O*) (*syntax_p*(*O*) ? *syntax_datum*(*O*) : (*O*))
#define *evaluate_incompatible*(*L*, *F*) **do**
    {
      lprint("incompatibility␣at␣line␣%d\n", (*L*));
      siglongjmp (∗(*F*), `LERR_INCOMPATIBLE`);
    }
    while (0)

⟨ Function declarations 21 ⟩ +≡
  void *evaluate*(**cell**, **sigjmp_buf** ∗);
  void *evaluate_program*(**cell**, **sigjmp_buf** ∗);
  void *combine*(**sigjmp_buf** ∗);
  void *validate_formals*(**bool**, **sigjmp_buf** ∗);
  void *validate_arguments*(**sigjmp_buf** ∗);
  void *validate_operative*(**sigjmp_buf** ∗);
  void *validate_primitive*(**sigjmp_buf** ∗);

249. Evaluation begins by saving the whole expression in *Arguments*. The control link and arguments registers must be empty.

The evaluation algorithm is written here taking very little advantage of syntax C offers, not even passing the result of one function as the direct argument of another. This is primarily because that's how the algorithm was originally written by Sussman & Steele (their goal was to write code to be translated directly to silicon) however it is also easier to describe and, in the author's opinion, understand than it would be if it were presented in a "higher-level" form than what is effectively dressed up machine code.

The algorithm as a whole consists of labeled chunks (as they will be referred to) of code which terminate by branching to another chunk. There is no "returning" to a partially-complete chunk. With few exceptions each chunk either carries out the next stage of evaluation or performs a conditional jump into another section (this is a restriction useful[1] to silicon which is not necessary in C but remains for familiarity).

Broadly speaking the algorithm takes the form of a loop starting at the *Begin* chunk after the expression to evaluate has been prepared in *Expression*. *Begin* dispatches based on the format of the expression — symbols are looked up in the environment, pairs are combined and re-evaluated and other atoms remain themselves. After evaluation is complete control will proceed to either *Finish* or *Return* and *evaluate* will **return** if computation has indeed finished, or dispatch to another chunk as directed by the head of the control link stack.

```
void evaluate(cell o, sigjmp_buf *failure)
{
    assert(null_p(CLINK) ∧ null_p(ARGS));
    assert(environment_p(ENV));
    EXPR ← o;
    LOG(ACC ← VOID);
Begin:
    EXPR ← evaluate_desyntax(EXPR);
    if (pair_p(EXPR)) goto Combine_Start;
    else if (¬symbol_p(EXPR)) goto Finish;
    LOG(ACC ← env_search(ENV, EXPR, true, failure));
    if (undefined_p(ACC)) {
        lprint("looking␣for␣");
        serial(EXPR, SERIAL_ROUND, 1, NIL, Λ, failure);
        lprint("\n");
        LOG(ACC ← VOID);
        siglongjmp(*failure, LERR_MISSING);
    }
    goto Return;
Evaluate: LOG(EXPR ← note_car(CLINK));
    LOG(CLINK ← note_cdr(CLINK));
    goto Begin;
⟨Evaluate a complex expression 252⟩
Finish: LOG(ACC ← EXPR);
Return:   /* Check CLINK to see if there is more work after one full evaluation. */
    if (null_p(CLINK)) return;   /* Accumulator (ACC) has the result. */
    else if (¬note_p(CLINK)) siglongjmp(*failure, LERR_INTERNAL);
    else if (note(CLINK) ≡ Sym_EVALUATE) goto Evaluate;
    else if (note(CLINK) ≡ Sym_COMBINE_APPLY) goto Combine_Apply;
    else if (note(CLINK) ≡ Sym_COMBINE_BUILD) goto Applicative_Build;
    else if (note(CLINK) ≡ Sym_COMBINE_DISPATCH) goto Combine_Dispatch;
    else if (note(CLINK) ≡ Sym_COMBINE_FINISH) goto Combine_Finish;
    else if (note(CLINK) ≡ Sym_COMBINE_OPERATE) goto Combine_Operate;
    else if (note(CLINK) ≡ Sym_OPERATIVE) goto Operative_Closure;
    else if (note(CLINK) ≡ Sym_APPLICATIVE) goto Applicative_Closure;
    else if (note(CLINK) ≡ Sym_CONDITIONAL) goto Conditional;
```

[1] I expect.

```
    else if (note(CLINK) ≡ Sym_ENVIRONMENT_P) goto Validate_Environment;
    else if (note(CLINK) ≡ Sym_EVALUATE_DISPATCH) goto Evaluate_Dispatch;
    else if (note(CLINK) ≡ Sym_SAVE_AND_EVALUATE) goto Save_And_Evaluate;
    else if (note(CLINK) ≡ Sym_ENVIRONMENT_M) goto Restore_Environment;
    else if (note(CLINK) ≡ Sym_DEFINITION) goto Mutate_Environment;
    else siglongjmp(*failure, LERR_INTERNAL);    /* Unknown note. */
  }
```

250.

```
void evaluate_program(cell o, sigjmp_buf *failure)
  {
    static int Sprogram ← 0;
    cell program;
    bool syntactic;
    sigjmp_buf cleanup;
    Verror reason ← LERR_NONE;

    stack_protect(1, o, failure);
    if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
    program ← env_search(Root, symbol_new_const("do"), true, &cleanup);
    syntactic ← syntax_p(SO(Sprogram));
    if (syntactic) {
      program ← cons(program, syntax_datum(SO(Sprogram)), &cleanup);
      program ← syntax_new(program, syntax_start(SO(Sprogram)), syntax_end(SO(Sprogram)),
          &cleanup);
    }
    else program ← cons(program, SO(Sprogram), &cleanup);
    stack_clear(1);
    evaluate(program, failure);
  }
```

251. While building and debugging the evaluator it has proven invaluable to get a trace of the activity but it is exceptionally noisy. However LossLess is still in development so the macro is still here, disabled.

```
#define LOG(cmd) cmd
#define DONTLOG(cmd) do
        {
          printf("%s\n", #cmd);
          cmd;
        }
        while (0)
```

252. A "complex expression" is a list who's first element is an applicative or operative combiner, which is to say a function/procedure or some other code who's arguments are (applicative) or are not (operative) themselves evaluated before calling it.

Combine_Start is entered when a pair is evaluated. First the list of partially evaluated arguments (of which the result of combining this pair (calling this function) will be part), the current environment and the combiner's own arguments are saved in the control stack. The combiner is left to be evaluated so that *Combine_Dispatch* later knows where to dispatch to.

```
⟨ Evaluate a complex expression 252 ⟩ ≡
Combine_Start:    /* Save any ARGS in progress and ENV on CLINK to resume later. */
  LOG(CLINK ← cons(ARGS, CLINK, failure));
  LOG(CLINK ← cons(ENV, CLINK, failure));
  LOG(ARGS ← lcdr(EXPR));    /* Save the combination's arguments. */
  LOG(CLINK ← note_new(Sym_COMBINE_DISPATCH, ARGS, CLINK, failure));
  LOG(EXPR ← lcar(EXPR));    /* Prepare to evaluate the combiner. */
  goto Begin;
Combine_Dispatch:    /* Apply or operate based on the evaluated combinator expression. */
  LOG(ARGS ← note_car(CLINK));    /* Restore the combination's arguments. */
  LOG(CLINK ← note_cdr(CLINK));
  if (operative_p(ACC)) goto Combine_Operate;
  else if (applicative_p(ACC)) goto Applicative_Start;
  else siglongjmp(*failure, LERR_UNCOMBINABLE);
```

See also sections 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, and 266.
This code is used in section 249.

253. When combination finished with its result in the accumulator the previous arguments and environment are popped from the control link stack.

```
⟨ Evaluate a complex expression 252 ⟩ +≡
Combine_Finish:    /* Restore the ENV and ARGS in place before evaluating the combinator. */
  LOG(EXPR ← note_car(CLINK));    /* May include the arguments to an operative. */
  LOG(CLINK ← note_cdr(CLINK));
  LOG(ENV ← lcar(CLINK));
  LOG(CLINK ← lcdr(CLINK));
  LOG(ARGS ← lcar(CLINK));
  LOG(CLINK ← lcdr(CLINK));
  goto Return;
```

254. Arguments to an applicative combiner are evaluated by the caller. This process begins in the *Applicative_Start* chunk by copying the unevaluated arguments into the expression register (in reverse) to validate that they are indeed a proper list. Control then repeatedly enters *Applicative_Pair* for each argument expression or *Combine_Apply* when the entire list has been evaluated.

```
⟨ Evaluate a complex expression 252 ⟩ +≡
Applicative_Start:    /* Save the applicative for later and evaluate its arguments. */
  LOG(CLINK ← note_new(Sym_COMBINE_APPLY, ACC, CLINK, failure));
  LOG(EXPR ← NIL);
Reverse_Arguments:
  if (pair_p(ARGS)) {
    LOG(ACC ← lcar(ARGS));
    LOG(EXPR ← cons(ACC, EXPR, failure));
    LOG(ARGS ← lcdr(ARGS));
    goto Reverse_Arguments;
  }
  else if (¬null_p(ARGS)) siglongjmp(*failure, LERR_IMPROPER);
Applicative_Dispatch:    /* ie. comefrom above & Applicative_Build. */
  if (pair_p(EXPR)) goto Applicative_Pair;
  else goto Combine_Apply;
```

255. The chunk *Applicative_Pair* extracts the next expression and returns to *Begin* to evaluate it, noting in the control link stack that the result will be used to continue building a combination's arguments.

