1. Introduction. `LossLess` is a programming language and environment similar to scheme. This document describes the implementation of a `LossLess` runtime written in C and `LossLess` itself will be described elsewhere. In unambiguous cases `LossLess` may be used to refer specifically to the implementation.

This code started off its life as `s9fes` by Nils M. Holm[1]. After a few iterations including being briefly ported to perl this rather different code is the result, although at its core it follows the same design.

`LossLess` is built as a library. The header file `lossless.h` can be included to link against it, which the REPL does.

```
⟨lossless.h 1⟩ ≡
#ifndef LOSSLESS_H
#define LOSSLESS_H
  ⟨API headers 5⟩
  ⟨Preprocessor definitions⟩
  ⟨Type definitions 6⟩
  ⟨API declarations 9⟩
#endif
```

2. The structure is of a virtual machine with a single accumulator register and a stack. There is a single entry point to the VM—*interpret*—called after parsed source code has been put into the accumulator, where the result will also be left.

```
  ⟨System headers 4⟩
  ⟨Preprocessor definitions⟩
  ⟨Global constants 112⟩
  ⟨Type definitions 6⟩
  ⟨Function declarations 8⟩
  ⟨Global variables 7⟩
```

3. ⟨Global initialisation 3⟩ ≡ /* This is located here to name it in full for CWEB's benefit */

See also sections 24, 52, 83, and 156.

This code is cited in section 70.

This code is used in section 71.

4. `LossLess` has few external dependencies, primarily *stdio* and *stdlib*, plus some obvious memory mangling functions from the C library there's no point in duplicating.

`LL_ALLOCATE` allows us to define a wrapper around *reallocarray* which is used to make it artificially fail during testing.

```
⟨System headers 4⟩ ≡
#include <ctype.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>    /* for memset */
#include <sys/types.h>
#ifndef LL_ALLOCATE
#define LL_ALLOCATE reallocarray
#endif
```

This code is used in section 2.

[1] `http://t3x.org/s9fes/`

5. ⟨ API headers 5 ⟩ ≡

```
#include <setjmp.h>
#include <stdlib.h>
```

This code is used in section 1.

6. The *boolean* and *predicate* C types are used to distinguish between *boolean*-returning functions reporting C truth (0 or 1) or *predicate*-returning functions reporting `LossLess` truth (`FALSE` or `TRUE`). Otherwise-untyped C macros always report C truth.

```
#define bfalse 0
#define btrue 1
```

⟨ Type definitions 6 ⟩ ≡

```
typedef int32_t cell;
typedef int boolean;
typedef cell predicate;
```

See also sections 66 and 199.

This code is used in sections 1 and 2.

7. Error Handling. Everything needs to be able to report errors and so even though the details will make little sense without a more complete understanding of `LossLess` the code and data to handle them come first in full.

When the VM begins it establishes two jump buffers. To understand jump buffers it's necessary to understand how C's stack works and we have enough stacks already.

The main thing to know is that whenever C code calls a function it grows its own stack with the caller's return address. When *setjmp* is called the position in this stack is saved. When jumping back to that position with *longjmp*, anything which has been added to C's stack since the corresponding call to *setjmp* is discarded which has the effect of returning to exactly the point in the program where the corresponding *setjmp* was called, this time with a non-zero return value (the value that was given as an argument to *longjmp*; this facility is not used by `LossLess` for anything and it always sends 1).

The other thing that you don't need to know is that sometimes C compilers can make the previous paragraph a tissue of lies.

```
#define ERR_UNIMPLEMENTED "unimplemented"
#define error(x, d) handle_error((x), NIL, (d))
#define ex_id car
#define ex_detail cdr
```

⟨ Global variables 7 ⟩ ≡

```
  volatile boolean Error_Handler = bfalse;
  jmp_buf Goto_Begin;
  jmp_buf Goto_Error;
```

See also sections 13, 17, 21, 23, 35, 42, 50, 67, 68, 74, 82, 131, 155, 182, and 192.

This code is used in section 2.

8. ⟨ Function declarations 8 ⟩ ≡

```
  void handle_error(char *, cell, cell) __dead;
  void warn(char *, cell);
```

See also sections 19, 30, 37, 44, 84, 104, 132, 167, 184, 191, 256, and 286.

This code is used in section 2.

9. ⟨ API declarations 9 ⟩ ≡

```
  extern volatile boolean Error_Handler;
  extern jmp_buf Goto_Begin;
  extern jmp_buf Goto_Error;
  void handle_error(char *, cell, cell) __dead;
  void warn(char *, cell);
```

See also sections 45, 70, 78, 81, and 105.

This code is used in section 1.

10. Raised errors may either be a C-'string'[1] when raised by an internal process or a *symbol* when raised at run-time.

If an error handler has been established then the *id* and *detail* are promoted to an *exception* object and the handler entered.

```
void handle_error(char *message, cell id, cell detail)
{
  int len;
  if (¬null_p(id)) {
    message = symbol_store(id);
    len = symbol_length(id);
  }
  else len = strlen(message);
  if (Error_Handler) {    /* TODO: Save Acc or rely on id being it? */
    vms_push(detail);
    if (null_p(id)) id = sym(message);
    Acc = atom(id, detail, FORMAT_EXCEPTION);
    vms_clear();
    longjmp(Goto_Error, 1);
  }
  printf("UNHANDLED␣ERROR:␣");
  for ( ; len--; message++) putchar(*message);
  putchar(':');
  putchar('␣');
  write_form(detail, 0);
  printf("\n");
  longjmp(Goto_Begin, 1);
}
```

11. Run-time errors are raised by the `OP_ERROR` opcode which passes control to *handle_error* (and never returns). The code which compiles **error** to emit this opcode comes later after the compiler has been defined.

```
⟨ Opcode implementations 11 ⟩ ≡
case OP_ERROR:
  handle_error(Λ, Acc, rts_pop(1));
  break;    /* superfluous */
```

See also sections 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 128, 129, and 130.
This code is used in section 79.

12. We additionally define *warn* here because where else is it going to go?

```
void warn(char *message, cell detail)
{
  printf("WARNING:␣%s:␣", message);
  write_form(detail, 0);
  printf("\n");
}
```

[1] C does not have strings, it has pointers to memory buffers that probably contain ASCII and might also happen to have a Λ in them somewhere.

13. Memory Management. The most commonly used data type in lisp-like languages is the *pair*, also called a "cons cell" for histerical raisins, which is a datum consisting of two equally-sized halves. For reasons that don't bear thinking about they are called the *car* for the "first" half and the *cdr* for the "second" half. In this code & document, **cell** refers to each half of a *pair*. **cell** is not used to refer to a whole cons cell in order to avoid confusion.

A *pair* in `LossLess` is stored in 2 equally-sized areas of memory. On 64-bit x86 implementations, which are all I'm considering at the moment, each half is 32 bits wide. Each pair additionally has an 8 bit tag (1 byte) associated with it, stored in a third array.

Internally a *pair* is represented by an offset into these memory areas. Negative numbers are therefore available for a few global constants.

The *pair*'s tag is treated as a bitfield. The garbage collector uses two bits (`TAG_MARK` and `TAG_STATE`). The other 6 bits are used to identify what data is stored in the **cell**s.

```
#define NIL −1      /∗ Not Λ, but not nil_p/nil? either ∗/
#define FALSE −2    /∗ Yes, ∗/
#define TRUE −3     /∗ really. ∗/
#define END_OF_FILE −4    /∗ stdio has EOF ∗/
#define VOID −5
#define UNDEFINED −6

#define TAG_NONE #00
#define TAG_MARK #80     /∗ GC mark bit ∗/
#define TAG_STATE #40    /∗ GC state bit ∗/
#define TAG_ACARP #20    /∗ CAR is a pair ∗/
#define TAG_ACDRP #10    /∗ CDR is a pair ∗/
#define TAG_FORMAT #3f   /∗ Mask lower 6 bits ∗/

#define HEAP_SEGMENT #8000

⟨ Global variables 7 ⟩ +≡
  cell ∗CAR = Λ;
  cell ∗CDR = Λ;
  char ∗TAG = Λ;
  cell Cells_Free = NIL;
  int Cells_Poolsize = 0;
  int Cells_Segment = HEAP_SEGMENT;
```

14. The pool is spread across CAR, CDR and TAG and starts off with a size of zero **cell**s, growing by *Cells_Segment* **cell**s each time it's enlarged. When the heap is enlarged newly allocated memory is set to zero and the segment size set to half of the total pool size.

```
#define ERR_OOM "out-of-memory"
#define enlarge_pool(p, m, t) do
        {
          void *n;
          n = LL_ALLOCATE((p), (m), sizeof (t));
          if (¬n) error (ERR_OOM, NIL);
          (p) = n;
        }
        while (0)
  void new_cells_segment(void)
  {
    enlarge_pool(CAR, Cells_Poolsize + Cells_Segment, cell);
    enlarge_pool(CDR, Cells_Poolsize + Cells_Segment, cell);
    enlarge_pool(TAG, Cells_Poolsize + Cells_Segment, char);
    bzero(CAR + Cells_Poolsize, Cells_Segment * sizeof(cell));
    bzero(CDR + Cells_Poolsize, Cells_Segment * sizeof(cell));
    bzero(TAG + Cells_Poolsize, Cells_Segment * sizeof(char));
    Cells_Poolsize += Cells_Segment;
    Cells_Segment = Cells_Poolsize/2;
  }
```

15. Preprocessor directives provide precidates to interrogate a *pair*'s tag and find out what it is.

Although not all of these *cXr* macros are used they are all defined here for completeness (and it's easier than working out which ones really are needed).

#define $special\_p(p)$ $((p) < 0)$
#define $boolean\_p(p)$ $((p) \equiv \texttt{FALSE} \lor (p) \equiv \texttt{TRUE})$
#define $eof\_p(p)$ $((p) \equiv \texttt{END\_OF\_FILE})$
#define $false\_p(p)$ $((p) \equiv \texttt{FALSE})$
#define $null\_p(p)$ $((p) \equiv \texttt{NIL})$
#define $true\_p(p)$ $((p) \equiv \texttt{TRUE})$
#define $void\_p(p)$ $((p) \equiv \texttt{VOID})$
#define $undefined\_p(p)$ $((p) \equiv \texttt{UNDEFINED})$

#define $mark\_p(p)$ $(\neg special\_p(p) \land (\texttt{TAG}[(p)]\ \&\ \texttt{TAG\_MARK}))$
#define $state\_p(p)$ $(\neg special\_p(p) \land (\texttt{TAG}[(p)]\ \&\ \texttt{TAG\_STATE}))$
#define $acar\_p(p)$ $(\neg special\_p(p) \land (\texttt{TAG}[(p)]\ \&\ \texttt{TAG\_ACARP}))$
#define $acdr\_p(p)$ $(\neg special\_p(p) \land (\texttt{TAG}[(p)]\ \&\ \texttt{TAG\_ACDRP}))$
#define $mark\_clear(p)$ $(\texttt{TAG}[(p)]\ \&= \sim\texttt{TAG\_MARK})$
#define $mark\_set(p)$ $(\texttt{TAG}[(p)]\ |= \texttt{TAG\_MARK})$
#define $state\_clear(p)$ $(\texttt{TAG}[(p)]\ \&= \sim\texttt{TAG\_STATE})$
#define $state\_set(p)$ $(\texttt{TAG}[(p)]\ |= \texttt{TAG\_STATE})$
#define $format(p)$ $(\texttt{TAG}[(p)]\ \&\ \texttt{TAG\_FORMAT})$

#define $tag(p)$ $(\texttt{TAG}[(p)])$

#define $car(p)$ $(\texttt{CAR}[(p)])$
#define $cdr(p)$ $(\texttt{CDR}[(p)])$
#define $caar(p)$ $(\texttt{CAR}[\texttt{CAR}[(p)]])$
#define $cadr(p)$ $(\texttt{CAR}[\texttt{CDR}[(p)]])$
#define $cdar(p)$ $(\texttt{CDR}[\texttt{CAR}[(p)]])$
#define $cddr(p)$ $(\texttt{CDR}[\texttt{CDR}[(p)]])$
#define $caaar(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CAR}[(p)]]])$
#define $caadr(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CDR}[(p)]]])$
#define $cadar(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CAR}[(p)]]])$
#define $caddr(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CDR}[(p)]]])$
#define $cdaar(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CAR}[(p)]]])$
#define $cdadr(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CDR}[(p)]]])$
#define $cddar(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CAR}[(p)]]])$
#define $cdddr(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CDR}[(p)]]])$
#define $caaaar(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CAR}[\texttt{CAR}[(p)]]]])$
#define $caaadr(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CAR}[\texttt{CDR}[(p)]]]])$
#define $caadar(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CDR}[\texttt{CAR}[(p)]]]])$
#define $caaddr(p)$ $(\texttt{CAR}[\texttt{CAR}[\texttt{CDR}[\texttt{CDR}[(p)]]]])$
#define $cadaar(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CAR}[\texttt{CAR}[(p)]]]])$
#define $cadadr(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CAR}[\texttt{CDR}[(p)]]]])$
#define $caddar(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CDR}[\texttt{CAR}[(p)]]]])$
#define $cadddr(p)$ $(\texttt{CAR}[\texttt{CDR}[\texttt{CDR}[\texttt{CDR}[(p)]]]])$
#define $cdaaar(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CAR}[\texttt{CAR}[(p)]]]])$
#define $cdaadr(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CAR}[\texttt{CDR}[(p)]]]])$
#define $cdadar(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CDR}[\texttt{CAR}[(p)]]]])$
#define $cdaddr(p)$ $(\texttt{CDR}[\texttt{CAR}[\texttt{CDR}[\texttt{CDR}[(p)]]]])$
#define $cddaar(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CAR}[\texttt{CAR}[(p)]]]])$
#define $cddadr(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CAR}[\texttt{CDR}[(p)]]]])$
#define $cdddar(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CDR}[\texttt{CAR}[(p)]]]])$
#define $cddddr(p)$ $(\texttt{CDR}[\texttt{CDR}[\texttt{CDR}[\texttt{CDR}[(p)]]]])$

16. Both *atoms* and cons cells are stored in *pairs*. The lower 6 bits of the tag define the format of data stored in that *pair*. The *atoms* are grouped into three types depending on whether both **cells** point to another *pair*, whether only the *cdr* does, or whether both **cells** are opaque. From this we obtain the core data types.

```
#define FORMAT_CONS (TAG_ACARP | TAG_ACDRP | #00)
#define FORMAT_APPLICATIVE (TAG_ACARP | TAG_ACDRP | #01)
#define FORMAT_OPERATIVE (TAG_ACARP | TAG_ACDRP | #02)
#define FORMAT_SYNTAX (TAG_ACARP | TAG_ACDRP | #03)
#define FORMAT_ENVIRONMENT (TAG_ACARP | TAG_ACDRP | #04)
#define FORMAT_EXCEPTION (TAG_ACARP | TAG_ACDRP | #05)
#define FORMAT_INTEGER (TAG_ACDRP | #00)   /* value : next/NIL */
#define FORMAT_SYMBOL (TAG_NONE | #00)   /* length : offset */
#define FORMAT_VECTOR (TAG_NONE | #01)   /* gc-index : offset */
#define FORMAT_COMPILER (TAG_NONE | #02)   /* offset : NIL */
#define atom_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≠ (TAG_ACARP | TAG_ACDRP)))
#define pair_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ (TAG_ACARP | TAG_ACDRP)))
#define applicative_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_APPLICATIVE))
#define compiler_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_COMPILER))
#define environment_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_ENVIRONMENT))
#define integer_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_INTEGER))
#define operative_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_OPERATIVE))
#define symbol_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_SYMBOL))
#define syntax_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_SYNTAX))
#define vector_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_VECTOR))
```

17. Allocating a new *pair* may require garbage collection to be performed. If the data being put into either half of the new *pair* is itself a *pair* it may be discarded by the collector. To avoid this happening the data are saved into preallocated temporary storage while a new *pair* is being located.

```
⟨ Global variables 7 ⟩ +≡
  cell Tmp_CAR = NIL;
  cell Tmp_CDR = NIL;
```

18. ⟨ Protected Globals 18 ⟩ ≡

```
  &Tmp_CAR, &Tmp_CDR ,
```

See also sections 36, 43, 69, 133, and 183.
This code is used in section 31.

19. ⟨ Function declarations 8 ⟩ +≡

```
  cell atom(cell, cell, char);
```

20. **#define** *cons*(*a*, *d*) *atom*((*a*), (*d*), FORMAT_CONS)

```
cell atom(cell ncar, cell ncdr, char ntag)
{
  cell r;
  if (null_p(Cells_Free)) {
    if (ntag & TAG_ACARP) Tmp_CAR = ncar;
    if (ntag & TAG_ACDRP) Tmp_CDR = ncdr;
    if (gc() ≤ (Cells_Poolsize/2)) {
      new_cells_segment();
      gc();
    }
    Tmp_CAR = Tmp_CDR = NIL;
  }
  r = Cells_Free;
  Cells_Free = cdr(Cells_Free);
  car(r) = ncar;
  cdr(r) = ncdr;
  tag(r) = ntag;
  return r;
}
```

21. Vectors. A *vector* stores a contiguous sequence of **cell**s, each referring to a *pair* on the heap. Unlike *pair*s *vector*s are compacted during garbage collection to avoid fragmentation.

Storage is largely the same as **cell**s except for how the free pointer is maintained: an index into the next unused **cell** in VECTOR.

⟨ Global variables 7 ⟩ +≡
cell ∗VECTOR = Λ;
int *Vectors_Free* = 0;
int *Vectors_Poolsize* = 0;
int *Vectors_Segment* = HEAP_SEGMENT;

22. **void** *new_vector_segment*(**void**)
{
cell ∗*new_vector*;
new_vector = LL_ALLOCATE(VECTOR, *Vectors_Poolsize* + *Vectors_Segment*, **sizeof**(**cell**));
if (*new_vector* ≡ Λ) **error** (ERR_OOM, NIL);
bzero(*new_vector* + *Vectors_Poolsize*, *Vectors_Segment* ∗ **sizeof**(**cell**));
VECTOR = *new_vector*;
Vectors_Poolsize += *Vectors_Segment*;
Vectors_Segment = *Vectors_Poolsize*/2;
}

23. When a *pair* holds a *vector* its tag is FORMAT_VECTOR, the *car* is used by the garbage collecter and the *cdr* is an index into VECTOR.

Each *vector* contains 2 additional pieces of metadata (which are **above** the index), the length of the *vector* and a reference back to the *pair* holding the *vector*.

A *vector* of length 0 is treated as a global constant akin to NIL but it must be stored in a variable and created during initialisation.

#define VECTOR_CELL 0
#define VECTOR_SIZE 1
#define VECTOR_HEAD 2
#define *vector_realsize*(*s*) ((*s*) + VECTOR_HEAD)
#define *vector_cell*(*v*) (VECTOR[*vector_offset*(*v*) − (VECTOR_HEAD − VECTOR_CELL)])
#define *vector_index*(*v*) (*car*(*v*))
#define *vector_length*(*v*) (VECTOR[*vector_offset*(*v*) − (VECTOR_HEAD − VECTOR_SIZE)])
#define *vector_offset*(*v*) (*cdr*(*v*))
#define *vector_ref*(*v*, *o*) (VECTOR[*vector_offset*(*v*) + (*o*)])

⟨ Global variables 7 ⟩ +≡
cell *Zero_Vector* = NIL;

24. ⟨ Global initialisation 3 ⟩ +≡
Zero_Vector = *vector_new_imp*(0, 0, 0);

25. Separate storage means separate garbage collection and a different allocator. *vector_new_imp*, again, is broadly similar to *atom* without the need for preallocated storage.

26. **cell** *vector_new_imp*(**int** *size*, **int** *fill_p*, **cell** *fill*)

```
{
  int wsize, off, i;
  cell r;
  wsize = vector_realsize(size);
  if (Vectors_Free + wsize ≥ Vectors_Poolsize) {
    gc_vectors();
    while (Vectors_Free + wsize ≥ (Vectors_Poolsize − (Vectors_Poolsize/2))) {
      new_vector_segment();
      gc_vectors();     /* Is this really necessary? */
    }
  }
  r = atom(NIL, NIL, FORMAT_VECTOR);
  off = Vectors_Free;
  Vectors_Free += wsize;
  vector_offset(r) = off + VECTOR_HEAD;     /* must be first */
  vector_length(r) = size;
  vector_cell(r) = r;
  vector_index(r) = 0;
  if (fill_p)
    for (i = VECTOR_HEAD; i ≤ size + (VECTOR_HEAD − 1); i++)
      vector_ref(r, i) = fill;
  return r;
}
```

27. **cell** *vector_new*(**int** *size*, **cell** *fill*)

```
{
  if (size ≡ 0) return Zero_Vector;
  return vector_new_imp(size, 1, fill);
}
```

28. *vector_new_list* turns a *list* of *pair*s into a *vector*.

```
cell vector_new_list(cell list, int len)
{
  cell r;
  int i;
  r = vector_new(len, 0);
  for (i = 0; i < len; i++) {
    vector_ref(r, i) = car(list);
    list = cdr(list);
  }
  return r;
}
```

29. Although a little early in the narrative *vector_sub* is defined here because it's the only other function substantially dealing with *vector* data.

```
cell vector_sub(cell src, int srcfrom, int srcto, int dstfrom, int dstto, cell fill)
{
  cell dst;
  int copy, i;
  copy = srcto − srcfrom;
  if (dstto < 0) dstto = dstfrom + copy;
  dst = vector_new_imp(dstto, 0, 0);
  for (i = 0; i < dstfrom; i++) vector_ref(dst, i) = fill;
  for (i = srcfrom; i < srcto; i++)
    vector_ref(dst, (dstfrom − srcfrom) + i) = vector_ref(src, i);
  for (i = dstfrom + copy; i < dstto; i++) vector_ref(dst, i) = fill;
  return dst;
}
```

30. Garbage Collection. The garbage collector is a straightforward mark and sweep collector. *mark* is called for every entry in `ROOTS` to recursively set the mark bit on every reachable *pair*, then the whole pool is scanned and any *pair*s which aren't marked are added to the free list.

⟨ Function declarations 8 ⟩ +≡
  int *gc*(**void**);
  int *gc_vectors*(**void**);

31. `ROOTS` is a Λ-terminated C array of objects to protect from collection. I can't think of any better way of declaring it but hard-coding it right here.

  cell ∗`ROOTS`[ ] = {⟨ Protected Globals 18 ⟩, Λ};

32.
```
void mark(cell next)
{
  cell parent, prev;
  int i;

  parent = prev = NIL;
  while (1) {
    if (¬(special_p(next) ∨ mark_p(next))) {
      if (vector_p(next)) {    /* S0 → S.1 */
        mark_set(next);
        vector_cell(next) = next;
        if (vector_length(next) > 0) {
          state_set(next);
          vector_index(next) = 0;
          prev = vector_ref(next, 0);
          vector_ref(next, 0) = parent;
          parent = next;
          next = prev;
        }
      }
      else if (¬acar_p(next) ∧ acdr_p(next)) {    /* S0 → S2 */
        prev = cdr(next);
        cdr(next) = parent;
        parent = next;
        next = prev;
        mark_set(parent);
      }
      else if (acar_p(next)) {    /* S0 → S1 */
        prev = car(next);
        car(next) = parent;
        mark_set(next);
        parent = next;
        next = prev;
        state_set(parent);
      }
      else {    /* S0 → S1 */
        mark_set(next);
      }
    }
    else {
      if (null_p(parent)) break;
      if (vector_p(parent)) {    /* S.1 → S.1/done */
        i = vector_index(parent);
        if ((i + 1) < vector_length(parent)) {
          prev = vector_ref(parent, i + 1);
          vector_ref(parent, i + 1) = vector_ref(parent, i);
          vector_ref(parent, i) = next;
          next = prev;
          vector_index(parent) = i + 1;
        }
        else {    /* S.1 → done */
          state_clear(parent);
          prev = parent;
```

```
          parent = vector_ref(prev, i);
          vector_ref(prev, i) = next;
          next = prev;
        }
      }
      else if (state_p(parent)) {      /* S1 → S2 */
        prev = cdr(parent);
        cdr(parent) = car(parent);
        car(parent) = next;
        state_clear(parent);
        next = prev;
      }
      else if (acdr_p(parent)) {      /* S2 → done */
        prev = parent;
        parent = cdr(prev);
        cdr(prev) = next;
        next = prev;
      }
      else {
        error(ERR_UNIMPLEMENTED, NIL);
      }
    }
  }
}
int gc(void)
{
  int count, sk, i;
  if (¬null_p(RTS)) {
    sk = vector_length(RTS);
    vector_length(RTS) = RTSp + 1;
  }
  for (i = 0; ROOTS[i]; i++) mark(*ROOTS[i]);
  for (i = SCHAR_MIN; i ≤ SCHAR_MAX; i++) {
    mark(Small_Int[(unsigned char) i]);
  }
  if (¬null_p(RTS)) vector_length(RTS) = sk;
  Cells_Free = NIL;
  count = 0;
  for (i = 0; i < Cells_Poolsize; i++) {
    if (¬mark_p(i)) {
      cdr(i) = Cells_Free;
      Cells_Free = i;
      count++;
    }
    else {
      mark_clear(i);
    }
  }
  return count;
}
```

33. *vector* garbage collection works by using the *pair*s garbage collector to scan `ROOTS` and determine which vectors are really in use then removes any which aren't from `VECTORS`, decrementing *Vectors_Free* if it can.

```
int gc_vectors(void)
{
  int to, from, d, i, r;
  ⟨Unmark all vectors 34⟩
  gc();
  from = to = 0;
  while (from < Vectors_Free) {
    d = vector_realsize(VECTOR[from + VECTOR_SIZE]);
    if (¬null_p(VECTOR[from + VECTOR_CELL])) {
      if (to ≠ from) {
        for (i = 0; i < d; i++) VECTOR[to + i] = VECTOR[from + i];
        vector_offset(VECTOR[to + VECTOR_CELL]) = to + VECTOR_HEAD;
      }
      to += d;
    }
    from += d;
  }
  r = Vectors_Free − to;
  Vectors_Free = to;
  return r;
}
```

34. To "unmark" a *vector*, all the links in `VECTOR` back to the cell which refers to it (*vector_cell*) are set to `NIL`. *gc* will re-set the link in any vectors that it can reach.

```
⟨Unmark all vectors 34⟩ ≡
  i = 0;
  while (i < Vectors_Free) {
    VECTOR[i + VECTOR_CELL] = NIL;
    i += vector_realsize(VECTOR[i + VECTOR_SIZE]);
  }
```

This code is used in section 33.

35. Objects. Although not `objects` per se, the first `objects` which will be defined are three stacks. We could define the run-time stack later because it's not used until the virtual machine is implemented but the implementations mirror each other and the internal VM stack is required before real objects can be defined. Also the runtime stack uses the VM stack in its implementation.

The compiler stack is included here because it's identical to the VM stack.

The VM stack is a pointer to the head of a *list*. This means that accessing the top few elements of the stack—especially pushing and popping a single `object`—is effectively free but accessing an arbitrary part of the stack requires an expensive walk over each item in turn.

