**1. Introduction.** LossLess is a programming language and environment similar to scheme. This document describes the implementation of a LossLess run-time 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.

All of the functions, variables, etc. used by LossLess are exported via lossless.h, even those which are nominally internal. Although this is not best practice for a library it makes this document less repetetive and facilitates easier testing.

⟨ lossless.h 1 ⟩ ≡
```
#ifndef LOSSLESS_H
#define LOSSLESS_H
  ⟨ System headers 4 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Complex definitions & macros 144 ⟩
  ⟨ Type definitions 5 ⟩
  ⟨ Function declarations 8 ⟩
  ⟨ Externalised global variables 7 ⟩
#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 ⟩
⟨ Complex definitions & macros 144 ⟩
⟨ Type definitions 5 ⟩
⟨ Function declarations 8 ⟩
⟨ Global variables 6 ⟩
```

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

See also sections 33, 69, 80, 103, 114, 182, and 195.

This code is cited in section 97.

This code is used in section 98.

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

**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 sections 1, 2, and 222.

**5.** 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 5 ⟩ ≡
```
  typedef int32_t cell;
  typedef int boolean;
  typedef cell predicate;
```
See also sections 90, 146, and 239.

This code is used in sections 1 and 2.

**6. 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) error_imp((x), NIL, (d))
#define ex_id car
#define ex_detail cdr
```

⟨ Global variables 6 ⟩ ≡

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

See also sections 12, 19, 25, 30, 47, 56, 66, 78, 91, 93, 101, 112, 148, 165, 180, 194, 220, 223, and 245.

This code is used in section 2.

**7.** ⟨ Externalised global variables 7 ⟩ ≡

```
extern volatile boolean Error_Handler;
extern jmp_buf Goto_Begin;
extern jmp_buf Goto_Error;
```

See also sections 13, 20, 26, 31, 42, 48, 57, 67, 79, 92, 94, 102, 113, 147, 166, 181, 221, 224, and 246.

This code is used in section 1.

**8.** ⟨ Function declarations 8 ⟩ ≡

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

See also sections 14, 22, 27, 36, 43, 51, 59, 70, 73, 81, 88, 97, 104, 109, 115, 135, 167, 206, 227, 240, 244, 471, and 501.

This code is used in sections 1 and 2.

**9.** 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 error_imp(char *message, cell id, cell detail)
{
  int len;
  char wbuf[BUFFER_SEGMENT] = {0};
  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);
  }
  write_form(detail, wbuf, BUFFER_SEGMENT, 0);
  printf("UNHANDLED␣ERROR:␣");
  for ( ; len--; message++) putchar(*message);
  printf(":␣%s\n", wbuf);
  longjmp(Goto_Begin, 1);
}
```

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

```
⟨ Opcode implementations 10 ⟩ ≡
case OP_ERROR:
  error_imp(Λ, Acc, rts_pop(1));
  break;      /* superfluous */
```

See also sections 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 162, 163, 164, and 507.

This code is used in section 110.

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

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

[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.

**12. 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 6 ⟩ +≡

```
cell *CAR = Λ;
cell *CDR = Λ;
char *TAG = Λ;
cell Cells_Free = NIL;
int Cells_Poolsize = 0;
int Cells_Segment = HEAP_SEGMENT;
```

**13.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern cell *CAR, *CDR, Cells_Free;
extern char *TAG;
extern int Cells_Poolsize, Cells_Segment;
```

**14.** ⟨ Function declarations 8 ⟩ +≡

```
void new_cells_segment(void);
```

**15.** ⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15 ⟩ ≡

```
free(CAR);
free(CDR);
free(TAG);
CAR = CDR = Λ;
TAG = Λ;
Cells_Free = NIL;
Cells_Poolsize = 0;
Cells_Segment = HEAP_SEGMENT;
```

See also sections 23, 28, 34, 50, 60, 68, 96, 169, and 226.

This code is used in section 98.

**16.** 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 ERR_OOM_P(p) do { if ((p) ≡ Λ) error (ERR_OOM, NIL); } while (0)
#define ERR_DOOM_P(p, d) do { if ((p) ≡ Λ) error (ERR_OOM, (d)); } while (0)
#define enlarge_pool(p, m, t) do
        {
          void *n;
          n = LL_ALLOCATE((p), (m), sizeof (t));
          ERR_OOM_P(n);
          (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;
  }
```

**17.** 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*) ≡ FALSE ∨ (*p*) ≡ TRUE)
**#define** *eof_p*(*p*) ((*p*) ≡ END_OF_FILE)
**#define** *false_p*(*p*) ((*p*) ≡ FALSE)
**#define** *null_p*(*p*) ((*p*) ≡ NIL)
**#define** *true_p*(*p*) ((*p*) ≡ TRUE)
**#define** *void_p*(*p*) ((*p*) ≡ VOID)
**#define** *undefined_p*(*p*) ((*p*) ≡ UNDEFINED)

**#define** *mark_p*(*p*) (¬*special_p*(*p*) ∧ (TAG[(*p*)] & TAG_MARK))
**#define** *state_p*(*p*) (¬*special_p*(*p*) ∧ (TAG[(*p*)] & TAG_STATE))
**#define** *acar_p*(*p*) (¬*special_p*(*p*) ∧ (TAG[(*p*)] & TAG_ACARP))
**#define** *acdr_p*(*p*) (¬*special_p*(*p*) ∧ (TAG[(*p*)] & TAG_ACDRP))
**#define** *mark_clear*(*p*) (TAG[(*p*)] &= ∼TAG_MARK)
**#define** *mark_set*(*p*) (TAG[(*p*)] |= TAG_MARK)
**#define** *state_clear*(*p*) (TAG[(*p*)] &= ∼TAG_STATE)
**#define** *state_set*(*p*) (TAG[(*p*)] |= TAG_STATE)
**#define** *format*(*p*) (TAG[(*p*)] & TAG_FORMAT)

**#define** *tag*(*p*) (TAG[(*p*)])

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

**18.** Both *atom*s and cons cells are stored in *pair*s. The lower 6 bits of the tag define the format of data stored in that *pair*. The *atom*s are grouped into three types depending on whether both **cell**s point to another *pair*, whether only the *cdr* does, or whether both **cell**s 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 exception_p(p) (¬special_p(p) ∧ ((tag(p) & TAG_FORMAT) ≡ FORMAT_EXCEPTION))
#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))
```

**19.** 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 6 ⟩ +≡
  cell Tmp_CAR = NIL;
  cell Tmp_CDR = NIL;
```

**20.**
```
⟨ Externalised global variables 7 ⟩ +≡
  extern cell Tmp_CAR, Tmp_CDR;
```

**21.**
```
⟨ Protected Globals 21 ⟩ ≡
  &Tmp_CAR, &Tmp_CDR ,
```

See also sections 32, 49, 58, 95, 168, and 225.

This code is used in section 41.

**22.**
```
⟨ Function declarations 8 ⟩ +≡
  cell atom(cell, cell, char);
```

**23.**
```
⟨ Pre-initialise Small_Int & other gc-sensitive buffers 15 ⟩ +≡
  Tmp_CAR = Tmp_CDR = NIL;
```

**24.** **#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;
}
```

**25. 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 6 ⟩ +≡
  cell *VECTOR = Λ;
  int Vectors_Free = 0;
  int Vectors_Poolsize = 0;
  int Vectors_Segment = HEAP_SEGMENT;
```

**26.**
```
⟨ Externalised global variables 7 ⟩ +≡
  extern cell *VECTOR;
  extern int Vectors_Free, Vectors_Poolsize, Vectors_Segment;
```

**27.**
```
⟨ Function declarations 8 ⟩ +≡
  void new_vector_segment(void);
```

**28.**
```
⟨ Pre-initialise Small_Int & other gc-sensitive buffers 15 ⟩ +≡
  free(VECTOR);
  VECTOR = Λ;
  Vectors_Free = Vectors_Poolsize = 0;
  Vectors_Segment = HEAP_SEGMENT;
```

**29.**
```
void new_vector_segment(void)
{
  cell *new_vector;
  new_vector = LL_ALLOCATE(VECTOR, Vectors_Poolsize + Vectors_Segment, sizeof(cell));
  ERR_OOM_P(new_vector);
  bzero(new_vector + Vectors_Poolsize, Vectors_Segment * sizeof(cell));
  VECTOR = new_vector;
  Vectors_Poolsize += Vectors_Segment;
  Vectors_Segment = Vectors_Poolsize/2;
}
```

**30.** 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 6 ⟩ +≡
  cell Zero_Vector = NIL;
```

**31.** ⟨ Externalised global variables 7 ⟩ +≡
```
extern cell Zero_Vector;
```

**32.** ⟨ Protected Globals 21 ⟩ +≡
```
&Zero_Vector ,
```

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

**34.** ⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15 ⟩ +≡
```
Zero_Vector = NIL;
```

**35.** 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.

**36.** ⟨ Function declarations 8 ⟩ +≡
```
cell vector_new(int, cell);
cell vector_new_imp(int, boolean, cell);
cell vector_new_list(cell, int);
cell vector_sub(cell, int, int, int, int, cell);
```

**37.**
```
cell vector_new_imp(int size, boolean 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();
  }
  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 = 0; i ≤ size; i++)
      vector_ref(r, i) = fill;
  return r;
}
```

**38.**
```
cell vector_new(int size, cell fill)
{
  if (size ≡ 0) return Zero_Vector;
  return vector_new_imp(size, btrue, fill);
}
```

**39.** *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;
}
```

**40.** 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;
}
```

**41. 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.

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 21 ⟩, Λ};
```

**42.**
```
⟨ Externalised global variables 7 ⟩ +≡
extern cell *ROOTS;
```

**43.**
```
⟨ Function declarations 8 ⟩ +≡
int gc(void);
int gc_vectors(void);
void mark(cell);
int sweep(void);
```

```
44.  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 sweep(void)
{
  int count, i;
  Cells_Free = NIL;
  count = 0;
  for (i = 0; i < Cells_Poolsize; i++) {
    if (¬mark_p(i)) {
      tag(i) = TAG_NONE;
      cdr(i) = Cells_Free;
      Cells_Free = i;
      count++;
    }
    else {
      mark_clear(i);
    }
  }
  return count;
}
int gc(void)
{
  int 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;
  return sweep( );
}
```

**45.** *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 46 ⟩
  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;
}
```

**46.** 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 46 ⟩ ≡
  i = 0;
  while (i < Vectors_Free) {
    VECTOR[i + VECTOR_CELL] = NIL;
    i += vector_realsize(VECTOR[i + VECTOR_SIZE]);
  }
```

This code is used in section 45.