⟨ Evaluate a complex expression 252 ⟩ +≡
```
Applicative_Pair:     /* Evaluate the next argument in the list. */
  LOG(ACC ← lcdr(EXPR));
  LOG(CLINK ← note_new(Sym_COMBINE_BUILD, ACC, CLINK, failure));
  LOG(EXPR ← lcar(EXPR));
  goto Begin;
```

256. Possibly misnamed (*Combine_Build*? *Sym_APPLICATIVE_BUILD*?), *Applicative_Build* is where control will arrive at following evaluation of the expression extracted by *Applicative_Pair* above. The result is appended to the growing arguments, which have now been re-reversed and will end up in the correct order, and control returns to *Applicative_Dispatch* to continue evaluating arguments or combine them into the result.

⟨ Evaluate a complex expression 252 ⟩ +≡
```
Applicative_Build:     /* Continue building a combination after evaluating one expression. */
  LOG(ARGS ← cons(ACC, ARGS, failure));
  LOG(EXPR ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
  goto Applicative_Dispatch;
```

257. After evaluating all of the arguments control will eventually be passed to *Combine_Apply* to restore the saved applicative combiner from the control link stack and then proceed to combination. An operative combiner skips all of the above evaluation and jumps straight in to *Combine_Operate* with the combiner already in the accumulator.

⟨ Evaluate a complex expression 252 ⟩ +≡
```
Combine_Apply:     /* Restore the saved applicative. */
  LOG(ACC ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
Combine_Operate:
  LOG(CLINK ← note_new(Sym_COMBINE_FINISH, EXPR, CLINK, failure));
  combine(failure);
  goto Return;     /* May have pushed further work to CLINK. */
```

258. Since a computer is at heart little more than a glorified transistor `LossLess` would be incomplete without conditional logic, implemented here.

⟨ Evaluate a complex expression 252 ⟩ +≡
```
Conditional:     /* Evaluate the consequent or alternate of a conditional operative. */
  LOG(EXPR ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
  LOG(EXPR ← false_p(ACC) ? lcdr(EXPR) : lcar(EXPR));
  goto Begin;
```

259. ⟨ Evaluate a complex expression 252 ⟩ +≡
```
Validate_Environment:
  if (¬environment_p(ACC)) evaluate_incompatible(__LINE__, failure);
  LOG(EXPR ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
#if 0     /* Test whether the environment can be mutated. */
  if (¬null_p(EXPR))
    if (¬environment_can_p(ACC, true_p(lcar(EXPR)), lcdr(EXPR)))
      evaluate_incompatible(__LINE__, failure);
#endif
  goto Return;
```

260. ⟨ Evaluate a complex expression 252 ⟩ +≡
```
Save_And_Evaluate: LOG(EXPR ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
  LOG(CLINK ← note_new(Sym_ENVIRONMENT_M, ACC, CLINK, failure));
  goto Begin;
Restore_Environment: LOG(ENV ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
  goto Return;
```

261. ⟨ Evaluate a complex expression 252 ⟩ +≡
```
Mutate_Environment: LOG(EXPR ← note_car(CLINK));
  LOG(CLINK ← note_cdr(CLINK));
  LOG(ARGS ← lcar(EXPR));
  LOG(EXPR ← lcdr(EXPR));
  if (true_p(ARGS)) LOG(env_define(ENV, EXPR, ACC, failure));
  else LOG(env_set(ENV, EXPR, ACC, failure));
  goto Return;
```

262. ⟨ Evaluate a complex expression 252 ⟩ +≡
```
Evaluate_Dispatch: LOG(EXPR ← note_cdr(CLINK));    /* For assert — replaced by Evaluate. */
  assert(note_p(EXPR) ∧ note(EXPR) ≡ Sym_COMBINE_FINISH);
  LOG(ENV ← ACC);
  goto Evaluate;
```

263. Primitive combiners are implemented right here in the evaluator (*combine* is only called from a single location in *evalulate*).

```
void combine(sigjmp_buf *failure)
{
  bool flag;
  int count;
  cell nsin, ndex;
  cell value;

  if (primitive_p(ACC)) {
    if (Iprimitive[primitive(ACC)].applicative ∨ Iprimitive[primitive(ACC)].min ≡ −1)
      validate_primitive(failure);
    switch (primitive(ACC)) {⟨ Implement primitive programs 264 ⟩}
  }
  else if (applicative_p(ACC)) LOG(CLINK ← note_new(Sym_APPLICATIVE, ACC, CLINK, failure));
  else if (operative_p(ACC)) LOG(CLINK ← note_new(Sym_OPERATIVE, ACC, CLINK, failure));
}
```

264. ⟨ Implement primitive programs 264 ⟩ ≡
```
default: siglongjmp(*failure, LERR_INTERNAL);
```
See also sections 267, 276, 277, 278, 279, 280, 283, 284, 285, 289, 290, 291, 292, 293, and 294.

This code is used in section 263.

265. Primitive in *Iprimitive*[*primitive*(ACC)] has from . *min* to . *max* arguments, to put into EXPR. If . *min* is -1 only . *max* is checked (for eg. 1 or 3 arguments).

This should be merged with *validate_arguments* and moved prior to evaluation (TODO).

```
void validate_primitive(sigjmp_buf *failure)
{
  int count;
  count ← 0;
  EXPR ← NIL;
  while (pair_p(ARGS)) {
    count ++;
    if (Iprimitive[primitive(ACC)].max ≠ 0 ∧ count > Iprimitive[primitive(ACC)].max)
      evaluate_incompatible(__LINE__, failure);
    EXPR ← cons(lcar(ARGS), EXPR, failure);
    ARGS ← lcdr(ARGS);
  }
  if (Iprimitive[primitive(ACC)].min ≥ 0 ∧ count < Iprimitive[primitive(ACC)].min)
    evaluate_incompatible(__LINE__, failure);
  if (Iprimitive[primitive(ACC)].max ∧ Iprimitive[primitive(ACC)].max ≠
        Iprimitive[primitive(ACC)].min) EXPR ← cons(fix(count), EXPR, failure);
  ARGS ← EXPR;
  EXPR ← NIL;
}
```

266. Entering a closure is similar regardless of whether it's applicative or operative, except that an operative closure needs access to the caller's environment in addition to its own.

The closure is "opened" by extending its environment and restoring its body to the *Environment* and *Expression* registers respectively and re-entering the evaluator (with the appropriate instructions added to the control link stack).

TODO: Refactor into one chunk?

```
⟨Evaluate a complex expression 252⟩ +≡
Operative_Closure:   /* EXPR has unevaluated arguments, ARGS unused. */
  LOG(EXPR ← note_car(CLINK));   /* Closure */
  LOG(CLINK ← note_cdr(CLINK));
  LOG(ACC ← lcar(EXPR));
  LOG(ARGS ← cons(ACC, ARGS, failure));   /* (Formals . Arguments) */
  LOG(EXPR ← lcdr(EXPR));
  LOG(ACC ← ENV);
  LOG(ENV ← lcar(EXPR));   /* Environment */
  LOG(EXPR ← lcdr(EXPR));
  LOG(ENV ← env_extend(ENV, failure));
  LOG(EXPR ← lcar(EXPR));   /* Body */
  LOG(validate_operative(failure));   /* Sets in ENV as required. */
  goto Begin;
Applicative_Closure: LOG(EXPR ← note_car(CLINK));   /* Closure */
  LOG(CLINK ← note_cdr(CLINK));
  LOG(ACC ← lcar(EXPR));   /* Formals */
  LOG(EXPR ← lcdr(EXPR));
  LOG(ENV ← lcar(EXPR));   /* Environment */
  LOG(EXPR ← lcdr(EXPR));
  LOG(ENV ← env_extend(ENV, failure));
  LOG(EXPR ← lcar(EXPR));   /* Body */
  LOG(validate_arguments(failure));   /* Copies from ARGS to ENV. */
  goto Begin;
```

267. Each kind of closure is created similarly — by validating its *formals* expression (ie. the argument names) and copying those, the body of code and the current run-time environment into the appropriately tagged object.

The `do` primitive (notably *not* the symbol representing it but the already-evaluated primitive itself) is prepended to the closure body in a manner which definitely needs improvement. This not only saves an unnecessary environment search at run-time but also means that the meaning of *this* `do` is fixed by the evaluator and cannot be overridden in *any* environment.

```
⟨ Implement primitive programs 264 ⟩ +≡
case PRIMITIVE_LAMBDA:      /* Return an applicative closure. */
case PRIMITIVE_VOV:      /* Return an operative closure. */
  flag ← (primitive(ACC) ≡ PRIMITIVE_LAMBDA);
  if (null_p(ARGS)) evaluate_incompatible(__LINE__, failure);
  LOG(ACC ← lcar(ARGS));      /* Formals */
  LOG(EXPR ← lcdr(ARGS));      /* Body */
  cell lame ← symbol_new_const("do");
  lame ← env_search(Root, lame, true, failure);
  LOG(EXPR ← cons(lame, EXPR, failure));
  LOG(validate_formals(flag, failure));
  LOG(ACC ← closure_new(flag, ARGS, ENV, EXPR, failure));
  break;
```

268. At this stage in evaluation the source code representing the formals is in the accumulator and the argument list is empty — these are swapped so the arguments are in the *Arguments* register — and the *Expression* register is occupied with the closure's body.

```
void validate_formals(bool is_applicative, sigjmp_buf *failure)
{
  static int Svargs ← 2, Svenv ← 1, Svcont ← 0;
  cell arg, state;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  LOG(ARGS ← ACC);      /* TODO: Also check all symbols are unique. */
  LOG(ACC ← NIL);
  if (is_applicative) {⟨ Validate applicative (lambda) formals 269 ⟩}
  else {⟨ Validate operative (vov) formals 270 ⟩}
}
```

269. The *formals* argument to an applicative (`lambda`) expression take the shape of a symbol or a (possibly improper) list of symbols. The symbols, which have bypassed the evaluator, are copied first into the accumulator (backwards) with their syntax wrapping removed, then copied back into the accumulator to restore their original order.