On the other hand the run-time stack is stored in a *vector* with a pointer *RTSp* to the current head of the stack, which is -1 if the stack is empty.

This has the obvious disadvantage that its storage space is finite and occasionally the whole stack will need to be copied into a new, larger *vector* (and conversely it may waste space or require occasional trimming). On the other hand random access to any part of the stack has the same (negligable) cost.

When it's not ambiguous "stack" in this document refers to the run-time stack; the VM stack is an implementation detail. In fact the run-time stack is also an implementation detail but the VM stack is an implementation detail of that implementation detail; do you like recursion yet?.

The main interface to each stack is its *push*/*pop*/*ref*/*clear* functions. There are some additional handlers for the run-time stack.

```
#define ERR_UNDERFLOW "underflow"
#define ERR_OVERFLOW "overflow"
#define CHECK_UNDERFLOW(s) if (null_p(s)) error (ERR_UNDERFLOW, VOID)
#define RTS_UNDERFLOW(p) if ((p) < −1) error (ERR_UNDERFLOW, RTS)
#define RTS_OVERFLOW(p) if ((p) > RTSp) error (ERR_OVERFLOW, RTS)
```

```
⟨ Global variables 7 ⟩ +≡
  cell CTS = NIL;
  cell RTS = NIL;
  cell VMS = NIL;
  int RTS_Size = 0;
  int RTSp = −1;
```

36. ⟨ Protected Globals 18 ⟩ +≡

```
  &CTS, &RTS, &VMS ,
```

37. ⟨ Function declarations 8 ⟩ +≡

```
  cell vms_pop(void);
  void vms_push(cell);
```

38. The VM and compiler stacks `VMS` and `CTS` are built on *list*s.

```
#define vms_clear() ((void) vms_pop())
  cell vms_pop(void)
  {
    cell r;
    CHECK_UNDERFLOW(VMS);
    r = car(VMS);
    VMS = cdr(VMS);
    return r;
  }
  void vms_push(cell item)
  { VMS = cons(item, VMS); }
  cell vms_ref(void)
  {
    CHECK_UNDERFLOW(VMS);
    return car(VMS);
  }
  void vms_set(cell item)
  {
    CHECK_UNDERFLOW(VMS);
    car(VMS) = item;
  }
```

39. `CTS` is treated identically to `VMS`. Using the C preprocessor for this would be unnecessarily inelegant so instead here is a delicious bowl of pasta.

```
#define cts_clear() ((void) cts_pop())
#define cts_reset() CTS = NIL
  cell cts_pop()
  {
    cell r;
    CHECK_UNDERFLOW(CTS);
    r = car(CTS);
    CTS = cdr(CTS);
    return r;
  }
  void cts_push(cell item)
  { CTS = cons(item, CTS); }
  cell cts_ref(void)
  {
    CHECK_UNDERFLOW(CTS);
    return car(CTS);
  }
  void cts_set(cell item)
  {
    CHECK_UNDERFLOW(CTS);
    car(CTS) = item;
  }
```

40. Being built on a *vector* the run-time stack needs to increase its size when it's full. Functions can call *rts_prepare* to ensure that the stack is big enough for their needs.

```
#define RTS_SEGMENT #1000
  void rts_prepare(int need)
  {
    int b, s;
    if (RTSp + need ≥ RTS_Size) {
      b = RTS_SEGMENT ∗ ((need + RTS_SEGMENT)/RTS_SEGMENT);
      s = RTS_Size + b;
      RTS = vector_sub(RTS, 0, RTS_Size, 0, s, UNDEFINED);
      RTS_Size = s;
    }
  }
```

41. Otherwise, the run-time stack has the same interface but a different implementation.

```
#define rts_clear(c) ((void) rts_pop(c))
#define rts_reset() Fp = RTSp = -1;
  cell rts_pop(int count)
  {
    RTS_UNDERFLOW(RTSp - count);
    RTSp -= count;
    return vector_ref(RTS, RTSp + 1);
  }
  void rts_push(cell o)
  {
    vms_push(o);
    rts_prepare(1);
    vector_ref(RTS, ++RTSp) = vms_pop();
  }
  cell rts_ref(int d)
  {
    RTS_UNDERFLOW(RTSp - d);
    RTS_OVERFLOW(RTSp - d);
    return vector_ref(RTS, RTSp - d);
  }
  cell rts_ref_abs(int d)
  {
    RTS_UNDERFLOW(d);
    RTS_OVERFLOW(d);
    return vector_ref(RTS, d);
  }
  void rts_set(int d, cell v)
  {
    RTS_UNDERFLOW(RTSp - d);
    RTS_OVERFLOW(RTSp - d);
    vector_ref(RTS, RTSp - d) = v;
  }
  void rts_set_abs(int d, cell v)
  {
    RTS_UNDERFLOW(d);
    RTS_OVERFLOW(d);
    vector_ref(RTS, d) = v;
  }
```

42. Symbols. With the basics in place, the first thing to define is *symbol*s; they're not needed yet but everything becomes easier with them extant and they depend on nothing but themselves since they are themselves.

*symbol*s are never garbage collected. This was not a conscious decision it just doesn't seem like it matters. Instead, every *symbol* once created is immediately added to the *Symbol_Table list*. When a reference to a *symbol* is requested, the object in this *list* is returned.

Eventually this should implement a hash table but I'm not making one of those this morning.

Owing to the nasty c-to-perl-to-c route that I've taken, combined with plans for vector/byte storage, the storage backing symbols is going to be hairy without explanation (for now it's a mini duplicate of vector storage).

```
#define sym(s) symbol((s), 1)
#define symbol_length car
#define symbol_offset cdr
#define symbol_store(s) (SYMBOL + symbol_offset(s))
⟨ Global variables 7 ⟩ +≡
  cell Symbol_Table = NIL;
  char *SYMBOL = Λ;
  int Symbol_Free = 0;
  int Symbol_Poolsize = 0;
```

43. ⟨ Protected Globals 18 ⟩ +≡

```
  &Symbol_Table ,
```

44. ⟨ Function declarations 8 ⟩ +≡

```
  cell symbol(char *, int);
```

45. ⟨ API declarations 9 ⟩ +≡

```
  cell symbol(char *, int);
```

46.

```
void symbol_expand(void)
{
    char *new;

    new = realloc(SYMBOL, Symbol_Poolsize + HEAP_SEGMENT);
    if (new ≡ Λ) error (ERR_OOM, NIL);
    Symbol_Poolsize += HEAP_SEGMENT;
    SYMBOL = new;
}
```

47. A *symbol* can "steal" storage from SYMBOL which results in an `object` which can be mostly treated like a normal *symbol*, used to compare a potentially new *symbol* with those currently stored in *Symbol_Table*. This is the closest that *symbol*s get to being garbage collected.

```
cell symbol_steal(char *cstr)
{
    cell r;
    int len;

    len = strlen(cstr);
    while (Symbol_Free + len > Symbol_Poolsize) symbol_expand();
    r = atom(len, Symbol_Free, FORMAT_SYMBOL);
    memcpy(SYMBOL + Symbol_Free, cstr, len);      /* Symbol_Free is not incremented here */
    return r;
}
```

48. Temporary *symbol*s compare byte-by-byte with existing *symbol*s. This is not efficient at all.

```
boolean symbol_same_p(cell maybe, cell match)
{
  char *pmaybe, *pmatch;
  int i, len;

  len = symbol_length(match);
  if (symbol_length(maybe) ≠ len) return bfalse;
  pmaybe = symbol_store(maybe);
  pmatch = symbol_store(match);
  if (maybe ≡ match)      /* This shouldn't happen */
    return btrue;
  for (i = 0; i < len; i++) {
    if (pmaybe[i] ≠ pmatch[i]) return bfalse;
  }
  return btrue;
}
```

49.

```
void symbol_reify(cell s)
{
  Symbol_Free += symbol_length(s);
  Symbol_Table = cons(s, Symbol_Table);
}
cell symbol(char *cstr, int permanent_p)
{
  cell st, s;

  s = symbol_steal(cstr);
  st = Symbol_Table;
  while (¬null_p(st)) {
    if (symbol_same_p(s, car(st))) return car(st);
    st = cdr(st);
  }
  if (permanent_p) symbol_reify(s);
  return s;
}
```

50. Numbers. The only numbers supported by this early implementation of `LossLess` are signed integers that fit in a single **cell** (ie. 32-bit integers).

The 256 numbers closest to 0 (ie. from $-$#`80` to $+$#`7f`) are preallocated during initialisation. If you live in a parallel universe where the char type isn't 8 bits then adjust those numbers accordingly.

#define *fixint_p*(*p*) (*integer_p*(*p*) ∧ *null_p*(*int_next*(*p*)))
#define *int_value*(*p*) ((**int**)(*car*(*p*)))
#define *int_next* *cdr*

⟨ Global variables 7 ⟩ +≡
  cell *Small_Int*[`UCHAR_MAX` + 1];

51. Even though the *Small_Int* objects are about to be created, in order to create objects garbage collection will happen and assume that *Small_Int* has already been initialised and attempt to protect data which don't exist from collection. This is a silly solution but I'm leaving it alone until I have a better memory model.

⟨ Pre-initialise *Small_Int* 51 ⟩ ≡
  for (*i* = 0; *i* < 256; *i*++) *Small_Int*[*i*] = `NIL`;

This code is used in section 71.

52. ⟨ Global initialisation 3 ⟩ +≡
  for (*i* = `SCHAR_MIN`; *i* ≤ `SCHAR_MAX`; *i*++)
    Small_Int[(**unsigned char**) *i*] = *int_new_imp*(*i*, `NIL`);

53. As with *vectors*, *int_new* checks whether it should return an *object* from *Small_Int* or build a new one.

```
cell int_new_imp(int value, cell next)
{
  if (¬null_p(next)) error (ERR_UNIMPLEMENTED, NIL);
  return atom((cell) value, next, FORMAT_INTEGER);
}
cell int_new(int value)
{
  if (value ≥ SCHAR_MIN ∧ value ≤ SCHAR_MAX)
    return Small_Int[(unsigned char) value];
  return int_new_imp(value, NIL);
}
```

54. Pairs & Lists. Of course *pair*s—and so by definition *list*s—have already been implemented but so far only enough to implement core features. Here we define handlers for operations specifically on *list* `objects`.

First to count it's length a *list* is simply walked from head to tail. It is not considered an error if the *list* is improper (or not a *list* at all). To indicate this case the returned length is negated.

```
int list_length(cell l)
{
  int c = 0;
  if (null_p(l)) return 0;
  for ( ; pair_p(l); l = cdr(l)) c++;
  if (¬null_p(l)) c = −(c + 1);
  return c;
}
```

55. A *list* is either NIL or a pair with one restriction, that its *cdr* must itself be a *list*. The size of the *list* is also counted to avoid walking it twice but nothing uses that (yet?).

```
predicate list_p(cell o, predicate improper_p, cell *sum)
{
  int c = 0;
  if (null_p(o)) {
    if (sum ≠ Λ) *sum = int_new(0);
    return TRUE;
  }
  while (pair_p(o)) {
    o = cdr(o);
    c++;
  }
  if (sum ≠ Λ) *sum = int_new(c);
  if (null_p(o)) return TRUE;
  if (sum ≠ Λ) *sum = int_new(−(c + 1));
  return improper_p;
}
```

56. A proper *list* can be reversed simply into a new *list*.

```
#define ERR_IMPROPER_LIST "improper-list"
  cell list_reverse(cell l, cell *improper, cell *sum)
  {
    cell saved, r;
    int c;
    saved = l;
    c = 0;
    vms_push(NIL);
    while (¬null_p(l)) {
      if (¬pair_p(l)) {
        r = vms_pop();
        if (improper ≠ Λ) {
          *improper = l;
          if (sum ≠ Λ) *sum = c;
          return r;
        }
        else error (ERR_IMPROPER_LIST, saved);
      }
      vms_set(cons(car(l), vms_ref()));
      l = cdr(l);
      c++;
    }
    if (sum ≠ Λ) *sum = int_new(c);
    return vms_pop();
  }
```

57. Reversing a *list* in-place means maintaining a link to the previous *pair* (or NIL) and replacing each *pair*'s *cdr*. The new head *pair* is returned, or FALSE if the *list* turned out to be improper.

```
  cell list_reverse_m(cell l, boolean error_p)
  {
    cell m, t, saved;
    saved = l;
    m = NIL;
    while (¬null_p(l)) {
      if (¬pair_p(l)) {
        if (¬error_p)    /* TODO: repair? */
          return FALSE;
        error (ERR_IMPROPER_LIST, saved);
      }
      t = cdr(l);
      cdr(l) = m;
      m = l;
      l = t;
    }
    return m;
  }
```

58. Environments. In order to associate a value with a *symbol* (a variable) they are paired together in an *environment*.

Like an onion or an ogre[1], an *environment* has layers. The top layer is both the current layer and the current *environment*. The bottom layer is the root *environment Root*.

An *environment* is stored in an *atom* with the *car* pointing to the previous layer (or `NIL` in the root *environment*).

The *cdr* is a *list* of association *pair*s representing the variables in that layer. An association *pair* is a proper *list* with two items: an identifier, in this case a *symbol*, and a value.

environment-handling functions and macros are generally named "env".

```
#define ERR_BOUND "already-bound"
#define ERR_UNBOUND "unbound"
#define env_empty() atom(NIL, NIL, FORMAT_ENVIRONMENT)
#define env_extend(e) atom((e), NIL, FORMAT_ENVIRONMENT)
#define env_layer cdr
#define env_parent car
#define env_empty_p(e) (environment_p(e) ∧ null_p(car(e)) ∧ null_p(cdr(e)))
#define env_root_p(e) (environment_p(e) ∧ null_p(car(e)))
```

59. Searching through an *environment* starts at its top layer and walks along each *pair*. If it encounters a *pair* who's *symbol* matches, the value is returned. If not then the search repeats layer by layer until the *environment* is exhausted and `UNDEFINED` is returned.

env_search does not raise an error if a *symbol* isn't found. This means that `UNDEFINED` is the only value which cannot be stored in a variable as there is no way to distinguish its return from this function.

```
cell env_search(cell haystack, cell needle)
{
  cell n;
  for ( ; ¬null_p(haystack); haystack = env_parent(haystack))
    for (n = env_layer(haystack); ¬null_p(n); n = cdr(n))
      if (caar(n) ≡ needle) return cadar(n);
  return UNDEFINED;
}
```

60.
```
cell env_here(cell haystack, cell needle)
{
  cell n;
  for (n = env_layer(haystack); ¬null_p(n); n = cdr(n))
    if (caar(n) ≡ needle) return cadar(n);
  return UNDEFINED;
}
```

---

[1] Or a cake.

61. To set a variable's value the *environment*'s top layer is first searched to see if the *symbol* is already bound. An **error** is raised if the symbol is bound (when running on behalf of *define!*) or not bound (when running on behalf of *set!*).

```
void env_set(cell e, cell name, cell value, boolean new_p)
{
  cell ass, t;
  ass = cons(name, cons(value, NIL));
  if (new_p) {⟨Mutate if unbound 63⟩}
  else {⟨Mutate if bound 62⟩}
}
```

62. Updating an already-bound variable means removing the existing binding from the *environment* and inserting the new binding. During the walk over the layer t is one pair ahead of the pair being considered so that when *name* is found t's *cdr* can be changed, snipping the old binding out, so the first pair is checked specially.

```
⟨Mutate if bound 62⟩ ≡
  if (null_p(env_layer(e))) error(ERR_UNBOUND, name);
  if (caar(env_layer(e)) ≡ name) {
    env_layer(e) = cons(ass, cdr(env_layer(e)));
    return;
  }
  for (t = env_layer(e); ¬null_p(cdr(t)); t = cdr(t)) {
    if (caadr(t) ≡ name) {
      cdr(t) = cddr(t);
      env_layer(e) = cons(ass, env_layer(e));
      return;
    }
  }
  error(ERR_UNBOUND, name);
```

This code is used in section 61.

63. The case is simpler if the *name* must **not** be bound already as the new binding can be prepended to the layer after searching with no need for special cases.

```
⟨Mutate if unbound 63⟩ ≡
  for (t = env_layer(e); ¬null_p(t); t = cdr(t))
    if (caar(t) ≡ name) error(ERR_BOUND, name);
  env_layer(e) = cons(ass, env_layer(e));
```

This code is used in section 61.

64. Values are passed to functions on the stack. *env_lift_stack* moves these values from the stack into an *environment*.

```
cell env_lift_stack(cell e, int nargs, cell formals)
{
  cell p, name, value, ass;
  vms_push(env_extend(e));
  p = NIL;    /* prepare a new layer */
  vms_push(p);
  while (nargs--) {
    if (pair_p(formals)) {
      name = car(formals);
      formals = cdr(formals);
    }
    else { name = formals; }
    value = rts_pop(1);
    if (¬null_p(name)) {
      ass = cons(name, cons(value, NIL));
      vms_set((p = cons(ass, p)));
    }
  }
  vms_pop();
  cdr(vms_ref()) = p;    /* place the new layer in the extended environment */
  return vms_pop();
}
```

65. Closures & Compilers. Finally we have data structures to save run-time state: *closure*s. The way the compiler and virtual machine work to get *closure* objects built is described below—here is only a description of their backing stores.

LossLess has two types of *closure*, *applicative* and *operative*. They store the same data in identical containers; the difference is in how they're used.

The data required to define a *closure* are a program & the *environment* to run it in. A *closure* in LossLess also contains the formals given in the **lambda** or **vov** expression that was used to define it.

Program code in LossLess is stored as compiled bytecode in a *vector* with an instruction pointer indicating the entry point (0 is not implied). The *closure*s then look like this:

$$(APPLICATIVE\ \langle formals\rangle\ \langle environment\rangle\ \langle code\rangle\ \langle pointer\rangle)$$

$$(OPERATIVE\ \langle formals\rangle\ \langle environment\rangle\ \langle code\rangle\ \langle pointer\rangle)$$

However the *environment*, code and pointer are never referred to directly until the closure is unpicked by OP_APPLY/OP_APPLY_TAIL. Instead the objects effectively look like this:

$$(A|O\ \langle formals\rangle\ .\ \langle opaque\_closure\rangle)$$

```
#define applicative_closure cdr
#define applicative_formals car
#define applicative_new(f, e, p, i) closure_new_imp(FORMAT_APPLICATIVE, (f), (e), (p), (i))
#define operative_closure cdr
#define operative_formals car
#define operative_new(f, e, p, i) closure_new_imp(FORMAT_OPERATIVE, (f), (e), (p), (i))
  cell closure_new_imp(char ntag, cell formals, cell env, cell prog, cell ip)
  {
    cell r;
    r = cons(int_new(ip), NIL);
    r = cons(prog, r);
    r = cons(env, r);
    return atom(formals, r, ntag);
  }
```

66. Other than closures, and required in order to make them, the evaluator uses "*compiler*" objects that compile LossLess source code to VM bytecode. Each *compiler* is described in the structure *primitive*, containing the native function pointer to it.

```
#define compiler_cname(c) COMPILER[car(c)].name
#define compiler_fn(c) COMPILER[car(c)].fn
⟨ Type definitions 6 ⟩ +≡
  typedef void (*native)(cell, cell, boolean);
  typedef struct {
    char *name;
    native fn;
  } primitive;
```

67. The contents of COMPILER are populated by the C compiler of this source. During initialisation *Root* then becomes the root environment filled with an association *pair* for each one.

```
⟨ Global variables 7 ⟩ +≡
  primitive COMPILER[] = { ⟨ List of opcode primitives 284 ⟩, {Λ, Λ} } ;
```

68. Virtual Machine. This implementation of `LossLess` compiles user source code to an internal bytecode representation which is then executed sequentially by a virtual machine (VM).

Additionally to the myriad stacks already mentioned, the VM maintains (global!) state primarily in 6 registers. Two of these are simple flags (booleans) which indicate whether interpretation should continue.

1. *Running* is a flag raised (1) when the VM begins and lowered by user code to indicate that it should halt cleanly. This flag is checked on the beginning of each iteration of the VM's main loop.

2. *Interrupt* is normally lowered (0) and is raised in response to external events such as a unix signal. Long-running operations—especially those which could potentially run unbounded—check frequently for the state of this flag and abort and return immediately when it's raised.

The other four registers represent the computation.

3. *Acc* is the accumulator. Opcodes generally read and/or write this register to do their work. This is where the final result of computation will be found.

4. *Env* holds the current *environment*. Changing this is the key to implementing *closure*s.

5. *Prog* is the compiled bytecode of the currently running computation, a *vector* of VM opcodes with their in-line arguments.

6. *Ip* is the instruction pointer. This is an **int**, not a **cell** and must be boxed to be used outside of the VM.

Root and *Prog_Main* are also defined here which hold, respectively, the root *environment* and the virtual machine's starting program.

⟨ Global variables 7 ⟩ +≡
boolean *Interrupt* = 0;
boolean *Running* = 0;
cell *Acc* = `NIL`;
cell *Env* = `NIL`;
cell *Prog* = `NIL`;
cell *Prog_Main* = `NIL`;
cell *Root* = `NIL`;
int *Ip* = 0;

69. ⟨ Protected Globals 18 ⟩ +≡
&*Acc*, &*Env*, &*Prog*, &*Prog_Main*, &*Root* ,

70. The `LossLess` virtual machine is initialised by calling the code snippets built into the ⟨ Global initialisation 3 ⟩ section then constructing the the root *environment* in *Root*.

Initialisation is divided into two phases. The first in *vm_init* sets up emergency jump points (which should never be reached) for errors which occur during initialisation or before the second phase.

The second phase establishes a jump buffer in *Goto_Begin* to support run-time errors that were not handled. It resets VM state which will not have had a chance to recover normally due to the computation aborting early.

The error handler's jump buffer *Goto_Error* on the other hand is established by *interpret* and does *not* reset any VM state, but does return to the previous jump buffer if the handler fails.

```
#define vm_init() do
        {
          if (setjmp(Goto_Begin)) {
            Acc = sym("ABORT");
            return EXIT_FAILURE;
          }
          if (setjmp(Goto_Error)) {
            Acc = sym("ABORT");
            return EXIT_FAILURE;
          }
          vm_init_imp();
        }
        while (0)
#define vm_prepare() do
        {
          setjmp(Goto_Begin);
          vm_prepare_imp();
        }
        while (0)
#define vm_runtime() do
        {
          if (setjmp(Goto_Error)) {
            Ip = −1;     /* TODO: call the handler */
            if (Ip < 0) longjmp(Goto_Begin, 1);
          }
        }
        while (0)
```

⟨ API declarations 9 ⟩ +≡

```
  extern boolean Interrupt;
  extern cell Acc;
  void vm_init_imp(void);
  void vm_prepare_imp(void);
  void vm_reset(void);
```

71. **void** *vm_init_imp*(**void**)

```
{
  cell t;
  int i;
  primitive *n;
  ⟨ Pre-initialise Small_Int 51 ⟩
  ⟨ Global initialisation 3 ⟩
  Prog_Main = compile_main();
  i = 0;
  Root = atom(NIL, NIL, FORMAT_ENVIRONMENT);
  for (n = COMPILER + i; n→fn ≠ Λ; n = COMPILER + (++i)) {
    t = atom(i, NIL, FORMAT_COMPILER);
    t = cons(t, NIL);
    t = cons(sym(n→name), t);
    env_layer(Root) = cons(t, env_layer(Root));
  }
  Env = Root;
}
```

72. **void** *vm_prepare_imp*(**void**)

```
{
  Acc = Prog = NIL;
  Env = Root;
  rts_reset();
}
```

73. **void** *vm_reset*(**void**)

```
{
  Prog = Prog_Main;
  Running = Interrupt = Ip = 0;
}
```

74. Frames. The VM enters a *closure*—aka. calls a function—by appending a *frame* header to the stack. A *frame* consists of any work-in-progress items on the stack followed by a fixed-size header. A *frame*'s header captures the state of computation at the time it's created which is what lets another subroutine run and then return. The *frame* header contains 4 objects: ≪*Ip Prog Env Fp*≫.

Fp is a quasi-register which points into the stack to the current *frame*'s header. It's saved when entering a *frame* and its value set to that of the stack pointer *RTSp*. *RTSp* is restored to the saved value when returning from a *frame*.

```
#define FRAME_HEAD 4
#define frame_ip(f) rts_ref_abs((f) + 1)
#define frame_prog(f) rts_ref_abs((f) + 2)
#define frame_env(f) rts_ref_abs((f) + 3)
#define frame_fp(f) rts_ref_abs((f) + 4)
#define frame_set_ip(f, v) rts_set_abs((f) + 1, (v));
#define frame_set_prog(f, v) rts_set_abs((f) + 2, (v));
#define frame_set_env(f, v) rts_set_abs((f) + 3, (v));
#define frame_set_fp(f, v) rts_set_abs((f) + 4, (v));
```

⟨ Global variables 7 ⟩ +≡

```
  int Fp = −1;
```

75. Creating a *frame* is pushing the header items onto the stack. Entering it is changing the VM's registers that are now safe. This is done in two stages for some reason.

```
  void frame_push(int ipdelta)
  {
    rts_push(int_new(Ip + ipdelta));
    rts_push(Prog);
    rts_push(Env);
    rts_push(int_new(Fp));
  }
  void frame_enter(cell e, cell p, cell i)
  {
    Env = e;
    Prog = p;
    Ip = i;
    Fp = RTSp − FRAME_HEAD;
  }
```

76. Leaving a *frame* means restoring the registers that were saved in it by *frame_push* and then returning *RTSp* and *Fp* to their previous values; *Fp* from the header and *RTSp* as the current *Fp* minus the *frame* header in case there were previously any in-progress items on top of the stack.

```
  void frame_leave(void)
  {
    int prev;
    Ip = int_value(frame_ip(Fp));
    Prog = frame_prog(Fp);
    Env = frame_env(Fp);
    prev = int_value(frame_fp(Fp));
    rts_clear(FRAME_HEAD);
    Fp = prev;
  }
```

77. Tail Recursion. TODO

This is a straight copy of what I wrote in perl which hasn't been used there. Looks about right. Might work.

```
void frame_consume(void)
{
  int src, dst, i;
  src = Fp;
  dst = int_value(frame_fp(src));      /* Copy the parts of the old frame header that are needed */
  frame_set_prog(src, frame_prog(dst));
  frame_set_ip(src, frame_ip(dst));
  frame_set_fp(src, frame_fp(dst));      /* Move the active frame over the top of the previous one */
  for (i = 1; i <= FRAME_HEAD; i++)
    rts_set_abs(dst + i, rts_ref_abs(src + i));
  rts_clear(src - dst);
  Fp -= src - dst;
}
```

78. Interpreter. The workhorse of the virtual machine is *interpret*. After being reset with *vm_reset*, parsed (but not compiled) source code is put into *Acc* and the VM can be started by calling *interpret*.

```
⟨ API declarations 9 ⟩ +≡
  void interpret(void);
```

79.

```
#define ERR_INTERRUPTED "interrupted"
  void interpret(void)
  {
    int ins;
    cell tmp;     /* not saved in ROOTS */
    vm_runtime();
    Running = 1;
    while (Running ∧ ¬Interrupt) {
      ins = int_value(vector_ref(Prog, Ip));
      switch (ins) {
        ⟨ Opcode implementations 11 ⟩
#ifdef LL_TEST
        ⟨ Testing implementations 186 ⟩
#endif
      }
    }
    if (Interrupt) error (ERR_INTERRUPTED, NIL);
  }
```

80. I/O. Before embarking on the meat of the interpreter a final detour to describe routines to parse a string (or stream) of source code into s-expressions, and because it's useful to see what's being done routines to write them back again.