**47. 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 run-time 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 6 ⟩ +≡

```
cell CTS = NIL;
cell RTS = NIL;
cell VMS = NIL;
int RTS_Size = 0;
int RTSp = −1;
```

**48.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern cell CTS, RTS, VMS;
extern int RTS_Size, RTSp;
```

**49.** ⟨ Protected Globals 21 ⟩ +≡

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

**50.** ⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15 ⟩ +≡

```
CTS = RTS = VMS = NIL;
RTS_Size = 0;
RTSp = −1;
```

**51.** ⟨ Function declarations 8 ⟩ +≡

```
cell cts_pop(void);
void cts_push(cell);
cell cts_ref(void);
void cts_set(cell);
cell rts_pop(int);
void rts_prepare(int);
void rts_push(cell);
cell rts_ref(int);
cell rts_ref_abs(int);
void rts_set(int, cell);
void rts_set_abs(int, cell);
cell vms_pop(void);
void vms_push(cell);
cell vms_ref(void);
void vms_set(cell);
```

**52.** 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;
}
```

**53.** 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;
  }
```

**54.** 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;
    }
  }
```

**55.** 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;
  }
```

**56. 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 6 ⟩ +≡
```
cell Symbol_Table = NIL;
char *SYMBOL = Λ;
int Symbol_Free = 0;
int Symbol_Poolsize = 0;
```

**57.** ⟨ Externalised global variables 7 ⟩ +≡
```
extern cell Symbol_Table;
extern char *SYMBOL;
extern int Symbol_Free, Symbol_Poolsize;
```

**58.** ⟨ Protected Globals 21 ⟩ +≡
```
&Symbol_Table ,
```

**59.** ⟨ Function declarations 8 ⟩ +≡
```
cell symbol(char *, boolean);
void symbol_expand(void);
void symbol_reify(cell);
boolean symbol_same_p(cell, cell);
cell symbol_steal(char *);
```

**60.** ⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15 ⟩ +≡
```
free(SYMBOL);
SYMBOL = Λ;
Symbol_Poolsize = Symbol_Free = 0;
Symbol_Table = NIL;
```

**61.**
```
void symbol_expand(void)
{
  char *new;
  new = realloc(SYMBOL, Symbol_Poolsize + HEAP_SEGMENT);
  ERR_OOM_P(new);
  Symbol_Poolsize += HEAP_SEGMENT;
  SYMBOL = new;
}
```

**62.** 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;
}
```

**63.** 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;
}
```

**64.**
```
void symbol_reify(cell s)
{
  Symbol_Free += symbol_length(s);
  Symbol_Table = cons(s, Symbol_Table);
}
```

**65.**
```
cell symbol(char *cstr, boolean 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;
}
```

**66. 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 smallint_p(p) (fixint_p(p) ∧ int_value(p) ≥ SCHAR_MIN ∧ int_value(p) ≤ SCHAR_MAX)
#define int_value(p) ((int)(car(p)))
#define int_next cdr
⟨ Global variables 6 ⟩ +≡
  cell Small_Int[UCHAR_MAX + 1];
```

**67.**
```
⟨ Externalised global variables 7 ⟩ +≡
  extern cell *Small_Int;
```

**68.** 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 & other gc-sensitive buffers 15 ⟩ +≡
  for (i = 0; i < 256; i++) Small_Int[i] = NIL;
```

**69.**
```
⟨ Global initialisation 3 ⟩ +≡
  for (i = SCHAR_MIN; i ≤ SCHAR_MAX; i++)
    Small_Int[(unsigned char) i] = int_new_imp(i, NIL);
```

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

```
⟨ Function declarations 8 ⟩ +≡
  cell int_new_imp(int, cell);
  cell int_new(int);
```

**71.**
```
cell int_new_imp(int value, cell next)
{
  if (¬null_p(next)) error (ERR_UNIMPLEMENTED, NIL);
  return atom((cell) value, next, FORMAT_INTEGER);
}
```

**72.**
```
cell int_new(int value)
{
  if (value ≥ SCHAR_MIN ∧ value ≤ SCHAR_MAX)
    return Small_Int[(unsigned char) value];
  return int_new_imp(value, NIL);
}
```

**73. 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* `object`s.

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.

⟨ Function declarations 8 ⟩ +≡

```
int list_length(cell);
predicate list_p(cell, predicate, cell *);
cell list_reverse_m(cell, boolean);
```

**74.**

```
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;
}
```

**75.** 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;
}
```

**76.** 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( );
  }
```

**77.** 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;
  }
```

**78. 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)))
⟨ Global variables 6 ⟩ +≡
  cell Sym_ERR_BOUND = NIL;
  cell Sym_ERR_UNBOUND = NIL;
```

**79.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern cell Sym_ERR_BOUND, Sym_ERR_UNBOUND;
```

**80.** ⟨ Global initialisation 3 ⟩ +≡

```
Sym_ERR_BOUND = sym(ERR_BOUND);
Sym_ERR_UNBOUND = sym(ERR_UNBOUND);
```

**81.** 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.

⟨ Function declarations 8 ⟩ +≡

```
cell env_here(cell, cell);
cell env_lift_stack(cell, cell);
cell env_search(cell, cell);
void env_set(cell, cell, cell, boolean);
```

**82.**

```
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;
}
```

[1] Or a cake.

**83.**
```
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;
}
```

**84.** 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!*).

```
#define env_set_fail(e) do
        {
          vms_clear( );
          error ((e), name);
        }
        while (0)
void env_set(cell e, cell name, cell value, boolean new_p)
{
  cell ass, t;
  ass = cons(name, cons(value, NIL));
  vms_push(ass);
  if (new_p) {⟨ Mutate if unbound 86 ⟩}
  else {⟨ Mutate if bound 85 ⟩}
}
```

**85.** 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 85 ⟩ ≡
  if (null_p(env_layer(e))) env_set_fail(ERR_UNBOUND);
  if (caar(env_layer(e)) ≡ name) {
    env_layer(e) = cons(ass, cdr(env_layer(e)));
    vms_clear( );
    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));
      vms_clear( );
      return;
    }
  }
  env_set_fail(ERR_UNBOUND);
```

This code is used in section 84.

**86.** 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 86 ⟩ ≡
  if (¬undefined_p(env_here(e, name))) env_set_fail(ERR_BOUND);
  env_layer(e) = cons(ass, env_layer(e));
  vms_clear( );
  return;
```

This code is used in section 84.

**87.** 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, cell formals)
{
  cell ass, name, p, r;
  vms_push(p = NIL);      /* prepare a new layer */
  while (¬null_p(formals)) {
    if (pair_p(formals)) {
      name = car(formals);
      formals = cdr(formals);
    }
    else {
      name = formals;
      formals = NIL;
    }
    if (null_p(name)) rts_clear(1);
    else {
      ass = cons(rts_pop(1), NIL);
      ass = cons(name, ass);
      p = cons(ass, p);
      vms_set(p);
    }
  }
  r = env_extend(e);
  env_layer(r) = p;
  vms_clear( );
  return r;
}
```

**88. 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,$(f),(e),(p),(i)$)

⟨ Function declarations 8 ⟩ +≡
**cell** *closure_new_imp*(**char**, **cell**, **cell**, **cell**, **cell**);

**89.** **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*);
}

**90.** 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 5 ⟩ +≡
**typedef void** (∗**native**)(**cell**, **cell**, **boolean**);
**typedef struct** {
**char** ∗*name*;
**native** *fn*;
} **primitive**;

**91.** 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 6 ⟩ +≡
**primitive** `COMPILER`[] = { ⟨ List of opcode primitives 499 ⟩, {Λ, Λ} } ;

**92.** ⟨ Externalised global variables 7 ⟩ +≡
**extern primitive** ∗`COMPILER`;

**93. 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 6 ⟩ +≡
  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;
```

**94.**
```
⟨ Externalised global variables 7 ⟩ +≡
  extern boolean Interrupt, Running;
  extern cell Acc, Env, Prog, Prog_Main, Root;
  extern int Ip;
```

**95.**
```
⟨ Protected Globals 21 ⟩ +≡
  &Acc, &Env, &Prog, &Prog_Main, &Root ,
```

**96.**
```
⟨ Pre-initialise Small_Int & other gc-sensitive buffers 15 ⟩ +≡
  Acc = Env = Prog = Prog_Main = Root = NIL;
  Interrupt = Running = Ip = 0;
```

**97.** 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)
```

⟨ Function declarations 8 ⟩ +≡

```
void vm_init_imp(void);
void vm_prepare_imp(void);
void vm_reset(void);
```

**98.** **void** *vm_init_imp*(**void**)

```
{
  cell t;
  int i;
  primitive *n;
  ⟨ Pre-initialise Small_Int & other gc-sensitive buffers 15 ⟩
  ⟨ 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;
}
```

**99.** **void** *vm_prepare_imp*(**void**)

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

**100.** **void** *vm_reset*(**void**)

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

**101. 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 6 ⟩ +≡

```
int Fp = -1;
```

**102.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern int Fp;
```

**103.** ⟨ Global initialisation 3 ⟩ +≡

```
Fp = -1;
```

**104.** 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.

⟨ Function declarations 8 ⟩ +≡

```
void frame_consume(void);
void frame_enter(cell, cell, cell);
void frame_leave(void);
void frame_push(int);
```

**105.**

```
void frame_push(int ipdelta)
{
  rts_push(int_new(Ip + ipdelta));
  rts_push(Prog);
  rts_push(Env);
  rts_push(int_new(Fp));
}
```

**106.**

```
void frame_enter(cell e, cell p, cell i)
{
  Env = e;
  Prog = p;
  Ip = i;
  Fp = RTSp - FRAME_HEAD;
}
```

**107.** 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;
}
```

**108. 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;
}
```

**109. 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*.

```
⟨ Function declarations 8 ⟩ +≡
  void interpret(void);
```

**110.**

```
#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 10 ⟩
#ifdef LL_TEST
        ⟨ Testing implementations 230 ⟩
#endif
      default: Interrupt = btrue;
      }
    }
    if (Interrupt) error (ERR_INTERRUPTED, NIL);
  }
```

**111. 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.

**112. 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" /* ⟪,@⟫ */
```

⟨ Global variables 6 ⟩ +≡

```
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;
```

**113.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern char Putback[2], *Read_Pointer;
extern int Read_Level;
extern cell Sym_ERR_UNEXPECTED, Sym_SYNTAX_DOTTED, Sym_SYNTAX_QUASI;
extern cell Sym_SYNTAX_QUOTE, Sym_SYNTAX_UNQUOTE, Sym_SYNTAX_UNSPLICE;
```

**114.** ⟨ 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);
```

**115.** ⟨ Function declarations 8 ⟩ +≡

```
int read_byte(void);
cell read_cstring(char *);
cell read_form(void);
cell read_list(cell);
cell read_number(void);
cell read_sexp(void);
cell read_symbol(void);
void unread_byte(char);
int useful_byte(void);
```

**116.**

```
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;
}
```

**117.** 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;
}
```

**118.** 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;
}
```

**119.** 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( );
}
```

**120.** *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 121 ⟩}
  error (ERR_UNEXPECTED, NIL);
}
```

**121.** 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 121 ⟩ ≡
case EOF:
  if (¬Read_Level) return END_OF_FILE;
  else error (ERR_ARITY_SYNTAX, NIL);
```

See also sections 123, 124, 125, 126, and 127.

This code is used in section 120.

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

**123.** ⟨ Reader forms 121 ⟩ +≡
```
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 (¬Read_Level) error (ERR_ARITY_SYNTAX, NIL);
  else return c ≡ ')' ? READ_CLOSE_PAREN : READ_CLOSE_BRACKET;
```

**124.** 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 121 ⟩ +≡
```
case '.':
  if (¬Read_Level) error (ERR_ARITY_SYNTAX, NIL);
  c = useful_byte( );
  if (c ≡ EOF) error (ERR_ARITY_SYNTAX, NIL);
  unread_byte(c);
  return READ_DOT;
```

**125.** Special forms and strings aren't supported yet.

⟨ Reader forms 121 ⟩ +≡
```
case '"': case '#': case '|': error (ERR_UNIMPLEMENTED, NIL);
```

**126.** 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 121 ⟩ +≡
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);
  }
```

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

```
⟨ Reader forms 121 ⟩ +≡
default:
  if (¬isprint(c)) error (ERR_ARITY_SYNTAX, NIL);
  unread_byte(c);
  if (isdigit(c)) return read_number( );
  else return read_symbol( );
```

**128.** *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 129 ⟩
    }
    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;
}
```

**129.** *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 129 ⟩ ≡
  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 130 ⟩}
```

This code is used in section 128.

**130.** 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 130 ⟩ ≡
  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 129.

**131.** If it's not a *list* or a *vector* (or a *string* (⟪"⟫), *special form* (⟪#⟫), *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);
  }
```

**132.** 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;
    ERR_OOM_P(buf = malloc(CHUNK_SIZE));
    s = CHUNK_SIZE;
    ⟨Read the first two bytes to check for a number 133⟩
    while (1) {⟨Read bytes until an invalid or terminating character 134⟩}
    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.

**133.** 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 133 ⟩ ≡
  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 132.

**134.** 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 134 ⟩ ≡
  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 132.