Each symbol should be unique but this is not validated (TODO).

⟨ Validate applicative (`lambda`) formals 269 ⟩ ≡

```
while (pair_p(evaluate_desyntax(ARGS))) {
  arg ← lcar(evaluate_desyntax(ARGS));
  LOG(ARGS ← lcdr(evaluate_desyntax(ARGS)));
  assert(syntax_p(arg));
  arg ← syntax_datum(arg);
  if (¬symbol_p(arg)) evaluate_incompatible(__LINE__, failure);
  LOG(ACC ← cons(arg, ACC, failure));
}
if (symbol_p(evaluate_desyntax(ARGS))) LOG(ARGS ← evaluate_desyntax(ARGS));
else if (¬null_p(evaluate_desyntax(ARGS))) evaluate_incompatible(__LINE__, failure);
while (¬null_p(ACC)) {
  LOG(ARGS ← cons(lcar(ACC), ARGS, failure));
  LOG(ACC ← lcdr(ACC));
}
```

This code is used in section 268.

270. The formals (or *informals*) argument to an operative (`vov`) expression is a list of lists of two symbols. The first such symbol is a variable name to bind to; the second symbol is what to bind to it, one of the caller's environment, (unevaluated) arguments or (unimplemented) continuation delimiter.

Each piece of state can be referenced once, and at least one piece of state must be (TODO: not demanding this may be profitable for global operators without arguments), and this is validated although each binding (variable) name must be unique and this is not (TODO).

```
#define Sym_VOV_ARGS (symbol_new_const("vov/args"))
#define Sym_VOV_ARGUMENTS (symbol_new_const("vov/arguments"))
#define Sym_VOV_CONT (symbol_new_const("vov/cont"))
#define Sym_VOV_CONTINUATION (symbol_new_const("vov/continuation"))
#define Sym_VOV_ENV (symbol_new_const("vov/env"))
#define Sym_VOV_ENVIRONMENT (symbol_new_const("vov/environment"))
#define save_vov_informal(O, S) do
        {
          if (¬null_p(SO(S))) evaluate_incompatible(__LINE__, failure);
          else LOG(SS((S), (O)));
        }
        while (0)

⟨ Validate operative (vov) formals 270 ⟩ ≡
  stack_reserve(3, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 3);
  ARGS ← evaluate_desyntax(ARGS);
  while (pair_p(ARGS)) {
    arg ← lcar(ARGS);
    arg ← evaluate_desyntax(arg);
    if (¬pair_p(arg)) evaluate_incompatible(__LINE__, failure);
    state ← lcdr(arg);
    arg ← lcar(arg);
    arg ← evaluate_desyntax(arg);
    if (¬symbol_p(arg) ∨ ¬pair_p(state) ∨ ¬null_p(lcdr(state)))
      evaluate_incompatible(__LINE__, failure);
    state ← lcar(state);
    state ← evaluate_desyntax(state);
    if (state ≡ Sym_VOV_ARGS ∨ state ≡ Sym_VOV_ARGUMENTS)
      save_vov_informal(arg, Svargs);
    else if (state ≡ Sym_VOV_ENV ∨ state ≡ Sym_VOV_ENVIRONMENT)
      save_vov_informal(arg, Svenv);
    else if (state ≡ Sym_VOV_CONT ∨ state ≡ Sym_VOV_CONTINUATION)
      save_vov_informal(arg, Svcont);
    else evaluate_incompatible(__LINE__, failure);
    LOG(ARGS ← lcdr(ARGS));
  }
  if (¬null_p(ARGS) ∨ (null_p(SO(Svargs)) ∧ null_p(SO(Svenv)) ∧ null_p(SO(Svcont))))
    evaluate_incompatible(__LINE__, failure);
  ARGS ← cons(SO(Svcont), ARGS, failure);
  ARGS ← cons(SO(Svenv), ARGS, failure);
  ARGS ← cons(SO(Svargs), ARGS, failure);
  stack_clear(3);
```

This code is used in section 268.

271. A closure object is simply the three pieces of virtual machine state saved in an opaque list.

272. From the *Applicative_Closure* evaluator chunk, the arguments have been evaluated and the number of them is validated while each is bound — in the extended environment — to the symbol named in the formals which have been saved in the accumulator.

Although the *env_define* call here will have the effect of forcing the formals symbols to be unique, this is the wrong place to rely on it.

```
void validate_arguments(sigjmp_buf *failure)
{
  static int Sname ← 1, Sarg ← 0;
  cell arg, name;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  stack_reserve(2, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 2);
  LOG(SS(Sname, evaluate_desyntax(ACC)));
  LOG(SS(Sarg, ARGS));
  while (pair_p(SO(Sname))) {
    LOG(name ← lcar(SO(Sname)));
    LOG(SS(Sname, lcdr(SO(Sname))));
    if (null_p(SO(Sarg))) evaluate_incompatible(__LINE__, failure);
    LOG(arg ← lcar(SO(Sarg)));
    LOG(SS(Sarg, lcdr(SO(Sarg))));
    LOG(env_define(ENV, name, arg, failure));
  }
  if (¬null_p(SO(Sname))) {
    LOG(assert(symbol_p(SO(Sname))));
    LOG(env_define(ENV, SO(Sname), SO(Sarg), failure));
  }
  else if (¬null_p(SO(Sarg))) evaluate_incompatible(__LINE__, failure);
  stack_clear(2);
}
```

273. An operative closure has in place of its formals a list of three symbols (or `NIL`). Their location in the list determines what will be bound to that symbol so the order must match that in *validate_formals* above.

The run-time environment of the caller has been saved in the accumulator prior to the closure's environment being restored and extended. Note that the arguments *still* have not been seen by the evaluator and so retain the syntax wrapping.

```
void validate_operative(sigjmp_buf *failure)
{
  static int Sinformal ← 0;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  assert(pair_p(ARGS));
  stack_push(lcar(ARGS), failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);
  ARGS ← lcdr(ARGS);
  assert(pair_p(SO(Sinformal)));
  if (symbol_p(lcar(SO(Sinformal)))) LOG(env_define(ENV, lcar(SO(Sinformal)), ARGS, failure));
  SS(Sinformal, lcdr(SO(Sinformal)));
  assert(pair_p(SO(Sinformal)));
  if (symbol_p(lcar(SO(Sinformal)))) LOG(env_define(ENV, lcar(SO(Sinformal)), ACC, failure));
  SS(Sinformal, lcdr(SO(Sinformal)));
  assert(pair_p(SO(Sinformal)));
  if (symbol_p(lcar(SO(Sinformal)))) siglongjmp (cleanup, LERR_UNIMPLEMENTED);
      /* Continuation */
  assert(null_p(lcdr(SO(Sinformal))));
  stack_clear(1);
}
```

274. Primitives. Core primitives are primarily operatives and some applicatives to manipulate lists, the environment and closures. Primitives which are applicative have their argument lists checked for length. Operatives enforce a maximum number of arguments (which may be zero) if the minimum indicated is -1 or that they will process the argument list entirely (minimum is zero) and the maximum value is not used.

⟨ Symbolic primitive identifiers 274 ⟩ ≡

```
PRIMITIVE_BREAK,
PRIMITIVE_CAR,
PRIMITIVE_CDR,
PRIMITIVE_CONS,
PRIMITIVE_CURRENT_ENVIRONMENT,
PRIMITIVE_DEFINE_M,
PRIMITIVE_DO,
PRIMITIVE_DUMP,
PRIMITIVE_EVAL,
PRIMITIVE_IF,
PRIMITIVE_LAMBDA,
PRIMITIVE_NULL_P,
PRIMITIVE_PAIR_P,
PRIMITIVE_ROOT_ENVIRONMENT,
PRIMITIVE_SET_M,
PRIMITIVE_VOV ,
```

See also section 287.

This code is used in section 186.

275. ⟨ Primitive definitions 275 ⟩ ≡

```
[PRIMITIVE_BREAK] ← {"break", NIL, false, −1, 0, },
    [PRIMITIVE_CURRENT_ENVIRONMENT] ← {"current-environment", NIL, false, −1, 0, },
    [PRIMITIVE_DO] ← {"do", NIL, false, 0, 0, },
    [PRIMITIVE_DEFINE_M] ← {"define!", NIL, false, 0, 0, },
    [PRIMITIVE_IF] ← {"if", NIL, false, −1, 3, },
    [PRIMITIVE_LAMBDA] ← {"lambda", NIL, false, 0, 0, },
    [PRIMITIVE_ROOT_ENVIRONMENT] ← {"root-environment", NIL, false, −1, 0, },
    [PRIMITIVE_SET_M] ← {"set!", NIL, false, 0, 0, },
    [PRIMITIVE_VOV] ← {"vov", NIL, false, 0, 0, },
[PRIMITIVE_CAR] ← {"car", NIL, true, 1, 1, },
    [PRIMITIVE_CDR] ← {"cdr", NIL, true, 1, 1, },
    [PRIMITIVE_CONS] ← {"cons", NIL, true, 2, 2, },
    [PRIMITIVE_DUMP] ← {"dump", NIL, true, 1, 1, },
    [PRIMITIVE_EVAL] ← {"eval", NIL, true, 1, 2, },
    [PRIMITIVE_NULL_P] ← {"null?", NIL, true, 1, 1, },
    [PRIMITIVE_PAIR_P] ← {"pair?", NIL, true, 1, 1, } ,
```

See also section 288.

This code is used in section 187.

276. The pair constructor `cons` along with its accessors `car`, `cdr`, etc. `ARGS` has been scanned sufficient to be certain it is a proper list (or `NIL`).