These routines use C's *stdio* for now to get a simple implementation finished.

81. Reader (or Parser). The s-expression reader is an ad-hoc LALR parser; a single byte is read to determine which type of form to parse. Bytes are then read one at a time to validate the syntax and create the appropriate `object`.

The reading routines call into themselves recursively (for which it cheats and relies on C's stack). To prevent it running out of control *Read_Level* records the recursion depth and *read_form* aborts if it exceeds `READER_MAX_DEPTH`.

The compiler's rather than the VM's stack is used for temporary storage so that error handling doesn't need to clean it up. This is safe provided the reader and compiler are never used simultaneously.

The parser often needs the byte that was used to determine which kind of form to parse (the one that was "looked ahead" at). *Putback* is a small buffer to contain this byte. In fact this buffer can hold *two* bytes to accomodate lisp's *unquote-splicing* operator ⟨⟨`,@`⟩⟩.

In order to perform tests of this primitive implementation the reader can be directed to "read" from a C-strings if *Read_Pointer* is set to a value other than Λ.

```
#define ERR_RECURSION "recursion"
#define ERR_UNEXPECTED "unexpected"
#define WARN_AMBIGUOUS_SYMBOL "ambiguous"
#define READER_MAX_DEPTH 1024 /* gotta pick something */
#define READ_SPECIAL −10
#define READ_DOT −10 /* ⟨⟨.⟩⟩ */
#define READ_CLOSE_BRACKET −11 /* ⟨⟨]⟩⟩ */
#define READ_CLOSE_PAREN −12 /* ⟨⟨)⟩⟩ */
#define SYNTAX_DOTTED "dotted" /* ⟨⟨.⟩⟩ */
#define SYNTAX_QUOTE "quote" /* ⟨⟨'⟩⟩ */
#define SYNTAX_QUASI "quasiquote" /* ⟨⟨`⟩⟩ */
#define SYNTAX_UNQUOTE "unquote" /* ⟨⟨,⟩⟩ */
#define SYNTAX_UNSPLICE "unquote-splicing" /* ⟨⟨,@⟩⟩ */
```

⟨ API declarations 9 ⟩ +≡
```
cell read_form(void);
```

82. ⟨ Global variables 7 ⟩ +≡
```
char Putback[2] = {'\0', '\0'};
int Read_Level = 0;
char *Read_Pointer = Λ;
cell Sym_ERR_UNEXPECTED = NIL;
cell Sym_SYNTAX_DOTTED = NIL;
cell Sym_SYNTAX_QUASI = NIL;
cell Sym_SYNTAX_QUOTE = NIL;
cell Sym_SYNTAX_UNQUOTE = NIL;
cell Sym_SYNTAX_UNSPLICE = NIL;
```

83. ⟨ Global initialisation 3 ⟩ +≡
```
Sym_ERR_UNEXPECTED = sym(ERR_UNEXPECTED);
Sym_SYNTAX_DOTTED = sym(SYNTAX_DOTTED);
Sym_SYNTAX_QUASI = sym(SYNTAX_QUASI);
Sym_SYNTAX_QUOTE = sym(SYNTAX_QUOTE);
Sym_SYNTAX_UNQUOTE = sym(SYNTAX_UNQUOTE);
Sym_SYNTAX_UNSPLICE = sym(SYNTAX_UNSPLICE);
```

84. ⟨ Function declarations 8 ⟩ +≡

```
cell read_symbol(void);
cell read_form(void);
cell read_list(cell);
cell read_number(void);
cell read_symbol(void);
void unread_byte(char);
```

85.

```
int read_byte(void)
{
  int r;
  if ((r = Putback[0]) ≠ '\0') {
    Putback[0] = Putback[1];
    Putback[1] = '\0';
    return r;
  }
  if (Read_Pointer ≠ Λ) {
    r = *Read_Pointer;
    if (r ≡ '\0') r = EOF;
    Read_Pointer++;
    return r;
  }
  return getchar();
}
void unread_byte(char c)
{
  Putback[1] = Putback[0];
  Putback[0] = c;
}
```

86. The internal test suite defined below needs to be able to evaluate code it supplies from hard-coded C-strings. The mechanism defined here to make this work is extremely brittle and not meant to be used by user code. Or for very long until it can be replaced by something less quonky.

```
cell read_cstring(char *src)
{
  cell r;
  Read_Pointer = src;
  r = read_form();
  Read_Pointer = Λ;
  return r;
}
```

87. Even this primitive parser should support primitive comments.

```
int useful_byte(void)
{
  int c;
  while (¬Interrupt) {
    c = read_byte();
    switch (c) {
    case '␣': case '\n': case '\r': case '\t': continue;
    case ';': c = read_byte();
      while (c ≠ '\n' ∧ ¬Interrupt) {      /* read up to but not beyond the next newline */
        c = read_byte();
        if (c ≡ EOF) return c;
      }
      break;    /* go around again */
    default: return c;    /* includes EOF (which ≠ END_OF_FILE) */
    }
  }
  return EOF;
}
```

88. The public entry point to the reader is *read_sexp*. This simply resets the reader's global state and calls *read_form*.

```
cell read_sexp(void)
{
  cts_clear();
  Read_Level = 0;
  Putback[0] = Putback[1] = '\0';
  return read_form();
}
```

89. *read_form* reads a single (in most cases) byte which it uses to determine which parser function to dispatch to. The parser function will then return a complete s-expression (or raise an error).

```
cell read_form(void)
{
  cell r;
  int c, n;
  if (Interrupt) return VOID;
  if (Read_Level > READER_MAX_DEPTH) error (ERR_RECURSION, NIL);
  c = useful_byte();
  switch (c) {⟨Reader forms 90⟩}
  error (ERR_UNEXPECTED, NIL);
}
```

90. Here are the different bytes which *read_form* can understand, starting with the non-byte value EOF which is an error if the reader is part-way through parsing an expression.

```
⟨Reader forms 90⟩ ≡
case EOF:
  if (¬Read_Level) return END_OF_FILE;
  else error (ERR_ARITY_SYNTAX, NIL);
```

See also sections 92, 93, 94, 95, and 96.

This code is used in section 89.

91. Lists and vectors are read in exactly the same way, differentiating by being told to expect the appropriate delimiter.

92. ⟨ Reader forms 90 ⟩ +≡

case '(': **return** *read_list*(READ_CLOSE_PAREN);
case '[': **return** *read_list*(READ_CLOSE_BRACKET);
case ')': **case** ']':
/* If *Read_Level* > 0 then *read_form* was called by *read_list*, otherwise *read_sexp* */
if ($\neg$*Read_Level*) **error** (ERR_ARITY_SYNTAX, NIL);
else return $c \equiv$ ')' ? READ_CLOSE_PAREN : READ_CLOSE_BRACKET;

93. A lone dot can only appear in a *list* and only before precisely one more expression. This is verified later by *read_list*.

⟨ Reader forms 90 ⟩ +≡

case '.':
if ($\neg$*Read_Level*) **error** (ERR_ARITY_SYNTAX, NIL);
$c =$ *useful_byte*( );
if ($c \equiv$ EOF) **error** (ERR_ARITY_SYNTAX, NIL);
unread_byte(c);
return READ_DOT;

94. Special forms and strings aren't supported yet.

⟨ Reader forms 90 ⟩ +≡

case '"': **case** '#': **case** '|': **error** (ERR_UNIMPLEMENTED, NIL);

95. In addition to the main syntactic characters, three other characters commonly have special meaning in lisps: ⟪'⟫, ⟪`⟫ and ⟪,⟫. ⟪,⟫ can also appear as ⟪,@⟫. Primarily these are for working with the macro expander.

In LossLess this syntax is unnecessary thanks to its first-class operatives but it's helpful so it's been retained. To differentiate between having parsed the syntactic form of these operators (eg. ⟪'foo⟫ or ⟪'(bar baz)⟫) and their symbolic form (eg. ⟪(quote . foo)⟫ or ⟪(quote bar baz)⟫) an otherwise ordinary *pair* with the operative's *symbol* in the *car* is created with the tag FORMAT_SYNTAX. These *syntax* objects are treated specially by the compiler and the writer.

⟨ Reader forms 90 ⟩ +≡
```
case '\'': case '`': n = useful_byte();
  if (n ≡ EOF) error (ERR_ARITY_SYNTAX, NIL);
  unread_byte(n);
  if (n ≡ ')' ∨ n ≡ ']') error (ERR_ARITY_SYNTAX, NIL);
  r = sym(c ≡ '`' ? SYNTAX_QUASI : SYNTAX_QUOTE);
  return atom(r, read_form(), FORMAT_SYNTAX);
case ',': c = read_byte();
  if (c ≡ EOF) error (ERR_ARITY_SYNTAX, NIL);
  if (c ≡ ')' ∨ c ≡ ']') error (ERR_ARITY_SYNTAX, NIL);
  if (c ≡ '@') {
    r = sym(SYNTAX_UNSPLICE);
    return atom(r, read_form(), FORMAT_SYNTAX);
  }
  else {
    unread_byte(c);
    r = sym(SYNTAX_UNQUOTE);
    return atom(r, read_form(), FORMAT_SYNTAX);
  }
```

96. Anything else is a number or a symbol (and this byte is part of it) provided it's ASCII.

⟨ Reader forms 90 ⟩ +≡
```
default:
  if (¬isprint(c)) error (ERR_ARITY_SYNTAX, NIL);
  unread_byte(c);
  if (isdigit(c)) return read_number();
  else return read_symbol();
```

97. *read_list* sequentially reads complete forms until it encounters the closing delimiter ⟪)⟫ or ⟪]⟫.

A pointer to the head of the *list* is saved and another pointer to its tail, *write*, is updated and used to insert the next object after it's been read, avoiding the need to reverse the *list* at the end.

```
cell read_list(cell delimiter)
{
  cell write, next, r;
  int count = 0;

  Read_Level++;
  write = cons(NIL, NIL);
  cts_push(write);
  while (1) {
    if (Interrupt) {
      cts_pop();
      Read_Level--;
      return VOID;
    }
    next = read_form();
    if (special_p(next)) {      /* These must return or terminate unless n is a 'real' special */
      ⟨ Handle terminable 'forms' during list construction 98 ⟩
    }
    count++;
    cdr(write) = cons(NIL, NIL);
    write = cdr(write);
    car(write) = next;
  }
  Read_Level--;
  r = cdr(cts_pop());
  if (delimiter ≡ READ_CLOSE_BRACKET)
    return vector_new_list(r, count);
  return count ? r : NIL;
}
```

98. *read_form* is expected to return an s-expression or raise an error if the input is invalid. In order to recognise when a closing parenthesis/bracket is read 3 'special' special forms are defined, READ_CLOSE_PAREN, READ_CLOSE_BRACKET and READ_DOT. Although these look and act like the other global constants they don't exist outside of the parser.

```
⟨ Handle terminable 'forms' during list construction 98 ⟩ ≡
  if (eof_p(next)) error (ERR_ARITY_SYNTAX, NIL);
  else if (next ≡ READ_CLOSE_BRACKET ∨ next ≡ READ_CLOSE_PAREN) {
    if (next ≠ delimiter) error (ERR_ARITY_SYNTAX, NIL);
    break;
  }
  else if (next ≡ READ_DOT) {⟨ Read dotted pair 99 ⟩}
```

This code is used in section 97.

99. Encountering a ⟪.⟫ requires more special care than it deserves, made worse because if a *list* is dotted, a *syntax* `object` is created instead of a normal s-expression so that the style in which it's written out will be in the same that was read in.

⟨Read dotted pair 99⟩ ≡
```
if (count < 1 ∨ delimiter ≠ READ_CLOSE_PAREN)
    /* There must be at least one item already and we must be parsing a list. */
  error (ERR_ARITY_SYNTAX, NIL);
next = read_form();
if (special_p(next) ∧ next ≤ READ_SPECIAL)
    /* Check that the next 'form' isn't one of ⟪.⟫, ⟪)⟫ or ⟪]⟫ */
  error (ERR_ARITY_SYNTAX, NIL);
cdr(write) = atom(sym(SYNTAX_DOTTED), next, FORMAT_SYNTAX);
next = read_form();
if (next ≠ delimiter)
    /* Check that the next 'form' is really the closing delimiter */
  error (ERR_ARITY_SYNTAX, NIL);
break;
```
This code is used in section 98.

100. If it's not a *list* or a *vector* (or a *string* (⟪"⟫), *specialform* (⟪#⟫), *raw symbol* (⟪|⟫) or *comment*) then the form being read is an *atom*. If the atom starts with a numeric digit then control proceeds directly to *read_number* otherwise *read_symbol* reads enough to determine whether the atom is a number beginning with ± or a valid or invalid symbol.

```
#define CHAR_TERMINATE "()[]\";␣\t\r\n"
#define terminable_p(c) strchr(CHAR_TERMINATE, (c))
  cell read_number(void)
  {
    char buf[12] = {0};   /* 2^32 is 10 digits, also ± and Λ */
    int c, i;
    long r;
    i = 0;
    while (1) {
      c = read_byte();
      if (c ≡ EOF)    /* TODO: If Read_Level is 0 is this an error? */
        error (ERR_ARITY_SYNTAX, NIL);
      if (i ≡ 0 ∧ (c ≡ '-' ∨ c ≡ '+')) buf[i++] = c;
      else if (isdigit(c)) buf[i++] = c;
      else if (¬terminable_p(c)) error (ERR_ARITY_SYNTAX, NIL);
      else {
        unread_byte(c);
        break;
      }
      if (i > 11) error (ERR_UNIMPLEMENTED, NIL);
    }
    r = atol(buf);
    if (r > INT_MAX ∨ r < INT_MIN) error (ERR_UNIMPLEMENTED, NIL);
    return int_new(r);
  }
```

101. Although LossLess specifices (read: would specify) that there are no restrictions on the value of a *symbol*'s label, memory permitting, an artificial limit is being placed on the length of *symbol*s of 16KB[1].

That said, there are no restrictions on the value of a *symbol*'s label, memory permitting. There are limits on what can be *parsed* as a *symbol* in source code. The limits on plain *symbol*s are primarily to avoid things that look vaguely like numbers to the human eye being parsed as *symbol*s when the programmer thinks they should be parsed as a number. This helps to avoid mistakes like '3..14159' and harder to spot human errors being silently ignored.

- A *symbol* must not begin with a numeric digit or a syntactic character (comments (⟪;⟫), whitespace and everything recognised by *read_form*).
- The syntactic characters ⟪(⟫, ⟪)⟫, ⟪[⟫, ⟪]⟫, ⟪;⟫ and ⟪"⟫ cannot appear anywhere in the *symbol*.

nb. This means that the following otherwise syntactic characters *are* permitted in a symbol provided they do not occupy the first byte: ⟪.⟫, ⟪,⟫, ⟪'⟫, ⟪`⟫, ⟪#⟫ and ⟪|⟫. You probably shouldn't do that lightly though.

- If the first character of a *symbol* is ⟪-⟫ or ⟪+⟫ then it cannot be followed a numeric digit. ++⟨*digit*⟩ is valid.
- A ⟪-⟫ character or ⟪+⟫ followed by a ⟪.⟫ is a valid if strange *symbol* but a warning should probably be emitted by the parser if it finds that.

```
#define CHUNK_SIZE #80
#define READSYM_EOF_P if (c ≡ EOF) error (ERR_ARITY_SYNTAX, NIL)
      /* TODO: If Read_Level is 0 is this an error? */
  cell read_symbol(void)
  {
    cell r;
    char *buf, *nbuf;
    int c, i, s;
    c = read_byte();
    READSYM_EOF_P;
    buf = malloc(CHUNK_SIZE);
    if (¬buf) error (ERR_OOM, NIL);
    s = CHUNK_SIZE;
    ⟨Read the first two bytes to check for a number 102⟩
    while (1) {⟨Read bytes until an invalid or terminating character 103⟩}
    buf[i] = '\0';    /* Λ-terminate the C-'string' */
    r = sym(buf);
    free(buf);
    return r;
  }
```

[1] 640KB was deemed to be far more than enough for anyone's needs.

102. Reading the first two bytes of a symbol is done specially to detect numbers beginning with ±. The first byte—which has already been read to check for EOF—is put into *buf* then if it matches ± the next byte is also read and also put into *buf*.

If that second byte is a digit then we're actually reading a number so put the bytes that were read so far into *Putback* and go to *read_number*, which will read them again. If the second byte is ⟪.⟫ then the *symbol* is valid but possibly a typo, so emit a warning and carry on.

⟨Read the first two bytes to check for a number 102⟩ ≡

```
buf[0] = c;
i = 1;
if (c ≡ '-' ∨ c ≡ '+') {
  c = read_byte();
  READSYM_EOF_P;
  buf[1] = c;
  i++;
  if (isdigit(buf[1])) {      /* This is a number! */
    unread_byte(buf[1]);
    unread_byte(buf[0]);
    free(buf);
    return read_number();
  }
  else if (buf[1] ≡ '.') warn(WARN_AMBIGUOUS_SYMBOL, NIL);
  else if (¬isprint(c)) error (ERR_ARITY_SYNTAX, NIL);
}
```

This code is used in section 101.

103. After the first two bytes we're definitely reading a *symbol* so anything goes except non-printable characters (which are an error) or syntactic terminators which indicate the end of the *symbol*.

⟨Read bytes until an invalid or terminating character 103⟩ ≡

```
c = read_byte();
READSYM_EOF_P;
if (terminable_p(c)) {
  unread_byte(c);
  break;
}
if (¬isprint(c)) error (ERR_ARITY_SYNTAX, NIL);
buf[i++] = c;
if (i ≡ s) {      /* Enlarge buf if it's now full (this will also allow the Λ-terminator to fit) */
  nbuf = realloc(buf, s *= 2);
  if (nbuf ≡ Λ) {
    free(buf);
    error (ERR_OOM, NIL);
  }
  buf = nbuf;
}
```

This code is used in section 101.

104. Writer. Although not an essential part of the language itself, the ability to display an s-expression to the user/programmer is obviously invaluable.

It is expected that this will (very!) shortly be changed to return a *string* representing the s-expression which can be passed on to an output routine but for the time being `LossLess` has no support for *string*s or output routines so the expression is written directly to *stdout*.

#define `WRITER_MAX_DEPTH` 1024 /* gotta pick something */

⟨ Function declarations 8 ⟩ +≡
void *write_form*(**cell**, **int**);

105. ⟨ API declarations 9 ⟩ +≡
void *write_form*(**cell**, **int**);

106. Opaque Objects. *applicative*s, *compiler*s and *operative*s don't have much to say.

```
boolean write_applicative(cell sexp, int depth__unused)
{
  if (¬applicative_p(sexp)) return bfalse;
  printf("#<applicative␣...>");
  return btrue;
}
boolean write_compiler(cell sexp, int depth__unused)
{
  if (¬compiler_p(sexp)) return bfalse;
  printf("#<compiler-%s>", compiler_cname(sexp));
  return btrue;
}
boolean write_operative(cell sexp, int depth__unused)
{
  if (¬operative_p(sexp)) return bfalse;
  printf("#<operative␣...>");
  return btrue;
}
```

107. As-Is Objects. *integer*s and *symbol*s print themselves.

```
boolean write_integer(cell sexp, int depth__unused)
{
  if (¬integer_p(sexp)) return bfalse;
  printf("%d", int_value(sexp));
  return btrue;
}
boolean write_symbol(cell sexp, int depth__unused)
{
  int i;
  if (¬symbol_p(sexp)) return bfalse;
  for (i = 0; i < symbol_length(sexp); i++) putchar(symbol_store(sexp)[i]);
  return btrue;
}
```

108. Secret Objects. The hidden *syntax* object prints its syntactic form and then itself.

```
boolean write_syntax(cell sexp, int depth)
{
  if (¬syntax_p(sexp)) return bfalse;
  else if (car(sexp) ≡ sym(SYNTAX_DOTTED)) printf(".␣");
  else if (car(sexp) ≡ sym(SYNTAX_QUASI)) printf("`");
  else if (car(sexp) ≡ sym(SYNTAX_QUOTE)) printf("'");
  else if (car(sexp) ≡ sym(SYNTAX_UNQUOTE)) printf(",");
  else if (car(sexp) ≡ sym(SYNTAX_UNSPLICE)) printf(",@");
  write_form(cdr(sexp), depth + 1);
  return btrue;
}
```

109. Environment Objects. An *environment* prints its own layer and then the layers above it.

```
boolean write_environment(cell sexp, int depth)
{
  if (¬environment_p(sexp)) return bfalse;
  printf("#<environment␣");
  write_form(env_layer(sexp), depth + 1);
  if (¬null_p(env_parent(sexp))) {
    printf("␣ON␣");
    write_form(env_parent(sexp), depth + 1);
    printf(">");
  }
  else printf("␣ROOT>");
  return btrue;
}
```

110. Sequential Objects. The routines for a *list* and *vector* are more or less the same – write each item in turn with whitespace after each form but the last, with the appropriate delimiters. *list*s also need to deal with being improper.

```
boolean write_list(cell sexp, int depth)
{
  if (¬pair_p(sexp)) return bfalse;
  printf("(");
  while (pair_p(sexp)) {
    write_form(car(sexp), depth + 1);
    if (pair_p(cdr(sexp)) ∨ syntax_p(cdr(sexp))) printf("␣");
    else if (¬null_p(cdr(sexp)) ∧ ¬pair_p(cdr(sexp)) ∧ ¬syntax_p(cdr(sexp))) printf("␣.␣");
    sexp = cdr(sexp);
  }
  if (¬null_p(sexp)) write_form(sexp, depth + 1);
  printf(")");
  return btrue;
}
boolean write_vector(cell sexp, int depth)
{
  int i;

  if (¬vector_p(sexp)) return bfalse;
  printf("[");
  for (i = 0; i < vector_length(sexp); i++) {
    write_form(vector_ref(sexp, i), depth + 1);
    if (i + 1 < vector_length(sexp)) printf("␣");
  }
  printf("]");
  return btrue;
}
```

111. *write_form* simply calls each writer in turn, stopping after the first one returning (C's) true.

```
void write_form(cell sexp, int depth)
{
  if (Interrupt) {
    if (¬depth) printf("...␣\n");
    return;
  }
  if (depth > WRITER_MAX_DEPTH) error (ERR_RECURSION, NIL);
  if (undefined_p(sexp)) printf("#><");   /* nothing should ever print this */
  else if (eof_p(sexp)) printf("#<eof>");
  else if (false_p(sexp)) printf("#f");
  else if (null_p(sexp)) printf("()");
  else if (true_p(sexp)) printf("#t");
  else if (void_p(sexp)) printf("#<>");
  else if (write_applicative(sexp, depth))   /* NOP */ ;
  else if (write_compiler(sexp, depth))   /* NOP */ ;
  else if (write_environment(sexp, depth))   /* NOP */ ;
  else if (write_integer(sexp, depth))   /* NOP */ ;
  else if (write_list(sexp, depth))   /* NOP */ ;
  else if (write_operative(sexp, depth))   /* NOP */ ;
  else if (write_symbol(sexp, depth))   /* NOP */ ;
  else if (write_syntax(sexp, depth))   /* NOP */ ;
  else if (write_vector(sexp, depth))   /* NOP */ ;
  else printf("#<wtf?>");   /* impossibru! */
}
```

112. Opcodes. With the core infrastucture out of the way we can finally turn our attention to the virtual machine implementation, or the implementation of the opcodes that the compiler will turn LossLess code into.

The opcodes that the virtual machine can perform must be declared before anything can be said about them. They take the form of an **enum**, this one unnamed. This list is sorted alphabetically for want of anything else.

Also defined here are *fetch* and *skip* which *opcode* implementations will use to obtain their argument(s) from *Prog* or advance *Ip*, respectively.

```
#define skip(d) Ip += (d)
#define fetch(d) vector_ref(Prog, Ip + (d))
```

⟨ Global constants 112 ⟩ ≡

```
  enum { OP_APPLY, OP_APPLY_TAIL, OP_CAR, OP_CDR,   /* 3 */
  OP_COMPILE, OP_CONS, OP_CYCLE, OP_ENVIRONMENT_P,   /* 7 */
  OP_ENV_MUTATE_M, OP_ENV_QUOTE, OP_ENV_ROOT, OP_ENV_SET_ROOT_M,   /* 11 */
  OP_ERROR, OP_HALT, OP_JUMP, OP_JUMP_FALSE,   /* 15 */
  OP_JUMP_TRUE, OP_LAMBDA, OP_LIST_P, OP_LIST_REVERSE,   /* 19 */
  OP_LIST_REVERSE_M, OP_LOOKUP, OP_NIL, OP_NOOP,   /* 23 */
  OP_NULL_P, OP_PAIR_P, OP_PEEK, OP_POP,   /* 27 */
  OP_PUSH, OP_QUOTE, OP_RETURN, OP_RUN,   /* 31 */
  OP_RUN_THERE, OP_SET_CAR_M, OP_SET_CDR_M, OP_SNOC,   /* 35 */
  OP_SWAP, OP_SYNTAX, OP_VOV ,
#ifdef LL_TEST
  ⟨ Testing opcodes 185 ⟩
#endif
  OPCODE_MAX } ;
```

See also section 113.

This code is used in section 2.

113. ⟨ Global constants 112 ⟩ +≡

```
#ifndef LL_TEST
  enum {   /* Ensure testing opcodes translate into undefined behaviour */
  OP_TEST_UNDEFINED_BEHAVIOUR = #f00f , ⟨ Testing opcodes 185 ⟩
  OPTEST_MAX } ;
#endif
```

114. Basic Flow Control. The most basic opcodes that the virtual machine needs are those which control whether to operate and where.

115. ⟨ Opcode implementations 11 ⟩ +≡

case `OP_HALT`:
  Running = 0;
  break;
case `OP_JUMP`:
  Ip = *int_value*(*fetch*(1));
  break;
case `OP_JUMP_FALSE`:
  if (*void_p*(*Acc*)) **error** (`ERR_UNEXPECTED`, `VOID`);
  else if (*false_p*(*Acc*)) *Ip* = *int_value*(*fetch*(1));
  else *skip*(2);
  break;
case `OP_JUMP_TRUE`:
  if (*void_p*(*Acc*)) **error** (`ERR_UNEXPECTED`, `VOID`);
  else if (*true_p*(*Acc*)) *Ip* = *int_value*(*fetch*(1));
  else *skip*(2);
  break;
case `OP_NOOP`:
  skip(1);
  break;

116. `OP_QUOTE` isn't really flow control but I don't know where else to put it.