**135. 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 BUFFER_SEGMENT 1024
#define WRITER_MAX_DEPTH 1024 /* gotta pick something */
#define append(b, r, c, s) do
        {
          ssize_t _l = strlen(c);
          if ((r) ≤ 0) return −1;
          if (strlcpy((b), (c), (r)) ≥ (size_t)(r)) return −(r);
          (s) += _l;
          (b) += _l;
          (r) −= _l;
        }
        while (0)
#define append_write(b, r, w, d, s) do
        {
          ssize_t _l = write_form((w), (b), (r), (d));
          if (_l ≤ 0) return −1;
          (s) += _l;
          (b) += _l;
          (r) −= _l;
        }
        while (0)
```

⟨ Function declarations 8 ⟩ +≡

```
ssize_t write_applicative(cell, char *, ssize_t, int);
ssize_t write_bytecode(cell, char *, ssize_t, int);
ssize_t write_compiler(cell, char *, ssize_t, int);
ssize_t write_environment(cell, char *, ssize_t, int);
ssize_t write_integer(cell, char *, ssize_t, int);
ssize_t write_list(cell, char *, ssize_t, int);
ssize_t write_operative(cell, char *, ssize_t, int);
ssize_t write_symbol(cell, char *, ssize_t, int);
ssize_t write_syntax(cell, char *, ssize_t, int);
ssize_t write_vector(cell, char *, ssize_t, int);
ssize_t write_form(cell, char *, ssize_t, int);
```

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

```
ssize_t write_applicative(cell sexp, char *buf, ssize_t rem, int depth__unused)
{
  ssize_t len = 0;
  if (¬applicative_p(sexp)) return 0;
  append(buf, rem, "#<applicative␣...>", len);
  return len;
}
ssize_t write_compiler(cell sexp, char *buf, ssize_t rem, int depth__unused)
{
  ssize_t len = 0;
  if (¬compiler_p(sexp)) return 0;
  append(buf, rem, "#<compiler␣", len);
  append(buf, rem, compiler_cname(sexp), len);
  if (rem ≡ 0) return −1;
  buf[0] = '>';
  buf[1] = '\0';
  return len + 1;
}
ssize_t write_operative(cell sexp, char *buf, ssize_t rem, int depth__unused)
{
  ssize_t len = 0;
  if (¬operative_p(sexp)) return 0;
  append(buf, rem, "#<operative␣...>", len);
  return len;
}
```

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

```
ssize_t write_integer(cell sexp, char *buf, ssize_t rem, int depth__unused)
{
  ssize_t len = 0;
  if (¬integer_p(sexp)) return 0;
  len = snprintf(buf, rem, "%d", int_value(sexp));
  if (len ≥ rem) return −1;
  return len;
}
ssize_t write_symbol(cell sexp, char *buf, ssize_t rem, int depth__unused)
{
  int i;
  if (¬symbol_p(sexp)) return 0;      /* TODO: unprintable (including zero-length) symbols */
  if (rem ≡ 0) return −1;
  for (i = 0; rem > 0 ∧ i < symbol_length(sexp); i++, rem−−) buf[i] = symbol_store(sexp)[i];
  if (i ≠ symbol_length(sexp)) {
    buf[i − 1] = '\0';
    return −i;
  }
  buf[i] = '\0';
  return i;
}
```

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

```
ssize_t write_syntax(cell sexp, char *buf, ssize_t rem, int depth)
{
  ssize_t len = 0;
  if (¬syntax_p(sexp)) return 0;
  else if (car(sexp) ≡ sym(SYNTAX_DOTTED)) append(buf, rem, ". ", len);
  else if (car(sexp) ≡ sym(SYNTAX_QUASI)) append(buf, rem, "`", len);
  else if (car(sexp) ≡ sym(SYNTAX_QUOTE)) append(buf, rem, "'", len);
  else if (car(sexp) ≡ sym(SYNTAX_UNQUOTE)) append(buf, rem, ",", len);
  else if (car(sexp) ≡ sym(SYNTAX_UNSPLICE)) append(buf, rem, ",@", len);
  append_write(buf, rem, cdr(sexp), depth + 1, len);
  return len;
}
```

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

```
ssize_t write_environment(cell sexp, char *buf, ssize_t rem, int depth)
{
  ssize_t len = 0;
  if (¬environment_p(sexp)) return 0;
  append(buf, rem, "#<environment␣", len);
  append_write(buf, rem, env_layer(sexp), depth, len);
  if (¬null_p(env_parent(sexp))) {
    append(buf, rem, "␣ON␣", len);
    append_write(buf, rem, env_parent(sexp), depth + 1, len);
    append(buf, rem, ">", len);
  }
  else append(buf, rem, "␣ROOT>", len);
  return len;
}
```

**140. Exception Objects.** These just need to look dangerous so they are in ALL CAPS.

```
ssize_t write_exception(cell sexp, char *buf, ssize_t rem, int depth)
{
  ssize_t len = 0;
  if (¬exception_p(sexp)) return 0;
  append(buf, rem, "#<EXCEPTION␣", len);
  append_write(buf, rem, ex_id(sexp), depth, len);
  append(buf, rem, "␣", len);
  append_write(buf, rem, ex_detail(sexp), depth + 1, len);
  append(buf, rem, ">", len);
  return len;
}
```

**141. 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.

```
ssize_t write_list(cell sexp, char *buf, ssize_t rem, int depth)
{
  ssize_t len = 0;
  if (¬pair_p(sexp)) return 0;
  append(buf, rem, "(", len);
  while (pair_p(sexp)) {
    append_write(buf, rem, car(sexp), depth + 1, len);
    if (pair_p(cdr(sexp)) ∨ syntax_p(cdr(sexp))) append(buf, rem, "␣", len);
    else if (¬null_p(cdr(sexp)) ∧ ¬pair_p(cdr(sexp)) ∧ ¬syntax_p(cdr(sexp)))
      append(buf, rem, "␣.␣", len);
    sexp = cdr(sexp);
  }
  if (¬null_p(sexp)) append_write(buf, rem, sexp, depth + 1, len);
  append(buf, rem, ")", len);
  return len;
}
ssize_t write_vector(cell sexp, char *buf, ssize_t rem, int depth)
{
  int i;
  ssize_t len = 0;
  if (¬vector_p(sexp)) return 0;
  append(buf, rem, "[", len);
  for (i = 0; i < vector_length(sexp); i++) {
    append_write(buf, rem, vector_ref(sexp, i), depth + 1, len);
    if (i + 1 < vector_length(sexp)) append(buf, rem, "␣", len);
  }
  append(buf, rem, "]", len);
  return len;
}
```

**142.** For the time being *write_bytecode* is only called (directly) by the unit tests; there is no object that represents bytecode for *write_form* to detect.

```
ssize_t write_bytecode(cell sexp, char *buf, ssize_t rem, int depth)
{
  int arg, ins, op;
  ssize_t len = 1;
  if (rem ≤ 0) return −1;
  *buf++ = '{';
  rem−−;
  op = 0;
  while (op < vector_length(sexp)) {
    if (op) append(buf, rem, "␣", len);
    ins = int_value(vector_ref(sexp, op));
    if (ins ≥ OPCODE_MAX) error (ERR_UNEXPECTED, NIL);
    append(buf, rem, OP[ins].name, len);
    for (arg = 1; arg ≤ OP[ins].nargs; arg++) {
      append(buf, rem, "␣", len);
      append_write(buf, rem, vector_ref(sexp, op + arg), depth + 1, len);
    }
    op += 1 + OP[ins].nargs;
  }
  append(buf, rem, "}", len);
  return len;
}
```

**143.** *write_form* simply calls each writer in turn, stopping after the first one returning a positive number of bytes written or a negative number indicating that the buffer is full.

```
ssize_t write_form(cell sexp, char *buf, ssize_t rem, int depth)
{
  ssize_t len = 0;
  if (Interrupt) {
    if (¬depth) append(buf, rem, "...␣", len);
    return len;
  }
  if (depth > WRITER_MAX_DEPTH) error (ERR_RECURSION, NIL);
  if (undefined_p(sexp)) append(buf, rem, "#><", len);      /* nothing should ever print this */
  else if (eof_p(sexp)) append(buf, rem, "#<eof>", len);
  else if (false_p(sexp)) append(buf, rem, "#f", len);
  else if (null_p(sexp)) append(buf, rem, "()", len);
  else if (true_p(sexp)) append(buf, rem, "#t", len);
  else if (void_p(sexp)) append(buf, rem, "#<>", len);
  else if ((len = write_applicative(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_compiler(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_environment(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_exception(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_integer(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_list(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_operative(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_symbol(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_syntax(sexp, buf, rem, depth)))      /* NOP */ ;
  else if ((len = write_vector(sexp, buf, rem, depth)))      /* NOP */ ;
  else append(buf, rem, "#<wtf?>", len);      /* impossibru! */
  return len;
}
```

**144. 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))
⟨ Complex definitions & macros 144 ⟩ ≡        /* Four per line */
  enum {
      OP_APPLY, OP_APPLY_TAIL, OP_CAR, OP_CDR,
      OP_COMPILE, OP_CONS, OP_CYCLE, OP_ENVIRONMENT_P,
      OP_ENV_MUTATE_M, OP_ENV_QUOTE, OP_ENV_ROOT, OP_ENV_SET_ROOT_M,
      OP_ERROR, OP_HALT, OP_JUMP, OP_JUMP_FALSE,
      OP_JUMP_TRUE, OP_LAMBDA, OP_LIST_P, OP_LIST_REVERSE,
      OP_LIST_REVERSE_M, OP_LOOKUP, OP_NIL, OP_NOOP,
      OP_NULL_P, OP_PAIR_P, OP_PEEK, OP_POP,
      OP_PUSH, OP_QUOTE, OP_RETURN, OP_RUN,
      OP_RUN_THERE, OP_SET_CAR_M, OP_SET_CDR_M, OP_SNOC,
      OP_SWAP, OP_SYMBOL_P, OP_SYNTAX, OP_VOV ,
#ifdef LL_TEST
  ⟨ Testing opcode names 228 ⟩
#endif
  OPCODE_MAX } ;
```

See also sections 145 and 249.

This code is used in sections 1 and 2.

**145.** In case testing opcodes are referred to outside the tests they are given numbers which will cause the interpreter to immediately abort. They are not printable.

```
⟨ Complex definitions & macros 144 ⟩ +≡
#ifndef LL_TEST
  enum { OP_TEST_UNDEFINED_BEHAVIOUR = #f00f, ⟨ Testing opcode names 228 ⟩ };
#endif
```

**146.**

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct {
    char *name;
    int nargs;
  } opcode;
```

**147.**

```
⟨ Externalised global variables 7 ⟩ +≡
  extern opcode OP[OPCODE_MAX];
```

**148.** ⟨ Global variables 6 ⟩ +≡

```
opcode OP[OPCODE_MAX] = {
    [OP_APPLY] = { .name = "OP_APPLY", .nargs = 1 },
    [OP_APPLY_TAIL] = { .name = "OP_APPLY_TAIL", .nargs = 1 },
    [OP_CAR] = { .name = "OP_CAR", .nargs = 0 },
    [OP_CDR] = { .name = "OP_CDR", .nargs = 0 },
    [OP_COMPILE] = { .name = "OP_COMPILE", .nargs = 0 },
    [OP_CONS] = { .name = "OP_CONS", .nargs = 0 },
    [OP_CYCLE] = { .name = "OP_CYCLE", .nargs = 0 },
    [OP_ENVIRONMENT_P] = { .name = "OP_ENVIRONMENT_P", .nargs = 0 },
    [OP_ENV_MUTATE_M] = { .name = "OP_ENV_MUTATE_M", .nargs = 2 },
    [OP_ENV_QUOTE] = { .name = "OP_ENV_QUOTE", .nargs = 0 },
    [OP_ENV_ROOT] = { .name = "OP_ENV_ROOT", .nargs = 0 },
    [OP_ENV_SET_ROOT_M] = { .name = "OP_ENV_SET_ROOT_M", .nargs = 0 },
    [OP_ERROR] = { .name = "OP_ERROR", .nargs = 0 },
    [OP_HALT] = { .name = "OP_HALT", .nargs = 0 },
    [OP_JUMP] = { .name = "OP_JUMP", .nargs = 1 },
    [OP_JUMP_FALSE] = { .name = "OP_JUMP_FALSE", .nargs = 1 },
    [OP_JUMP_TRUE] = { .name = "OP_JUMP_TRUE", .nargs = 1 },
    [OP_LAMBDA] = { .name = "OP_LAMBDA", .nargs = 1 },
    [OP_LIST_P] = { .name = "OP_LIST_P", .nargs = 2 },
    [OP_LIST_REVERSE] = { .name = "OP_LIST_REVERSE", .nargs = 2 },
    [OP_LIST_REVERSE_M] = { .name = "OP_LIST_REVERSE_M", .nargs = 0 },
    [OP_LOOKUP] = { .name = "OP_LOOKUP", .nargs = 0 },
    [OP_NIL] = { .name = "OP_NIL", .nargs = 0 },
    [OP_NOOP] = { .name = "OP_NOOP", .nargs = 0 },
    [OP_NULL_P] = { .name = "OP_NULL_P", .nargs = 0 },
    [OP_PAIR_P] = { .name = "OP_PAIR_P", .nargs = 0 },
    [OP_PEEK] = { .name = "OP_PEEK", .nargs = 0 },
    [OP_POP] = { .name = "OP_POP", .nargs = 0 },
    [OP_PUSH] = { .name = "OP_PUSH", .nargs = 0 },
    [OP_QUOTE] = { .name = "OP_QUOTE", .nargs = 1 },
    [OP_RETURN] = { .name = "OP_RETURN", .nargs = 0 },
    [OP_RUN] = { .name = "OP_RUN", .nargs = 0 },
    [OP_RUN_THERE] = { .name = "OP_RUN_THERE", .nargs = 0 },
    [OP_SET_CAR_M] = { .name = "OP_SET_CAR_M", .nargs = 0 },
    [OP_SET_CDR_M] = { .name = "OP_SET_CDR_M", .nargs = 0 },
    [OP_SNOC] = { .name = "OP_SNOC", .nargs = 0 },
    [OP_SWAP] = { .name = "OP_SWAP", .nargs = 0 },
    [OP_SYMBOL_P] = { .name = "OP_SYMBOL_P", .nargs = 0 },
    [OP_SYNTAX] = { .name = "OP_SYNTAX", .nargs = 1 },
    [OP_VOV] = { .name = "OP_VOV", .nargs = 1 },