⟨ Implement primitive programs 264 ⟩ +≡
```
case PRIMITIVE_CONS: LOG(Tmp_DEX ← lcar(ARGS));
  LOG(ARGS ← lcdr(ARGS));
  LOG(Tmp_SIN ← lcar(ARGS));
  LOG(ARGS ← lcdr(ARGS));
  assert(null_p(ARGS));
  LOG(ACC ← cons(Tmp_SIN, Tmp_DEX, failure));    /* Tmp_* reset by cons. */
  break;
case PRIMITIVE_CAR: LOG(EXPR ← lcar(ARGS));
  LOG(ARGS ← lcdr(ARGS));
  assert(null_p(ARGS));
  if (¬pair_p(EXPR)) evaluate_incompatible(__LINE__, failure);
  LOG(ACC ← lcar(EXPR));
  break;
case PRIMITIVE_CDR: LOG(EXPR ← lcar(ARGS));
  LOG(ARGS ← lcdr(ARGS));
  assert(null_p(ARGS));
  if (¬pair_p(EXPR)) evaluate_incompatible(__LINE__, failure);
  LOG(ACC ← lcdr(EXPR));
  break;
```

277. Perform a list of evaluations sequentially, terminating with the result of the last in the accumulator. This validates that the body of the expression is a proper list building up instructions to evaluate in a new list in the accumulator.

To avoid reversing and re-reversing the list a pointer to the tail is kept in the *Expression* register which is otherwise empty. This algorithm is rather odd in that the new control link node is created with the accumulator (the list head) in its tail position which is then replaced. This is so that the evaluator does not require any temporary storage other than the five evaluator registers.

⟨ Implement primitive programs 264 ⟩ +≡
```
case PRIMITIVE_DO:    /* (Operative) */
  LOG(EXPR ← note_new(Sym_EVALUATE, VOID, NIL, failure));
  LOG(ACC ← EXPR);
  while (¬null_p(ARGS)) {
    if (¬pair_p(ARGS)) evaluate_incompatible(__LINE__, failure);
    LOG(value ← note_new(Sym_EVALUATE, evaluate_desyntax(lcar(ARGS)), NIL, failure));
    LOG(note_set_cdr_m(ACC, value));
    LOG(ACC ← value);
    LOG(ARGS ← lcdr(ARGS));
    ARGS ← evaluate_desyntax(ARGS);
  }
  LOG(note_set_cdr_m(ACC, CLINK));
  LOG(CLINK ← EXPR);
  break;
```

278. ⟨ Implement primitive programs 264 ⟩ +≡
```
case PRIMITIVE_CURRENT_ENVIRONMENT: assert(null_p(ARGS));
  LOG(ACC ← ENV);
  break;
case PRIMITIVE_ROOT_ENVIRONMENT: assert(null_p(ARGS));
  LOG(ACC ← Root);
  break;
```

279. Set the stage to evaluate an expression and branch to the evaluation of one of two other expressions depending on its outcome's truth.

The test is saved in the *Expression* register with the consequent and alternate (in case the test evaluates to false) expressions saved in a pair in the control link stack. An alternate expression is optional and in such a case a false test result will evaluate to `VOID`.

⟨ Implement primitive programs 264 ⟩ +≡
case `PRIMITIVE_IF`: /* (Operative) */
  `LOG`(*count* ← *fix_value*(*lcar*(`ARGS`)));
  `LOG`(`ARGS` ← *lcdr*(`ARGS`));
  if (*count* ≡ 3) *validated_argument*(`ACC`, `ARGS`, *false*, *defined_p*, *failure*);
  else if (*count* ≡ 1) `LOG`(`ACC` ← `VOID`); /* No alternate */
  else *evaluate_incompatible*(`__LINE__`, *failure*);
  validated_argument(`EXPR`, `ARGS`, *false*, *defined_p*, *failure*);
  `LOG`(`EXPR` ← *cons*(`EXPR`, `ACC`, *failure*));
  validated_argument(`ACC`, `ARGS`, *false*, *defined_p*, *failure*);
  `LOG`(`CLINK` ← *note_new*(*Sym_CONDITIONAL*, `EXPR`, `CLINK`, *failure*));
  `LOG`(`CLINK` ← *note_new*(*Sym_EVALUATE*, `ACC`, `CLINK`, *failure*));
  break;

280. Debugging.

⟨ Implement primitive programs 264 ⟩ +≡
case `PRIMITIVE_DUMP`: *lprint*(`"DUMP␣"`);
  `ACC` ← *lcar*(`ARGS`);
  serial(`ACC`, `SERIAL_DETAIL`, 42, `NIL`, Λ, *failure*);
  lprint(`"\n"`);
case `PRIMITIVE_BREAK`: *breakpoint*( );
  break;

281. ⟨ Function declarations 21 ⟩ +≡
  void *breakpoint*(**void**);

282. **void** *breakpoint*(**void**)
  {
    printf(`"Why␣did␣we␣ever␣allow␣GNU?\n"`);
  }

283. (define! ¡env¿ ¡sym¿ ¡expr¿)
or (define! ¡env¿ (¡sym¿ . ¡formals¿) . ¡body¿) == (define! ¡env¿ ¡sym¿ (lambda ¡formals¿ . ¡body¿))
== (define! ¡env¿ ¡pair?¿ . ¡rest¿) == (define! ¡env¿ ,(car ¡pair¿) (lambda ,(cdr ¡pair) . ¡rest¿))

⟨ Implement primitive programs 264 ⟩ +≡

```
case PRIMITIVE_DEFINE_M: case PRIMITIVE_SET_M: flag ← (primitive(ACC) ≡ PRIMITIVE_DEFINE_M);
  if (¬pair_p(ARGS)) evaluate_incompatible(__LINE__, failure);
  LOG(ACC ← lcar(ARGS));    /* Environment to mutate. */
  ARGS ← lcdr(ARGS);
  if (¬pair_p(ARGS)) evaluate_incompatible(__LINE__, failure);
  EXPR ← lcar(ARGS);    /* Binding label. */
  EXPR ← evaluate_desyntax(EXPR);
  if (pair_p(EXPR)) {    /* Applicative closure: (label . formals) */
    LOG(ARGS ← lcdr(ARGS));    /* Closure body. */
    LOG(value ← lcdr(EXPR));    /* Applicative formals. */
    LOG(EXPR ← lcar(EXPR));    /* Real binding label. */
    EXPR ← evaluate_desyntax(EXPR);
    LOG(ARGS ← cons(value, ARGS, failure));
    LOG(ARGS ← cons(Iprimitive[PRIMITIVE_LAMBDA].box, ARGS, failure));
  }
  else if (symbol_p(EXPR)) {
    LOG(ARGS ← lcdr(ARGS));
    if (¬pair_p(ARGS)) evaluate_incompatible(__LINE__, failure);
    LOG(value ← lcar(ARGS));    /* Value (after evaluation). */
    LOG(ARGS ← lcdr(ARGS));
    if (¬null_p(ARGS)) evaluate_incompatible(__LINE__, failure);
    LOG(ARGS ← value);
  }
  else evaluate_incompatible(__LINE__, failure);
  LOG(EXPR ← cons(predicate(flag), EXPR, failure));
  LOG(CLINK ← note_new(Sym_DEFINITION, EXPR, CLINK, failure));
  LOG(CLINK ← note_new(Sym_SAVE_AND_EVALUATE, ARGS, CLINK, failure));
  LOG(CLINK ← note_new(Sym_ENVIRONMENT_P, EXPR, CLINK, failure));
  LOG(CLINK ← note_new(Sym_EVALUATE, ACC, CLINK, failure));
  break;
```

284.

```
#define primitive_predicate(O) do {
      ACC ← lcar(ARGS);
      ARGS ← lcdr(ARGS);
      assert(null_p(ARGS));
      ACC ← predicate(O(ACC));
    } while (0); break
```

⟨ Implement primitive programs 264 ⟩ +≡

```
case PRIMITIVE_NULL_P: primitive_predicate(null_p);
case PRIMITIVE_PAIR_P: primitive_predicate(pair_p);
```

285. ⟨ Implement primitive programs 264 ⟩ +≡

```
case PRIMITIVE_EVAL: LOG(count ← fix_value(lcar(ARGS)));
  LOG(ARGS ← lcdr(ARGS));
  if (count ≡ 2) validated_argument(ACC, ARGS, false, environment_p, failure);
  else if (count ≡ 1) LOG(ACC ← ENV);
  else evaluate_incompatible(__LINE__, failure);
  LOG(EXPR ← lcar(ARGS));    /* Expression to evaluate. */
  LOG(CLINK ← note_new(Sym_EVALUATE_DISPATCH, EXPR, CLINK, failure));
  break;
```

286. **#define** *validated_argument*(VAR, ARGS, NULLABLE, PREDICATE, FAILURE) **do**

```
{
  (VAR) ← lcar(ARGS);
  (ARGS) ← lcdr(ARGS);
  if ((¬(NULLABLE) ∧ null_p(VAR)) ∨ ¬PREDICATE(VAR))
    evaluate_incompatible(__LINE__, (FAILURE));
}
while (0)
```

287. Object primitives.

⟨ Symbolic primitive identifiers 274 ⟩ +≡
PRIMITIVE_NEW_TREE_PLAIN_NODE, PRIMITIVE_NEW_TREE_BIWARD_NODE,
PRIMITIVE_NEW_TREE_SINWARD_NODE, PRIMITIVE_NEW_TREE_DEXWARD_NODE,
PRIMITIVE_TREE_SIN_HAS_THREAD_P, PRIMITIVE_TREE_DEX_HAS_THREAD_P,
PRIMITIVE_TREE_DEX_IS_LIVE_P, PRIMITIVE_TREE_SIN_IS_LIVE_P, PRIMITIVE_TREE_P,
PRIMITIVE_TREE_PLAIN_P, PRIMITIVE_TREE_RETHREAD_M, PRIMITIVE_TREE_SIN_THREADABLE_P,
PRIMITIVE_TREE_DEX_THREADABLE_P, PRIMITIVE_TREE_DATUM,