⟨ Opcode implementations 11 ⟩ +≡

case `OP_QUOTE`:
  Acc = *fetch*(1);
  skip(2);
  break;

117. Pairs & Lists. `OP_CAR`, `OP_CDR`, `OP_NULL_P` and `OP_PAIR_P` are self explanatory.

⟨ Opcode implementations 11 ⟩ +≡
case `OP_CAR`:
  Acc = *car*(*Acc*);
  skip(1);
  break;
case `OP_CDR`:
  Acc = *cdr*(*Acc*);
  skip(1);
  break;
case `OP_NULL_P`:
  Acc = *null_p*(*Acc*) ? `TRUE` : `FALSE`;
  skip(1);
  break;
case `OP_PAIR_P`:
  Acc = *pair_p*(*Acc*) ? `TRUE` : `FALSE`;
  skip(1);
  break;

118. `OP_CONS` consumes one stack item (for the *cdr*) and puts the new pair in *Acc*. `OP_SNOC` does the opposite, pushing *Acc*'s *cdr* to the stack and leaving its *car* in *Acc*.

⟨ Opcode implementations 11 ⟩ +≡
case `OP_CONS`:
  Acc = *cons*(*Acc*, *rts_pop*(1));
  skip(1);
  break;
case `OP_SNOC`:
  rts_push(*cdr*(*Acc*));
  Acc = *car*(*Acc*);
  skip(1);
  break;

119. Cons cell mutators clear take an item from the stack and clear *Acc*.

⟨ Opcode implementations 11 ⟩ +≡
case `OP_SET_CAR_M`:
  car(*rts_pop*(1)) = *Acc*;
  Acc = `VOID`;
  skip(1);
  break;
case `OP_SET_CDR_M`:
  cdr(*rts_pop*(1)) = *Acc*;
  Acc = `VOID`;
  skip(1);
  break;

120. Other Objects. There is not much to say about these.

⟨ Opcode implementations 11 ⟩ +≡

case `OP_LIST_P`:
  if $(\neg \mathit{false\_p}(\mathit{fetch}(2)))$ **error** (`ERR_UNIMPLEMENTED`, `NIL`);
  $\mathit{Acc} = \mathit{list\_p}(\mathit{Acc}, \mathit{fetch}(1), \Lambda)$;
  $\mathit{skip}(3)$;
  break;
case `OP_LIST_REVERSE`:
  if $(\neg \mathit{true\_p}(\mathit{fetch}(1)) \lor \neg \mathit{false\_p}(\mathit{fetch}(2)))$ **error** (`ERR_UNIMPLEMENTED`, `NIL`);
  $\mathit{Acc} = \mathit{list\_reverse}(\mathit{Acc}, \Lambda, \Lambda)$;
  $\mathit{skip}(3)$;
  break;
case `OP_LIST_REVERSE_M`:
  $\mathit{Acc} = \mathit{list\_reverse\_m}(\mathit{Acc}, \mathit{btrue})$;
  $\mathit{skip}(1)$;
  break;
case `OP_SYNTAX`:
  $\mathit{Acc} = \mathit{atom}(\mathit{fetch}(1), \mathit{Acc},$ `FORMAT_SYNTAX`$)$;
  $\mathit{skip}(2)$;
  break;

121. Stack. `OP_PUSH` and `OP_POP` push the `object` in *Acc* onto the stack, or remove the top stack `object` into *Acc*, respectively. `OP_PEEK` is `OP_POP` without removing the item from the stack.

`OP_SWAP` swaps the `object` in *Acc* with the `object` on top of the stack.

`OP_CYCLE` swaps the top two stack items with each other.

`OP_NIL` pushes a `NIL` straight onto the stack without the need to quote it first.

⟨ Opcode implementations 11 ⟩ +≡

```
case OP_CYCLE:
  tmp = rts_ref(0);
  rts_set(0, rts_ref(1));
  rts_set(1, tmp);
  skip(1);
  break;
case OP_PEEK:
  Acc = rts_ref(0);
  skip(1);
  break;
case OP_POP:
  Acc = rts_pop(1);
  skip(1);
  break;
case OP_PUSH:
  rts_push(Acc);
  skip(1);
  break;
case OP_SWAP:
  tmp = Acc;
  Acc = rts_ref(0);
  rts_set(0, tmp);
  skip(1);
  break;
case OP_NIL:
  rts_push(NIL);
  skip(1);
  break;
```

122. Environments. Get or mutate *environment* objects. `OP_ENV_SET_ROOT_M` isn't used yet.

⟨ Opcode implementations 11 ⟩ +≡

```
case OP_ENVIRONMENT_P:
  Acc = environment_p(Acc) ? TRUE : FALSE;
  skip(1);
  break;
case OP_ENV_MUTATE_M:
  env_set(rts_pop(1), fetch(1), Acc, true_p(fetch(2)));
  Acc = VOID;
  skip(3);
  break;
case OP_ENV_QUOTE:
  Acc = Env;
  skip(1);
  break;
case OP_ENV_ROOT:
  Acc = Root;
  skip(1);
  break;
case OP_ENV_SET_ROOT_M:
  Root = Acc;    /* Root is 'lost'! */
  skip(1);
  break;
```

123. To look up the value of a variable in an *environment* we use `OP_LOOKUP` which calls the (recursive) *env_search*, interpreting the `UNDEFINED` it might return.

⟨ Opcode implementations 11 ⟩ +≡

```
case OP_LOOKUP:
  vms_push(Acc);
  Acc = env_search(Env, vms_ref());
  if (undefined_p(Acc)) {
    Acc = vms_pop();
    error (ERR_UNBOUND, Acc);
  }
  vms_pop();
  skip(1);
  break;
```

124. Closures. A *closure* is the combination of code to interpret and an *environment* to interpret it in. Usually a closure has arguments—making it useful—although in some cases a closure may work with global state or be idempotent.

In order to apply the arguments (if any) to the *closure* it must be entered by one of the opcodes `OP_APPLY` or `OP_APPLY_TAIL`. `OP_APPLY_TAIL` works identically to `OP_APPLY` and then consumes the stack frame which `OP_APPLY` created, allowing for *proper tail recursion* with further support from the compiler.

```
⟨ Opcode implementations 11 ⟩ +≡
case OP_APPLY:
  ⟨ Enter a closure 125 ⟩
  break;
case OP_APPLY_TAIL:
  ⟨ Enter a closure 125 ⟩
  frame_consume( );
  break;
case OP_RETURN:
  frame_leave( );
  break;
```

125. Whether in tail position or not, entering a *closure* is the same.

```
⟨ Enter a closure 125 ⟩ ≡
  {
    cell e, i, p;
    tmp = fetch(2);
    e = env_lift_stack(cadr(tmp), int_value(fetch(1)), car(tmp));
    p = caddr(tmp);
    i = int_value(cadddr(tmp));
    frame_push(3);
    frame_enter(e, p, i);
  }
```

This code is used in section 124.

126. Creating a closure in the first place follows an identical procedure whether it's an applicative or an operative but creates a different type of `object` in each case.

```
⟨ Opcode implementations 11 ⟩ +≡
case OP_LAMBDA:    /* The applicative */
  Acc = applicative_new(rts_pop(1), Env, Prog, int_value(fetch(1)));
  skip(2);
  break;
case OP_VOV:    /* The operative */
  Acc = operative_new(rts_pop(1), Env, Prog, int_value(fetch(1)));
  skip(2);
  break;
```

127. Compiler. The compiler needs to instruct the interpreter to compile more code and then run it, so these *opcodes* do that. `OP_COMPILE` compiles an s-expression into `LossLess` bytecode.

128. ⟨ Opcode implementations 11 ⟩ +≡

case `OP_COMPILE`:
  Acc = *compile*(*Acc*);
  skip(1);
  break;

129. `OP_RUN` interprets the bytecode in *Acc* in the current *environment*; the VM's live state is saved into a new stack *frame* then that *frame* is entered by executing the bytecode in *Acc*, starting at instruction 0.

⟨ Opcode implementations 11 ⟩ +≡

case `OP_RUN`:
  frame_push(1);
  frame_enter(*Env*, *Acc*, 0);
  break;

130. `OP_RUN_THERE` is like `OP_RUN` except that the *environment* to interpret the bytecode in is taken from the stack rather than staying in the active *environment*.

⟨ Opcode implementations 11 ⟩ +≡

case `OP_RUN_THERE`:
  vms_push(*rts_pop*(1));
  frame_push(1);
  frame_enter(*vms_pop*( ), *Acc*, 0);
  break;

131. Compiler. Speaking of the compiler, we can now turn our attention to writing it. The compiler is not advanced in any way but it is a little unusual. Due to the nature of first-class operatives, how to compile any expression can't be known until the combinator has been evaluated (read: compiled and then interpreted) in order to distinguish an applicative from an operative so that it knows whether to evaluate the arguments in the expression. I don't know if this qualifies it for a *Just-In-Time* compiler; I think *Finally-Able-To* is more suitable.

The compiler uses a small set of C macros which grow and fill *Compilation*—a *vector* holding the compilation in-progress.

```
#define ERR_COMPILE_DIRTY "compiler"
#define ERR_UNCOMBINABLE "uncombinable"
#define COMPILATION_SEGMENT #80
#define emitop(o) emit(int_new(o))
#define emitq(o) do { emitop(OP_QUOTE); emit(o); }
        while (0)    /* C... */
#define patch(i, v) (vector_ref(Compilation, (i)) = (v))
#define undot(p) ((syntax_p(p) ∧ car(p) ≡ Sym_SYNTAX_DOTTED) ? cdr(p) : (p))
```

⟨ Global variables 7 ⟩ +≡

```
int Here = 0;
cell Compilation = NIL;
```

132. ⟨ Function declarations 8 ⟩ +≡

```
cell compile(cell);
cell compile_main(void);
void compile_car(cell, cell, boolean);
void compile_cdr(cell, cell, boolean);
void compile_conditional(cell, cell, boolean);
void compile_cons(cell, cell, boolean);
void compile_define_m(cell, cell, boolean);
void compile_env_current(cell, cell, boolean);
void compile_env_root(cell, cell, boolean);
void compile_error(cell, cell, boolean);
void compile_eval(cell, cell, boolean);
void compile_expression(cell, boolean);
void compile_lambda(cell, cell, boolean);
void compile_null_p(cell, cell, boolean);
void compile_pair_p(cell, cell, boolean);
void compile_quasiquote(cell, cell, boolean);
void compile_quote(cell, cell, boolean);
void compile_set_car_m(cell, cell, boolean);
void compile_set_cdr_m(cell, cell, boolean);
void compile_set_m(cell, cell, boolean);
void compile_vov(cell, cell, boolean);
```

133. ⟨ Protected Globals 18 ⟩ +≡

```
&Compilation ,
```

134. **void** *emit*(**cell** *bc*)

```
{
  int l;
  l = vector_length(Compilation);
  if (Here ≥ l)
    Compilation = vector_sub(Compilation, 0, l,
        0, l + COMPILATION_SEGMENT,
        OP_HALT);
  vector_ref(Compilation, Here++) = bc;
}
```

135. While compiling it frequently occurs that the value to emit isn't known at the time it's being emitted. The most common and obvious example of this is a forward jump who's address must immediately follow the opcode but the address won't be known until more compilation has been performed.

To make this work *comefrom* emits a `NIL` as a placeholder and returns its offset, which can later be passed in the first argument of *patch* to replace the `NIL` with the desired address etc.

```
int comefrom(void)
{
  emit(NIL);
  return Here − 1;
}
```

136. Compilation begins by preparing *Compilation* and `CTS` then recursively walks the tree in *source* dispatching to individual compilation routines to emit the appropriate bytecode.

```
cell compile(cell source)
{
  cell r;
  vms_push(source);
  Compilation = vector_new(COMPILATION_SEGMENT, int_new(OP_HALT));
  Here = 0;
  cts_reset();
  compile_expression(source, 1);
  emitop(OP_RETURN);
  r = vector_sub(Compilation, 0, Here, 0, Here, VOID);
  Compilation = NIL;
  vms_clear();
  if (¬null_p(CTS)) error (ERR_COMPILE_DIRTY, source);
  return r;
}
```

137. *compile_main* is used during initialisation to build the bytecode ≪`OP_COMPILE OP_RUN OP_HALT`≫ which is the program installed initially into the virtual machine.

```
cell compile_main(void)
{
  cell r;
  r = vector_new_imp(3, 0, 0);
  vector_ref(r, 0) = int_new(OP_COMPILE);
  vector_ref(r, 1) = int_new(OP_RUN);
  vector_ref(r, 2) = int_new(OP_HALT);
  return r;
}
```

138. The first job of the compiler is to figure out what type of expression it's compiling, chiefly whether it's a *list* to combine or an *atom* which is itself.

```
void compile_expression(cell sexp, int tail_p)
{
  if (¬pair_p(sexp) ∧ ¬syntax_p(sexp)) {⟨Compile an atom 139⟩}
  else {⟨Compile a combiner 140⟩}
}
```

139. The only *atom* which doesn't evaluate to itself is a *symbol*. A *symbol* being evaluated references a variable which must be looked up in the active environment.

```
⟨Compile an atom 139⟩ ≡
  if (symbol_p(sexp)) {
    emitq(sexp);
    emitop(OP_LOOKUP);
  }
  else { emitq(sexp); }
```

This code is used in section 138.

140. Combining a *list* requires more work. This is also where operatives obtain the property of being first-class objects by delaying compilation of all but the first expression in the *list* until after that compiled bytecode has been interpreted.

```
⟨Compile a combiner 140⟩ ≡
  cell args, combiner;

  combiner = car(sexp);
  args = undot(cdr(sexp));
  ⟨Search Root for syntactic combiners 141⟩
  if (compiler_p(combiner)) {⟨Compile native combiner 142⟩}
  else if (applicative_p(combiner)) {⟨Compile applicative combiner 150⟩}
  else if (operative_p(combiner)) {⟨Compile operative combiner 159⟩}
  else if (symbol_p(combiner) ∨ pair_p(combiner)) {⟨Compile unknown combiner 143⟩}
  else { error (ERR_UNCOMBINABLE, combiner); }
```

This code is used in section 138.

141. If the combiner (*sexp*'s *car*) is a *syntax* object then it represents the result of parsing (for example) ⟪`'(expression)`⟫ into ⟪`(quote expression)`⟫ and it must always mean the *real quote* operator, so *syntax* combiners are always looked for directly (and only) in *Root*.

```
⟨Search Root for syntactic combiners 141⟩ ≡
  if (syntax_p(sexp)) {
    cell c;

    c = env_search(Root, combiner);
    if (undefined_p(c)) error (ERR_UNBOUND, combiner);    /* should never happen */
    combiner = c;
  }
```

This code is used in section 140.

142. A native compiler is simple; look up its address in `COMPILER` and go there. The individual native compilers are defined below.

```
⟨Compile native combiner 142⟩ ≡
  compiler_fn(combiner)(combiner, args, tail_p);
```

This code is used in section 140.

143. If the compiler doesn't know whether *combiner* is applicative or operative then that must be determined before *args* can be considered.

⟨ Compile unknown combiner 143 ⟩ ≡

```
emitq(args);
emitop(OP_PUSH);  /* save args onto the stack */
compile_expression(combiner, 0);  /* evaluate the combiner, leaving it in Acc */
emitop(OP_CONS);  /* rebuild sexp with the evaluated combiner */
emitop(OP_COMPILE);  /* continue compiling sexp */
emitop(OP_RUN);  /* run that code in the same environment */
```

This code is used in section 140.

144. Function Bodies. Nearly everything has arguments to process and it's nearly always done in the same way. *arity* and *arity_next* work in concert to help the compiler implementations check how many arguments there are (but not their value or type) and raise any errors encountered.

arity pushes the minimum required arguments onto the compiler stack (in reverse) and returns a pointer to the rest of the argument list.

```
#define ERR_ARITY_EXTRA "extra"
#define ERR_ARITY_MISSING "missing"
#define ERR_ARITY_SYNTAX "syntax"
#define arity_error(e, c, a) error ((e), cons((c), (a)))
  cell arity(cell op, cell args, int min, boolean more_p)
  {
    cell a = args;
    int i = 0;
    for ( ; i < min; i++) {
      if (null_p(a)) {
        if (compiler_p(op) ∨ operative_p(op)) arity_error(ERR_ARITY_SYNTAX, op, args);
        else arity_error(ERR_ARITY_MISSING, op, args);
      }
      if (¬pair_p(a)) arity_error(ERR_ARITY_SYNTAX, op, args);
      cts_push(car(a));
      a = cdr(a);
    }
    if (min ∧ ¬more_p ∧ ¬null_p(a)) {
      if (pair_p(a)) arity_error(ERR_ARITY_EXTRA, op, args);
      else arity_error(ERR_ARITY_SYNTAX, op, args);
    }
    return a;
  }
```

145. *arity_next*, given the remainder of the arguments that were returned from *arity*, checks whether another one is present and whether it's allowed to be, then returns a value suitable for another call to *arity_next*.

```
  cell arity_next(cell op, cell args, cell more, boolean required_p, boolean last_p)
  {
    if (null_p(more)) {
      if (required_p) arity_error(ERR_ARITY_MISSING, op, args);
      else {
        cts_push(UNDEFINED);
        return NIL;
      }
    }
    else if (¬pair_p(more))
      arity_error(ERR_ARITY_SYNTAX, op, args);
    else if (last_p ∧ ¬null_p(cdr(more))) {
      if (operative_p(op) ∧ pair_p(cdr(more))) arity_error(ERR_ARITY_EXTRA, op, args);
      else arity_error(ERR_ARITY_SYNTAX, op, args);
    }
    cts_push(car(more));
    return cdr(more);
  }
```

146. *closure* bodies, and the contents of a *begin* expression, are compiled by simply walking the list and recursing into *compile_expression* for everything on it. When compiling the last item in the list the *tail_p* flag is raised so that the expression can use `OP_APPLY_TAIL` if appropriate, making tail recursion proper.

```
void compile_list(cell op, cell sexp, boolean tail_p)
{
  boolean t;
  cell body, next, this;
  body = undot(sexp);
  t = null_p(body);
  if (t) {
    emitq(VOID);
    return;
  }
  while (¬t) {
    if (¬pair_p(body)) arity_error(ERR_ARITY_SYNTAX, op, sexp);
    this = car(body);
    next = undot(cdr(body));
    t = null_p(next);
    compile_expression(this, t ∧ tail_p);
    body = next;
  }
}
```

147. Closures (Applicatives & Operatives). The first thing to understand is that at their core *applicatives* and *operatives* work in largely the same way and have the same internal representation:

• The static *environment* which will expand into a local *environment* when entering the *closure*. This is where the variables that were "closed over" are stored.

• The program which the *closure* will perform, as compiled bytecode and a starting instruction pointer.

• A list of formals naming any arguments which will be passed to the *closure*, so that they can be put into the newly-extended *environment*.

Entering a *closure* means extracting these saved values and restoring them to the virtual machine's registers, *Env*, *Prog* & *Ip*.

A *closure* can (usually does) have arguments and it's how they're handled that differentiates an *applicative* from an *operative*.

148. The main type of *closure* everyone is familiar with already even if they don't know it is a function[1] or *applicative*.

An *applicative* is created in response to evaluating a **lambda** expression. The bytecode which does this evaluating is created by *compile_lambda*.

```
void compile_lambda(cell op, cell args, boolean tail_p)
{
  cell body, in, formals, f;
  int begin_address, comefrom_end;
  body = arity(op, args, 1, 1);
  body = undot(body);
  formals = cts_pop();
  formals = undot(formals);
  if (¬symbol_p(formals)) {⟨Process lambda formals 149⟩}
  emitq(formals);    /* push formals onto the stack */
  emitop(OP_PUSH);
  emitop(OP_LAMBDA);    /* create the applicative */
  begin_address = comefrom();    /* start address; argument to OP_LAMBDA */
  emitop(OP_JUMP);    /* jump over the compiled closure body */
  comefrom_end = comefrom();
  patch(begin_address, int_new(Here));
  compile_list(op, body, tail_p);    /* compile the code that entering the closure will interpret */
  emitop(OP_RETURN);    /* returns from the closure at run-time */
  patch(comefrom_end, int_new(Here));
}
```

[1] The word "function" is horribly misused everywhere and this trend will continue without my getting in its way.

149. If the *formals* given in the **lambda** expression are not in fact a single *symbol* then it must be a list of *symbol*s which is verified here. At the same time if the list is a dotted pair then the *syntax* wrapper is removed.

⟨ Process lambda formals 149 ⟩ ≡
```
cts_push(f = cons(NIL, NIL));
in = formals;
while (pair_p(in)) {
  if (¬symbol_p(car(in)) ∧ ¬null_p(car(in))) arity_error(ERR_ARITY_SYNTAX, op, args);
  cdr(f) = cons(car(in), NIL);
  f = cdr(f);
  in = undot(cdr(in));
}
if (¬null_p(in)) {
  if (¬symbol_p(in) ∧ ¬null_p(in)) arity_error(ERR_ARITY_SYNTAX, op, args);
  cdr(f) = in;
}
formals = cdr(cts_pop());
```
This code is used in section 148.

150. To enter this *closure* at run-time—aka. to call the function returned by **lambda**—the arguments it's called with must be evaluated (after being arity checked) then OP_APPLY or OP_APPLY_TAIL enters the *closure*, consuming a stack *frame* in the latter case.

The arguments and the formals saved in the *applicative* are walked together and saved in *direct*. If the formals list ends in a dotted *pair* then the remainder of the arguments are saved in *collect*.

When *collect* and *direct* have been prepared, being a copy of the unevaluated arguments in reverse order, they are walked again to emit the opcodes which will evaluate each argument and put the results onto the stack.

⟨ Compile applicative combiner 150 ⟩ ≡
```
cell collect, direct, formals, a;
int nargs = 0;

formals = applicative_formals(combiner);
cts_push(direct = NIL);
a = undot(args);
⟨ Look for required arguments 151 ⟩
⟨ Look for optional arguments 152 ⟩
if (pair_p(a)) arity_error(ERR_ARITY_EXTRA, combiner, args);
else if (¬null_p(a)) arity_error(ERR_ARITY_SYNTAX, combiner, args);
⟨ Evaluate optional arguments into a list 154 ⟩
⟨ Evaluate required arguments onto the stack 153 ⟩
cts_clear();
emitop(tail_p ? OP_APPLY_TAIL : OP_APPLY);
emit(int_new(nargs));
emit(combiner);
```
This code is used in section 140.

151. It's a syntax error if the arguments are not a proper list, otherwise there is nothing much to say about this.

⟨ Look for required arguments 151 ⟩ ≡
```
while (pair_p(formals)) {
  if (¬pair_p(a)) {
    if (null_p(a)) arity_error(ERR_ARITY_SYNTAX, combiner, args);
    else arity_error(ERR_ARITY_SYNTAX, combiner, args);
  }
  direct = cons(car(a), direct);
  cts_set(direct);
  a = undot(cdr(a));
  formals = cdr(formals);
  nargs++;
}
```
This code is used in section 150.

152. If the *applicative* formals indicate that it can be called with a varying number of arguments then that counts as one more argument which will be a list of whatever arguments remain.

⟨ Look for optional arguments 152 ⟩ ≡
```
if (symbol_p(formals)) {
  nargs++;
  cts_push(collect = NIL);
  while (pair_p(a)) {
    collect = cons(car(a), collect);
    cts_set(collect);
    a = undot(cdr(a));
  }
}
```
This code is used in section 150.

153. To perform the evaluation, each argument in the (now reversed) list *direct* is compiled followed by an OP_PUSH to save the result on the stack.

⟨ Evaluate required arguments onto the stack 153 ⟩ ≡
```
while (¬null_p(direct)) {
  compile_expression(car(direct), 0);
  emitop(OP_PUSH);
  direct = cdr(direct);
}
```
This code is used in section 150.

154. If the *applicative* expects a varying number of arguments then the (also reversed) list in *collect* is compiled in the same way but before `OP_PUSH`, `OP_CONS` removes the growing list from the stack and prepends the new result to it and it's this *list* which is pushed.

⟨ Evaluate optional arguments into a *list* 154 ⟩ ≡
```
if (symbol_p(formals)) {
  emitop(OP_NIL);
  while (¬null_p(collect)) {
    compile_expression(car(collect), 0);
    emitop(OP_CONS);
    emitop(OP_PUSH);
    collect = cdr(collect);
  }
  cts_clear();
}
```
This code is used in section 150.

155. Analogous to *compile_lambda* for *applicative*s is *compile_vov* for *operative*s. An *operative closure* is a simpler than an *applicative* because the arguments are not evaluated. Instead *compile_vov* needs to handle **vov**'s very different way of specifying its formals.

Resembling *let* rather than **lambda**, **vov**'s formals specify what run-time detail the *operative* needs: The unevaluated arguments, the active environment and/or (unimplemented) a continuation delimiter. To do this each entry in the formals list is an association pair with the *symbol*ic name for that detail associated with another symbol specifying what: *vov/arguments*, *vov/environment* or *vov/continuation*. Because no-one wants RSI these have the abbreviations *vov/args*, *vov/env* and *vov/cont*.

⟨ Global variables 7 ⟩ +≡
```
cell Sym_vov_args = UNDEFINED;
cell Sym_vov_args_long = UNDEFINED;
cell Sym_vov_cont = UNDEFINED;
cell Sym_vov_cont_long = UNDEFINED;
cell Sym_vov_env = UNDEFINED;
cell Sym_vov_env_long = UNDEFINED;
```

156. ⟨ Global initialisation 3 ⟩ +≡
```
Sym_vov_args = sym("vov/args");
Sym_vov_args_long = sym("vov/arguments");
Sym_vov_cont = sym("vov/cont");
Sym_vov_cont_long = sym("vov/continuation");
Sym_vov_env = sym("vov/env");
Sym_vov_env_long = sym("vov/environment");
```

157. **void** *compile_vov*(**cell** *op*, **cell** *args*, **boolean** *tail_p*)

```
{
  cell body, formals;
  int begin_address, comefrom_end;
  cell a = NIL;
  cell c = NIL;
  cell e = NIL;
  body = arity(op, args, 1, 1);
  body = undot(body);
  formals = cts_pop();
  formals = undot(formals);
  ⟨Scan operative informals 158⟩
  emitop(OP_NIL);    /* push formals onto the stack */
  emitq(c); emitop(OP_CONS); emitop(OP_PUSH);
  emitq(e); emitop(OP_CONS); emitop(OP_PUSH);
  emitq(a); emitop(OP_CONS); emitop(OP_PUSH);
  emitop(OP_VOV);    /* create the operative */
    /* The rest of compile_vov is identical to compile_lambda: */
  begin_address = comefrom();    /* start address; argument to opcode */
  emitop(OP_JUMP);    /* jump over the compiled closure body */
  comefrom_end = comefrom();
  patch(begin_address, int_new(Here));
  compile_list(op, body, tail_p);    /* compile the code that entering the closure will interpret */
  emitop(OP_RETURN);    /* return from the run-time closure */
  patch(comefrom_end, int_new(Here));    /* finish building the closure */
}
```

158. To scan the "informals" three variables, *a*, *c* and *e* are prepared with NIL representing the *symbol* for the arguments, *continuation* and *environment* respectively. Each "informal" is checked in turn using *arity* and the appropriate placeholder's NIL replaced with the *symbol*.

```
⟨Scan operative informals 158⟩ ≡
  cell r, s;
  if (¬pair_p(formals)) arity_error(ERR_ARITY_SYNTAX, op, args);
#define CHECK_AND_ASSIGN(v)
  {
    if (¬null_p(v)) arity_error(ERR_ARITY_SYNTAX, op, args);
    (v) = s;
  }
  while (pair_p(formals)) {
    arity(op, car(formals), 2, 0);
    r = cts_pop();
    s = cts_pop();
    if (¬symbol_p(s)) arity_error(ERR_ARITY_SYNTAX, op, args);
    else if (r ≡ Sym_vov_args ∨ r ≡ Sym_vov_args_long) CHECK_AND_ASSIGN(a)
    else if (r ≡ Sym_vov_env ∨ r ≡ Sym_vov_env_long) CHECK_AND_ASSIGN(e)
    else if (r ≡ Sym_vov_cont ∨ r ≡ Sym_vov_cont_long) CHECK_AND_ASSIGN(c)
    formals = cdr(formals);
  }
  if (¬null_p(formals)) arity_error(ERR_ARITY_SYNTAX, op, args);
```

This code is used in section 157.