#ifdef LL_TEST
  ⟨ Testing opcodes 229 ⟩
#endif
  } ;
```

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

**150.** ⟨ Opcode implementations 10 ⟩ +≡

```
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;
```

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

⟨ Opcode implementations 10 ⟩ +≡

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

**152. Pairs & Lists.** OP_CAR, OP_CDR, OP_NULL_P and OP_PAIR_P are self explanatory.

⟨ Opcode implementations 10 ⟩ +≡
```
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;
```

**153.** 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 10 ⟩ +≡
```
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;
```

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

⟨ Opcode implementations 10 ⟩ +≡
```
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;
```

**155. Other Objects.** There is not much to say about these.

⟨ Opcode implementations 10 ⟩ +≡

```
case OP_LIST_P:
  if (¬false_p(fetch(2))) error (ERR_UNIMPLEMENTED, NIL);
  Acc = list_p(Acc, fetch(1), Λ);
  skip(3);
  break;
case OP_LIST_REVERSE:
  if (¬true_p(fetch(1)) ∨ ¬false_p(fetch(2))) error (ERR_UNIMPLEMENTED, NIL);
  Acc = list_reverse(Acc, Λ, Λ);
  skip(3);
  break;
case OP_LIST_REVERSE_M:
  Acc = list_reverse_m(Acc, btrue);
  skip(1);
  break;
case OP_SYNTAX:
  Acc = atom(fetch(1), Acc, FORMAT_SYNTAX);
  skip(2);
  break;
```

**156. 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 10 ⟩ +≡

```
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;
```

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

⟨ Opcode implementations 10 ⟩ +≡

```
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;
```

**158.** 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 10 ⟩ +≡

```
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;
```

**159. 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 was created, allowing for *proper tail recursion* with further support from the compiler.

```
⟨ Opcode implementations 10 ⟩ +≡
case OP_APPLY:
  case OP_APPLY_TAIL:
  tmp = fetch(1);
  vms_push(env_lift_stack(cadr(tmp), car(tmp)));
  frame_push(2);
  frame_enter(vms_pop( ), caddr(tmp), int_value(cadddr(tmp)));
  if (ins ≡ OP_APPLY_TAIL) frame_consume( );
  break;
case OP_RETURN:
  frame_leave( );
  break;
```

**160.** 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 10 ⟩ +≡
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;
```

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

**162.** ⟨ Opcode implementations 10 ⟩ +≡

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

**163.** `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 10 ⟩ +≡

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

**164.** `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 10 ⟩ +≡

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

**165. 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 6 ⟩ +≡
  int Here = 0;
  cell Compilation = NIL;
```

**166.** ⟨ Externalised global variables 7 ⟩ +≡

```
  extern int Here;
  extern cell Compilation;
```

**167.** ⟨ Function declarations 8 ⟩ +≡
```
cell arity(cell, cell, int, boolean);
cell arity_next(cell, cell, cell, boolean, boolean);
int comefrom(void);
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_list(cell, cell, boolean);
void compile_null_p(cell, cell, boolean);
void compile_pair_p(cell, cell, boolean);
void compile_quasicompiler(cell, cell, cell, int, 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_symbol_p(cell, cell, boolean);
void compile_vov(cell, cell, boolean);
void emit(cell);
```

**168.** ⟨ Protected Globals 21 ⟩ +≡
```
&Compilation ,
```

**169.** ⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15 ⟩ +≡
```
Compilation = NIL;
```

**170.**
```
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;
}
```

**171.** 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;
}
```

**172.** 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;
  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 = Zero_Vector;
  if (¬null_p(CTS)) error (ERR_COMPILE_DIRTY, source);
  return r;
}
```

**173.** *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;
}
```

**174.** 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 175 ⟩}
  else {⟨ Compile a combiner 176 ⟩}
}
```

**175.** 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 175 ⟩ ≡
  if (symbol_p(sexp)) {
    emitq(sexp);
    emitop(OP_LOOKUP);
  }
  else { emitq(sexp); }
```

This code is used in section 174.

**176.** 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 176 ⟩ ≡
  cell args, combiner;
  combiner = car(sexp);
  args = undot(cdr(sexp));
  ⟨ Search Root for syntactic combiners 177 ⟩
  if (compiler_p(combiner)) {⟨ Compile native combiner 178 ⟩}
  else if (applicative_p(combiner)) {⟨ Compile applicative combiner 189 ⟩}
  else if (operative_p(combiner)) {⟨ Compile operative combiner 198 ⟩}
  else if (symbol_p(combiner) ∨ pair_p(combiner)) {⟨ Compile unknown combiner 179 ⟩}
  else { error (ERR_UNCOMBINABLE, combiner); }
```

This code is used in section 174.

**177.** 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 177 ⟩ ≡
  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 176.

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

```
⟨ Compile native combiner 178 ⟩ ≡
  compiler_fn(combiner)(combiner, args, tail_p);
```

This code is used in section 176.

**179.** 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 179 ⟩ ≡
  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 176.

**180. 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)))
```

⟨ Global variables 6 ⟩ +≡

```
cell Sym_ERR_ARITY_EXTRA = NIL;
cell Sym_ERR_ARITY_MISSING = NIL;
cell Sym_ERR_ARITY_SYNTAX = NIL;
```

**181.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern cell Sym_ERR_ARITY_EXTRA, Sym_ERR_ARITY_MISSING, Sym_ERR_ARITY_SYNTAX;
```

**182.** ⟨ Global initialisation 3 ⟩ +≡

```
Sym_ERR_ARITY_EXTRA = sym(ERR_ARITY_EXTRA);
Sym_ERR_ARITY_MISSING = sym(ERR_ARITY_MISSING);
Sym_ERR_ARITY_SYNTAX = sym(ERR_ARITY_SYNTAX);
```

**183.**

```
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;
}
```

**184.** *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);
}
```

**185.** *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;
  }
}
```

**186. Closures (Applicatives & Operatives).** The first thing to understand is that at their core *applicative*s and *operative*s 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*.

**187.** 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 188 ⟩}
  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.

**188.** 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 188 ⟩ ≡
  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 187.

**189.** 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 189 ⟩ ≡
  cell collect, direct, formals, a;
  int nargs = 0;
  formals = applicative_formals(combiner);
  cts_push(direct = NIL);
  a = undot(args);
  ⟨ Look for required arguments 190 ⟩
  ⟨ Look for optional arguments 191 ⟩
  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 193 ⟩
  ⟨ Evaluate required arguments onto the stack 192 ⟩
  cts_clear( );
  emitop(tail_p ? OP_APPLY_TAIL : OP_APPLY);
  emit(combiner);
```

This code is used in section 176.

**190.** 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 190 ⟩ ≡
  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 189.

**191.** 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 191 ⟩ ≡
  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 189.

**192.** 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 192 ⟩ ≡
  while (¬null_p(direct)) {
    compile_expression(car(direct), 0);
    emitop(OP_PUSH);
    direct = cdr(direct);
  }
```

This code is used in section 189.

**193.** 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 193 ⟩ ≡
  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 189.

**194.** 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 6 ⟩ +≡
  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;
```

**195.**

```
⟨ 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");
```

**196.**
```
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 197 ⟩
  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 */
}
```

**197.** 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 197 ⟩ ≡
  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 196.

**198.** 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 198 ⟩ ≡

```
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(combiner);
```

This code is used in section 176.

**199. 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.

**200. 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, btrue);
  sexp = cts_pop( );
  arity_next(op, args, more, bfalse, btrue);
  eenv = cts_pop( );
  if (¬undefined_p(eenv)) {
    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(undefined_p(eenv) ? OP_RUN : OP_RUN_THERE);
}
```

**201. 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);
}
```

**202. 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);
}
```

**203. 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);
}
```

**204. 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); }
```

**205.** 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.

**206.** 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);
```

**207.**

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

**208.** 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 209 ⟩}
  else if (vector_p(arg)) { error (ERR_UNIMPLEMENTED, NIL); }
  else if (syntax_p(arg)) {⟨ Quasiquote syntax 210 ⟩}
  else {
    emitq(arg);
    if (in_list_p) emitop(OP_CONS);
  }
}
```

[1] Yo!

**209.** 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 209 ⟩ ≡
  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 208.

**210.** The quote & unquote syntax is where the quasicompiler starts to get interesting. *quote*s and *quasiquote*s (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 210 ⟩ ≡
  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 211 and 212.

This code is used in section 208.

**211.** *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 210 ⟩ +≡
  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);
    }
```

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

```
⟨ Quasiquote syntax 210 ⟩ +≡
  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 213 ⟩}
    }
    else error (ERR_UNIMPLEMENTED, NIL);
```

**213. Splicing Lists.** If not recursing back into the quasicompiler at a lower depth then we are quasi-compiling 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 run-time whether we are splicing into the tail position – consider constructs like `` `(,@foo ,@bar) `` where `bar` evaluates to `NIL`.

```
⟨ Compile unquote-splicing 213 ⟩ ≡
  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 214, 215, and 216.

This code is used in section 212.

**214.** `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 213 ⟩ +≡
  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);
```

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

```
⟨ Compile unquote-splicing 213 ⟩ +≡
  patch(goto_list_p, int_new(Here));
  emitop(OP_POP);
  emitop(OP_SWAP);
  emitop(OP_JUMP_TRUE); goto_finish = comefrom( );
```

**216.** 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 213 ⟩ +≡
  emitop(OP_POP);
  emitop(OP_LIST_REVERSE); emit(TRUE); emit(FALSE);
  ⟨ Walk through the splicing list 217 ⟩
```

**217.** ⟨ Walk through the splicing list 217 ⟩ ≡

```
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 218.

This code is used in section 216.