PRIMITIVE_NEW_ROPE_PLAIN_NODE, PRIMITIVE_ROPE_P, PRIMITIVE_ROPE_PLAIN_P,
PRIMITIVE_ROPE_SIN_THREADABLE_P, PRIMITIVE_ROPE_DEX_THREADABLE_P,
PRIMITIVE_ROPE_SEGMENT ,

288. ⟨ Primitive definitions 275 ⟩ +≡
[PRIMITIVE_NEW_TREE_PLAIN_NODE] ← {"new-tree%plain-node", NIL, *true*, 3, 3, },
[PRIMITIVE_NEW_TREE_BIWARD_NODE] ← {"new-tree%bi-threadable-node", NIL, *true*, 3, 3, },
[PRIMITIVE_NEW_TREE_DEXWARD_NODE] ← {"new-tree%sin-threadable-node", NIL, *true*, 3, 3, },
[PRIMITIVE_NEW_TREE_SINWARD_NODE] ← {"new-tree%dex-threadable-node", NIL, *true*, 3, 3, },
[PRIMITIVE_TREE_SIN_HAS_THREAD_P] ← {"tree/sin-has-thread?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_DEX_HAS_THREAD_P] ← {"tree/dex-has-thread?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_SIN_IS_LIVE_P] ← {"tree/live-sin", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_DEX_IS_LIVE_P] ← {"tree/live-dex", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_P] ← {"tree?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_PLAIN_P] ← {"tree%plain?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_RETHREAD_M] ← {"tree/rethread!", NIL, *true*, 3, 3, },
[PRIMITIVE_TREE_SIN_THREADABLE_P] ← {"tree%sin-threadable?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_DEX_THREADABLE_P] ← {"tree%dex-threadable?", NIL, *true*, 1, 1, },
[PRIMITIVE_TREE_DATUM] ← {"tree/datum", NIL, *true*, 1, 1, },

[PRIMITIVE_ROPE_P] ← {"rope?", NIL, *true*, 1, 1, },
[PRIMITIVE_ROPE_PLAIN_P] ← {"rope%plain?", NIL, *true*, 1, 1, },
[PRIMITIVE_ROPE_SIN_THREADABLE_P] ← {"rope%sin-threadable?", NIL, *true*, 1, 1, },
[PRIMITIVE_ROPE_DEX_THREADABLE_P] ← {"rope%dex-threadable?", NIL, *true*, 1, 1, },
[PRIMITIVE_NEW_ROPE_PLAIN_NODE] ← {"new-rope%plain-node", NIL, *true*, 1, 3, },
[PRIMITIVE_ROPE_SEGMENT] ← {"rope/segment", NIL, *true*, 1, 1, } ,

289. ⟨ Implement primitive programs 264 ⟩ +≡
case PRIMITIVE_ROPE_P: ACC ← *predicate*(*rope_p*(*lcar*(ARGS)));
break;
case PRIMITIVE_ROPE_PLAIN_P: ACC ← *predicate*(*plain_rope_p*(*lcar*(ARGS)));
break;
case PRIMITIVE_ROPE_SIN_THREADABLE_P: ACC ← *predicate*(*rope_sin_threadable_p*(*lcar*(ARGS)));
break;
case PRIMITIVE_ROPE_DEX_THREADABLE_P: ACC ← *predicate*(*rope_dex_threadable_p*(*lcar*(ARGS)));
break;

290. ⟨ Implement primitive programs 264 ⟩ +≡

case `PRIMITIVE_NEW_ROPE_PLAIN_NODE`: *count* ← *fix_value*(*lcar*(`ARGS`));
  `ARGS` ← *lcdr*(`ARGS`);
  if (*count* ≡ 3) {
    validated_argument(*ndex*, `ARGS`, *true*, *plain_rope_p*, *failure*);
    validated_argument(*nsin*, `ARGS`, *true*, *plain_rope_p*, *failure*);
  }
  else if (*count* ≠ 1) *evaluate_incompatible*(`__LINE__`, *failure*);
  else *nsin* ← *ndex* ← `NIL`;
  validated_argument(`ACC`, `ARGS`, *true*, *rope_p*, *failure*);
  if (*null_p*(`ACC`)) `ACC` ← *rope_node_new_length*(*false*, *false*, 0, *nsin*, *ndex*, *failure*);
  else `ACC` ← *rope_node_new_clone*(*false*, *false*, `ACC`, *nsin*, *ndex*, *failure*);
  break;

291. ⟨ Implement primitive programs 264 ⟩ +≡

case `PRIMITIVE_TREE_P`: `ACC` ← *predicate*(*tree_p*(*lcar*(`ARGS`)));
  break;
case `PRIMITIVE_TREE_PLAIN_P`: `ACC` ← *predicate*(*plain_tree_p*(*lcar*(`ARGS`)));
  break;
case `PRIMITIVE_TREE_SIN_THREADABLE_P`: `ACC` ← *predicate*(*tree_sin_threadable_p*(*lcar*(`ARGS`)));
  break;
case `PRIMITIVE_TREE_DEX_THREADABLE_P`: `ACC` ← *predicate*(*tree_dex_threadable_p*(*lcar*(`ARGS`)));
  break;

292. ⟨ Implement primitive programs 264 ⟩ +≡

case `PRIMITIVE_TREE_SIN_HAS_THREAD_P`:
  if (¬*tree_sin_threadable_p*(*lcar*(`ARGS`))) `ACC` ← `LFALSE`;
  else `ACC` ← *tree_sin_has_thread_p*(*lcar*(`ARGS`));
  break;
case `PRIMITIVE_TREE_DEX_HAS_THREAD_P`:
  if (¬*tree_dex_threadable_p*(*lcar*(`ARGS`))) `ACC` ← `LFALSE`;
  else `ACC` ← *tree_dex_has_thread_p*(*lcar*(`ARGS`));
  break;
case `PRIMITIVE_TREE_SIN_IS_LIVE_P`: `ACC` ← *predicate*(*dryad_sin_p*(*lcar*(`ARGS`)));
  break;
case `PRIMITIVE_TREE_DEX_IS_LIVE_P`: `ACC` ← *predicate*(*dryad_dex_p*(*lcar*(`ARGS`)));
  break;

293. ⟨ Implement primitive programs 264 ⟩ +≡

case `PRIMITIVE_NEW_TREE_PLAIN_NODE`: *validated_argument*(*ndex*, `ARGS`, *true*, *plain_tree_p*, *failure*);
  validated_argument(*nsin*, `ARGS`, *true*, *plain_tree_p*, *failure*);
#**if** 0
  `ACC` ← *tree_node_new*(*lcar*(`ARGS`), *nsin*, *ndex*, *failure*);
#**endif**
  break;
case `PRIMITIVE_NEW_TREE_BIWARD_NODE`:
  validated_argument(*ndex*, `ARGS`, *true*, *tree_threadable_p*, *failure*);
  validated_argument(*nsin*, `ARGS`, *true*, *tree_threadable_p*, *failure*);
#**if** 0
  `ACC` ← *tree_threadable_node_new*(*lcar*(`ARGS`), *nsin*, *ndex*, *failure*);
#**endif**
  break;
case `PRIMITIVE_NEW_TREE_SINWARD_NODE`:
  validated_argument(*ndex*, `ARGS`, *true*, *tree_sin_threadable_p*, *failure*);
  validated_argument(*nsin*, `ARGS`, *true*, *tree_sin_threadable_p*, *failure*);
  if (*tree_dex_threadable_p*(*nsin*) ∨ *tree_dex_threadable_p*(*ndex*))
    evaluate_incompatible(`__LINE__`, *failure*);
#**if** 0
  `ACC` ← *tree_sin_threadable_node_new*(*lcar*(`ARGS`), *nsin*, *ndex*, *failure*);
#**endif**
  break;
case `PRIMITIVE_NEW_TREE_DEXWARD_NODE`:
  validated_argument(*ndex*, `ARGS`, *true*, *tree_dex_threadable_p*, *failure*);
  validated_argument(*nsin*, `ARGS`, *true*, *tree_dex_threadable_p*, *failure*);
  if (*tree_sin_threadable_p*(*nsin*) ∨ *tree_sin_threadable_p*(*ndex*))
    evaluate_incompatible(`__LINE__`, *failure*);
#**if** 0
  `ACC` ← *tree_dex_threadable_node_new*(*lcar*(`ARGS`), *nsin*, *ndex*, *failure*);
#**endif**
  break;

case `PRIMITIVE_TREE_RETHREAD_M`: *validated_argument*(*ndex*, `ARGS`, *false*, *boolean_p*, *failure*);
  validated_argument(*nsin*, `ARGS`, *false*, *boolean_p*, *failure*);
  if (¬*tree_p*(*lcar*(`ARGS`))) *evaluate_incompatible*(`__LINE__`, *failure*);
#**if** 0
  `ACC` ← *treeish_rethread_m*(*lcar*(`ARGS`), *true_p*(*nsin*), *true_p*(*ndex*), *failure*);
#**endif**
  break;

294. ⟨ Implement primitive programs 264 ⟩ +≡

case `PRIMITIVE_TREE_DATUM`: *validated_argument*(*value*, `ARGS`, *false*, *tree_p*, *failure*);
  `ACC` ← *dryad_datum*(*value*);
  break;

295. Serialisation.

```
#define serial_printable_p(O) ((O) ≥ '␣' ∧ (O) < 0x7f)
#define SERIAL_SILENT 0
#define SERIAL_HUMAN 1
#define SERIAL_ROUND 2
#define SERIAL_DETAIL 3
```