159. Entering an *operative* involves pushing the 3 desired run-time properties, or `NIL`, onto the stack as though arguments to an *applicative closure* (remember that the unevaluated run-time arguments of the *closure* are potentially one of those run-time properties).

⟨ Compile operative combiner 159 ⟩ ≡

```
cell a, c, e, f;
f = operative_formals(combiner);
a = ¬null_p(car(f)); f = cdr(f);
e = ¬null_p(car(f)); f = cdr(f);
c = ¬null_p(car(f)); f = cdr(f);
if (c) error (ERR_UNIMPLEMENTED, NIL);
else emitop(OP_NIL);
if (e) {
  emitop(OP_ENV_QUOTE);
  emitop(OP_PUSH);
}
else emitop(OP_NIL);
if (a) {
  emitq(args);
  emitop(OP_PUSH);
}
else emitop(OP_NIL);
emitop(tail_p ? OP_APPLY_TAIL : OP_APPLY);
emit(int_new(3));
emit(combiner);
```

This code is used in section 140.

160. Conditionals (if). Although you could define a whole language with just **lambda** and **vov**[1] that way lies Church Numerals and other madness, so we will define the basic conditional, **if**.

```
void compile_conditional(cell op, cell args, boolean tail_p)
{
  cell alternate, condition, consequent, more;
  int jump_false, jump_true;
  more = arity(op, args, 2, 1);
  arity_next(op, args, more, 0, 1);
  alternate = cts_pop();
  consequent = cts_pop();
  condition = cts_pop();
  compile_expression(condition, 0);
  emitop(OP_JUMP_FALSE);
  jump_false = comefrom();
  compile_expression(consequent, tail_p);
  emitop(OP_JUMP);
  jump_true = comefrom();
  patch(jump_false, int_new(Here));
  if (undefined_p(alternate)) emitq(VOID);
  else compile_expression(alternate, tail_p);
  patch(jump_true, int_new(Here));
}
```

[1] In fact I think conditionals can be achieved in both somehow, so you only need one.

161. Run-time Evaluation (eval). **eval** must evaluate its 1 or 2 arguments in the current environment, and then enter the environment described by the second to execute the program in the first.

```
void compile_eval(cell op, cell args, boolean tail_p__unused)
{
  cell more, sexp, eenv;
  int goto_env_p;
  more = arity(op, args, 1, 1);
  sexp = cts_pop();
  arity_next(op, args, more, 0, 1);
  eenv = cts_pop();
  if (undefined_p(eenv)) {
    emitop(OP_ENV_QUOTE);
    emitop(OP_PUSH);
  }
  else {
    compile_expression(eenv, 0);
    emitop(OP_PUSH);
    emitop(OP_ENVIRONMENT_P);
    emitop(OP_JUMP_TRUE);
    goto_env_p = comefrom();
    emitq(Sym_ERR_UNEXPECTED);
    emitop(OP_ERROR);
    patch(goto_env_p, int_new(Here));
  }
  compile_expression(sexp, 0);
  emitop(OP_COMPILE);
  emitop(OP_RUN_THERE);
}
```

162. Run-time Errors. **error** expects a symbol an the first position and an optional form to evaluate in the second.

```
void compile_error(cell op, cell args, boolean tail_p __unused)
{
  cell id, more, value;
  more = arity(op, args, 1, 1);
  arity_next(op, args, more, 0, 1);
  value = cts_pop();
  id = cts_pop();
  if (¬symbol_p(id)) arity_error(ERR_ARITY_SYNTAX, op, args);
  if (undefined_p(value)) emitq(NIL);
  else compile_expression(value, 0);
  emitop(OP_PUSH);
  emitq(id);
  emitop(OP_ERROR);
}
```

163. Cons Cells. These operators have been written out directly despite the obvious potential for refactoring into reusable pieces. This is short-lived until more compiler routines have been written and the similarity patterns between them become apparent.

Cons cells are defined by the *cons*, *car*, *cdr*, *null?* and *pair?* symbols with *set-car!* and *set-cdr!* providing for mutation.

```
void compile_cons(cell op, cell args, boolean tail_p__unused)
{    /* pattern 0; arity ≡ (O, O) */
  cell ncar, ncdr;

  arity(op, args, 2, 0);
  ncdr = cts_pop();
  ncar = cts_pop();
  compile_expression(ncdr, 0);
  emitop(OP_PUSH);
  compile_expression(ncar, 0);
  emitop(OP_CONS);
}
void compile_car(cell op, cell args, boolean tail_p__unused)
{    /* pattern 1; arity = (OP_PAIR_P) */
  int comefrom_pair_p;

  arity(op, args, 1, 0);
  compile_expression(cts_pop(), 0);
  emitop(OP_PUSH);
  emitop(OP_PAIR_P);
  emitop(OP_JUMP_TRUE);
  comefrom_pair_p = Here;
  emit(NIL);
  emitq(sym(ERR_UNEXPECTED));    /* TODO */
  emitop(OP_ERROR);
  patch(comefrom_pair_p, int_new(Here));
  emitop(OP_POP);
  emitop(OP_CAR);
}
void compile_cdr(cell op, cell args, boolean tail_p__unused)
{
  int comefrom_pair_p;

  arity(op, args, 1, 0);
  compile_expression(cts_pop(), 0);
  emitop(OP_PUSH);
  emitop(OP_PAIR_P);
  emitop(OP_JUMP_TRUE);
  comefrom_pair_p = Here;
  emit(NIL);
  emitq(sym(ERR_UNEXPECTED));    /* TODO */
  emitop(OP_ERROR);
  patch(comefrom_pair_p, int_new(Here));
  emitop(OP_POP);
  emitop(OP_CDR);    /* this is the only difference from the above */
}
void compile_null_p(cell op, cell args, boolean tail_p__unused)
{    /* pattern 2 = predicate */
```

```
    arity(op, args, 1, 0);
    compile_expression(cts_pop(), 0);
    emitop(OP_NULL_P);
}
void compile_pair_p(cell op, cell args, boolean tail_p __unused)
{
    arity(op, args, 1, 0);
    compile_expression(cts_pop(), 0);
    emitop(OP_PAIR_P);
}
void compile_set_car_m(cell op, cell args, boolean tail_p __unused)
{    /* pattern 3 = arity = (OP_PAIR_P, O) */
    cell value, object;
    int goto_pair_p;

    arity(op, args, 2, 0);
    value = cts_pop();
    object = cts_pop();
    compile_expression(object, bfalse);
    emitop(OP_PUSH);
    emitop(OP_PAIR_P);
    emitop(OP_JUMP_TRUE);
    goto_pair_p = comefrom();
    emitq(Sym_ERR_UNEXPECTED);
    emitop(OP_ERROR);
    patch(goto_pair_p, int_new(Here));
    compile_expression(value, bfalse);
    emitop(OP_SET_CAR_M);
}
void compile_set_cdr_m(cell op, cell args, boolean tail_p __unused)
{
    cell value, object;
    int goto_pair_p;

    arity(op, args, 2, 0);
    value = cts_pop();
    object = cts_pop();
    compile_expression(object, bfalse);
    emitop(OP_PUSH);
    emitop(OP_PAIR_P);
    emitop(OP_JUMP_TRUE);
    goto_pair_p = comefrom();
    emitq(Sym_ERR_UNEXPECTED);
    emitop(OP_ERROR);
    patch(goto_pair_p, int_new(Here));
    compile_expression(value, bfalse);
    emitop(OP_SET_CDR_M);
}
```

164. Environment. The *environment* mutators are the same except for the flag given to the final opcode.

```
void compile_set_m(cell op, cell args, boolean tail_p__unused)
{   /* pattern 4, arity = ( OP_ENV_P #<> symbol ? ) */
  cell env, name, value;
  int goto_env_p;

  arity(op, args, 3, bfalse);
  value = cts_pop();
  name = cts_pop();
  env = cts_pop();
  if (¬symbol_p(name)) error (ERR_ARITY_SYNTAX, NIL);
  compile_expression(env, bfalse);
  emitop(OP_PUSH);
  emitop(OP_ENVIRONMENT_P);
  emitop(OP_JUMP_TRUE);
  goto_env_p = comefrom();
  emitq(Sym_ERR_UNEXPECTED);
  emitop(OP_ERROR);
  patch(goto_env_p, int_new(Here));
  compile_expression(value, bfalse);
  emitop(OP_ENV_MUTATE_M);
  emit(name);
  emit(FALSE);
}

void compile_define_m(cell op, cell args, boolean tail_p__unused)
{
  cell env, name, value;
  int goto_env_p;

  arity(op, args, 3, bfalse);
  value = cts_pop();
  name = cts_pop();
  env = cts_pop();
  if (¬symbol_p(name)) error (ERR_ARITY_SYNTAX, NIL);
  compile_expression(env, bfalse);
  emitop(OP_PUSH);
  emitop(OP_ENVIRONMENT_P);
  emitop(OP_JUMP_TRUE);
  goto_env_p = comefrom();
  emitq(Sym_ERR_UNEXPECTED);
  emitop(OP_ERROR);
  patch(goto_env_p, int_new(Here));
  compile_expression(value, bfalse);
  emitop(OP_ENV_MUTATE_M);
  emit(name);
  emit(TRUE);
}

void compile_env_root(cell op, cell args, boolean tail_p__unused)
{   /* pattern 5 = no args */
  arity(op, args, 0, bfalse);
  emitop(OP_ENV_ROOT);
}
```

```
void compile_env_current(cell op, cell args, boolean tail_p __unused)
{
    arity(op, args, 0, bfalse);
    emitop(OP_ENV_QUOTE);
}
```

165. Quotation & Quasiquotation. A quoted object is one which is not evaluated and we have an opcode to do just that, used by many of the implementations above.

void *compile_quote*(**cell** *op__unused*, **cell** *args*, **boolean** *tail_p__unused*)
{ *emitq*(*args*); }

166. Quasiquoting an object is almost, but not quite, entirely different. The end result is the same however—a run-time object which (almost) exactly matches the unevaluated source code that it was created from.

A quasiquoted object is converted into its final form by changing any *unquote* (and *unquote-splicing*) within it to the result of evaulating them. This is complicated enough because we're now writing a compiler within our compiler[1] but additionally the quasiquoted object may contain quasiquoted objects, changing the nature of the inner-*unquote* operators.

167. The compiler for compiling quasiquoted code only calls directly into the recursive quasicompiler engine (let's call it the quasicompiler).

⟨Function declarations 8⟩ +≡
void *compile_quasicompiler*(**cell**, **cell**, **cell**, **int**, **boolean**);

168. **void** *compile_quasiquote*(**cell** *op*, **cell** *args*, **boolean** *tail_p__unused*)
{ /* pattern Q */
compile_quasicompiler(*op*, *args*, *args*, 0, *bfalse*);
}

169. As with any compiler, the first task is to figure out what sort of expression is being quasicompiled. Atoms are themselves. Otherwise lists and vectors must be recursively compiled item-by-item, and the syntactic operators must operate when encountered.

Quasiquoting *vector*s is not supported but I'm not anticipating it being difficult, just not useful yet.

```
void compile_quasicompiler(cell op, cell oargs, cell arg, int depth, boolean in_list_p)
{
  if (pair_p(arg)) {⟨ Quasiquote a pair/list 170 ⟩}
  else if (vector_p(arg)) { error (ERR_UNIMPLEMENTED, NIL); }
  else if (syntax_p(arg)) {⟨ Quasiquote syntax 171 ⟩}
  else {
    emitq(arg);
    if (in_list_p) emitop(OP_CONS);
  }
}
```

[1] Yo!

170. Dealing first with the simple case of a list the quasicompiler reverses the list to find its tail, which may or may not be NIL, and recursively calling *compile_quasicompiler* for every item.

After each item has been quasicompiled it will be combined with the transformed list being grown on top of the stack.

When quasicompiling the list's tail there is no partial list to prepend it to so the quasicompiler is entered in atomic mode. *compile_quasicompiler* can be relied on to handle the tail of a proper or improper list.

⟨ Quasiquote a pair/list 170 ⟩ ≡

```
cell todo, tail;
tail = NIL;
todo = list_reverse(arg, &tail, Λ);
compile_quasicompiler(op, oargs, tail, depth, bfalse);
for ( ; ¬null_p(todo); todo = cdr(todo)) {
  emitop(OP_PUSH);      /* Push the list so far */
  compile_quasicompiler(op, oargs, car(todo), depth, btrue);
}
if (in_list_p) emitop(OP_CONS);
```

This code is used in section 169.

171. The quote & unquote syntax is where the quasicompiler starts to get interesting. *quotes* and *quasiquotes* (and a *dotted* tail) recurse back into the quasicompiler to emit the transformation of the quoted object, then re-apply the syntax operator.

depth is increased when recursing into a *quasiquote* so that the compiler knows whether to evaluate an unquote operator.

⟨ Quasiquote syntax 171 ⟩ ≡

```
int d;
if (car(arg) ≡ Sym_SYNTAX_DOTTED
      ∨ car(arg) ≡ Sym_SYNTAX_QUOTE
      ∨ car(arg) ≡ Sym_SYNTAX_QUASI) {
  d = (car(arg) ≡ Sym_SYNTAX_QUASI) ? 1 : 0;
  compile_quasicompiler(op, oargs, cdr(arg), depth + d, bfalse);
  emitop(OP_SYNTAX);
  emit(car(arg));
  if (in_list_p) emitop(OP_CONS);
}
```

See also sections 172 and 173.

This code is used in section 169.

172. *unquote* evaluates the unquoted object. If quasiquote is quasicompiling an inner quasiquote then the unquoted object isn't evaluated but compiled at a decreased *depth*. This enables the correct unquoting-or-not of quasiquoting quasiquoted quotes.

⟨ Quasiquote syntax 171 ⟩ +≡

```
else
  if (car(arg) ≡ Sym_SYNTAX_UNQUOTE) {
    if (depth > 0) {
      compile_quasicompiler(op, oargs, cdr(arg), depth − 1, bfalse);
      emitop(OP_SYNTAX);
      emit(Sym_SYNTAX_UNQUOTE);
    }
    else compile_expression(cdr(arg), bfalse);
    if (in_list_p) emitop(OP_CONS);
  }
```

173. Similarly to *unquote*, *unquote-splicing* recurses back into the quasicompiler at a lower depth when unquoting an inner quasiquote.

⟨ Quasiquote syntax 171 ⟩ +≡

```
else
  if (car(arg) ≡ Sym_SYNTAX_UNSPLICE) {
    if (depth > 0) {
      compile_quasicompiler(op, oargs, cdr(arg), depth − 1, bfalse);
      emitop(OP_SYNTAX);
      emit(Sym_SYNTAX_UNSPLICE);
      if (in_list_p) emitop(OP_CONS);
    }
    else {⟨ Compile unquote-splicing 174 ⟩}
  }
  else error (ERR_UNIMPLEMENTED, NIL);
```

174. Splicing Lists. If not recursing back into the quasicompiler at a lower depth then we are quasicompiling at the lowest depth and need to do the work.

When splicing into the tail position of a list we can replace its `NIL` with the evaluation with minimal further processing. Unfortunately we don't know until runtime whether we are splicing into the tail position – consider constructs like `` `(,@foo ,@bar) `` where `bar` evaluates to `NIL`.

```
⟨ Compile unquote-splicing 174 ⟩ ≡
  int goto_inject_iterate, goto_inject_start, goto_finish;
  int goto_list_p, goto_null_p, goto_nnull_p;
  if (¬in_list_p) error (ERR_UNEXPECTED, arg);
  emitop(OP_PEEK);
  emitop(OP_NULL_P);
  emitop(OP_JUMP_TRUE); goto_null_p = comefrom();
  emitop(OP_PUSH);     /* save FALSE */
  emitop(OP_JUMP); goto_nnull_p = comefrom();
  patch(goto_null_p, int_new(Here));
  emitop(OP_SWAP);     /* become the tail, save TRUE */
  patch(goto_nnull_p, int_new(Here));
```

See also sections 175, 176, and 177.

This code is used in section 173.

175. `FALSE` or `TRUE` is now atop the stack indicating whether a new list is being built otherwise the remainder of the list is left on the stack. Now we can evaluate and validate the expression.

```
⟨ Compile unquote-splicing 174 ⟩ +≡
  compile_expression(cdr(arg), 0);
  emitop(OP_PUSH);
  emitop(OP_LIST_P); emit(TRUE); emit(FALSE);
  emitop(OP_JUMP_TRUE); goto_list_p = comefrom();
  emitq(Sym_ERR_UNEXPECTED);
  emitop(OP_ERROR);
```

176. If we have a list we can leave it as-is if we were originally in the tail position.

```
⟨ Compile unquote-splicing 174 ⟩ +≡
  patch(goto_list_p, int_new(Here));
  emitop(OP_POP);
  emitop(OP_SWAP);
  emitop(OP_JUMP_TRUE); goto_finish = comefrom();
```

177. Splicing a list into the middle of another list is done item-by-item in reverse. A small efficiency could be gained here by not walking the list a second time (the first to validate it above) at the cost of more complex bytecode.

By now the evaluated list to splice in is first on the stack followed by the partial result.

```
⟨ Compile unquote-splicing 174 ⟩ +≡
  emitop(OP_POP);
  emitop(OP_LIST_REVERSE); emit(TRUE); emit(FALSE);
  ⟨ Walk through the splicing list 178 ⟩
```

178. ⟨ Walk through the splicing list 178 ⟩ ≡

$emitop$(OP_JUMP); $goto\_inject\_start = comefrom()$;
$goto\_inject\_iterate = Here$;
$emitop$(OP_POP);
$emitop$(OP_SNOC);
$emitop$(OP_CYCLE);
$emitop$(OP_CONS);
$emitop$(OP_SWAP);

See also section 179.

This code is used in section 177.

179. If this was the last item (the first of the evaluated list's) or the evaluation was NIL then we're done otherwise we go around again. This is also where the loop starts to handle the case of evaluating an empty list.

⟨ Walk through the splicing list 178 ⟩ +≡

$patch(goto\_inject\_start, int\_new(Here))$;
$emitop$(OP_PUSH);
$emitop$(OP_NULL_P);
$emitop$(OP_JUMP_FALSE); $emit(int\_new(goto\_inject\_iterate))$;
$emitop$(OP_POP);
$patch(goto\_finish, int\_new(Here))$;
$emitop$(OP_POP);

180. Testing. A comprehensive test suite is planned for `LossLess` but a testing tool would be no good if it wasn't itself reliable. During the build process test binaries are produced with additional functionality exposing internal data structures and processes necessary to test the compiled `LossLess` executable.

C's preprocessor is used to define an alternate entry-point to `LossLess` by renaming *main* and adding testing opcodes and operators. Other testing functions unused by the `LossLess` runtime are expected to be removed by the C-compiler or linker rather than obscuring this source code by drowning it in preprocessor directives.

```
#define test_copy_env() Env
#define test_compare_env(o) ((o) ≡ Env)
#define test_is_env(o, e) ((o) ≡ (e))
```

⟨ Test executable wrapper 180 ⟩ ≡

```
#define LL_TEST 1
#include "lossless.c"
  void test_main(void);

  int main(int argc __unused, char **argv __unused)
  {
    volatile boolean first = btrue;
#ifndef LLT_BARE_TEST
    vm_init();
    if (argc > 1) error(ERR_ARITY_EXTRA, NIL);
    vm_prepare();
#else
    setjmp(Goto_Begin);
    setjmp(Goto_Error);
#endif
    if (¬first) {
      printf("Bail␣out!␣Unhandled␣exception␣in␣test\n");
      return EXIT_FAILURE;
    }
    first = bfalse;
    test_main();
    tap_plan(0);
    return EXIT_SUCCESS;
  }
```

This code is used in sections 181, 198, 239, 251, 260, 269, 276, and 283.

181. The heap tests need to operate before the VM has been initialised.

⟨ Allocator test executable wrapper 181 ⟩ ≡

```
#define LLT_BARE_TEST 1
#define LL_ALLOCATE fallible_reallocarray
  void *fallible_reallocarray();    /* no size_t yet */
  ⟨ Test executable wrapper 180 ⟩
  int Allocate_Success = −1;
  void *fallible_reallocarray(void *ptr, size_t nmemb, size_t size)
  {
    return Allocate_Success−− ? reallocarray(ptr, nmemb, size) : Λ;
  }
```

This code is used in sections 202 and 223.

182. Tests need to be able to save data from the maw of the garbage collector.

⟨ Global variables 7 ⟩ +≡
```
cell Tmp_Test = NIL;
```

183. ⟨ Protected Globals 18 ⟩ +≡
```
#ifdef LL_TEST
  &Tmp_Test ,
#endif
```

184. Some tests need to examine a snapshot of the interpreter's run-time state which they do by calling *test!probe*.

⟨ Function declarations 8 ⟩ +≡
```
void compile_testing_probe(cell, cell, boolean);
void compile_testing_probe_app(cell, cell, boolean);
cell testing_build_probe(cell);
```

185. ⟨ Testing opcodes 185 ⟩ ≡
```
OP_TEST_PROBE ,
```
This code is used in sections 112 and 113.

186. ⟨ Testing implementations 186 ⟩ ≡
```
case OP_TEST_PROBE:
  Acc = testing_build_probe(rts_pop(1));
  skip(1);
  break;
```
This code is used in section 79.

187. ⟨ Testing primitives 187 ⟩ ≡
```
{"test!probe", compile_testing_probe},
{"test!probe-applying", compile_testing_probe_app} ,
```
This code is used in section 284.

188.
```
void compile_testing_probe(cell op__unused, cell args, boolean tail_p__unused)
{
  emitop(OP_PUSH);
  emitq(args);
  emitop(OP_TEST_PROBE);
}
```

189. This variant evaluates its run-time arguments first.

```
void compile_testing_probe_app(cell op __unused, cell args, boolean tail_p __unused)
{
    emitop(OP_PUSH);
    cts_push(args = list_reverse(args, Λ, Λ));
    emitq(NIL);
    for ( ; pair_p(args); args = cdr(args)) {
        emitop(OP_PUSH);
        compile_expression(car(args), bfalse);
        emitop(OP_CONS);
    }
    cts_pop();
    emitop(OP_TEST_PROBE);
}
```

190. TODO: This should make a deep copy of the objects not merely reference them.

```
#define probe_push(n, o) do {
        vms_push(cons((o), NIL));
        vms_set(cons(sym(n), vms_ref()));
        t = vms_pop();
        vms_set(cons(t, vms_ref()));
    } while (0)
    cell testing_build_probe(cell was_Acc)
    {
        cell t;

        vms_push(NIL);
        probe_push("Acc", was_Acc);
        probe_push("Args", Acc);
        probe_push("Env", Env);
        return vms_pop();
    }
#undef probe_push
```

191. The Perl ecosystem has a well-deserved reputation for its thorough testing regime and the quality (if not necessarily the quality) of the results so LossLess is deliberately aping the interfaces that were developed there.

The LossLess internal tests are a collection of test "script"s each of which massages some LossLess function or other and then reports what happened in a series of binary pass/fail "test"s. A test in this sense isn't the performance of any activity but comparing the result of having *already performed* some activity with the expected outcome. Any one action normally requires a lot of individual tests to confirm the validity of its result. Occasionally "test" refers to a collection of these tests which are performed together, which is a bad habit.