**218.** 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 217 ⟩ +≡

```
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);
```

**219. Testing.** A comprehensive test suite is planned for LossLess but a testing tool would be no good if it wasn't itself reliable, which these primarily unit tests work towards. In addition to the main library lossless.o two libraries with extra functionality needed by the tests are created: t/lltest.o and t/llalloc.o which additionally to extra operators wraps reallocarray to test memory allocation.

```
⟨ t/lltest.c 219 ⟩ ≡
#define LL_TEST
#include "../lossless.c"      /* C source */
```

**220.**
```
⟨ Global variables 6 ⟩ +≡
#ifdef LL_TEST
  int Allocate_Success = -1;
#endif
```

**221.**
```
⟨ Externalised global variables 7 ⟩ +≡
#ifdef LL_TEST
  extern int Allocate_Success;
#endif
```

**222.**
```
⟨ t/llalloc.c 222 ⟩ ≡
#define LL_ALLOCATE fallible_reallocarray
  ⟨ System headers 4 ⟩
  void *fallible_reallocarray(void *, size_t, size_t);
#define LL_TEST
#include "../lossless.c"      /* C source */
  void *fallible_reallocarray(void *ptr, size_t nmemb, size_t size)
  {
    return Allocate_Success-- ? reallocarray(ptr, nmemb, size) : Λ;
  }
```

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

```
⟨ Global variables 6 ⟩ +≡
  cell Tmp_Test = NIL;
```

**224.**
```
⟨ Externalised global variables 7 ⟩ +≡
  extern cell Tmp_Test;
```

**225.**
```
⟨ Protected Globals 21 ⟩ +≡
#ifdef LL_TEST
  &Tmp_Test ,
#endif
```

**226.**
```
⟨ Pre-initialise Small_Int & other gc-sensitive buffers 15 ⟩ +≡
#ifdef LL_TEST
  Tmp_Test = NIL;
#endif
```

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

```
#define object_copy(o, d, p) object_copy_imp((o), (d), (p), 0)
#define object_copyref(o, d) object_copyref_imp((o), (d), 0)
⟨ Function declarations 8 ⟩ +≡
  void compile_testing_probe(cell, cell, boolean);
  void compile_testing_probe_app(cell, cell, boolean);
  boolean object_compare(char *, size_t, cell, boolean);
  int object_copy_imp(cell, char *, boolean, int);
  int object_copyref_imp(cell, cell *, int);
  size_t object_sizeof(cell);
  size_t object_sizeofref(cell);
  cell testing_build_probe(cell);
```

**228.** ⟨ Testing opcode names 228 ⟩ ≡

```
OP_TEST_PROBE ,
```

This code is used in sections 144 and 145.

**229.** ⟨ Testing opcodes 229 ⟩ ≡

```
[OP_TEST_PROBE] = { . name = "OP_VOV" , . nargs = 1 } ,
```

This code is used in section 148.

**230.** ⟨ Testing implementations 230 ⟩ ≡

```
case OP_TEST_PROBE:
  Acc = testing_build_probe(rts_pop(1));
  skip(1);
  break;
```

This code is used in section 110.

**231.** ⟨ Testing primitives 231 ⟩ ≡

```
{"test!probe", compile_testing_probe},
{"test!probe-applying", compile_testing_probe_app} ,
```

This code is used in section 499.

**232.**

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

**233.** 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);
}
```

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

```
size_t object_sizeof(cell o)
{
  size_t s;
  int i;
  if (special_p(o)) return sizeof(char);
  s = sizeof(char) + 2 * sizeof(cell);
  if (acar_p(o)) s += object_sizeof(car(o));
  if (acdr_p(o)) s += object_sizeof(cdr(o));
  if (vector_p(o)) {
    s += (vector_length(o) + VECTOR_HEAD) * sizeof(cell);
    for (i = 0; i < vector_length(o); i++) s += object_sizeof(vector_ref(o, i));
  }
  return s;
}
```

**235.** **int** *object_copy_imp*(**cell** *o*, **char** ∗*dst*, **boolean** *offset_p*, **int** *p*)

```
{
  int i;
  if (special_p(o)) dst[p++] = (char) o;
  else {
    bcopy(&tag(o), dst + p, sizeof(char));
    p++;
    if (¬vector_p(o))      /* car is gc's index */
      bcopy(&car(o), dst + p, sizeof(cell));
    else bzero(dst + p, sizeof(cell));
    p += sizeof(cell);
    if (¬vector_p(o) ∨ offset_p) bcopy(&cdr(o), dst + p, sizeof(cell));
    else bzero(dst + p, sizeof(cell));
    p += sizeof(cell);
    if (acar_p(o)) p = object_copy_imp(car(o), dst, offset_p, p);
    if (acdr_p(o)) p = object_copy_imp(cdr(o), dst, offset_p, p);
    if (vector_p(o)) {
      bcopy(&(vector_ref(o, 0)) − VECTOR_HEAD, dst + p, sizeof(cell) ∗ (vector_length(o) + VECTOR_HEAD));
      p += sizeof(cell) ∗ (vector_length(o) + VECTOR_HEAD);
      for (i = 0; i < vector_length(o); i++) p = object_copy_imp(vector_ref(o, i), dst, offset_p, p);
    }
  }
  return p;
}
```

**236.** **boolean** *object_compare*(**char** ∗*buf1*, **size_t** *len*, **cell** *o2*, **boolean** *offset_p*)

```
{
  char *buf2;
  boolean r;
  if (object_sizeof(o2) ≠ len) return bfalse;
  ERR_OOM_P(buf2 = malloc(len));
  object_copy(o2, buf2, offset_p);
  r = (bcmp(buf1, buf2, len) ≡ 0) ? btrue : bfalse;
  free(buf2);
  return r;
}
```

**237.** **size_t** *object_sizeofref*(**cell** *o*)

```
{
  int i;
  size_t s = 1;
  if (special_p(o)) return s;
  if (acar_p(o)) s += object_sizeofref(car(o));
  if (acdr_p(o)) s += object_sizeofref(cdr(o));
  if (vector_p(o))
    for (i = 0; i < vector_length(o); i++) s += object_sizeofref(vector_ref(o, i));
  return s;
}
```

**238.** **int** *object_copyref_imp*(**cell** *o*, **cell** ∗*dst*, **int** *p*)
```
{
  int i;
  dst[p++] = o;
  if (special_p(o)) return p;
  if (acar_p(o)) p = object_copyref_imp(car(o), dst, p);
  if (acdr_p(o)) p = object_copyref_imp(cdr(o), dst, p);
  if (vector_p(o))
    for (i = 0; i < vector_length(o); i++) p = object_copyref_imp(vector_ref(o, i), dst, p);
  return p;
}
```

**239.** **#define** SIZEOF_LLT_COPY (**sizeof**(**size_t**) + **sizeof**(**cell**))

⟨ Type definitions 5 ⟩ +≡
```
typedef struct {
  size_t size;
  cell origin;
  char buf[];
} llt_copy;
```

**240.** ⟨ Function declarations 8 ⟩ +≡
```
llt_copy *llt_copy_object(cell o, boolean with_offset_p);
```

**241.**
```
/* TODO merge these */
llt_copy *llt_copy_object(cell o, boolean with_offset_p)
{
  size_t s = object_sizeof(o);
  llt_copy *r;
  r = malloc(s + SIZEOF_LLT_COPY);
  ERR_OOM_P(r);
  r->size = s;
  r->origin = o;
  object_copy(o, r->buf, with_offset_p);
  return r;
}
```

**242.**

```
#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
```

**243.**

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

⟨ Old test executable wrapper 243 ⟩ ≡

```
#define LL_TEST 1
#include "lossless.h"
  void test_main(void);
  int main(int argc__unused, char **argv__unused)
  {
    volatile boolean first = btrue;
    vm_init();
    if (argc > 1) error (ERR_ARITY_EXTRA, NIL);
    vm_prepare();
    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 454, 466, 475, 484, 491, and 498.

**244.** 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 *);
  int test_count_free_list(void);
  char *test_msgf(char *, const char *, char *, ...);
  void test_vm_state(char *, int);
#endif
```

**245.** ⟨ Global variables 6 ⟩ +≡

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

**246.** ⟨ Externalised global variables 7 ⟩ +≡

```
extern int Test_Plan, Next_Test;
```

**247.**

```
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);
}
```

**248.** **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;
}
```

**249.** 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
⟨ Complex definitions & macros 144 ⟩ +≡
#define tmsgf(...) test_msgf (msg, prefix, __VA_ARGS__)
```

**250.** **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;
}
```

**251.** 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; */
  }
```

**252.**

```
  int test_count_free_list(void)
  {
    int r = 0;
    cell c = Cells_Free;
    if (¬Cells_Poolsize) return 0;
    while (¬null_p(c)) {
      r++;
      c = cdr(c);
    }
    return r;
  }
```

**253. 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 harness, without obscuring it with the more complicated unit test framework below.

⟨ t/sanity.c 253 ⟩ ≡

```
#define LL_TEST
#include "lossless.h"
  int main( )
  {
    tap_plan(1);
    vm_init( );
    vm_prepare( );
    vm_reset( );
    interpret( );
    tap_pass("LossLess␣compiles␣and␣runs");
  }
```

**254. 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.

```
⟨ t/llt.h 254 ⟩ ≡
#ifndef LLT_H
#define LLT_H
  ⟨ Unit test fixture header 255 ⟩
  typedef struct llt_Fixture llt_Fixture;      /* user-defined */
  typedef void (*llt_thunk)(llt_Fixture *);
  typedef boolean (*llt_unit)(llt_Fixture *);
  typedef llt_Fixture *(*llt_fixture)(void);
  extern llt_fixture Test_Fixtures[];      /* user-defined */
#define fmsgf(...)test_msgf (buf, fix.name, __VA_ARGS__)
#define fpmsgf(...)test_msgf (buf, fix→name, __VA_ARGS__)
  llt_Fixture *llt_alloc(size_t);
  boolean llt_main(llt_Fixture *);
  llt_Fixture *llt_prepare(void);
#endif      /* LLT_H */
```

**255.** Unit test fixtures are defiend in a **llt_Fixture** structure which is only declared in this header; it is up to each unit test to implement its own **llt_Fixture** with this common header.

```
⟨ Unit test fixture header 255 ⟩ ≡
#define LLT_FIXTURE_HEADER
      const char *name;
      const char *suffix;
      int id;
      int max;
      llt_thunk prepare;
      llt_thunk act;
      llt_unit test;
      llt_thunk destroy;
      boolean skip_gc_p      /* no semicolon */
```

This code is used in section 254.

**256.** The vast majority (all, so far) of unit tests follow the same simple structure. There are plans for more interactive tests but they aren't necessary yet.

```
⟨ Unit test header 256 ⟩ ≡
#define LL_TEST
#include "lossless.h"
#include "llt.h"
```

This code is used in sections 264, 283, 296, 329, 339, 354, 376, and 439.

**257.** ⟨ Unit test body 257 ⟩ ≡

```
int main(int argc__unused, char **argv__unused)
{
  llt_Fixture *suite;
  if (argc > 1) {
    printf("usage: %s", argv[0]);
    return EXIT_FAILURE;
  }
#ifndef LLT_NOINIT
  vm_init();
#endif
  suite = llt_prepare();
  llt_main(suite);
  free(suite);
  tap_plan(0);
}
```

See also sections 258, 259, and 262.

This code is used in sections 264, 283, 296, 329, 339, 354, 376, and 439.

**258.** ⟨ Unit test body 257 ⟩ +≡

```
llt_Fixture *llt_alloc(size_t n)
{
  llt_Fixture *f;
  size_t i;
  ERR_OOM_P(f = calloc(n, sizeof(llt_Fixture)));
  for (i = 0; i < n; i++) f[i].max = n;
  return f;
}
```

**259.** ⟨ Unit test body 257 ⟩ +≡

```
boolean llt_main(llt_Fixture *suite)
{
  int i;
  int d, f0, f1;
  boolean all, ok;
  char buf[TEST_BUFSIZE] = {0}, *name;
  all = btrue;
  for (i = 0; i < suite->max; i++) {
    if (suite[i].suffix) snprintf(buf, TEST_BUFSIZE, "%s (%s)", suite[i].name, suite[i].suffix);
    else snprintf(buf, TEST_BUFSIZE, "%s", suite[i].name);
    ⟨ Unit test a single fixture 260 ⟩
    if ((d = f0 - f1) > 0 ∧ ¬suite[i].skip_gc_p) {⟨ Repeat the fixture with garbage collection 261 ⟩}
    tap_more(all, ok, buf);
  }
  return all;
}
```

**260.** ⟨ Unit test a single fixture 260 ⟩ ≡

```
name = (char *) suite[i].name;
suite[i].name = (char *) buf;
if (suite[i].prepare) suite[i].prepare(suite + i);
f0 = test_count_free_list( );
suite[i].act(suite + i);
f1 = test_count_free_list( );
ok = suite[i].test(suite + i);
if (suite[i].destroy) suite[i].destroy(suite + i);
suite[i].name = name;
```

This code is used in section 259.

**261.** This is substantially the same as the previous section except that after the fixture is prepared *cons* is called repeatedly to waste cells before the fixture's action is taken.

⟨ Repeat the fixture with garbage collection 261 ⟩ ≡

```
int j, k;
for (j = d; j ≥ 0; j−−) {
  sprintf(buf, "%s: trigger gc at %d free cells", name, j);
  name = (char *) suite[i].name;
  suite[i].name = buf;
  if (suite[i].prepare) suite[i].prepare(suite + i);
  d = test_count_free_list( );
  for (k = 0; k < d − j; k++) cons(NIL, NIL);
  suite[i].act(suite + i);
  ok = suite[i].test(suite + i) ∧ ok;
  if (suite[i].destroy) suite[i].destroy(suite + i);
  suite[i].name = name;
}
if (suite[i].suffix) snprintf(buf, TEST_BUFSIZE, "%s (%s)", name, suite[i].suffix);
else snprintf(buf, TEST_BUFSIZE, "%s", name);
suite[i].name = buf;
```

This code is used in section 259.

**262.** ⟨ Unit test body 257 ⟩ +≡

```
llt_Fixture *llt_prepare(void)
{
  llt_fixture *s = Test_Fixtures, *t;
  llt_Fixture *r = Λ, *q, *p;
  int c = 0, i;
  int f = sizeof(llt_Fixture);
  for (t = s; *t; t++) {
    q = (*t)( );
    p = reallocarray(r, c + q->max, f);
    ERR_OOM_P(p);
    r = p;
    bcopy(q, r + c, f * q->max);
    c += q->max;
    free(q);
  }
  for (i = 0; i < c; i++) {
    r[i].id = i;
    r[i].max = c;
  }
  return r;
}
```

**263. 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.

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.

**264.** This unit test relies on the VM being uninitialised so that it can safely switch out the heap pointers. The *save_CAR*, *save_CDR* & *save_TAG* pointers in the fixture are convenience pointers into *heapcopy*.