⟨ Function declarations 21 ⟩ +≡

```
void gc_serial(cell, cell);
void serial(cell, int, int, cell, cell *, sigjmp_buf *);
void serial_append_imp(cell, char *, int, sigjmp_buf *);
int serial_cycle(cell, cell);
char *serial_deduplicate(cell, int, cell, cell, sigjmp_buf *);
void serial_escape(char *, int, cell, sigjmp_buf *);
void serial_imp(cell, int, int, bool, cell, cell, sigjmp_buf *);
void serial_rope(cell, int, int, cell, cell, sigjmp_buf *);
```

296.

what: 3 debug: all detail 2 round-trip: strings/symbols formatted, impossible-to-print objects abort 1 descriptive: (unimplemented) 0 silent (to collect a list of recursive points)

how: strings escaped vs. raw maximum depth numeric base

buffer is a segment where the serialised result will be put. Can only be `NIL` if *detail* is 0 or *dprint* is enabled.

```
void serial(cell o, int detail, int maxdepth, cell buffer, cell *cycles, sigjmp_buf *failure)
{
  static int Sobject ← 0;
  size_t count;
  Oheap *h, *heap;
  bool compacting;
  cell ignored ← Null_Array;     /* In case of special_p(o). */
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;

  assert(defined_p(o));
  assert(maxdepth ≥ 0);
  if (detail) assert(null_p(buffer));
  else assert((LDEBUG_P ∧ null_p(buffer)) ∨ segment_p(buffer));
  if (cycles ≡ Λ) cycles ← &ignored;
  stack_protect(1, o, failure);
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) unwind(failure, reason, false, 1);

  if (¬special_p(o)) {
    ⟨ Determine which heap an object is in 297 ⟩
    ⟨ Look for cyclic data structures 298 ⟩
  }
  if (detail) serial_imp(SO(Sobject), detail, maxdepth, true, buffer, *cycles, failure);
  stack_clear(1);
}
```

297. ⟨ Determine which heap an object is in 297 ⟩ ≡

```
heap ← Theap;
h ← Sheap;
while (h ≠ Λ)
  if (h ≡ ATOM_TO_HEAP(o)) {
    heap ← Sheap;
    break;
  }
  else h ← h→next;
```

This code is used in section 296.

298. ⟨ Look for cyclic data structures 298 ⟩ ≡

```
*cycles ← array_new_imp(8, LFALSE, FORM_ARRAY, &cleanup);
pointer_set_datum_m(*cycles, fix(0));
compacting ← (ATOM_TO_HEAP(o)→pair ≠ Λ);
count ← 0;    /* Ignored. */
*cycles ← gc_mark(heap, *cycles, compacting, Λ, &count);
SS(Sobject, gc_mark(heap, SO(Sobject), compacting, cycles, &count));
if (compacting) gc_compacting(heap, false);
else gc_sweeping(heap, false);
if (null_p(*cycles)) siglongjmp (cleanup, LERR_OOM);
count ← 2 * fix_value(pointer_datum(*cycles));
*cycles ← array_grow_m(*cycles, count − array_length(*cycles), LFALSE, failure);
count /= 2;
pointer_set_datum_m(*cycles, fix(count));
#if 0
serial_imp(SO(Sobject), SERIAL_SILENT, maxdepth, false, Λ, *cycles, failure);
pointer_set_datum_m(*cycles, fix(0));
for (i ← 0; i < count; i++) {
  j ← fix_value(pointer_datum(*cycles));
  if (true_p(array_ref(*cycles, i + count))) {
    if (i ≠ j) array_set_m(*cycles, j++, array_ref(*cycles, i));
    pointer_set_datum_m(*cycles, fix(j));
  }
}
for ( ; i < j * 2; i++) array_set_m(*cycles, j, LFALSE);
#endif
```

This code is used in section 296.

299. When it's collecting the object being serialised, the garbage collector is instructed to call this function whenever it encounters a reference it has already seen in that round of collection.

```
void gc_serial(cell cycles, cell found)
{
  int i, len;
  sigjmp_buf cleanup;
  Verror reason ← LERR_NONE;
  if (null_p(cycles)) return;
  if (failure_p(reason ← sigsetjmp(cleanup, 1))) return;    /* FIXME: The error is lost. */
  len ← fix_value(pointer_datum(cycles));
  for (i ← 0; i < len; i++)
    if (array_ref(cycles, i) ≡ found) return;
  if (len ≥ array_length(cycles)) array_grow_m(cycles, array_length(cycles), LFALSE, &cleanup);
  array_set_m(cycles, len, found);
  pointer_set_datum_m(cycles, fix(len + 1));
}
```

300. `#define serial_append(B, C, L, F) do`

```
    {
      if (LDEBUG_P ∧ null_p(B))
        for (int _i ← 0; _i < (L); _i++) lput((C)[_i]);
      else serial_append_imp((B), (C), (L), (F));
    }
    while (0)
void serial_append_imp(cell buffer, char *content, int length, sigjmp_buf *failure)
{
  int i, off;
  assert(segment_p(buffer));
  off ← fix_value(pointer_datum(buffer));
  i ← segment_length(buffer) − off − 1;
  if (i > length) i ← length;
  for ( ; i ≥ 0; i−−) segment_address(buffer)[off + i] ← content[i];
  segment_address(buffer)[off + i] ← '\0';
  pointer_set_datum_m(buffer, fix(off + i));
  if (i ≠ length) siglongjmp (*failure, LERR_OVERFLOW);
}
```

301. Uses C stack (calls itself) but doesn't have to.
Buffer may be NIL if debugging and LDEBUG_P so *lprint*/*lput* will work.

```
void serial_imp(cell o, int detail, int maxdepth, bool prefix, cell buffer, cell cycles, sigjmp_buf
      *failure)
{
  cell p;
  int i, length ← 0;
  char *append ← Λ, buf[FIX_BASE10 + 2];
  assert(maxdepth ≥ 0);
  assert((LDEBUG_P ∧ null_p(buffer)) ∨ segment_p(buffer));
  assert(array_p(cycles));
  if (special_p(o)) {⟨Serialise a unique object 302⟩}
  else if (symbol_p(o)) {⟨Serialise a symbol 303⟩}
  else append ← serial_deduplicate(o, detail, buffer, cycles, failure);
  if (append ≡ Λ) {
    ⟨Serialise an object 307⟩    /* An unterminated if/else if chain. */
    else {    /* Will go away when ⟨Serialise an object 307⟩ is complete. */
      lprint("%2x?\n", form(o));
      assert(¬"unknown␣type");
    }
  }
  if (append ≡ Λ) {
    lprint("%p:␣%x\n", o, special_p(o) ? −1 : form(o));
    siglongjmp (*failure, LERR_UNPRINTABLE);
  }
  if (¬length) {
    length ← append[0];
    append++;
  }
  if (detail) serial_append(buffer, append, length, failure);
}
```

302. ⟨ Serialise a unique object 302 ⟩ ≡

```
  if (null_p(o)) append ← "\002()";
  else if (false_p(o)) append ← "\002#f";
  else if (true_p(o)) append ← "\002#t";
  else if (void_p(o)) {   /* Licence to disappear. */
    if (detail ≡ SERIAL_DETAIL) append ← "\007#<void>";
#if 0
    else if (detail ≠ SERIAL_ROUND) append ← "\000";
#endif
  }
  else if (eof_p(o)) {   /* The terminator will not be back. */
    if (detail ≡ SERIAL_DETAIL) append ← "\021#<schwarzenegger>";
  }
  else if (undefined_p(o)) {   /* Ecce res qui est faba. */
    if (detail ≡ SERIAL_DETAIL) append ← "\006#<zen>";
  }
  else {
    assert(fix_p(o));
    i ← fix_value(o);
    if (¬i) append ← "\0010";
    else {
      if (i < 0) i ← −i;
      append ← buf + FIX_BASE10 + 2;
      *−− append ← '\0';   /* Terminator. */
      *−− append ← 0;   /* Length. */
      while (i) {
        *(append − 1) ← *(append) + 1;
        *append −− ← (i % 10) + '0';
        i /= 10;
      }
    }
    if (fix_value(o) < 0) {
      *(append − 1) ← *(append) + 1;
      *append −− ← '-';
    }
  }
```

This code is used in section 301.

303. ⟨ Serialise a symbol 303 ⟩ ≡

```
  append ← symbol_buffer(o);
  length ← symbol_length(o);
  if (detail ∧ maxdepth)
    for (i ← 0; i < length; i++)
      if (¬serial_printable_p(append[i])) {
        if (detail) {
          serial_append(buffer, "#|", 2, failure);
          serial_escape(append, length, buffer, failure);
          serial_append(buffer, "|", 1, failure);
        }
        append ← "\0";
        length ← 0;
        break;
      }
```

This code is used in section 301.