This design is modelled on the Test Anything Protocol and the test scripts call an API that looks suspiciously like a tiny version of *Test::Simple*.

tap_plan is optionally called before the test script starts if the total number of tests is known in advance and then again at the end of testing with an argument of 0 to emit exactly one test plan.

```
#define tap_fail(m) tap_ok(bfalse, (m))
#define tap_pass(m) tap_ok(btrue, (m))
#define tap_again(t, r, m) tap_ok(((t) = ((t) ∧ (r))), (m))    /* intentional assignment */
#define tap_more(t, r, m) (t) &= tap_ok((r), (m))
#define tap_or(p, m) if (¬tap_ok((p), (m)))
```

⟨ Function declarations 8 ⟩ +≡

```
#ifdef LL_TEST
  void tap_plan(int);
  boolean tap_ok(boolean, char *);
#endif
```

192. ⟨ Global variables 7 ⟩ +≡

```
  boolean Test_Passing = btrue;
  int Test_Plan = −1;
  int Next_Test = 1;    /* not 0 */
```

193.

```
void tap_plan(int plan)
  {
    if (plan ≡ 0) {
      if (Test_Plan < 0) printf("1..%d\n", Next_Test − 1);
      else if (Next_Test − 1 ≠ Test_Plan) {
        printf("#␣Planned␣%3$d␣%1$s␣but␣ran␣%2$s%4$d!\n", (Test_Plan ≡ 1 ? "test" : "tests"),
            (Next_Test ≤ Test_Plan ? "only␣" : ""), Test_Plan, Next_Test − 1);
        Test_Passing = bfalse;
      }
      return;
    }
    if (Test_Plan > 0) error ("plan-exists", int_new(Test_Plan));
    if (plan < 0) error (ERR_UNEXPECTED, cons(sym("test-plan"), int_new(plan)));
    Test_Plan = plan;
    printf("1..%d\n", plan);
  }
```

194. **boolean** *tap_ok*(**boolean** *result*, **char** **message*)

```
{
  printf("%s␣%d␣%s\n", (result ? "ok" : "not␣ok"),
      Next_Test++,
      (message ∧ *message) ? message : "?");
  if (result) return btrue;
  return Test_Passing = bfalse;
}
```

195. LossLess is a programming language and so a lot of its tests involve code. *test_vmsgf* formats messages describing tests which involve code (or any other s-expression) in a consistent way. The caller is expected to maintain its own buffer of TEST_BUFSIZE bytes a pointer to which goes in and out so that the function can be used in-line.

tmsgf hardcodes the names of the variables a function passes into *test_vmsgf* for brevity.

```
#define TEST_BUFSIZE 1024
#define tmsgf(...) test_msgf (msg, prefix, __VA_ARGS__)
  char *test_msgf(char *tmsg, const char *tsrc, char *fmt, ...)
  {
    char ttmp[TEST_BUFSIZE] = {0};
    int ret;
    va_list ap;
    va_start(ap, fmt);
    ret = vsnprintf(ttmp, TEST_BUFSIZE, fmt, ap);
    va_end(ap);
    snprintf(tmsg, TEST_BUFSIZE, "%s:␣%s", tsrc, ttmp);
    return tmsg;
  }
```

196. The majority of tests validate some parts of the VM state, which parts is controlled by the *flags* parameter.

```
#define TEST_VMSTATE_RUNNING #01
#define TEST_VMSTATE_NOT_RUNNING #00
#define TEST_VMSTATE_INTERRUPTED #02
#define TEST_VMSTATE_NOT_INTERRUPTED #00
#define TEST_VMSTATE_VMS #04
#define TEST_VMSTATE_CTS #08
#define TEST_VMSTATE_RTS #10
#define TEST_VMSTATE_STACKS (TEST_VMSTATE_VMS | TEST_VMSTATE_CTS | TEST_VMSTATE_RTS)
#define TEST_VMSTATE_ENV_ROOT #20
#define TEST_VMSTATE_PROG_MAIN #40
#define test_vm_state_full(p)
        test_vm_state((p), TEST_VMSTATE_NOT_RUNNING | TEST_VMSTATE_NOT_INTERRUPTED |
            TEST_VMSTATE_ENV_ROOT | TEST_VMSTATE_PROG_MAIN | TEST_VMSTATE_STACKS)
#define test_vm_state_normal(p)
        test_vm_state((p), TEST_VMSTATE_NOT_RUNNING | TEST_VMSTATE_NOT_INTERRUPTED |
            TEST_VMSTATE_PROG_MAIN | TEST_VMSTATE_STACKS)    /* ¬TEST_VMSTATE_ENV_ROOT */
  void test_vm_state(char *prefix, int flags)
  {
    char msg[TEST_BUFSIZE] = {0};
    if (flags & TEST_VMSTATE_RUNNING) tap_ok(Running, tmsgf("(==␣Running␣1)"));
    else tap_ok(¬Running, tmsgf("(==␣Running␣0)"));
    if (flags & TEST_VMSTATE_INTERRUPTED) tap_ok(Interrupt, tmsgf("(==␣Interrupt␣1)"));
    else tap_ok(¬Interrupt, tmsgf("(==␣Interrupt␣0)"));
    if (flags & TEST_VMSTATE_VMS) tap_ok(null_p(VMS), tmsgf("(null?␣VMS)"));
    if (flags & TEST_VMSTATE_CTS) tap_ok(null_p(CTS), tmsgf("(null?␣CTS)"));
    if (flags & TEST_VMSTATE_RTS) tap_ok(RTSp ≡ −1, tmsgf("(==␣RTSp␣-1)"));
    if (flags & TEST_VMSTATE_ENV_ROOT) tap_ok(Env ≡ Root, tmsgf("(==␣Env␣Root)"));
    if (flags & TEST_VMSTATE_PROG_MAIN) {
      tap_ok(Prog ≡ Prog_Main, tmsgf("Prog_Main␣is␣returned␣to"));
      tap_ok(Ip ≡ vector_length(Prog_Main) − 1, tmsgf("Prog_Main␣is␣completed"));
    }
        /* TODO? Others: root unchanged; */
  }
```

197. Each LossLess internal test script is a function named “*test_...*”. Other prefixes are also used for supporting functions. Arguments to the testing binary determine which test script to run or enter a standard REPL in the testing *environment* if there are none.

TODO: Add a banner and run-time detection to make it clear the testing environment is not for production use.

198. Sanity Test. This seemingly pointless test achieves two goals: the test harness can run it first and can abort the entire test suite if it fails, and it provides a simple demonstration of how individual test scripts interact with the outside world, without obscuring it with any actual testing.

```
⟨ t/sanity.c 198 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    tap_plan(1);
    tap_pass("LossLess␣compiles␣and␣runs");
  }
```

199. Unit Tests. This is the very boring process of laboriously checking that each function or otherwise segregable unit of code does what it says on the tin. For want of a better model to follow I've taken inspiration from Mike Bland's article "Goto Fail, Heartbleed, and Unit Testing Culture" describing how he created unit tests for the major OpenSSL vulnerabilities known as "goto fail" and "Heartbleed". The article itself is behind some sort of Google wall but Martin Fowler has reproduced it at https://martinfowler.com/articles/testing-culture.html.

```
⟨ Type definitions 6 ⟩ +≡
#define LLTF_BASE_HEADER
        const char *name;
  test_fixture_thunk prepare; test_fixture_thunk destroy
  typedef struct lltf_Base lltf_Base;
  typedef void (*test_fixture_thunk)(struct lltf_Base *);
  struct lltf_Base {
    LLTF_BASE_HEADER;
  };
  typedef boolean (*test_unit)(void);
```

200.

```
#define test_single(s) test_single_imp(s, 0)
#define test_suite(s) test_suite_imp(s, 0)
  void test_single_imp(test_unit suite, int id)
  {
    char msg[TEST_BUFSIZE] = {0};
    boolean ok;
    ok = suite();
    tap_ok(ok, test_msgf(msg, "Test case", "%d", id));
  }
  void test_suite_imp(test_unit *suite, int start)
  {
    int i;
    for (i = start; *suite; suite++, i++) test_single_imp(*suite, i);
  }
```

201. Heap Allocation. The first units we test are the memory allocators because I've already found embarrassing bugs there proving that even that "obvious" code needs manual verification. To do that we will need to be able to make *reallocarray* fail without actually exhausting the system's memory. A global counter is decremented each time this variant is called and returns Λ if it reaches zero.

202. ⟨ t/cell-heap.c 202 ⟩ ≡

```
⟨ Allocator test executable wrapper 181 ⟩
⟨ Unit test the heap allocator 203 ⟩
test_unit llt_Grow_Pool_suite[ ] = {
    llt_Grow_Pool__Immediate_Fail, llt_Grow_Pool__Second_Fail, llt_Grow_Pool__Third_Fail,
    llt_Grow_Pool__Full_Success, llt_Grow_Pool__Full_Immediate_Fail, llt_Grow_Pool__Full_Second_Fail,
    llt_Grow_Pool__Full_Third_Fail, Λ
};
void test_main(void)
{
  printf("#␣The␣many␣unhandled␣out-of-memory␣errors""␣are␣expected␣and␣harmless.\n");
  test_single(llt_Grow_Pool__Initial_Success);
  test_suite_imp(llt_Grow_Pool_suite, 1);
}
```

203. This method of implementing unit tests has us pose 5 questions:

1. *What is the contract fulfilled by the code under test?*

new_cells_segment performs 3, or 5 if each allocation is counted seperately, actions: Enlarge each of `CAR`, `CDR` & `TAG` in turn, checking for out-of-memory for each; zero-out the newly-allocated range of memory; update the global counters *Cells_Poolsize* & *Cells_Segment*.

There is no return value but either the heap will have been enlarged or one of 3 (mostly identical) errors will have been raised.

2. *What preconditions are required, and how are they enforced?*

Cells_Segment describes how much the pool will grow by. If *Cells_Poolsize* is 0 the three pointers must be Λ otherwise they each point to an area of allocated memory *Cells_Poolsize* elements wide. There is no explicit enforcement.

3. *What postconditions are guaranteed?*

IFF there was an allocation error for any of the 3 pools, the pointer under question will not have changed but those reallocated before it may have. *Cells_Poolsize* & *Cells_Segment* will be unchanged. Any newly-allocated memory should not be considered available

Otherwise `CAR`, `CDR` & `TAG` will point to still-valid memory but possibly at the same address.

The newly allocated memory will have been zerod.

Cells_Poolsize & *Cells_Segment* will have been enlarged.

new_cells_segment also guarantees that previously-allocated data will not have changed but it's safe for now to rely on *reallocarray* getting that right.

4. *What example inputs trigger different behaviors?*

Chiefly there are two classes of inputs, whether or not *Cells_Poolsize* is 0, and whether allocation succeeds for each of the 3 attempts.

5. *What set of tests will trigger each behavior and validate each guarantee?*

Eight tests, four starting from no heap and four from a heap with data in it. One for success and one for each potentially failed allocation.

⟨ Unit test the heap allocator 203 ⟩ ≡

```
enum lltf_Grow_Pool_result {
  LLTF_GROW_POOL_SUCCESS, LLTF_GROW_POOL_FAIL_CAR, LLTF_GROW_POOL_FAIL_CDR,
      LLTF_GROW_POOL_FAIL_TAG
};
```

See also sections 204, 205, 206, 207, 208, 213, 214, 215, 216, 217, 218, 219, 220, and 221.

This code is used in section 202.

204. The test fixture describes how to perform the test and what to expect out of it. *fix.expect* is an enum which identifies the checks to carry out after performing the test.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
typedef struct {
  LLTF_BASE_HEADER;
  enum lltf_Grow_Pool_result expect;
  int allocations;
  int Poolsize;
  int Segment;
  cell *CAR;
  cell *CDR;
  char *TAG;
  char *heapcopy;
  cell *save_CAR;
  cell *save_CDR;
  char *save_TAG;
} lltf_Grow_Pool;
```

205. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
void llt_Grow_Pool_prepare(lltf_Grow_Pool *);
void llt_Grow_Pool_destroy(lltf_Grow_Pool *);

lltf_Grow_Pool llt_Grow_Pool_fix(const char *name)
{
  lltf_Grow_Pool fix;

  bzero(&fix, sizeof (fix));
  fix.name = name;
  fix.prepare = (test_fixture_thunk) llt_Grow_Pool_prepare;
  fix.destroy = (test_fixture_thunk) llt_Grow_Pool_destroy;
  fix.expect = LLTF_GROW_POOL_SUCCESS;
  fix.allocations = −1;
  fix.Segment = HEAP_SEGMENT;
  return fix;
}
```

206. This unit test relies on the VM being uninitialised so that it can safely switch out the heap pointers.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
void llt_Grow_Pool_prepare(lltf_Grow_Pool *fix)
{
  if (fix→Poolsize) {
    int cs = fix→Poolsize;
    fix→heapcopy = reallocarray(Λ, cs, 2 * sizeof(cell) + sizeof(char));
    fix→save_CAR = (cell *) fix→heapcopy;
    fix→save_CDR = (cell *)(fix→heapcopy + sizeof(cell) * cs);
    fix→save_TAG = fix→heapcopy + sizeof(cell) * cs * 2;
    bcopy(fix→CAR, fix→save_CAR, sizeof(cell) * cs);
    bcopy(fix→CDR, fix→save_CDR, sizeof(cell) * cs);
    bcopy(fix→TAG, fix→save_TAG, sizeof(char) * cs);
  }
  CAR = fix→CAR;
  CDR = fix→CDR;
  TAG = fix→TAG;
  Cells_Poolsize = fix→Poolsize;
  Cells_Segment = fix→Segment;
}
```

207. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
void llt_Grow_Pool_destroy(lltf_Grow_Pool *fix)
{
  free(CAR);
  free(CDR);
  free(TAG);
  free(fix→heapcopy);
  CAR = CDR = Λ;
  TAG = Λ;
  Cells_Poolsize = 0;
  Cells_Segment = HEAP_SEGMENT;
}
```

208. There is not much for this test to do apart from prepare state and call *new_cells_segment* then validate that the memory was, or was not, correctly reallocated.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool_exec(lltf_Grow_Pool fix)
{
  char msg[TEST_BUFSIZE] = {0};
  boolean ok;
  jmp_buf save_jmp;
  if (fix.prepare) fix.prepare((lltf_Base *) &fix);
  Allocate_Success = fix.allocations;
  memcpy(&save_jmp, &Goto_Begin, sizeof(jmp_buf));
  if (¬setjmp(Goto_Begin)) new_cells_segment();
  Allocate_Success = −1;
  memcpy(&Goto_Begin, &save_jmp, sizeof(jmp_buf));
  switch (fix.expect) {
  case LLTF_GROW_POOL_SUCCESS:
    ⟨ Unit test allocations, validate success 209 ⟩
    break;    /* TODO: test for bzero */
  case LLTF_GROW_POOL_FAIL_CAR:
    ⟨ Unit test allocations, validate car failure 210 ⟩
    break;
  case LLTF_GROW_POOL_FAIL_CDR:
    ⟨ Unit test allocations, validate cdr failure 211 ⟩
    break;
  case LLTF_GROW_POOL_FAIL_TAG:
    ⟨ Unit test allocations, validate tag failure 212 ⟩
    break;
  }
  if (fix.destroy) fix.destroy((lltf_Base *) &fix);
  return ok;
}
```

209. ⟨ Unit test allocations, validate success 209 ⟩ ≡

```
ok = tap_ok(Cells_Poolsize ≡ (fix.Poolsize + fix.Segment), test_msgf(msg, fix.name,
    "Cells_Poolsize␣is␣increased"));
tap_more(ok, Cells_Segment ≡ (fix.Poolsize + fix.Segment)/2, test_msgf(msg, fix.name,
    "Cells_Segment␣is␣increased"));
tap_more(ok, CAR ≠ CDR ∧ CAR ≠ (cell *) TAG, test_msgf(msg, fix.name, "CAR,␣CDR␣&␣TAG␣are␣unique"));
tap_more(ok, CAR ≠ Λ, test_msgf(msg, fix.name, "CAR␣is␣not␣NULL"));
tap_more(ok, ¬bcmp(CAR, fix.save_CAR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
    "CAR␣heap␣is␣unchanged"));
tap_more(ok, CDR ≠ Λ, test_msgf(msg, fix.name, "CDR␣is␣not␣NULL"));
tap_more(ok, ¬memcmp(CDR, fix.save_CDR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
    "CDR␣heap␣is␣unchanged"));
tap_more(ok, TAG ≠ Λ, test_msgf(msg, fix.name, "TAG␣is␣not␣NULL"));
tap_more(ok, ¬memcmp(TAG, fix.save_TAG, sizeof(char) * fix.Poolsize), test_msgf(msg, fix.name,
    "TAG␣heap␣is␣unchanged"));
```

This code is used in section 208.

210. ⟨ Unit test allocations, validate car failure 210 ⟩ ≡

```
ok = tap_ok(Cells_Poolsize ≡ fix.Poolsize, test_msgf(msg, fix.name,
      "Cells_Poolsize␣is␣not␣increased"));
tap_more(ok, Cells_Segment ≡ fix.Segment, test_msgf(msg, fix.name,
      "Cells_Segment␣is␣not␣increased"));
tap_more(ok, CAR ≡ fix.CAR, test_msgf(msg, fix.name, "CAR␣is␣unchanged"));
tap_more(ok, CDR ≡ fix.CDR, test_msgf(msg, fix.name, "CDR␣is␣unchanged"));
tap_more(ok, TAG ≡ fix.TAG, test_msgf(msg, fix.name, "TAG␣is␣unchanged"));
```

This code is used in section 208.

211. ⟨ Unit test allocations, validate cdr failure 211 ⟩ ≡

```
ok = tap_ok(Cells_Poolsize ≡ fix.Poolsize, test_msgf(msg, fix.name,
      "Cells_Poolsize␣is␣not␣increased"));
tap_more(ok, Cells_Segment ≡ fix.Segment, test_msgf(msg, fix.name,
      "Cells_Segment␣is␣not␣increased"));
tap_more(ok, ¬memcmp(CAR, fix.save_CAR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
      "CAR␣heap␣is␣unchanged"));
tap_more(ok, CDR ≡ fix.CDR, test_msgf(msg, fix.name, "CDR␣is␣unchanged"));
tap_more(ok, TAG ≡ fix.TAG, test_msgf(msg, fix.name, "TAG␣is␣unchanged"));
```

This code is used in section 208.

212. ⟨ Unit test allocations, validate tag failure 212 ⟩ ≡

```
ok = tap_ok(Cells_Poolsize ≡ fix.Poolsize, test_msgf(msg, fix.name,
      "Cells_Poolsize␣is␣not␣increased"));
tap_more(ok, Cells_Segment ≡ fix.Segment, test_msgf(msg, fix.name,
      "Cells_Segment␣is␣not␣increased"));
tap_more(ok, ¬memcmp(CAR, fix.save_CAR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
      "CAR␣heap␣is␣unchanged"));
tap_more(ok, ¬memcmp(CDR, fix.save_CDR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
      "CDR␣heap␣is␣unchanged"));
tap_more(ok, TAG ≡ fix.TAG, test_msgf(msg, fix.name, "TAG␣is␣unchanged"));
```

This code is used in section 208.

213. This tests that allocation is successful the first time the heap is ever allocated. It is the simplest test in this unit.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Initial_Success(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  return llt_Grow_Pool_exec(fix);
}
```

214. If the very first call to *reallocarray* fails then everything should remain unchanged.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Immediate_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_CAR;
  fix.allocations = 0;
  return llt_Grow_Pool_exec(fix);
}
```

215. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Second_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_CDR;
  fix.allocations = 1;
  return llt_Grow_Pool_exec(fix);
}
```

216. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Third_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_TAG;
  fix.allocations = 2;
  return llt_Grow_Pool_exec(fix);
}
```

217. Data already on the heap must be preserved exactly.

⟨ Unit test the heap allocator 203 ⟩ +≡

```
void lltf_Grow_Pool__fill(lltf_Grow_Pool *fix)
{
  size_t i;
  fix→CAR = reallocarray(Λ, fix→Poolsize, sizeof(cell));
  fix→CDR = reallocarray(Λ, fix→Poolsize, sizeof(cell));
  fix→TAG = reallocarray(Λ, fix→Poolsize, sizeof(char));
  for (i = 0; i < (fix→Poolsize * sizeof(cell))/sizeof(int); i++) *(((int *) fix→CAR) + i) = rand();
  for (i = 0; i < (fix→Poolsize * sizeof(cell))/sizeof(int); i++) *(((int *) fix→CDR) + i) = rand();
  for (i = 0; i < (fix→Poolsize * sizeof(char))/sizeof(int); i++) *(((int *) fix→TAG) + i) = rand();
}
```

218. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Full_Success(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Pool__fill(&fix);
  return llt_Grow_Pool_exec(fix);
}
```

219. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Full_Immediate_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_CAR;
  fix.allocations = 0;
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Pool__fill(&fix);
  return llt_Grow_Pool_exec(fix);
}
```

220. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Full_Second_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_CDR;
  fix.allocations = 1;
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Pool__fill(&fix);
  return llt_Grow_Pool_exec(fix);
}
```

221. ⟨ Unit test the heap allocator 203 ⟩ +≡

```
boolean llt_Grow_Pool__Full_Third_Fail(void)
{
  lltf_Grow_Pool fix = llt_Grow_Pool_fix(__func__);
  fix.expect = LLTF_GROW_POOL_FAIL_TAG;
  fix.allocations = 2;
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Pool__fill(&fix);
  return llt_Grow_Pool_exec(fix);
}
```

222. Vector Heap. Testing the vector's heap is the same but simpler because it has 1 not 3 possible error conditions so this section is duplicated from the previous without further explanation.

223. ⟨ t/vector-heap.c 223 ⟩ ≡

```
⟨ Allocator test executable wrapper 181 ⟩
⟨ Unit test the vector allocator 224 ⟩
test_unit llt_Grow_Vector_Pool_suite[] = {
    llt_Grow_Vector_Pool__Empty_Fail, llt_Grow_Vector_Pool__Full_Success,
    llt_Grow_Vector_Pool__Full_Fail, Λ
};
void test_main(void)
{
  printf("#␣The␣many␣unhandled␣out-of-memory␣errors""␣are␣expected␣and␣harmless.\n");
  test_single(llt_Grow_Vector_Pool__Empty_Success);
  test_suite_imp(llt_Grow_Vector_Pool_suite, 1);
}
```

224. ⟨ Unit test the vector allocator 224 ⟩ ≡

```
enum lltf_Grow_Vector_Pool_result {
  LLTF_GROW_VECTOR_POOL_SUCCESS, LLTF_GROW_VECTOR_POOL_FAIL
};
```

See also sections 225, 226, 227, 228, 229, 232, 233, 234, 235, and 236.
This code is used in section 223.

225. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
typedef struct {
  LLTF_BASE_HEADER;
  enum lltf_Grow_Vector_Pool_result expect;
  int allocations;
  int Poolsize;
  int Segment;
  cell *VECTOR;
  cell *save_VECTOR;
} lltf_Grow_Vector_Pool;
```

226. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
void llt_Grow_Vector_Pool_prepare(lltf_Grow_Vector_Pool *);
void llt_Grow_Vector_Pool_destroy(lltf_Grow_Vector_Pool *);
lltf_Grow_Vector_Pool llt_Grow_Vector_Pool_fix(const char *name)
{
  lltf_Grow_Vector_Pool fix;
  bzero(&fix, sizeof (fix));
  fix.name = name;
  fix.prepare = (test_fixture_thunk) llt_Grow_Vector_Pool_prepare;
  fix.destroy = (test_fixture_thunk) llt_Grow_Vector_Pool_destroy;
  fix.expect = LLTF_GROW_VECTOR_POOL_SUCCESS;
  fix.allocations = −1;
  fix.Segment = HEAP_SEGMENT;
  return fix;
}
```

227. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
void llt_Grow_Vector_Pool_prepare(lltf_Grow_Vector_Pool *fix)
{
  if (fix→Poolsize) {
    int cs = fix→Poolsize;
    fix→save_VECTOR = reallocarray(Λ, cs, sizeof(cell));
    bcopy(fix→VECTOR, fix→save_VECTOR, sizeof(cell) * cs);
  }
  VECTOR = fix→VECTOR;
  Vectors_Poolsize = fix→Poolsize;
  Vectors_Segment = fix→Segment;
}
```

228. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
void llt_Grow_Vector_Pool_destroy(lltf_Grow_Vector_Pool *fix)
{
  free(VECTOR);
  free(fix→save_VECTOR);
  VECTOR = Λ;
  Vectors_Poolsize = 0;
  Vectors_Segment = HEAP_SEGMENT;
}
```

229. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
boolean llt_Grow_Vector_Pool_exec(lltf_Grow_Vector_Pool fix)
{
  char msg[TEST_BUFSIZE] = {0};
  boolean ok;
  jmp_buf save_jmp;
  if (fix.prepare) fix.prepare((lltf_Base *) &fix);
  Allocate_Success = fix.allocations;
  memcpy(&save_jmp, &Goto_Begin, sizeof(jmp_buf));
  if (¬setjmp(Goto_Begin)) new_vector_segment();
  Allocate_Success = −1;
  memcpy(&Goto_Begin, &save_jmp, sizeof(jmp_buf));
  switch (fix.expect) {
  case LLTF_GROW_VECTOR_POOL_SUCCESS:
    ⟨ Unit test vector allocations, validate success 230 ⟩
    break;    /* TODO: test for bzero */
  case LLTF_GROW_VECTOR_POOL_FAIL:
    ⟨ Unit test vector allocations, validate failure 231 ⟩
    break;
  }
  if (fix.destroy) fix.destroy((lltf_Base *) &fix);
  return ok;
}
```

230. ⟨ Unit test vector allocations, validate success 230 ⟩ ≡

```
ok = tap_ok(Vectors_Poolsize ≡ (fix.Poolsize + fix.Segment), test_msgf(msg, fix.name,
    "Vectors_Poolsize␣is␣increased"));
tap_more(ok, Vectors_Segment ≡ (fix.Poolsize + fix.Segment)/2, test_msgf(msg, fix.name,
    "Vectors_Segment␣is␣increased"));
tap_more(ok, VECTOR ≠ Λ, test_msgf(msg, fix.name, "VECTOR␣is␣not␣NULL"));
tap_more(ok, ¬bcmp(VECTOR, fix.save_VECTOR, sizeof(cell) * fix.Poolsize), test_msgf(msg, fix.name,
    "VECTOR␣heap␣is␣unchanged"));
```

This code is used in section 229.

231. ⟨ Unit test vector allocations, validate failure 231 ⟩ ≡

```
ok = tap_ok(Vectors_Poolsize ≡ fix.Poolsize, test_msgf(msg, fix.name,
    "Vectors_Poolsize␣is␣not␣increased"));
tap_more(ok, Vectors_Segment ≡ fix.Segment, test_msgf(msg, fix.name,
    "Vectors_Segment␣is␣not␣increased"));
tap_more(ok, VECTOR ≡ fix.VECTOR, test_msgf(msg, fix.name, "VECTOR␣is␣unchanged"));
```

This code is used in section 229.

232. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
boolean llt_Grow_Vector_Pool__Empty_Success(void)
{
  lltf_Grow_Vector_Pool fix = llt_Grow_Vector_Pool_fix(__func__);
  return llt_Grow_Vector_Pool_exec(fix);
}
```

233. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
boolean llt_Grow_Vector_Pool__Empty_Fail(void)
{
  lltf_Grow_Vector_Pool fix = llt_Grow_Vector_Pool_fix(__func__);
  fix.expect = LLTF_GROW_VECTOR_POOL_FAIL;
  fix.allocations = 0;
  return llt_Grow_Vector_Pool_exec(fix);
}
```

234. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
void lltf_Grow_Vector_Pool__fill(lltf_Grow_Vector_Pool *fix)
{
  size_t i;
  fix→VECTOR = reallocarray(Λ, fix→Poolsize, sizeof(cell));
  for (i = 0; i < (fix→Poolsize * sizeof(cell))/sizeof(int); i++) *(((int *) fix→VECTOR) + i) = rand();
}
```

235. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
boolean llt_Grow_Vector_Pool__Full_Success(void)
{
  lltf_Grow_Vector_Pool fix = llt_Grow_Vector_Pool_fix(__func__);
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Vector_Pool__fill(&fix);
  return llt_Grow_Vector_Pool_exec(fix);
}
```

236. ⟨ Unit test the vector allocator 224 ⟩ +≡

```
boolean llt_Grow_Vector_Pool__Full_Fail(void)
{
  lltf_Grow_Vector_Pool fix = llt_Grow_Vector_Pool_fix(__func__);
  fix.expect = LLTF_GROW_VECTOR_POOL_FAIL;
  fix.allocations = 0;
  fix.Poolsize = HEAP_SEGMENT;
  lltf_Grow_Vector_Pool__fill(&fix);
  return llt_Grow_Vector_Pool_exec(fix);
}
```

237. Garbage Collector.

238. Objects.

239. Pair Integration. With the basic building blocks' interactions tested we arrive at the critical integration between the compiler and the interpreter.

Calling the following tests integration tests may be thought of as a bit of a misnomer; if so consider them unit tests of the integration tests which are to follow in pure LossLess code.

Starting with *pair*s tests that *cons*, *car*, *cdr*, *null?*, *pair?*, *set-car!* & *set-cdr!* return their result and don't do anything strange. This code is extremely boring and repetetive.

```
⟨ t/pair.c 239 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    boolean ok, okok;
    cell marco, polo, t, water;      /* t is not saved from destruction */
    char *prefix = Λ;
    char msg[TEST_BUFSIZE] = {0};
    marco = sym("marco?");
    polo = sym("polo!");
    water = sym("fish␣out␣of␣water!");
    ⟨ Test integrating cons 240 ⟩
    ⟨ Test integrating car 241 ⟩
    ⟨ Test integrating cdr 242 ⟩
    ⟨ Test integrating null? 243 ⟩
    ⟨ Test integrating pair? 246 ⟩
    ⟨ Test integrating set-car! 249 ⟩
    ⟨ Test integrating set-cdr! 250 ⟩
  }
```

240. These tests could perhaps be made more thorough but I'm not sure what it would achieve. Testing the non-mutating calls is basically the same: Prepare & interpret code that will call the operator and then test that the result is correct and that internal state is (not) changed as expected.