```
⟨ t/cell-heap.c 264 ⟩ ≡
#define LLT_NOINIT
  ⟨ Unit test header 256 ⟩
  enum llt_Grow_Pool_result {
    LLT_GROW_POOL_SUCCESS, LLT_GROW_POOL_FAIL_CAR, LLT_GROW_POOL_FAIL_CDR,
        LLT_GROW_POOL_FAIL_TAG
  };
  struct llt_Fixture {
    LLT_FIXTURE_HEADER;
    enum llt_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;
  };
  ⟨ Unit test body 257 ⟩
  ⟨ Unit test: grow heap pool 265 ⟩
  llt_fixture Test_Fixtures[] = {
      llt_Grow_Pool__Initial_Success, 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
  };
```

**265.** ⟨ Unit test: grow heap pool 265 ⟩ ≡

```
void llt_Grow_Pool_prepare(llt_Fixture *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;
}
```

See also sections 266, 267, 268, 273, 274, 275, 276, 277, 278, 279, 280, 281, and 282.

This code is used in section 264.

**266.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

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

**267.** 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: grow heap pool 265 ⟩ +≡

```
void llt_Grow_Pool_act(llt_Fixture *fix)
{
  jmp_buf save_jmp;
  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));
}
```

**268.** ⟨ Unit test: grow heap pool 265 ⟩ +≡
```
boolean llt_Grow_Pool_test(llt_Fixture *fix)
{
  boolean ok;
  char buf[TEST_BUFSIZE] = {0};
  switch (fix→expect) {
  case LLT_GROW_POOL_SUCCESS:
    ⟨ Unit test part: grow heap pool, validate success 269 ⟩
    break;      /* TODO: test for bzero */
  case LLT_GROW_POOL_FAIL_CAR:
    ⟨ Unit test part: grow heap pool, validate car failure 270 ⟩
    break;
  case LLT_GROW_POOL_FAIL_CDR:
    ⟨ Unit test part: grow heap pool, validate cdr failure 271 ⟩
    break;
  case LLT_GROW_POOL_FAIL_TAG:
    ⟨ Unit test part: grow heap pool, validate tag failure 272 ⟩
    break;
  }
  return ok;
}
```

**269.** ⟨ Unit test part: grow heap pool, validate success 269 ⟩ ≡
```
ok = tap_ok(Cells_Poolsize ≡ (fix→Poolsize + fix→Segment), fpmsgf("Cells_Poolsize is increased"));
tap_more(ok, Cells_Segment ≡ (fix→Poolsize + fix→Segment)/2, fpmsgf("Cells_Segment is increased"));
tap_more(ok, CAR ≠ CDR ∧ CAR ≠ (cell *) TAG, fpmsgf("CAR, CDR & TAG are unique"));
tap_more(ok, CAR ≠ Λ, fpmsgf("CAR is not NULL"));
tap_more(ok, ¬bcmp(CAR, fix→save_CAR, sizeof(cell)*fix→Poolsize), fpmsgf("CAR heap is unchanged"));
tap_more(ok, CDR ≠ Λ, fpmsgf("CDR is not NULL"));
tap_more(ok, ¬memcmp(CDR, fix→save_CDR, sizeof(cell) * fix→Poolsize),
    fpmsgf("CDR heap is unchanged"));
tap_more(ok, TAG ≠ Λ, fpmsgf("TAG is not NULL"));
tap_more(ok, ¬memcmp(TAG, fix→save_TAG, sizeof(char) * fix→Poolsize),
    fpmsgf("TAG heap is unchanged"));
```
This code is used in section 268.

**270.** ⟨ Unit test part: grow heap pool, validate car failure 270 ⟩ ≡
```
ok = tap_ok(Cells_Poolsize ≡ fix→Poolsize, fpmsgf("Cells_Poolsize is not increased"));
tap_more(ok, Cells_Segment ≡ fix→Segment, fpmsgf("Cells_Segment is not increased"));
tap_more(ok, CAR ≡ fix→CAR, fpmsgf("CAR is unchanged"));
tap_more(ok, CDR ≡ fix→CDR, fpmsgf("CDR is unchanged"));
tap_more(ok, TAG ≡ fix→TAG, fpmsgf("TAG is unchanged"));
```
This code is used in section 268.

**271.** ⟨ Unit test part: grow heap pool, validate cdr failure 271 ⟩ ≡
```
ok = tap_ok(Cells_Poolsize ≡ fix→Poolsize, fpmsgf("Cells_Poolsize is not increased"));
tap_more(ok, Cells_Segment ≡ fix→Segment, fpmsgf("Cells_Segment is not increased"));
tap_more(ok, ¬memcmp(CAR, fix→save_CAR, sizeof(cell) * fix→Poolsize),
    fpmsgf("CAR heap is unchanged"));
tap_more(ok, CDR ≡ fix→CDR, fpmsgf("CDR is unchanged"));
tap_more(ok, TAG ≡ fix→TAG, fpmsgf("TAG is unchanged"));
```
This code is used in section 268.

**272.** ⟨ Unit test part: grow heap pool, validate tag failure 272 ⟩ ≡

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

This code is used in section 268.

**273.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_Grow_Pool_prepare;
  fix→destroy = llt_Grow_Pool_destroy;
  fix→act = llt_Grow_Pool_act;
  fix→test = llt_Grow_Pool_test;
  fix→skip_gc_p = btrue;
  fix→expect = LLT_GROW_POOL_SUCCESS;
  fix→allocations = -1;
  fix→Segment = HEAP_SEGMENT;
  return fix;
}
```

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

⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Initial_Success(void)
{
  return llt_Grow_Pool_fix(llt_alloc(1), __func__);
}
```

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

⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Immediate_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_CAR;
  fix→allocations = 0;
  return fix;
}
```

**276.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Second_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_CDR;
  fix→allocations = 1;
  return fix;
}
```

**277.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Third_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_TAG;
  fix→allocations = 2;
  return fix;
}
```

**278.** Data already on the heap must be preserved exactly.

⟨ Unit test: grow heap pool 265 ⟩ +≡

```
void llt_Grow_Pool__fill(llt_Fixture *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();
}
```

**279.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Full_Success(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→Poolsize = HEAP_SEGMENT;
  llt_Grow_Pool__fill(fix);
  return fix;
}
```

**280.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Full_Immediate_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_CAR;
  fix→allocations = 0;
  fix→Poolsize = HEAP_SEGMENT;
  llt_Grow_Pool__fill(fix);
  return fix;
}
```

**281.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Full_Second_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_CDR;
  fix→allocations = 1;
  fix→Poolsize = HEAP_SEGMENT;
  llt_Grow_Pool__fill(fix);
  return fix;
}
```

**282.** ⟨ Unit test: grow heap pool 265 ⟩ +≡

```
llt_Fixture *llt_Grow_Pool__Full_Third_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_POOL_FAIL_TAG;
  fix→allocations = 2;
  fix→Poolsize = HEAP_SEGMENT;
  llt_Grow_Pool__fill(fix);
  return fix;
}
```

**283. 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.

```
⟨ t/vector-heap.c 283 ⟩ ≡
#define LLT_NOINIT
  ⟨ Unit test header 256 ⟩
  enum llt_Grow_Vector_Pool_result {
    LLT_GROW_VECTOR_POOL_SUCCESS, LLT_GROW_VECTOR_POOL_FAIL
  };
  struct llt_Fixture {
    LLT_FIXTURE_HEADER;
    enum llt_Grow_Vector_Pool_result expect;
    int allocations;
    int Poolsize;
    int Segment;
    cell *VECTOR;
    cell *save_VECTOR;
  };
  ⟨ Unit test body 257 ⟩
  ⟨ Unit test: grow vector pool 284 ⟩
  llt_fixture Test_Fixtures[] = {
      llt_Grow_Vector_Pool__Empty_Success, llt_Grow_Vector_Pool__Empty_Fail,
      llt_Grow_Vector_Pool__Full_Success, llt_Grow_Vector_Pool__Full_Fail, Λ
  };
```

**284.** ⟨ Unit test: grow vector pool 284 ⟩ ≡

```
  void llt_Grow_Vector_Pool_prepare(llt_Fixture *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;
  }
```

See also sections 285, 286, 287, 290, 291, 292, 293, 294, and 295.

This code is used in section 283.

**285.** ⟨ Unit test: grow vector pool 284 ⟩ +≡

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

**286.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
void llt_Grow_Vector_Pool_act(llt_Fixture *fix)
{
  jmp_buf save_jmp;
  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));
}
```

**287.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
boolean llt_Grow_Vector_Pool_test(llt_Fixture *fix)
{
  boolean ok;
  char buf[TEST_BUFSIZE] = {0};
  switch (fix→expect) {
  case LLT_GROW_VECTOR_POOL_SUCCESS:
    ⟨ Unit test part: grow vector pool, validate success 288 ⟩
    break;      /* TODO: test for bzero */
  case LLT_GROW_VECTOR_POOL_FAIL:
    ⟨ Unit test part: grow vector pool, validate failure 289 ⟩
    break;
  }
  return ok;
}
```

**288.** ⟨ Unit test part: grow vector pool, validate success 288 ⟩ ≡
```
ok = tap_ok(Vectors_Poolsize ≡ (fix→Poolsize + fix→Segment),
    fpmsgf("Vectors_Poolsize␣is␣increased"));
tap_more(ok, Vectors_Segment ≡ (fix→Poolsize + fix→Segment)/2,
    fpmsgf("Vectors_Segment␣is␣increased"));
tap_more(ok, VECTOR ≠ Λ, fpmsgf("VECTOR␣is␣not␣NULL"));
tap_more(ok, ¬bcmp(VECTOR, fix→save_VECTOR, sizeof(cell) * fix→Poolsize),
    fpmsgf("VECTOR␣heap␣is␣unchanged"));
```
This code is used in section 287.

**289.** ⟨ Unit test part: grow vector pool, validate failure 289 ⟩ ≡
```
ok = tap_ok(Vectors_Poolsize ≡ fix→Poolsize, fpmsgf("Vectors_Poolsize␣is␣not␣increased"));
tap_more(ok, Vectors_Segment ≡ fix→Segment, fpmsgf("Vectors_Segment␣is␣not␣increased"));
tap_more(ok, VECTOR ≡ fix→VECTOR, fpmsgf("VECTOR␣is␣unchanged"));
```
This code is used in section 287.

**290.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
llt_Fixture *llt_Grow_Vector_Pool_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_Grow_Vector_Pool_prepare;
  fix→destroy = llt_Grow_Vector_Pool_destroy;
  fix→act = llt_Grow_Vector_Pool_act;
  fix→test = llt_Grow_Vector_Pool_test;
  fix→skip_gc_p = btrue;
  fix→expect = LLT_GROW_VECTOR_POOL_SUCCESS;
  fix→allocations = -1;
  fix→Segment = HEAP_SEGMENT;
  return fix;
}
```