304. **void** *serial_escape*(**char** ∗*append*, **int** *length*, **cell** *buffer*, **sigjmp_buf** ∗*failure*)

```
{
  int i, j;
  char ascii[4] ← {'#', 'x', '\0', '\0'};
  for (i ← 0; i < length; i++) {
    if (append[i] ≡ '#') serial_append(buffer, "##", 2, failure);
    else if (append[i] ≡ '|') serial_append(buffer, "#|", 2, failure);
    else if (append[i] ≡ '\n' ∨ append[i] ≡ '\t' ∨ serial_printable_p(append[i]))
      serial_append(buffer, append + i, 1, failure);
    else {
      j ← (append[i] & 0xf0) ≫ 4;
      ascii[2] ← int_to_hexscii(j, false);
      j ← (append[i] & 0x0f) ≫ 0;
      ascii[3] ← int_to_hexscii(j, false);
      serial_append(buffer, ascii, 4, failure);
    }
  }
}
```

305. **int** *serial_cycle*(**cell** *cycles*, **cell** *candidate*)

```
{
  int r;
  for (r ← 0; r < fix_value(pointer_datum(cycles)); r++)
    if (array_ref(cycles, r) ≡ candidate) return r;
  return −1;
}
```

306. The first time a cycle is encountered identify and print it, later occurrences refer back to it.

```
char *serial_deduplicate(cell o, int detail, cell buffer, cell cycles, sigjmp_buf *failure)
{
  int c, i;
  assert((LDEBUG_P ∧ null_p(buffer)) ∨ segment_p(buffer));
  assert(array_p(cycles));
  c ← serial_cycle(cycles, o);
  if (c ≡ −1) return Λ;
  i ← c + fix_value(pointer_datum(cycles));
  if (¬detail) array_set_m(cycles, i, LTRUE);
  else if (true_p(array_ref(cycles, i))) {
    serial_append(buffer, "##", 2, failure);
    serial_imp(fix(c), SERIAL_ROUND, 1, true, buffer, cycles, failure);
  }
  else {
    serial_append(buffer, "#=", 2, failure);
    serial_imp(fix(c), SERIAL_ROUND, 1, true, buffer, cycles, failure);
    serial_append(buffer, "␣", 1, failure);
    array_set_m(cycles, i, LTRUE);
    return Λ;
  }
  return "\0";
}
```

307. This is a long chain of **if**/**else if** beginning here. Not sure I like that.

⟨ Serialise an object 307 ⟩ ≡

```
if (pair_p(o)) {
   if (¬maxdepth ∧ detail ≠ SERIAL_ROUND) append ← "\005(...)";
   else if (maxdepth) {
      if (detail) serial_append(buffer, "(", 1, failure);
      serial_imp(lcar(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      for (o ← lcdr(o); pair_p(o); o ← lcdr(o)) {
         if (detail) serial_append(buffer, "␣", 1, failure);
         serial_imp(lcar(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      }
      if (¬null_p(o)) {
         if (detail) serial_append(buffer, "␣.␣", 1, failure);
         serial_imp(o, detail, maxdepth − 1, true, buffer, cycles, failure);
      }
      if (detail) serial_append(buffer, ")", 1, failure);
      append ← "\0";
   }
}
```

See also sections 308, 309, 311, 312, 313, 314, 315, 316, 317, 318, and 319.

This code is cited in section 301.

This code is used in section 301.

308. ⟨ Serialise an object 307 ⟩ +≡

```
else
   if (array_p(o)) {
      if (¬maxdepth ∧ detail ≠ SERIAL_ROUND) append ← "\005[...]";
      else if (maxdepth) {
         if (detail) serial_append(buffer, "[", 1, failure);
         for (i ← 0; i < array_length(o); i++) {
            serial_imp(array_ref(o, i), detail, maxdepth − 1, true, buffer, cycles, failure);
            if (detail ∧ i < array_length(o) − 1) serial_append(buffer, "␣", 1, failure);
         }
         if (detail) serial_append(buffer, "]", 1, failure);
         append ← "\0";
      }
   }
```

309. ⟨ Serialise an object 307 ⟩ +≡

```
else
   if (rope_p(o)) {
      if (detail ≡ SERIAL_DETAIL) {
         serial_append(buffer, "", (int) sizeof (""), failure);
         serial_rope(o, detail, maxdepth, buffer, cycles, failure);
         serial_append(buffer, "", (int) sizeof (""), failure);
      }
      else {
         if (detail) serial_append(buffer, "|", 1, failure);
         serial_rope(o, detail, maxdepth, buffer, cycles, failure);
         if (detail) serial_append(buffer, "|", 1, failure);
      }
      append ← "\0";
   }
```

310.
```
void serial_rope(cell o, int detail, int maxdepth, cell buffer, cell cycles, sigjmp_buf *failure)
{
   cell p;
   assert(rope_p(o));
   if (maxdepth < 0 ∧ detail ≡ SERIAL_ROUND) siglongjmp (*failure, LERR_UNPRINTABLE);
   else if (maxdepth < 0) {
     if (detail) serial_append(buffer, "...", 3, failure);
   }
   else {
     p ← rope_prev(o, failure);
     if (null_p(p)) {
       if (detail ≡ SERIAL_DETAIL) serial_append(buffer, "()", 2, failure);
     }
     else if (serial_deduplicate(p, detail, buffer, cycles, failure) ≡ Λ)
       serial_rope(p, detail, maxdepth − 1, buffer, cycles, failure);
     if (detail ≡ SERIAL_DETAIL) serial_append(buffer, "␣|", 2, failure);
     if (detail) serial_escape(rope_buffer(o), rope_blength(o), buffer, failure);
     if (detail ≡ SERIAL_DETAIL) serial_append(buffer, "|␣", 2, failure);
     p ← rope_next(o, failure);
     if (null_p(p)) {
       if (detail ≡ SERIAL_DETAIL) serial_append(buffer, "()", 2, failure);
     }
     else if (serial_deduplicate(p, detail, buffer, cycles, failure) ≡ Λ)
       serial_rope(p, detail, maxdepth − 1, buffer, cycles, failure);
   }
}
```

311. ⟨Serialise an object 307⟩ +≡
```
else
   if (primitive_p(o) ∧ detail ≠ SERIAL_ROUND) {
     if (detail) {
       serial_append(buffer, "#{primitive␣", 13, failure);
       serial_append(buffer, Iprimitive[primitive(o)].label, (int) strlen(Iprimitive[primitive(o)].label),
           failure);
       serial_append(buffer, "}", 1, failure);
     }
     append ← "\0";
   }
```

312. ⟨Serialise an object 307⟩ +≡
```
else
   if (environment_p(o) ∧ detail ≠ SERIAL_ROUND) {
     if (¬maxdepth) append ← "\015<ENVIRONMENT>";
     else {
       if (detail) serial_append(buffer, "#{environment␣", 14, failure);
       serial_imp(env_layer(o), detail, maxdepth − 1, true, buffer, cycles, failure);
       if (detail) serial_append(buffer, "␣on␣", 4, failure);
       serial_imp(env_previous(o), detail, maxdepth − 1, true, buffer, cycles, failure);
       if (detail) serial_append(buffer, "}", 1, failure);
       append ← "\0";
     }
   }
```

313. `table free/length (id . value)` (*id* ← *hash* % *length*).

```
⟨ Serialise an object 307 ⟩ +≡
  else
    if (keytable_p(o) ∧ detail ≠ SERIAL_ROUND) {
      if (¬maxdepth) append ← "\014#{table␣...}";
      else {
        if (detail) serial_append(buffer, "#{table␣", 8, failure);
        serial_imp(fix(keytable_free(o)), SERIAL_ROUND, 1, true, buffer, cycles, failure);
        if (detail) serial_append(buffer, "/", 1, failure);
        serial_imp(fix(keytable_length(o)), SERIAL_ROUND, 1, true, buffer, cycles, failure);
        for (i ← 0; i < keytable_length(o); i++) {
          if (¬null_p(keytable_ref(o, i))) {
            if (detail) serial_append(buffer, "␣(", 2, failure);
            serial_imp(fix(i), detail, maxdepth − 1, true, buffer, cycles, failure);
            if (detail) serial_append(buffer, "␣.␣", 3, failure);
            serial_imp(keytable_ref(o, i), detail, maxdepth − 1, true, buffer, cycles, failure);
            if (detail) serial_append(buffer, ")", 1, failure);
          }
        }
        if (detail) serial_append(buffer, "}", 1, failure);
        append ← "\0";
      }
    }
```

314.

```
⟨ Serialise an object 307 ⟩ +≡
  else
    if (closure_p(o) ∧ detail ≠ SERIAL_ROUND) {
      if (¬maxdepth) append ← "\016#{closure␣...}";
      else {
        if (detail) {
          serial_append(buffer, "#{", 2, failure);
          if (applicative_p(o)) serial_append(buffer, "applicative␣", 12, failure);
          else serial_append(buffer, "operative␣", 10, failure);
        }
        serial_imp(closure_formals(o), detail, maxdepth − 1, true, buffer, cycles, failure);
        if (detail) serial_append(buffer, "␣", 1, failure);
        serial_imp(closure_body(o), detail, maxdepth − 1, true, buffer, cycles, failure);
        if (detail) serial_append(buffer, "␣in␣", 4, failure);
        serial_imp(closure_environment(o), detail, maxdepth − 1, true, buffer, cycles, failure);
        if (detail) serial_append(buffer, "}", 1, failure);
        append ← "\0";
      }
    }
```

315. ⟨ Serialise an object 307 ⟩ +≡

```
else
  if (dlist_p(o) ∧ detail ≠ SERIAL_ROUND) {
    if (¬maxdepth) append ← "\006<LIST>";
    else {
      if (prefix) {
        if (detail) serial_append(buffer, "#{dlist␣", 8, failure);
        serial_imp(dlist_datum(o), detail, maxdepth − 1, true, buffer, cycles, failure);
        p ← dlist_next(o);
        while (p ≠ o) {
          if (detail) serial_append(buffer, "␣::␣", 4, failure);
          serial_imp(p, detail, maxdepth, false, buffer, cycles, failure);
          p ← dlist_next(p);
        }
        if (detail) serial_append(buffer, "}", 1, failure);
      }
      else serial_imp(dlist_datum(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      append ← "\0";
    }
  }
```

316. ⟨ Serialise an object 307 ⟩ +≡

```
else
  if (note_p(o) ∧ detail ≠ SERIAL_ROUND) {
    if (¬maxdepth) append ← "\006<NOTE>";
    else {
      serial_imp(note(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "", (int) sizeof (""), failure);
      serial_imp(note_car(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "␣␣", (int) sizeof ("␣␣"), failure);
      serial_imp(note_cdr(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "", (int) sizeof (""), failure);
      append ← "\0";
    }
  }
```