```
⟨ Test integrating cons 240 ⟩ ≡
  vm_reset();
  Acc = read_cstring(prefix = "(cons␣24␣42)");
  interpret();
  ok = tap_ok(pair_p(Acc), tmsgf("pair?"));
  tap_again(ok, integer_p(car(Acc)) ∧ int_value(car(Acc)) ≡ 24, tmsgf("car"));
  tap_again(ok, integer_p(car(Acc)) ∧ int_value(cdr(Acc)) ≡ 42, tmsgf("cdr"));
  test_vm_state_full(prefix);
```

This code is used in section 239.

241.
```
⟨ Test integrating car 241 ⟩ ≡
  vm_reset();
  t = cons(int_new(42), polo);
  t = cons(synquote_new(t), NIL);
  Tmp_Test = Acc = cons(sym("car"), t);
  prefix = "(car␣'(42␣.␣polo))";
  interpret();
  tap_ok(integer_p(Acc) ∧ int_value(Acc) ≡ 42, tmsgf("integer?"));
  test_vm_state_full(prefix);
```

This code is used in section 239.

242. ⟨ Test integrating cdr 242 ⟩ ≡

```
vm_reset();
Acc = cons(sym("cdr"), t);
prefix = "(cdr␣'(42␣.␣polo))";
interpret();
tap_ok(symbol_p(Acc) ∧ Acc ≡ polo, tmsgf("symbol?"));
test_vm_state_full(prefix);
```

This code is used in section 239.

243. ⟨ Test integrating null? 243 ⟩ ≡

```
vm_reset();
t = cons(NIL, NIL);
Acc = cons(sym("null?"), t);
prefix = "(null?␣())";
interpret();
tap_ok(true_p(Acc), tmsgf("true?"));
test_vm_state_full(prefix);
```

See also sections 244 and 245.

This code is used in section 239.

244. ⟨ Test integrating null? 243 ⟩ +≡

```
vm_reset();
t = cons(synquote_new(polo), NIL);
Acc = cons(sym("null?"), t);
prefix = "(null?␣'polo!)";
interpret();
tap_ok(false_p(Acc), tmsgf("false?"));
test_vm_state_full(prefix);
```

245. ⟨ Test integrating null? 243 ⟩ +≡

```
vm_reset();
t = synquote_new(cons(NIL, NIL));
Acc = cons(sym("null?"), cons(t, NIL));
prefix = "(null?␣'(()))";
interpret();
tap_ok(false_p(Acc), tmsgf("false?"));
test_vm_state_full(prefix);
```

246. ⟨ Test integrating pair? 246 ⟩ ≡

```
vm_reset();
Acc = cons(sym("pair?"), cons(NIL, NIL));
prefix = "(pair?␣())";
interpret();
tap_ok(false_p(Acc), tmsgf("false?"));
test_vm_state_full(prefix);
```

See also sections 247 and 248.

This code is used in section 239.

247. ⟨ Test integrating pair? 246 ⟩ +≡

```
vm_reset();
t = cons(synquote_new(polo), NIL);
Acc = cons(sym("pair?"), t);
prefix = "(pair?␣'polo!)";
interpret();
tap_ok(false_p(Acc), tmsgf("false?"));
test_vm_state_full(prefix);
```

248. ⟨ Test integrating pair? 246 ⟩ +≡

```
vm_reset();
t = synquote_new(cons(NIL, NIL));
Acc = cons(sym("pair?"), cons(t, NIL));
prefix = "(pair?␣'(()))";
interpret();
tap_ok(true_p(Acc), tmsgf("true?"));
test_vm_state_full(prefix);
```

249. Testing that pair mutation works correctly requires some more work. A pair is created and saved in *Tmp_Test* then the code which will be interpreted is created by hand to inject that pair directly and avoid looking for its value in an *environment*.

TODO: duplicate these tests for symbols that are looked up.

⟨ Test integrating set-car! 249 ⟩ ≡

```
vm_reset();
Tmp_Test = cons(marco, water);
t = cons(synquote_new(polo), NIL);
t = cons(synquote_new(Tmp_Test), t);
Acc = cons(sym("set-car!"), t);
prefix = "(set-car!␣'(marco␣.␣|fish␣out␣of␣water!|)␣'polo!)";
interpret();
ok = tap_ok(void_p(Acc), tmsgf("void?"));
okok = tap_ok(ok ∧ pair_p(Tmp_Test), tmsgf("(pair?␣T)"));
tap_again(ok, symbol_p(car(Tmp_Test)) ∧ car(Tmp_Test) ≡ polo, tmsgf("(eq?␣(car␣T)␣'polo!)"));
tap_again(okok, symbol_p(cdr(Tmp_Test)) ∧ cdr(Tmp_Test) ≡ water,
    tmsgf("(eq?␣(cdr␣T)␣'|fish␣out␣of␣water!|)"));
```

This code is used in section 239.

250. ⟨ Test integrating set-cdr! 250 ⟩ ≡

```
vm_reset();
Tmp_Test = cons(water, marco);
t = cons(synquote_new(polo), NIL);
t = cons(synquote_new(Tmp_Test), t);
Acc = cons(sym("set-cdr!"), t);
prefix = "(set-cdr!␣'(|fish␣out␣of␣water!|␣.␣marco)␣'polo!)";
interpret();
ok = tap_ok(void_p(Acc), tmsgf("void?"));
okok = tap_ok(ok ∧ pair_p(Tmp_Test), tmsgf("(pair?␣T)"));
tap_again(ok, symbol_p(car(Tmp_Test)) ∧ car(Tmp_Test) ≡ water,
    tmsgf("(eq?␣(car␣T)␣'|fish␣out␣of␣water!|)"));
tap_again(okok, symbol_p(cdr(Tmp_Test)) ∧ cdr(Tmp_Test) ≡ polo, tmsgf("(eq?␣(cdr␣T)␣'polo!)"));
```

This code is used in section 239.

251. Integrating eval. Although useful to write, and they weeded out some dumb bugs, the real difficulty is in ensuring the correct *environment* is in place at the right time.

We'll skip **error** for now and start with **eval**. Again this test isn't thorough but I think it's good enough for now. The important tests are that the arguments to **eval** are evaluated in the compile-time environment in which the **eval** is located, and that the program which the first argument evaluates to is itself evaluated in the environment the second argument evaluates to.

```
⟨ t/eval.c 251 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    cell t, m, p;
    char *prefix;
    char msg[TEST_BUFSIZE] = {0};
    ⟨ Test integrating eval 252 ⟩
  }
```

See also section 257.

252. The first test of **eval** calls into it without needing to look up any of its arguments. The program to be evaluated calls *test!probe* and its result is examined. First evaluating in the current environment which is here *Root*.

```
⟨ Test integrating eval 252 ⟩ ≡
  vm_reset();
  Acc = read_cstring((prefix = "(eval␣'(test!probe))"));
  interpret();
  t = assoc_value(Acc, sym("Env"));
  tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
  tap_ok(t ≡ Root, tmsgf("(eq?␣(assoc-value␣T␣'Env)␣Root)"));
    /* TODO: Is it worth testing that Acc ≡ Prog ≡ [ OP_TEST_PROBE OP_RETURN ]? */
  test_vm_state_full(prefix);
```

See also sections 253, 254, and 255.

This code is used in section 251.

253. And then testing with a second argument of an artificially-constructed environment.

The probing symbol is given a different name to shield against it being found in *Root* and fooling the tests into passing.

```
⟨ Test integrating eval 252 ⟩ +≡
  vm_reset();
  Tmp_Test = env_empty();
  env_set(Tmp_Test, sym("alt-test!probe"), env_search(Root, sym("test!probe")), TRUE);
  Acc = read_cstring((prefix = "(eval␣'(alt-test!probe)␣E)"));
  cddr(Acc) = cons(Tmp_Test, NIL);
  interpret();
  t = assoc_value(Acc, sym("Env"));
  tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
  tap_ok(t ≡ Tmp_Test, tmsgf("(eq?␣(assoc-value␣T␣'Env)␣E)"));
  test_vm_state_full(prefix);
```

254. Testing that **eval**'s arguments are evaluated in the correct *environment* is a little more difficult. The *environment* with variables to supply **eval**'s arguments is constructed. These are the program source and another artificial *environment* which the program should be evaluated in.

t, m & p are protected throughout as they are only links to somewhere in the outer *environment* which is protected by *Tmp_Test*.

⟨ Test integrating **eval** 252 ⟩ +≡

```
Tmp_Test = env_empty();   /* outer environment */
env_set(Tmp_Test, sym("eval"), env_search(Root, sym("eval")), TRUE);
env_set(Tmp_Test, sym("alt-test!probe"), env_search(Root, sym("error")), TRUE);
t = read_cstring("(alt-test!probe␣'oops)");   /* program; oops in case we end up in error */
env_set(Tmp_Test, sym("testing-program"), t, TRUE);
m = env_empty();   /* evaluation environmant */
env_set(Tmp_Test, sym("testing-environment"), m, TRUE);
env_set(m, sym("alt-test!probe"), env_search(Root, sym("test!probe")), TRUE);
env_set(m, sym("testing-environment"), env_empty(), TRUE);
p = read_cstring("(error␣wrong-program)");
env_set(m, sym("testing-program"), p, TRUE);
```

255. **eval** is then called in the newly-constructed *environment* by putting it in *Env* before calling *interpret*, mimicking what *frame_push* would do when entering the closure the *environment* represents.

⟨ Test integrating **eval** 252 ⟩ +≡

```
vm_reset();
prefix = "(eval␣testing-program␣testing-environment)";
Acc = read_cstring(prefix);
Env = Tmp_Test;
interpret();
t = assoc_value(Acc, sym("Env"));
tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
tap_ok(t ≡ m, tmsgf("(eq?␣(assoc-value␣T␣'Env)␣E)"));
test_integrate_eval_unchanged(prefix, Tmp_Test, m);
test_vm_state_normal(prefix);
tap_ok(Env ≡ Tmp_Test, tmsgf("(unchanged?␣Env)"));
```

256. Neither of the two environments should be changed at all. That is *inner* should have exactly `alt-test!probe`, `testing-environment` & `testing-program`, *outer* should have the same symbols with the different values as above and also **eval**.

⟨ Function declarations 8 ⟩ +≡

```
#define TEST_EVAL_FOUND(var)
  if (undefined_p(var)) (var) = cadar(t);
  else fmore = btrue;
#define TEST_EVAL_FIND
  feval = fprobe = fenv = fprog = UNDEFINED;
  fmore = bfalse;
  while (¬null_p(t)) {
    if (caar(t) ≡ sym("alt-test!probe")) { TEST_EVAL_FOUND(fprobe); }
    else if (caar(t) ≡ sym("eval")) { TEST_EVAL_FOUND(feval); }
    else if (caar(t) ≡ sym("testing-environment")) { TEST_EVAL_FOUND(fenv); }
    else if (caar(t) ≡ sym("testing-program")) { TEST_EVAL_FOUND(fprog); }
    else fmore = btrue;
    t = cdr(t);
  }
  void test_integrate_eval_unchanged(char *, cell, cell);
```

257. ⟨ t/eval.c 251 ⟩ +≡

```
  void test_integrate_eval_unchanged(char *prefix, cell outer, cell inner)
  {
    boolean oki, oko, fmore;
    cell fenv, feval, fprobe, fprog;
    cell oeval, oprobe;
    cell iprobe;
    cell t;
    char msg[TEST_BUFSIZE] = {0};
    ⟨ Test the outer environment when testing eval 258 ⟩
    ⟨ Test the inner environment when testing eval 259 ⟩
  }
```

258. ⟨ Test the outer environment when testing **eval** 258 ⟩ ≡

```
  oko = tap_ok(environment_p(outer), tmsgf("(environment?␣outer)"));
  tap_ok(env_root_p(outer), tmsgf("(environment.is-root?␣outer)"));
  if (oko) {
    oeval = env_search(Root, sym("eval"));
    oprobe = env_search(Root, sym("error"));
    t = env_layer(outer);
    TEST_EVAL_FIND
    if (¬undefined_p(fprog)) oki = list_p(fprog, FALSE, &t) ∧ int_value(t) ≡ 2;
        /* TODO: write for match(fprog, read_cstring("(alt-test!probe␣'oops)")) */
  }
  tap_again(oko, ¬fmore ∧ feval ≡ oeval ∧ fprobe ≡ oprobe ∧ fenv ≡ inner,
      tmsgf("outer␣environment␣is␣unchanged"));
```

This code is used in section 257.

259. ⟨ Test the inner environment when testing **eval** 259 ⟩ ≡

```
oki = tap_ok(environment_p(inner), tmsgf("(environment?␣inner)"));
tap_ok(env_root_p(inner), tmsgf("(environment.is-root?␣inner)"));
if (oki) {
  iprobe = env_search(Root, sym("test!probe"));
  t = env_layer(inner);
  TEST_EVAL_FIND
  if (¬undefined_p(fprog)) oki = list_p(fprog, FALSE, &t) ∧ int_value(t) ≡ 2;
}
tap_again(oki, ¬fmore ∧ undefined_p(feval) ∧ fprobe ≡ iprobe ∧ env_empty_p(fenv),
    tmsgf("inner␣environment␣is␣unchanged"));
```

This code is used in section 257.

260. Conditional Integration. Before testing conditional interaction with *environments* it's reassuring to know that **if**'s syntax works the way that's expected of it, namely that when only the conequent is provided without an alternate it is as though the alternate was the value `VOID`, and that a call to it has no unexpected side-effects.

```
⟨ t/if.c 260 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    cell fcorrect, tcorrect, fwrong, twrong;
    cell talt, tcons, tq;
    cell marco, polo, t;
    char *prefix = Λ;
    char msg[TEST_BUFSIZE] = {0};
    fcorrect = sym("correct-false");
    fwrong = sym("wrong-false");
    tcorrect = sym("correct-true");
    twrong = sym("wrong-true");
    talt = sym("test-alternate");
    tcons = sym("test-consequent");
    tq = sym("test-query");
    marco = sym("marco?");
    polo = sym("polo!");
    ⟨ Sanity test if's syntax 261 ⟩
    ⟨ Test integrating if 265 ⟩
  }
```

261. Four tests make sure **if**'s arguments work as advertised. These are the only tests of the 2-argument form of **if**.

`(if #t 'polo!)` ⇒ `polo!`:

```
⟨ Sanity test if's syntax 261 ⟩ ≡
  vm_reset();
  t = cons(synquote_new(polo), NIL);
  t = cons(TRUE, t);
  Acc = cons(sym("if"), t);
  prefix = "(if␣#t␣'polo!)";
  interpret();
  tap_ok(symbol_p(Acc) ∧ Acc ≡ polo, tmsgf("symbol?"));
  test_vm_state_full(prefix);
```

See also sections 262, 263, and 264.

This code is used in section 260.

262. `(if #f 'marco?)` ⇒ `VOID`:

```
⟨ Sanity test if's syntax 261 ⟩ +≡
  vm_reset();
  t = cons(synquote_new(marco), NIL);
  t = cons(FALSE, t);
  Acc = cons(sym("if"), t);
  prefix = "(if␣#f␣'marco?)";
  interpret();
  tap_ok(void_p(Acc), tmsgf("void?"));
  test_vm_state_full(prefix);
```

263. `(if #t 'marco? 'polo!)` ⇒ `marco?`:

⟨ Sanity test **if**'s syntax 261 ⟩ +≡
  vm_reset( );
  t = *cons*(*synquote_new*(*polo*), NIL);
  t = *cons*(*synquote_new*(*marco*), t);
  t = *cons*(TRUE, t);
  Acc = *cons*(*sym*("if"), t);
  prefix = "(if␣#t␣'marco?␣'polo!)";
  interpret( );
  tap_ok(*symbol_p*(*Acc*) ∧ *Acc* ≡ *marco*, *tmsgf*("symbol?"));
  test_vm_state_full(*prefix*);

264. `(if #f 'marco? 'polo!)` ⇒ `polo!`:

⟨ Sanity test **if**'s syntax 261 ⟩ +≡
  vm_reset( );
  t = *cons*(*synquote_new*(*polo*), NIL);
  t = *cons*(*synquote_new*(*marco*), t);
  t = *cons*(FALSE, t);
  Acc = *cons*(*sym*("if"), t);
  prefix = "(if␣#f␣'marco?␣'polo!)";
  interpret( );
  tap_ok(*symbol_p*(*Acc*) ∧ *Acc* ≡ *polo*, *tmsgf*("symbol?"));
  test_vm_state_full(*prefix*);

265. To confirm that **if**'s arguments are evaluated in the correct *environment* *Root* is replaced with a duplicate and invalid variants of the symbols inserted into it. This is then extended into a new *environment* with the desired version of the four symbols **if**, `test-query`, `test-consequent` and `test-alternate`.

⟨ Test integrating **if** 265 ⟩ ≡
  t = *env_layer*(*Tmp_Test* = *Root*);
  Root = *env_empty*( );
  for (; ¬*null_p*(t); t = *cdr*(t))
    if (*caar*(t) ≠ *sym*("if")) *env_set*(*Root*, *caar*(t), *cadar*(t), *btrue*);
  env_set(*Root*, *sym*("if"), *env_search*(*Tmp_Test*, *sym*("error")), *btrue*);
  env_set(*Root*, *talt*, *fwrong*, *btrue*);
  env_set(*Root*, *tcons*, *twrong*, *btrue*);
  env_set(*Root*, *tq*, VOID, *btrue*);
  Env = *env_extend*(*Root*);
  env_set(*Env*, *sym*("if"), *env_search*(*Tmp_Test*, *sym*("if")), *btrue*);
  env_set(*Env*, *talt*, *fcorrect*, *btrue*);
  env_set(*Env*, *tcons*, *tcorrect*, *btrue*);
  env_set(*Env*, *tq*, VOID, *btrue*);

See also sections 266, 267, and 268.

This code is used in section 260.

266. The test is performed with *test-query* resolving to `#f` & `#t`.

⟨ Test integrating **if** 265 ⟩ +≡
```
vm_reset();
env_set(Env, tq, FALSE, bfalse);
t = cons(talt, NIL);
t = cons(tcons, t);
t = cons(tq, t);
Acc = cons(sym("if"), t);
prefix = "(let␣((query␣#f))␣(if␣query␣consequent␣alternate))";
t = Env;
interpret();
tap_ok(symbol_p(Acc) ∧ Acc ≡ fcorrect, tmsgf("symbol?"));
test_vm_state_normal(prefix);
tap_ok(Env ≡ t, tmsgf("(unchanged?␣Env)"));
```

267. ⟨ Test integrating **if** 265 ⟩ +≡
```
vm_reset();
env_set(Env, tq, TRUE, bfalse);
t = cons(talt, NIL);
t = cons(tcons, t);
t = cons(tq, t);
Acc = cons(sym("if"), t);
prefix = "(let␣((query␣#t))␣(if␣query␣consequent␣alternate))";
t = Env;
interpret();
tap_ok(symbol_p(Acc) ∧ Acc ≡ tcorrect, tmsgf("symbol?"));
test_vm_state_normal(prefix);
tap_ok(Env ≡ t, tmsgf("(unchanged?␣Env)"));
```

268. It is important that the real *Root* is restored at the end of these tests in order to perform any more testing.

⟨ Test integrating **if** 265 ⟩ +≡
```
Root = Tmp_Test;
```

269. Applicatives. Testing **lambda** here is mostly concerned with verifying that the correct environment is stored in the closure it creates and then extended when it is entered.

These tests (and **vov**, below) could be performed using higher-level testing and *current-environment* but a) there is no practically usable `LossLess` language yet and b) I have a feeling I may want to write deeper individual tests.

```
⟨ t/lambda.c 269 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    boolean ok;
    cell ie, oe, len;
    cell t, m, p;
    cell sn, si, sin, sinn, so, sout, soutn;
    char *prefix;
    char msg[TEST_BUFSIZE] = {0};
      /* Although myriad these variables' scope is small and they are not used between the sections */
    sn = sym("n");
    si = sym("inner");
    sin = sym("in");
    sinn = sym("in-n");
    so = sym("outer");
    sout = sym("out");
    soutn = sym("out-n");
    ⟨ Test calling lambda 270 ⟩
    ⟨ Test entering an applicative closure 271 ⟩
    ⟨ Applicative test passing an applicative 272 ⟩
    ⟨ Applicative test passing an operative 273 ⟩
    ⟨ Applicative test returning an applicative 274 ⟩
    ⟨ Applicative test returning an operative 275 ⟩
  }
```

270. An applicative closes over the local *environment* that was active at the point **lambda** was compiled.

```
#define TEST_AB "(lambda␣x)"
#define TEST_AB_PRINT "(lambda␣x␣...)"

⟨ Test calling lambda 270 ⟩ ≡
  Env = env_extend(Root);
  Tmp_Test = test_copy_env();
  Acc = read_cstring(TEST_AB);
  prefix = TEST_AB_PRINT;
  vm_reset();
  interpret();
  ok = tap_ok(applicative_p(Acc), tmsgf("applicative?"));
  tap_again(ok, applicative_formals(Acc) ≡ sym("x"), tmsgf("formals"));
  if (ok) t = applicative_closure(Acc);
  tap_again(ok, environment_p(car(t)), tmsgf("environment?"));
  tap_again(ok, test_is_env(car(t), Tmp_Test), tmsgf("closure"));
  if (ok) t = cdr(t);
  tap_again(ok, car(t) ≠ Prog, tmsgf("prog"));    /* & what? */
  test_vm_state_normal(prefix);
  tap_ok(test_compare_env(Tmp_Test), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

271. When entering an applicative closure the *environment* it closed over at compile-time is extended (into a new frame which is removed when leaving the closure).

```
#define TEST_AC "(lambda␣x␣(test!probe))"
#define TEST_AC_PRINT "("TEST_AC")"

⟨ Test entering an applicative closure 271 ⟩ ≡
  Env = env_extend(Root);
  Tmp_Test = cons(test_copy_env(), NIL);
  Acc = read_cstring(TEST_AC);
  vm_reset();
  interpret();
  Env = env_extend(Root);
  cdr(Tmp_Test) = test_copy_env();
  t = read_cstring("(LAMBDA)");
  car(t) = Acc;
  Acc = t;
  prefix = TEST_AC_PRINT;
  vm_reset();
  interpret();
  t = assoc_value(Acc, sym("Env"));
  ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
  tap_again(ok, test_is_env(env_parent(t), car(Tmp_Test)),
      tmsgf("(eq?␣(assoc-value␣T␣'Env)␣(env.parent␣E))"));
  test_vm_state_normal(prefix);
  tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

272. Given that we can compile and enter an applicative closure, this test assures that we can correctly enter a closure that's passed as an argument to it. The expression being evaluated is: `((lambda`$_0$ `(L`$_1$ `. x0) (L`$_1$ `(test!probe`$_0$`))) (lambda`$_1$ `(T`$_0$ `. x1) (test!probe`$_1$`)))` except that the same technique as the previous test compiles each expression in its own *environment*.

Entering the outer closure extends the *environment* E_0 to E_1 which will be contained in the probe result that's an argument to the inner closure.

Entering the inner closure extends its *environment* E_2 to E_3.

```
#define TEST_ACA_INNER "(lambda␣(T␣.␣x1)␣(test!probe))"
#define TEST_ACA_OUTER "(lambda␣(L␣.␣x0)␣(L␣(test!probe)))"
#define TEST_ACA "("TEST_ACA_OUTER"LAMBDA)"
#define TEST_ACA_PRINT "("TEST_ACA_OUTER"␣(LAMBDA))"
```

⟨ Applicative test passing an *applicative* 272 ⟩ ≡

```
Env = env_extend(Root);    /* E2 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ACA_INNER);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);    /* E0 */
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ACA);
cadr(Acc) = vms_pop();
prefix = TEST_ACA_PRINT;
vm_reset();
interpret();
t = assoc_value(Acc, sym("Env"));    /* E3 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣inner)"));
if (ok) p = env_search(t, sym("T"));
if (ok) m = assoc_value(p, sym("Env"));    /* E1 */
tap_again(ok, environment_p(m), tmsgf("(environment?␣outer)"));
tap_again(ok, m ≠ t, tmsgf("(eq?␣outer␣inner)"));
tap_again(ok, test_is_env(env_parent(m), cdr(Tmp_Test)), tmsgf("(parent?␣outer)"));
tap_again(ok, test_is_env(env_parent(t), car(Tmp_Test)), tmsgf("(parent?␣inner)"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

273. This is the same test, passing/entering an *operative*. The key difference is that the inner *operative* must evaluate its arguments itself. Additionally *test!probe* is an operative so an applicative variant is called: `(vov ((A vov/args) (E vov/env)) (test!probe-applying (eval (car A) E)))`.

The same *environments* are in play as in the previous test with the addition that E_1 will be passed into the inner closure in *vov/environment*.

```
#define TEST_ACO_INNER_BODY "(test!probe-applying␣(eval␣(car␣A)␣E))"
#define TEST_ACO_INNER "(vov␣((A␣vov/args)␣(E␣vov/env))"
          TEST_ACO_INNER_BODY")"
#define TEST_ACO_OUTER "(lambda␣(V␣.␣x0)␣(V␣(test!probe)))"
#define TEST_ACO "("TEST_ACO_OUTER"VOV)"
#define TEST_ACO_PRINT "((LAMBDA)␣(vov␣(...)␣"TEST_ACO_INNER_BODY")"
```

⟨ Applicative test passing an *operative* 273 ⟩ ≡

```
Env = env_extend(Root);    /* E2 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ACO_INNER);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);    /* E0 */
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ACO);
cadr(Acc) = vms_pop();
prefix = TEST_ACO_PRINT;
vm_reset();
interpret();
t = assoc_value(Acc, sym("Env"));    /* E3 */
p = car(assoc_value(Acc, sym("Args")));
m = assoc_value(p, sym("Env"));    /* E1 */
ok = tap_ok(environment_p(m), tmsgf("(environment?␣outer)"));
tap_again(ok, test_is_env(env_parent(m), cdr(Tmp_Test)), tmsgf("(parent?␣outer)"));
ok = tap_ok(environment_p(t), tmsgf("(environment?␣inner)"));
if (ok) p = env_search(t, sym("E"));    /* E1 */
tap_again(ok, environment_p(p), tmsgf("(environment?␣E)"));
tap_again(ok, test_is_env(p, m), tmsgf("operative␣environment"));
tap_ok(¬test_is_env(m, t), tmsgf("(eq?␣outer␣inner)"));
tap_again(ok, test_is_env(env_parent(t), car(Tmp_Test)), tmsgf("(parent?␣inner)"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

274. Similar to applicatives which call into another closure are applicatives which return one. Starting with an *applicative*-returning-*applicative* `(lambda (outer n) (lambda (inner n) (test!probe)))`.

This is a function which takes two arguments, *outer* and *n* and creates another function which closes over them and takes two of its own arguments, *inner* and *n*.

The test calls this by evaluating `((X 'out 'out-n) 'in 'in-n)` with the above code inserted in the `X` position.

When the inner lambda is evaluating *test!probe* its local *environment* E_2 should be an extension of the dynamic *environment* E_1 that was created when entering the outer closure. E_1 should be an extension of the run-time *environment* E_0 when the closure was built.