**291.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
llt_Fixture *llt_Grow_Vector_Pool__Empty_Success(void)
{
  return llt_Grow_Vector_Pool_fix(llt_alloc(1), __func__);
}
```

**292.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
llt_Fixture *llt_Grow_Vector_Pool__Empty_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Vector_Pool_fix(llt_alloc(1), __func__);
  fix→expect = LLT_GROW_VECTOR_POOL_FAIL;
  fix→allocations = 0;
  return fix;
}
```

**293.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
void llt_Grow_Vector_Pool__fill(llt_Fixture *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( );
}
```

**294.** ⟨ Unit test: grow vector pool 284 ⟩ +≡
```
llt_Fixture *llt_Grow_Vector_Pool__Full_Success(void)
{
  llt_Fixture *fix = llt_Grow_Vector_Pool_fix(llt_alloc(1), __func__);
  fix→Poolsize = HEAP_SEGMENT;
  llt_Grow_Vector_Pool__fill(fix);
  return fix;
}
```

**295.** ⟨ Unit test: grow vector pool 284 ⟩ +≡

```
llt_Fixture *llt_Grow_Vector_Pool__Full_Fail(void)
{
  llt_Fixture *fix = llt_Grow_Vector_Pool_fix(llt_alloc(1), __func__);
  fix->expect = LLT_GROW_VECTOR_POOL_FAIL;
  fix->allocations = 0;
  fix->Poolsize = HEAP_SEGMENT;
  llt_Grow_Vector_Pool__fill(fix);
  return fix;
}
```

**296. Garbage Collector.** There are three parts to the garbage collector, each building on the last. The inner-most component is *mark* which searches the heap for any data which are in use.

1. *What is the contract fulfilled by the code under test?*
2. *What preconditions are required, and how are they enforced?*
3. *What postconditions are guaranteed?*

Given a **cell**, it and any objects it refers to—recursively, including internal components of atoms—will have their mark flag raised. No other objects will be affected and no other changes will be made to the objects which are. The global constants (specials) are ignored.

*mark*'s main complication is that it's a linear implementation of a recursive algorithm. It can't use any of the real stacks to keep track of the recursion so it uses the individual cells its scanning as an impromptu stack. This heap mutation needs to have no visible external effect despite mutating every **cell** that's considered.

4. *What example inputs trigger different behaviors?*

Global constants and cells already marked vs. unmarked cells. Obviously different objects will be marked in their own way.

Constants aside, the different types of object come in one of 5 categories: pairs, vectors, atomic pairs, atomic lists (the car is opaque) and pure atoms (which are entirely opaque). These are referred to as P, V, A & L respectively.

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

A test for each type of object—P, V, A & L as well as globals—created without any nesting and one for each recursive combination up to a depth of 3.

⟨ t/gc-mark.c 296 ⟩ ≡

```
⟨ Unit test header 256 ⟩
enum llt_GC_Mark_flat {
  LLT_GC_MARK_SIMPLE_ATOM, LLT_GC_MARK_SIMPLE_LONG_ATOM, LLT_GC_MARK_SIMPLE_PAIR,
      LLT_GC_MARK_SIMPLE_VECTOR
};
enum llt_GC_Mark_recursion {
  LLT_GC_MARK_RECURSIVE_PA, LLT_GC_MARK_RECURSIVE_PL, LLT_GC_MARK_RECURSIVE_PP,
      LLT_GC_MARK_RECURSIVE_PV, LLT_GC_MARK_RECURSIVE_PLL, LLT_GC_MARK_RECURSIVE_VA,
      LLT_GC_MARK_RECURSIVE_VL, LLT_GC_MARK_RECURSIVE_VP, LLT_GC_MARK_RECURSIVE_VV,
      LLT_GC_MARK_RECURSIVE_VLL, LLT_GC_MARK_RECURSIVE_LL, LLT_GC_MARK_RECURSIVE_LLL,
      LLT_GC_MARK_RECURSIVE_PPA, LLT_GC_MARK_RECURSIVE_PPL, LLT_GC_MARK_RECURSIVE_PPP,
      LLT_GC_MARK_RECURSIVE_PPV, LLT_GC_MARK_RECURSIVE_PVA, LLT_GC_MARK_RECURSIVE_PVL,
      LLT_GC_MARK_RECURSIVE_PVP, LLT_GC_MARK_RECURSIVE_PVV, LLT_GC_MARK_RECURSIVE_VPA,
      LLT_GC_MARK_RECURSIVE_VPL, LLT_GC_MARK_RECURSIVE_VPP, LLT_GC_MARK_RECURSIVE_VPV,
      LLT_GC_MARK_RECURSIVE_VVA, LLT_GC_MARK_RECURSIVE_VVL, LLT_GC_MARK_RECURSIVE_VVP,
      LLT_GC_MARK_RECURSIVE_VVV
};
struct llt_Fixture {
  LLT_FIXTURE_HEADER;
  cell safe;
  char *copy;
  size_t len;
  boolean proper_pair_p;
  enum llt_GC_Mark_recursion complex;
  enum llt_GC_Mark_flat simplex;
};
⟨ Unit test body 257 ⟩
⟨ Unit test: garbage collector mark 297 ⟩
```

```
llt_fixture Test_Fixtures[] = {
    llt_GC_Mark__Global, llt_GC_Mark__Atom, llt_GC_Mark__Long_Atom, llt_GC_Mark__Pair,
    llt_GC_Mark__Vector, llt_GC_Mark__Recursive_P, llt_GC_Mark__Recursive_V,
    llt_GC_Mark__Recursive_L, llt_GC_Mark__Recursive_PP, llt_GC_Mark__Recursive_PV,
    llt_GC_Mark__Recursive_VP, llt_GC_Mark__Recursive_VV, Λ
};
```

**297.** These tests work by serialising the object under test into a buffer before and after performing the test to check for changes, and recursively walking the data structure using C's stack to look for the mark flag.

```
⟨ Unit test: garbage collector mark 297 ⟩ ≡
  boolean llt_GC_Mark_is_marked_p(cell c)
  {
    return special_p(c) ∨ (mark_p(c)
        ∧ (¬acar_p(c) ∨ llt_GC_Mark_is_marked_p(car(c)))
        ∧ (¬acdr_p(c) ∨ llt_GC_Mark_is_marked_p(cdr(c))));
  }
```

See also sections 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 322, 323, 324, 325, 326, 327, and 328.

This code is used in section 296.

**298.** Of course after the mark phase of garbage collection live objects *have* been changed because that's the whole point so serialising the post-mark object as-is wouldn't work. Instead the flag is (recursively) lowered first reverting the only change that *mark* should have made.

```
⟨ Unit test: garbage collector mark 297 ⟩ +≡
  void llt_GC_Mark_unmark_m(cell c)
  {
    int i;
    if (special_p(c)) return;
    mark_clear(c);
    if (acar_p(c)) llt_GC_Mark_unmark_m(car(c));
    if (acdr_p(c)) llt_GC_Mark_unmark_m(cdr(c));
    if (vector_p(c))
      for (i = 0; i < vector_length(c); i++)
        llt_GC_Mark_unmark_m(vector_ref(c, i));
  }
```

**299.** Objects need to be created in various combinations to create the recursive structures to test.

```
#define llt_GC_Mark_mkatom sym
⟨ Unit test: garbage collector mark 297 ⟩ +≡
  cell llt_GC_Mark_mklong(int x, int y)
  {
    cell r;
    vms_push(int_new(y));
    r = int_new(x);
    cdr(r) = vms_pop( );
    return r;
  }
```

**300.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
cell llt_GC_Mark_mklonglong(int x, int y, int z)
{
  cell r;
  vms_push(int_new(z));
  r = int_new(y);
  cdr(r) = vms_pop( );
  vms_push(r);
  r = int_new(x);
  cdr(r) = vms_pop( );
  return r;
}
```

**301.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
cell llt_GC_Mark_mkpair(boolean proper_p)
{
  cell r = cons(VOID, UNDEFINED);
  if (proper_p) cdr(r) = NIL;
  return r;
}
```

**302.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
cell llt_GC_Mark_mkvector(void)
{
  cell r;
  int i, j;
  r = vector_new_imp(abs(UNDEFINED), 0, NIL);
  for (i = 0, j = -1; j ≥ UNDEFINED; i++, j--) vector_ref(r, i) = j;
  return r;
}
```

**303.** Preparing and running the tests. This is where the object under test (created below) gets serialised.

⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
void llt_GC_Mark_prepare(llt_Fixture *fix)
{
  fix→len = object_sizeof(fix→safe);
  ERR_OOM_P(fix→copy = malloc(fix→len));
  object_copy(fix→safe, fix→copy, btrue);
}
```

**304.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
void llt_GC_Mark_destroy(llt_Fixture *fix)
{
  free(fix→copy);
}
```

**305.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
void llt_GC_Mark_act(llt_Fixture *fix)
{
  mark(fix→safe);
}
```

**306.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
boolean llt_GC_Mark_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE];
  boolean ok;
  ok = tap_ok(llt_GC_Mark_is_marked_p(fix→safe), fpmsgf("the object is fully marked"));
  llt_GC_Mark_unmark_m(fix→safe);
  tap_again(ok, object_compare(fix→copy, fix→len, fix→safe, btrue), fpmsgf("the object is unchanged"));
  return ok;
}
```

**307.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_GC_Mark_prepare;
  fix→destroy = llt_GC_Mark_destroy;
  fix→act = llt_GC_Mark_act;
  fix→test = llt_GC_Mark_test;
  fix→safe = NIL;
  return fix;
}
```

**308.** This defines 6 test cases, one for each global object, which need no further preparation.

⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
#define mkfix(n, o) do
  {
    llt_GC_Mark_fix(f + (n), __func__);
    f[(n)].suffix = #o;
    f[(n)].safe = (o);
  }
  while (0)
  llt_Fixture *llt_GC_Mark__Global(void)
  {
    llt_Fixture *f = llt_alloc(6);
    mkfix(0, NIL);
    mkfix(1, FALSE);
    mkfix(2, TRUE);
    mkfix(3, END_OF_FILE);
    mkfix(4, VOID);
    mkfix(5, UNDEFINED);
    return f;
  }
#undef mkfix
```

**309.** Four test cases test each of the other object types without triggering recursion.

```
⟨ Unit test: garbage collector mark 297 ⟩ +≡
  void llt_GC_Mark__PLAV_prepare(llt_Fixture *fix)
  {
    switch (fix→simplex) {
    case LLT_GC_MARK_SIMPLE_ATOM:
      fix→safe = llt_GC_Mark_mkatom("forty-two");
      break;
    case LLT_GC_MARK_SIMPLE_LONG_ATOM:
      fix→safe = int_new(42);      /* nb. doesn't use mklong */
      break;
    case LLT_GC_MARK_SIMPLE_PAIR:
      fix→safe = llt_GC_Mark_mkpair(fix→proper_pair_p);
      break;
    case LLT_GC_MARK_SIMPLE_VECTOR:
      fix→safe = llt_GC_Mark_mkvector( );
      break;
    }
    llt_GC_Mark_prepare(fix);
  }
```

**310.**
```
⟨ Unit test: garbage collector mark 297 ⟩ +≡
  llt_Fixture *llt_GC_Mark__Atom(void)
  {
    llt_Fixture *f = llt_alloc(1);
    llt_GC_Mark_fix(f, __func__);
    f→simplex = LLT_GC_MARK_SIMPLE_ATOM;
    f→prepare = llt_GC_Mark__PLAV_prepare;
    return f;
  }
```

**311.**
```
⟨ Unit test: garbage collector mark 297 ⟩ +≡
  llt_Fixture *llt_GC_Mark__Long_Atom(void)
  {
    llt_Fixture *f = llt_alloc(1);
    llt_GC_Mark_fix(f, __func__);
    f→simplex = LLT_GC_MARK_SIMPLE_LONG_ATOM;
    f→prepare = llt_GC_Mark__PLAV_prepare;
    return f;
  }
```