317. ⟨ Serialise an object 307 ⟩ +≡

```
else
  if (syntax_p(o) ∧ detail ≠ SERIAL_ROUND) {
    if (¬maxdepth) append ← "\010<SYNTAX>";
    else {
      if (detail) serial_append(buffer, "#{syntax␣", 9, failure);
      serial_imp(syntax_datum(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "␣", 1, failure);
      serial_imp(syntax_start(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "␣", 1, failure);
      serial_imp(syntax_end(o), detail, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "}", 1, failure);
      append ← "\0";
    }
  }
```

318. ⟨ Serialise an object 307 ⟩ +≡
```
else
  if (lexeme_p(o) ∧ detail ≠ SERIAL_ROUND) {
    if (¬maxdepth) append ← "\015#{lexeme␣...}";
    else {
      if (detail) serial_append(buffer, "#{lexeme␣", 9, failure);
      serial_imp(fix(lexeme(o)→cat), SERIAL_ROUND, 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "␣@", 2, failure);
      serial_imp(fix(lexeme(o)→tboffset), SERIAL_ROUND, 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, ":", 1, failure);
      serial_imp(fix(lexeme(o)→blength), SERIAL_ROUND, 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "␣of␣", 4, failure);
      serial_imp(lexeme_twine(o), SERIAL_ROUND, maxdepth − 1, true, buffer, cycles, failure);
      if (detail) serial_append(buffer, "}", 1, failure);
      append ← "\0";
    }
  }
```

319. ⟨ Serialise an object 307 ⟩ +≡
```
else
  if (keytable_p(o)) {
    append ← Λ;
  }
```

320. Miscellanea.

⟨ Function declarations 21 ⟩ +≡
 int *high_bit*(**digit**);

321. **int** *high_bit*(**digit** *o*)

```
{
  int i ← CELL_BITS;
  while (−−i) if (o & (1ULL ≪ i)) return i + 1;
  return o;
}
```

322. ⟨ Repair the system headers 322 ⟩ ≡

```
#ifdef __GNUC__      /* & clang */
#define Lunused __attribute__((__unused__))
#else
#define Lunused      /* noisy compiler */
#endif
#ifdef __GNUC__      /* &clang */
#define Lnoreturn __attribute__ ((__noreturn__))
#else
#ifdef _Noreturn
#define Lnoreturn _Noreturn
#else
#define Lnoreturn      /* noisy compiler */
#endif
#endif
#ifdef LDEBUG
#define LDEBUG_P true
#else
#define LDEBUG_P false
#endif
#if EOF ≡ −1
#define FAIL −2
#else
#define FAIL −1
#endif
#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))
```

This code is used in sections 2 and 3.

323. Junkyard.

324. Symbols.

325. Constants.

⟨ External symbols 16 ⟩ +≡
```
cell lapi_NIL(void);
cell lapi_FALSE(void);
cell lapi_TRUE(void);
cell lapi_VOID(void);
cell lapi_EOF(void);
cell lapi_UNDEFINED(void);
```

326. ⟨ ffi.c 5 ⟩ +≡
```
cell lapi_NIL(void)
{
  return NIL;
}
cell lapi_FALSE(void)
{
  return LFALSE;
}
cell lapi_TRUE(void)
{
  return LTRUE;
}
cell lapi_VOID(void)
{
  return VOID;
}
cell lapi_EOF(void)
{
  return LEOF;
}
cell lapi_UNDEFINED(void)
{
  return UNDEFINED;
}
```

327. Accessors.

⟨ External symbols 16 ⟩ +≡
```
bool lapi_null_p(cell);
bool lapi_false_p(cell);
bool lapi_true_p(cell);
bool lapi_pair_p(cell);
bool lapi_symbol_p(cell);
cell lapi_cons(bool, cell, cell, sigjmp_buf *);
cell lapi_car(cell, sigjmp_buf *);
cell lapi_cdr(cell, sigjmp_buf *);
void lapi_set_car_m(cell, cell, sigjmp_buf *);
void lapi_set_cdr_m(cell, cell, sigjmp_buf *);
```

328. **bool** *lapi_null_p*(**cell** *o*)
```
{
  return null_p(o);
}
bool lapi_false_p(cell o)
{
  return false_p(o);
}
bool lapi_true_p(cell o)
{
  return true_p(o);
}
bool lapi_pair_p(cell o)
{
  return pair_p(o);
}
bool lapi_symbol_p(cell o)
{
  return symbol_p(o);
}
```

329. **cell** *lapi_cons*(**bool** *share*, **cell** *ncar*, **cell** *ncdr*, **sigjmp_buf** ∗*failure*)
```
{
  if (¬defined_p(ncar) ∨ ¬defined_p(ncdr)) siglongjmp (∗failure, LERR_INCOMPATIBLE);
  return atom(share ? Sheap : Theap, ncar, ncdr, FORM_PAIR, failure);
}
cell lapi_car(cell o, sigjmp_buf ∗failure)
{
  if (¬pair_p(o)) siglongjmp (∗failure, LERR_INCOMPATIBLE);
  return lcar(o);
}
cell lapi_cdr(cell o, sigjmp_buf ∗failure)
{
  if (¬pair_p(o)) siglongjmp (∗failure, LERR_INCOMPATIBLE);
  return lcdr(o);
}
void lapi_set_car_m(cell o, cell value, sigjmp_buf ∗failure)
{
  if (¬pair_p(o) ∨ ¬defined_p(value)) siglongjmp (∗failure, LERR_INCOMPATIBLE);
  lcar_set_m(o, value);
}
void lapi_set_cdr_m(cell o, cell value, sigjmp_buf ∗failure)
{
  if (¬pair_p(o) ∨ ¬defined_p(value)) siglongjmp (∗failure, LERR_INCOMPATIBLE);
  lcdr_set_m(o, value);
}
```

330. **cell** *lapi_env_search*(**cell** *env*, **cell** *label*, **sigjmp_buf** *∗failure*)
{
  cell *r*;
  if (*null_p*(*env*)) *env* ← *Environment*;
  if (¬*environment_p*(*env*) ∨ ¬*symbol_p*(*label*)) **siglongjmp** (∗*failure*, LERR_INCOMPATIBLE);
  r ← *env_search*(*env*, *label*, *true*, *failure*);
  if (*undefined_p*(*r*)) **siglongjmp** (∗*failure*, LERR_MISSING);
  else return *r*;
}
void *lapi_env_define*(**cell** *env*, **cell** *label*, **cell** *value*, **sigjmp_buf** *∗failure*)
{
  if (*null_p*(*env*)) *env* ← *Environment*;
  if (¬*environment_p*(*env*) ∨ ¬*symbol_p*(*label*) ∨ ¬*defined_p*(*value*))
    siglongjmp (∗*failure*, LERR_INCOMPATIBLE);
  env_define(*env*, *label*, *value*, *failure*);
}
void *lapi_env_set*(**cell** *env*, **cell** *label*, **cell** *value*, **sigjmp_buf** *∗failure*)
{
  if (*null_p*(*env*)) *env* ← *Environment*;
  if (¬*environment_p*(*env*) ∨ ¬*symbol_p*(*label*) ∨ ¬*defined_p*(*value*))
    siglongjmp (∗*failure*, LERR_INCOMPATIBLE);
  env_set(*env*, *label*, *value*, *failure*);
}
void *lapi_env_clear*(**cell** *env*, **cell** *label*, **sigjmp_buf** *∗failure*)
{
  if (*null_p*(*env*)) *env* ← *Environment*;
  if (¬*environment_p*(*env*) ∨ ¬*symbol_p*(*label*)) **siglongjmp** (∗*failure*, LERR_INCOMPATIBLE);
  env_clear(*env*, *label*, *failure*);
}
void *lapi_env_unset*(**cell** *env*, **cell** *label*, **sigjmp_buf** *∗failure*)
{
  if (*null_p*(*env*)) *env* ← *Environment*;
  if (¬*environment_p*(*env*) ∨ ¬*symbol_p*(*label*)) **siglongjmp** (∗*failure*, LERR_INCOMPATIBLE);
  env_unset(*env*, *label*, *failure*);
}

331. ⟨ Function declarations 21 ⟩ +≡
cell *lapi_Accumulator*(**cell**);
cell *lapi_User_Register*(**cell**);

332. **cell** *lapi_Accumulator*(**cell** *new*)
{
  if (*defined_p*(*new*)) *Accumulator* ← *new*;
  return *Accumulator*;
}
cell *lapi_User_Register*(**cell** *new*)
{
  if (*defined_p*(*new*)) *User_Register* ← *new*;
  return *User_Register*;
}

333. TODO.

```
void mem_init(void)
{
  sigjmp_buf failed, *failure ← &failed;
  Verror reason ← LERR_NONE;
  cell x;
  int i;
  if (failure_p(reason ← sigsetjmp(failed, 1))) {
    fprintf(stderr, "FATAL␣Initialisation␣error␣%u:␣%s.\n", reason, Ierror[reason].message);
    abort();
  }
  ⟨ Save register locations 55 ⟩
  ⟨ Initialise storage 35 ⟩
  ⟨ Register primitive operators 188 ⟩
  ⟨ Prepare constants & symbols 182 ⟩
  ENV ← env_extend(Root, failure);
}
```

334.

```
int lprint(char *format, ...)
{
  va_list args;
  int r;
  assert(LDEBUG_P);
  va_start(args, format);
  r ← vfprintf(stdout, format, args);
  va_end(args);
  return r;
}
int lput(int c)
{
  assert(LDEBUG_P);
  return putchar(c);
}
```

335. ⟨ Function declarations 21 ⟩ +≡

```
void mem_init(void);
int lprint(char *, ...);
int lput(int);
```

336. `#define WARN() fprintf(stderr, "WARNING:␣You␣probably␣don't␣want␣to␣do␣that.\n");`

LOSSLESS