```
#define TEST_ARA_INNER "(lambda␣(inner␣n)␣(test!probe))"
#define TEST_ARA_BUILD "(lambda␣(outer␣n)␣"TEST_ARA_INNER")"
#define TEST_ARA_PRINT TEST_ARA_BUILD
#define TEST_ARA_CALL "((LAMBDA␣'out␣'out-n)␣'in␣'in-n)"
```

⟨ Applicative test returning an *applicative* 274 ⟩ ≡

```
Env = env_extend(Root);     /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ARA_BUILD);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ARA_CALL);
caar(Acc) = vms_pop();
prefix = TEST_ARA_PRINT;
vm_reset();
interpret();
ie = assoc_value(Acc, sym("Env"));     /* E2 */
ok = tap_ok(environment_p(ie), tmsgf("(environment?␣inner)"));
tap_again(ok, env_search(ie, sn) ≡ sinn, tmsgf("(eq?␣n␣'in-n)"));
tap_again(ok, env_search(ie, si) ≡ sin, tmsgf("(eq?␣inner␣'in)"));
tap_again(ok, env_search(ie, so) ≡ sout, tmsgf("(eq?␣outer␣'out)"));
if (ok) oe = env_parent(ie);     /* E1 */
tap_again(ok, environment_p(oe), tmsgf("(environment?␣outer)"));
tap_again(ok, env_search(oe, sn) ≡ soutn, tmsgf("(eq?␣n␣'out-n)"));
tap_again(ok, undefined_p(env_search(oe, si)), tmsgf("(defined?␣inner)"));
tap_again(ok, env_search(oe, so) ≡ sout, tmsgf("(eq?␣outer␣'out)"));
tap_again(ok, test_is_env(env_parent(oe), car(Tmp_Test)), tmsgf("(parent?␣outer)"));     /* E0 */
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

275. Finally, an applicative closing over an operative it returns looks similar: `(vov ((A vov/args) (E vov/env)) (test!probe-applying A E))`

Again the same *environments* are in play although this time the operative's arguments are unevaluated and E_3, the run-time environment, is passed in *vov/environment*.

```
#define TEST_ARO_INNER_BODY "(test!probe-applying␣A␣E)"
#define TEST_ARO_INNER "(vov␣((A␣vov/args)␣(E␣vov/env))"TEST_ARO_INNER_BODY")"
#define TEST_ARO_BUILD "(lambda␣(outer␣n)"TEST_ARO_INNER")"
#define TEST_ARO_CALL "((LAMBDA␣'out␣'out-n)␣'in␣'in-n)"
#define TEST_ARO_PRINT "(LAMBDA␣(vov␣(...)␣"TEST_ARO_INNER_BODY"))"
```

⟨ Applicative test returning an *operative* 275 ⟩ ≡

```
Env = env_extend(Root);     /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ARO_BUILD);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);     /* E3 */
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ARO_CALL);
caar(Acc) = vms_pop();
prefix = TEST_ARO_PRINT;
vm_reset();
interpret();
ie = assoc_value(Acc, sym("Env"));     /* E2 */
ok = tap_ok(environment_p(ie), tmsgf("(environment?␣inner)"));
tap_again(ok, undefined_p(env_here(ie, sn)), tmsgf("(lifted?␣n)"));
tap_again(ok, undefined_p(env_here(ie, so)), tmsgf("(lifted?␣outer)"));
tap_again(ok, env_search(ie, sn) ≡ soutn, tmsgf("(eq?␣n␣'out-n)"));
tap_again(ok, env_search(ie, so) ≡ sout, tmsgf("(eq?␣outer␣'out)"));
if (ok) oe = env_parent(ie);     /* E1 */
tap_again(ok, environment_p(oe), tmsgf("(environment?␣outer)"));
tap_again(ok, env_search(ie, sn) ≡ soutn, tmsgf("(eq?␣n␣'out-n)"));
tap_again(ok, env_search(ie, so) ≡ sout, tmsgf("(eq?␣outer␣'out)"));
tap_again(ok, undefined_p(env_search(oe, sym("A"))), tmsgf("(defined?␣A)"));
tap_again(ok, undefined_p(env_search(oe, sym("E"))), tmsgf("(defined?␣E)"));
tap_again(ok, test_is_env(env_parent(oe), car(Tmp_Test)), tmsgf("(parent?␣outer)"));     /* E0 */
if (ok) t = env_search(ie, sym("A"));
tap_again(ok, true_p(list_p(t, FALSE, &len)), tmsgf("(list?␣A)"));
tap_again(ok, int_value(len) ≡ 2, tmsgf("length"));
tap_again(ok, syntax_p(car(t)) ∧ cdar(t) ≡ sin ∧ syntax_p(cadr(t)) ∧ cdadr(t) ≡ sinn,
      tmsgf("unevaluated"));
tap_again(ok, test_is_env(env_search(ie, sym("E")), cdr(Tmp_Test)), tmsgf("(eq?␣E␣Env)"));
   /* E3 */
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 269.

276. Operatives. Testing **vov** follows the same plan as **lambda** with the obvious changes to which environment is expected to be found where and care taken to ensure that arguments are evaluated when appropriate.

```
⟨ t/vov.c 276 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    boolean ok;
    cell t, m, p;
    cell sn, si, sin, sinn, so, sout, soutn;
    char *prefix;
    char msg[TEST_BUFSIZE] = {0};
    sn = sym("n");
    si = sym("inner");
    sin = sym("in");
    sinn = sym("in-n");
    so = sym("outer");
    sout = sym("out");
    soutn = sym("out-n");
    ⟨ Test calling vov 277 ⟩
    ⟨ Test entering an operative closure 278 ⟩
    ⟨ Operative test passing an applicative 279 ⟩
    ⟨ Operative test passing an operative 280 ⟩
    ⟨ Operative test returning an applicative 281 ⟩
    ⟨ Operative test returning an operative 282 ⟩
  }
```

277.

```
#define TEST_OB "(vov␣((E␣vov/env)))"
#define TEST_OB_PRINT "(vov␣((E␣vov/env))␣...)"
⟨ Test calling vov 277 ⟩ ≡
  Env = env_extend(Root);
  Tmp_Test = test_copy_env();
  Acc = read_cstring(TEST_OB);
  prefix = TEST_OB_PRINT;
  vm_reset();
  interpret();
  ok = tap_ok(operative_p(Acc), tmsgf("operative?"));
  tap_again(ok, pair_p(t = operative_formals(Acc)), tmsgf("formals"));
  if (ok) t = operative_closure(Acc);
  tap_again(ok, environment_p(car(t)), tmsgf("environment?"));
  tap_again(ok, car(t) ≡ Env, tmsgf("closure"));
  if (ok) t = cdr(t);
  tap_again(ok, car(t) ≠ Prog, tmsgf("prog"));     /* & what? */
  test_vm_state_normal(prefix);
  tap_ok(test_compare_env(Tmp_Test), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

278. Upon entering an operative closure:

1. The run-time *environment* E_0 when it was created is extended to a new *environment* E_1 contaning the 1-3 **vov** arguments.

2. The run-time *environment* E_2 when it was entered is passed to the **vov** in the argument in the *vov/environment* (or *vov/env*) position.

3. Upon leaving it the stack and the run-time *environment* are restored unchanged.

```
#define TEST_OC "(vov␣((A␣vov/args)␣(E␣vov/env))␣(test!probe-applying␣A␣E))"
#define TEST_OC_PRINT "((vov␣(...)␣(test!probe-applying␣A␣E)))"
```

⟨ Test entering an *operative* closure 278 ⟩ ≡

```
Env = env_extend(Root);     /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_OC);
vm_reset();
interpret();

Env = env_extend(Root);     /* E2 */
cdr(Tmp_Test) = test_copy_env();
t = read_cstring("(VOV)");
car(t) = Acc;
Acc = t;
prefix = TEST_OC_PRINT;
vm_reset();
interpret();

t = assoc_value(Acc, sym("Env"));     /* E1 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
tap_again(ok, test_is_env(env_parent(t), car(Tmp_Test)),
    tmsgf("(eq?␣(assoc-value␣T␣'Env)␣(env.parent␣E))"));
if (ok) p = env_search(t, sym("E"));     /* E2 */
tap_again(ok, environment_p(p), tmsgf("(environment?␣E)"));
tap_again(ok, test_is_env(p, cdr(Tmp_Test)), tmsgf("(eq?␣T␣(current-environment))"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

279. Calling an applicative inside an operative closure is no different from any other function call. An operative closure is entered with the result of **lambda** as an argument: `((VOV) (lambda x1 (test!probe)))`.

Operative's arguments are not evaluated so whether a **lambda** expression, variable lookup or whatever the operative evaluates its argument in the caller's *environment* then calls into it along with its own probe: `((vov (...) (cons ((eval (car A) E)) (test!probe))) (LAMBDA))`.

The operative's compile-time *environment* E_0 is extended up entering it to E_1. The run-time *environment* E_2 is extended when entering the callee's applicative and is passed to the operative.

```
#define TEST_OCA_INNER "(lambda␣x1␣(test!probe))"
#define TEST_OCA_OUTER
        "(vov␣((A␣vov/args)␣(E␣vov/env))""(cons␣((eval␣(car␣A)␣E))␣(test!probe)))"
#define TEST_OCA "("TEST_OCA_OUTER"LAMBDA)"
#define TEST_OCA_PRINT "((VOV)␣"TEST_OCA_INNER")"
```

⟨ Operative test passing an *applicative* 279 ⟩ ≡

```
Env = env_extend(Root);   /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_OCA_INNER);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);   /* E2 */
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_OCA);
cadr(Acc) = vms_pop();
prefix = TEST_OCA_PRINT;
vm_reset();
interpret();
t = assoc_value(cdr(Acc), sym("Env"));   /* E1 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣(cdr␣T)␣'Env))"));
tap_again(ok, test_is_env(env_parent(t), cdr(Tmp_Test)), tmsgf("(parent?␣E)"));   /* E0 */
tap_again(ok, test_is_env(env_search(t, sym("E")), cdr(Tmp_Test)), tmsgf("(eq?␣E␣vov/env)"));
p = assoc_value(car(Acc), sym("Env"));   /* E3 */
ok = tap_ok(environment_p(p), tmsgf("(environment?␣(assoc-value␣(car␣T)␣'Env))"));
tap_again(ok, test_is_env(env_parent(p), car(Tmp_Test)), tmsgf("(parent?␣E')"));   /* E2 */
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

280. To verify calling an operative argument to an operative closure there are three tests to perform:

1. The run-time *environment* E_2 in the inner operative is an extension of the one it was originally created with E_1.

2. The run-time *environment* E_1 in the outer operative is an extension of its compile-time *environment* E_0.

3. E_1 is the *vov/environment* argument of the inner operative.

```
#define TEST_OCO_INNER "(vov␣((yE␣vov/env))␣(test!probe))"
#define TEST_OCO_OUTER "(vov␣((xA␣vov/args)␣(xE␣vov/env))"
            "(cons␣((eval␣(car␣xA)␣xE))␣(test!probe)))"
#define TEST_OCO "("TEST_OCO_OUTERTEST_OCO_INNER")"
#define TEST_OCO_PRINT "((VOV)␣"TEST_OCO_INNER")"
```

⟨ Operative test passing an *operative* 280 ⟩ ≡

```
Env = env_extend(Root);    /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_OCO_INNER);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_OCO);
cadr(Acc) = vms_pop();
prefix = TEST_OCO_PRINT;
vm_reset();
interpret();
t = assoc_value(car(Acc), sym("Env"));    /* E2 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣(car␣T)␣'Env))"));
tap_again(ok, test_is_env(env_parent(t), car(Tmp_Test)), tmsgf("(parent?␣E)"));    /* E1 */
m = env_here(t, sym("yE"));    /* E1 */
tap_again(ok, ¬undefined_p(m), tmsgf("(env.exists?␣E␣yE)"));
p = assoc_value(cdr(Acc), sym("Env"));    /* E1 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣(cdr␣T)␣'Env))"));
tap_again(ok, test_is_env(m, p), tmsgf("operative␣environment"));
tap_ok(¬test_is_env(p, t), tmsgf("(eq?␣E'␣E)"));
tap_again(ok, test_is_env(env_parent(p), cdr(Tmp_Test)), tmsgf("(parent?␣E')"));    /* E0 */
tap_again(ok, ¬undefined_p(env_here(p, sym("xE"))), tmsgf("(env.exists?␣E'␣xE)"));
tap_again(ok, ¬undefined_p(env_here(p, sym("xA"))), tmsgf("(env.exists?␣E'␣xA)"));
tap_ok(test_is_env(p, m), tmsgf("(eq?␣E'␣yE)"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

281. Building applicatives and operatives within an operative requires extra care to evaluate code in the correct *environment*.

The *environment* E_1 that a returned applicative closes over, and will extend into E_2 when it's entered, is the local *environment* of the operative.

The outer operative evaluates its two arguments in its caller's *environment* E_0, saving them in *outer* and n in turn, and then calls **lambda**.

```
#define TEST_ORA_INNER "(lambda␣(inner␣n)␣(test!probe))"
#define TEST_ORA_MIXUP "(define!␣(current-environment)␣inner␣'out)""(define!␣(current-e\
        nvironment)␣outer␣(eval␣(car␣yA)␣yE))""(define!␣(current-environment)␣n␣(eval␣\
        (car␣(cdr␣yA))␣yE))"
#define TEST_ORA_BUILD
    "(vov␣((yA␣vov/args)␣(yE␣vov/env))"TEST_ORA_MIXUPTEST_ORA_INNER")"
#define TEST_ORA_CALL "((VOV␣'out␣'out-n)␣'in␣'in-n)"
#define TEST_ORA_PRINT "(vov␣(...)␣(lambda␣(inner␣n)␣(test!probe)))"
```

⟨ Operative test returning an *applicative* 281 ⟩ ≡

```
Env = env_extend(Root);     /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ORA_BUILD);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ORA_CALL);
caar(Acc) = vms_pop();
prefix = TEST_ORA_PRINT;
vm_reset();
interpret();
t = assoc_value(Acc, sym("Env"));     /* E2 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣(cdr␣T)␣'Env))"));
m = env_here(t, sym("n"));
tap_again(ok, m ≡ sinn, tmsgf("(eq?␣(env.here␣E␣n)␣'in-n)"));
m = env_here(t, sym("inner"));
tap_again(ok, m ≡ sin, tmsgf("(eq?␣(env.here␣E␣inner)␣'in)"));
tap_again(ok, undefined_p(env_here(t, sym("outer"))), tmsgf("(exists-here?␣E␣outer)"));
m = env_search(t, sym("outer"));
tap_again(ok, m ≡ sout, tmsgf("(eq?␣(env.lookup␣E␣inner)␣'out)"));
if (ok) p = env_parent(t);     /* E1 */
tap_again(ok, ¬undefined_p(env_here(p, sym("yE"))), tmsgf("(exists?␣(env.parent␣E)␣yE)"));
tap_again(ok, test_is_env(env_parent(p), car(Tmp_Test)), tmsgf("(env.parent?␣E')"));     /* E0 */
m = env_here(p, sym("n"));
tap_again(ok, m ≡ soutn, tmsgf("(eq?␣(env.here␣E␣n)␣'out-n)"));
m = env_here(p, sym("inner"));
tap_again(ok, m ≡ sout, tmsgf("(eq?␣(env.here␣E␣inner)␣'out)"));
m = env_here(p, sym("outer"));
tap_again(ok, m ≡ sout, tmsgf("(eq?␣(env.lookup␣E␣inner)␣'out)"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

282. Closing over an operative within an operative requires even more care that the correct environment is used so that the returned operative has access to its creator's local environment.

The creating operative extends the *environment* E_0 it closes over and this *environment* E_1 is then closed over by the returned operative. E_1 is extended upon entering the inner operative into *environment* E_2.

The same run-time *environment* E_3 is passed as an argument to the each operative.

```
#define TEST_ORO_INNER_BODY "(test!probe-applying␣(eval␣'(test!probe)␣oE))"
#define TEST_ORO_INNER "(vov␣((oE␣vov/env))"TEST_ORO_INNER_BODY")"
#define TEST_ORO_BUILD "(vov␣((A␣vov/args)␣(E␣vov/env))"TEST_ORO_INNER")"
#define TEST_ORO_CALL "((VOV␣'out␣'out-n)␣'in␣'in-n)"
#define TEST_ORO_PRINT "(VOV␣(vov␣(...)␣(test!probe␣(eval␣'(test!probe)␣E))))"
```

⟨ Operative test returning an *operative* 282 ⟩ ≡

```
Env = env_extend(Root);   /* E0 */
Tmp_Test = cons(test_copy_env(), NIL);
Acc = read_cstring(TEST_ORO_BUILD);
vm_reset();
interpret();
vms_push(Acc);
Env = env_extend(Root);   /* E3 */
cdr(Tmp_Test) = test_copy_env();
Acc = read_cstring(TEST_ORO_CALL);
caar(Acc) = vms_pop();
prefix = TEST_ORO_PRINT;
vm_reset();
interpret();
t = assoc_value(Acc, sym("Env"));   /* E2 */
ok = tap_ok(environment_p(t), tmsgf("(environment?␣(assoc-value␣T␣'Env))"));
if (ok) m = env_here(t, sym("oE"));   /* E3 */
tap_again(ok, environment_p(m), tmsgf("(environment?␣oE)"));
tap_again(ok, m ≡ cdr(Tmp_Test), tmsgf("(eq?␣E␣Env)"));
if (ok) m = env_parent(t);   /* E1 */
tap_again(ok, ¬undefined_p(env_here(m, sym("A"))), tmsgf("(env.exists?␣E'␣A)"));
if (ok) p = env_here(m, sym("E"));   /* E3 */
tap_again(ok, ¬undefined_p(env_here(m, sym("E"))), tmsgf("(env.exists?␣E'␣E)"));
tap_again(ok, p ≡ cdr(Tmp_Test), tmsgf("(eq?␣E''␣Env)"));
tap_again(ok, env_parent(m) ≡ car(Tmp_Test), tmsgf("(eq?␣(env.parent␣E')␣Env)"));
test_vm_state_normal(prefix);
tap_ok(test_compare_env(cdr(Tmp_Test)), tmsgf("(unchanged?␣Env)"));
```

This code is used in section 276.

283. Exceptions. When an error occurs at run-time it has the option (unimplemented) to be handled at run-time but if it isn't then control returns to before the beginning of the main loop. Each time around the main loop, *interpret* begins by calling *vm_reset* but that explicitely *doesn't* change the *environment* to allow for run-time mutation and expects that well-behaved code will clear the stack correctly.

These exception tests enter a closure, which creates a stack frame, and call **error** within it. The tests then ensure that the *environment* and stack are ready to compute again.

There is no actual support for exception handlers so the interpreter will halt and jump back *Goto_Begin*.

```
#define GOTO_FAIL "((lambda␣x␣(error␣fail)))"
```

```
⟨ t/exception.c 283 ⟩ ≡
  ⟨ Test executable wrapper 180 ⟩
  void test_main(void)
  {
    volatile boolean first = btrue;
    volatile boolean failed = bfalse;    /* WARNING: ERROR: SUCCESS */
    boolean ok;
    Error_Handler = btrue;
    vm_prepare();
    if (first) {
      first = bfalse;
      vm_reset();
      Acc = read_cstring(GOTO_FAIL);
      interpret();
    }
    else failed = btrue;
    ok = tap_ok(failed, "an␣error␣is␣raised");
    test_vm_state(GOTO_FAIL, TEST_VMSTATE_RUNNING | TEST_VMSTATE_NOT_INTERRUPTED |
        TEST_VMSTATE_ENV_ROOT | TEST_VMSTATE_STACKS);
  }
```

284. TODO.

⟨ List of opcode primitives 284 ⟩ ≡ /* Core: */
{"error", *compile_error*}, {"eval", *compile_eval*}, {"if", *compile_conditional*}, {"lambda", *compile_lambda*}, {"vov", *compile_vov*}, {"quote", *compile_quote*}, {"quasiquote", *compile_quasiquote*}, /* Pairs: */
{"car", *compile_car*}, {"cdr", *compile_cdr*}, {"cons", *compile_cons*}, {"null?", *compile_null_p*}, {"pair?", *compile_pair_p*}, {"set-car!", *compile_set_car_m*}, {"set-cdr!", *compile_set_cdr_m*}, /* Mutation: */
{"current-environment", *compile_env_current*}, {"root-environment", *compile_env_root*}, {"set!", *compile_set_m*}, {"define!", *compile_define_m*},
#**ifdef** LL_TEST
⟨ Testing primitives 187 ⟩
#**endif**

This code is used in section 67.

285. REPL. The *main* loop is a simple repl.

```
⟨ repl.c 285 ⟩ ≡
#include <stdio.h>
#include "lossless.h"
  int main(int argc, char **argv __unused)
  {
    vm_init();
    if (argc > 1) {
      printf("usage:␣%s", argv[0]);
      return EXIT_FAILURE;
    }
    vm_prepare();
    while (1) {
      vm_reset();
      printf(">␣");
      Acc = read_form();
      if (eof_p(Acc) ∨ Interrupt) break;
      interpret();
      if (¬void_p(Acc)) {
        write_form(Acc, 0);
        printf("\n");
      }
    }
    if (Interrupt) printf("Interrupted");
    return EXIT_SUCCESS;
  }
```

286. Association Lists.

⟨ Function declarations 8 ⟩ +≡

```
cell assoc_member(cell, cell);
cell assoc_content(cell, cell);
cell assoc_value(cell, cell);
```

287.

```
cell assoc_member(cell alist, cell needle)
{
  if (¬symbol_p(needle)) error (ERR_ARITY_SYNTAX, NIL);
  if (¬list_p(alist, FALSE, Λ)) error (ERR_ARITY_SYNTAX, NIL);
  for ( ; pair_p(alist); alist = cdr(alist))
    if (caar(alist) ≡ needle) return car(alist);
  return FALSE;
}
cell assoc_content(cell alist, cell needle)
{
  cell r;

  r = assoc_member(alist, needle);
  if (¬pair_p(r)) error (ERR_UNEXPECTED, r);
  return cdr(r);
}
cell assoc_value(cell alist, cell needle)
{
  cell r;

  r = assoc_member(alist, needle);
  if (¬pair_p(cdr(r))) error (ERR_UNEXPECTED, r);
  return cadr(r);
}
```

288. Misc.

289. **#define** $synquote\_new(o)$ $atom(Sym\_SYNTAX\_QUOTE, (o), \texttt{FORMAT\_SYNTAX})$
/* */

290. Index.

⟨ API declarations 9, 45, 70, 78, 81, 105 ⟩ Used in section 1.
⟨ API headers 5 ⟩ Used in section 1.
⟨ Allocator test executable wrapper 181 ⟩ Used in sections 202 and 223.
⟨ Applicative test passing an *applicative* 272 ⟩ Used in section 269.
⟨ Applicative test passing an *operative* 273 ⟩ Used in section 269.
⟨ Applicative test returning an *applicative* 274 ⟩ Used in section 269.
⟨ Applicative test returning an *operative* 275 ⟩ Used in section 269.
⟨ Compile a combiner 140 ⟩ Used in section 138.
⟨ Compile an atom 139 ⟩ Used in section 138.
⟨ Compile applicative combiner 150 ⟩ Used in section 140.
⟨ Compile native combiner 142 ⟩ Used in section 140.
⟨ Compile operative combiner 159 ⟩ Used in section 140.
⟨ Compile unknown combiner 143 ⟩ Used in section 140.
⟨ Compile unquote-splicing 174, 175, 176, 177 ⟩ Used in section 173.
⟨ Enter a *closure* 125 ⟩ Used in section 124.
⟨ Evaluate optional arguments into a *list* 154 ⟩ Used in section 150.
⟨ Evaluate required arguments onto the stack 153 ⟩ Used in section 150.
⟨ Function declarations 8, 19, 30, 37, 44, 84, 104, 132, 167, 184, 191, 256, 286 ⟩ Used in section 2.
⟨ Global constants 112, 113 ⟩ Used in section 2.
⟨ Global initialisation 3, 24, 52, 83, 156 ⟩ Cited in section 70. Used in section 71.
⟨ Global variables 7, 13, 17, 21, 23, 35, 42, 50, 67, 68, 74, 82, 131, 155, 182, 192 ⟩ Used in section 2.
⟨ Handle terminable 'forms' during *list* construction 98 ⟩ Used in section 97.
⟨ List of opcode primitives 284 ⟩ Used in section 67.
⟨ Look for optional arguments 152 ⟩ Used in section 150.
⟨ Look for required arguments 151 ⟩ Used in section 150.
⟨ Mutate if bound 62 ⟩ Used in section 61.
⟨ Mutate if unbound 63 ⟩ Used in section 61.
⟨ Opcode implementations 11, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 128, 129, 130 ⟩ Used in section 79.
⟨ Operative test passing an *applicative* 279 ⟩ Used in section 276.
⟨ Operative test passing an *operative* 280 ⟩ Used in section 276.
⟨ Operative test returning an *applicative* 281 ⟩ Used in section 276.
⟨ Operative test returning an *operative* 282 ⟩ Used in section 276.
⟨ Pre-initialise *Small_Int* 51 ⟩ Used in section 71.
⟨ Process lambda formals 149 ⟩ Used in section 148.
⟨ Protected Globals 18, 36, 43, 69, 133, 183 ⟩ Used in section 31.
⟨ Quasiquote a pair/list 170 ⟩ Used in section 169.
⟨ Quasiquote syntax 171, 172, 173 ⟩ Used in section 169.
⟨ Read bytes until an invalid or terminating character 103 ⟩ Used in section 101.
⟨ Read dotted pair 99 ⟩ Used in section 98.
⟨ Read the first two bytes to check for a number 102 ⟩ Used in section 101.
⟨ Reader forms 90, 92, 93, 94, 95, 96 ⟩ Used in section 89.
⟨ Sanity test **if**'s syntax 261, 262, 263, 264 ⟩ Used in section 260.
⟨ Scan operative informals 158 ⟩ Used in section 157.
⟨ Search *Root* for syntactic combiners 141 ⟩ Used in section 140.
⟨ System headers 4 ⟩ Used in section 2.
⟨ Test calling **lambda** 270 ⟩ Used in section 269.
⟨ Test calling **vov** 277 ⟩ Used in section 276.
⟨ Test entering an *applicative* closure 271 ⟩ Used in section 269.
⟨ Test entering an *operative* closure 278 ⟩ Used in section 276.
⟨ Test executable wrapper 180 ⟩ Used in sections 181, 198, 239, 251, 260, 269, 276, and 283.
⟨ Test integrating car 241 ⟩ Used in section 239.
⟨ Test integrating cdr 242 ⟩ Used in section 239.

LossLess Programming Environment