**312.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark__Pair(void)
{
  llt_Fixture *f = llt_alloc(2);
  llt_GC_Mark_fix(f + 0, __func__);
  llt_GC_Mark_fix(f + 1, __func__);
  f[0].simplex = f[1].simplex = LLT_GC_MARK_SIMPLE_PAIR;
  f[0].prepare = f[1].prepare = llt_GC_Mark__PLAV_prepare;
  f[0].proper_pair_p = btrue;
  return f;
}
```

**313.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark__Vector(void)
{
  llt_Fixture *f = llt_alloc(1);
  llt_GC_Mark_fix(f, __func__);
  f→simplex = LLT_GC_MARK_SIMPLE_VECTOR;
  f→prepare = llt_GC_Mark__PLAV_prepare;
  return f;
}
```

**314.** Preparing the recursive test cases involves a lot of repetetive and methodical code.

⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
void llt_GC_Mark__Recursive_prepare_imp(llt_Fixture *fix, enum llt_GC_Mark_recursion c)
{
  switch (c) {
    ⟨ Unit test part: prepare plain pairs 315 ⟩
    ⟨ Unit test part: prepare plain vectors 316 ⟩
    ⟨ Unit test part: prepare atomic lists 317 ⟩
    ⟨ Unit test part: prepare pairs in pairs 318 ⟩
    ⟨ Unit test part: prepare vectors in pairs 319 ⟩
    ⟨ Unit test part: prepare pairs in vectors 320 ⟩
    ⟨ Unit test part: prepare vectors in vectors 321 ⟩
  }
}
void llt_GC_Mark__Recursive_prepare(llt_Fixture *fix)
{
  llt_GC_Mark__Recursive_prepare_imp(fix, fix→complex);
  Tmp_Test = NIL;
  llt_GC_Mark_prepare(fix);
}
```

**315.** ⟨ Unit test part: prepare plain pairs 315 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_PA: fix→safe = llt_GC_Mark_mkpair(bfalse);
  car(fix→safe) = llt_GC_Mark_mkatom("forty-two");
  cdr(fix→safe) = llt_GC_Mark_mkatom("twoty-four");
  break;
case LLT_GC_MARK_RECURSIVE_PL: fix→safe = llt_GC_Mark_mkpair(bfalse);
  car(fix→safe) = llt_GC_Mark_mklong(2048, 42);
  cdr(fix→safe) = llt_GC_Mark_mklong(8042, 24);
  break;
case LLT_GC_MARK_RECURSIVE_PP: fix→safe = llt_GC_Mark_mkpair(bfalse);
  car(fix→safe) = llt_GC_Mark_mkpair(btrue);
  cdr(fix→safe) = llt_GC_Mark_mkpair(bfalse);
  break;
case LLT_GC_MARK_RECURSIVE_PV: fix→safe = llt_GC_Mark_mkpair(bfalse);
  car(fix→safe) = llt_GC_Mark_mkvector( );
  cdr(fix→safe) = llt_GC_Mark_mkvector( );
  break;
case LLT_GC_MARK_RECURSIVE_PLL: fix→safe = llt_GC_Mark_mkpair(bfalse);
  car(fix→safe) = llt_GC_Mark_mklonglong(1024, 2048, 42);
  cdr(fix→safe) = llt_GC_Mark_mklonglong(4201, 4820, 24);
  break;
```

This code is used in section 314.

**316.** ⟨ Unit test part: prepare plain vectors 316 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_VA: fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = llt_GC_Mark_mkatom("42");
  vector_ref(fix→safe, 2) = llt_GC_Mark_mkatom("24");
  break;
case LLT_GC_MARK_RECURSIVE_VL: fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = llt_GC_Mark_mklong(2048, 42);
  vector_ref(fix→safe, 2) = llt_GC_Mark_mklong(8042, 24);
  break;
case LLT_GC_MARK_RECURSIVE_VP: fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = llt_GC_Mark_mkpair(btrue);
  vector_ref(fix→safe, 2) = llt_GC_Mark_mkpair(bfalse);
  break;
case LLT_GC_MARK_RECURSIVE_VV: fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 2) = llt_GC_Mark_mkvector( );
  break;
case LLT_GC_MARK_RECURSIVE_VLL: fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = llt_GC_Mark_mklonglong(1024, 2048, 42);
  vector_ref(fix→safe, 2) = llt_GC_Mark_mklonglong(4201, 4820, 24);
  break;
```

This code is used in section 314.

**317.** ⟨ Unit test part: prepare atomic lists 317 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_LL: fix→safe = llt_GC_Mark_mklong(1024, 42);
  break;
case LLT_GC_MARK_RECURSIVE_LLL: fix→safe = llt_GC_Mark_mklonglong(1024, 2048, 42);
  break;
```

This code is used in section 314.

**318.** ⟨ Unit test part: prepare pairs in pairs 318 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_PPA:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PA);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PA);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PPL:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PL);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PL);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PPP:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PP);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PP);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PPV:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PV);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_PV);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
```

This code is used in section 314.

**319.** ⟨ Unit test part: prepare vectors in pairs 319 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_PVA:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VA);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VA);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PVL:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VL);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VL);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PVP:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VP);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VP);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_PVV:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VV);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VV);
  fix→safe = cons(fix→safe, Tmp_Test);
  break;
```

This code is used in section 314.

**320.** ⟨ Unit test part: prepare pairs in vectors 320 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_VPA:
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PA);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PA);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VPL:
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PL);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PL);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VPP:
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PP);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PP);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VPV:
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PV);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix,LLT_GC_MARK_RECURSIVE_PV);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
```

This code is used in section 314.

**321.** ⟨ Unit test part: prepare vectors in vectors 321 ⟩ ≡

```
case LLT_GC_MARK_RECURSIVE_VVA:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VA);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VA);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VVL:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VL);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VL);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VVP:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VP);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VP);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
case LLT_GC_MARK_RECURSIVE_VVV:
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VV);
  Tmp_Test = fix→safe;
  llt_GC_Mark__Recursive_prepare_imp(fix, LLT_GC_MARK_RECURSIVE_VV);
  Tmp_Test = cons(fix→safe, Tmp_Test);
  fix→safe = llt_GC_Mark_mkvector( );
  vector_ref(fix→safe, 4) = car(Tmp_Test);
  vector_ref(fix→safe, 2) = cdr(Tmp_Test);
  break;
```

This code is used in section 314.

**322.** **#define** *llt_GC_Mark_recfix*$(f, n, c)$ **do**
```
{
  llt_GC_Mark_fix(f + (n), __func__);
  f[(n)].prepare = llt_GC_Mark__Recursive_prepare;
  f[(n)].complex = (c);
  f[(n)].suffix = #c;
}
while (0)
```

⟨ Unit test: garbage collector *mark* 297 ⟩ +≡
```
llt_Fixture *llt_GC_Mark__Recursive_P(void)
{
  llt_Fixture *f = llt_alloc(5);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_PA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_PL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_PP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_PV);
  llt_GC_Mark_recfix(f, 4, LLT_GC_MARK_RECURSIVE_PLL);
  return f;
}
```

**323.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡
```
llt_Fixture *llt_GC_Mark__Recursive_V(void)
{
  llt_Fixture *f = llt_alloc(5);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_VA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_VL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_VP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_VV);
  llt_GC_Mark_recfix(f, 4, LLT_GC_MARK_RECURSIVE_VLL);
  return f;
}
```

**324.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡
```
llt_Fixture *llt_GC_Mark__Recursive_L(void)
{
  llt_Fixture *f = llt_alloc(2);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_LL);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_LLL);
  return f;
}
```

**325.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡
```
llt_Fixture *llt_GC_Mark__Recursive_PP(void)
{
  llt_Fixture *f = llt_alloc(4);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_PPA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_PPL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_PPP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_PPV);
  return f;
}
```

**326.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark__Recursive_PV(void)
{
  llt_Fixture *f = llt_alloc(4);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_PVA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_PVL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_PVP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_PVV);
  return f;
}
```

**327.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark__Recursive_VP(void)
{
  llt_Fixture *f = llt_alloc(4);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_VPA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_VPL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_VPP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_VPV);
  return f;
}
```

**328.** ⟨ Unit test: garbage collector *mark* 297 ⟩ +≡

```
llt_Fixture *llt_GC_Mark__Recursive_VV(void)
{
  llt_Fixture *f = llt_alloc(4);
  llt_GC_Mark_recfix(f, 0, LLT_GC_MARK_RECURSIVE_VVA);
  llt_GC_Mark_recfix(f, 1, LLT_GC_MARK_RECURSIVE_VVL);
  llt_GC_Mark_recfix(f, 2, LLT_GC_MARK_RECURSIVE_VVP);
  llt_GC_Mark_recfix(f, 3, LLT_GC_MARK_RECURSIVE_VVV);
  return f;
}
```

**329. Sweep.**

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

All cells which are marked will become unmarked and otherwise unchanged. All other cells will be on the free list (in an insignificant order). The size of the free list will be returned.

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

The pool need not have been initialised in which case the free list and return value are `NIL` and 0 respectively. Live objects should be already marked for which we define *llt_GC_Sweep_mark_m* which also counts the size of the object being marked.

3. *What postconditions are guaranteed?*

The *car* content of a cell put into the free list is unchanged but this doesn't matter.

4. *What example inputs trigger different behaviors?*

Exactly 2: The size of the pool and the set of marked cells.

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

There are three tests. The simplest is to verify that *sweep* is effectively a no-op when there is no pool. The other two both prepare a dud object which should be returned to the free list and test whether *sweep* works correctly both with and without a live object.

⟨ `t/gc-sweep.c` 329 ⟩ ≡

```
⟨ Unit test header 256 ⟩
struct llt_Fixture {
  LLT_FIXTURE_HEADER;
  boolean preinit_p;
  cell safe;
  cell *safe_buf;
  size_t expect;
  int ret_val;
};
⟨ Unit test body 257 ⟩
⟨ Unit test: garbage collector sweep 330 ⟩
llt_fixture Test_Fixtures[] = {
    llt_GC_Sweep__Empty_Pool, llt_GC_Sweep__Used_Pool, Λ
};
```

**330.** ⟨ Unit test: garbage collector *sweep* 330 ⟩ ≡

```
size_t llt_GC_Sweep_mark_m(cell c)
{
  int i;
  size_t count = 0;
  if (special_p(c)) return 0;
  mark_set(c);
  count++;
  if (acar_p(c)) count += llt_GC_Sweep_mark_m(car(c));
  if (acdr_p(c)) count += llt_GC_Sweep_mark_m(cdr(c));
  if (vector_p(c))
    for (i = 0; i < vector_length(c); i++)
      count += llt_GC_Sweep_mark_m(vector_ref(c, i));
  return count;
}
```

See also sections 331, 332, 333, 334, 335, 336, 337, and 338.

This code is used in section 329.

**331.** To test *sweep* when there is no pool there's no need to actually remove the pool. In other cases a few cells are consumed from the free list and ignored.

```
⟨ Unit test: garbage collector sweep 330 ⟩ +≡
  void llt_GC_Sweep_prepare(llt_Fixture *fix)
  {
    if (fix→preinit_p) {
      Cells_Poolsize = 0;
      return;
    }
    vms_push(cons(NIL, NIL));
    cons(NIL, vms_pop( ));
  }
```

**332.** The VM is fully reset after every test.

```
⟨ Unit test: garbage collector sweep 330 ⟩ +≡
  void llt_GC_Sweep_destroy(llt_Fixture *fix__unused)
  {
    free(fix→safe_buf);
    vm_init_imp( );
  }
```

**333.**

```
⟨ Unit test: garbage collector sweep 330 ⟩ +≡
  void llt_GC_Sweep_act(llt_Fixture *fix)
  {
    fix→ret_val = sweep( );
  }
```

**334.** ⟨ Unit test: garbage collector *sweep* 330 ⟩ +≡

```
boolean llt_GC_Sweep_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE] = {0};
  cell f;
  boolean ok, mark_ok_p, free_ok_p;
  int i, rem;
  rem = Cells_Poolsize - fix→expect;
  ok = tap_ok(fix→ret_val ≡ rem, fpmsgf("sweep returns the number of free cells (%d)", rem));
  i = test_count_free_list();
  tap_more(ok, i ≡ rem, fpmsgf("the number of free cells is correct (%d)", rem));
  mark_ok_p = btrue;
  for (i = 0; i < (int) fix→expect; i++)
    if (mark_p(fix→safe_buf[i]))
      mark_ok_p = bfalse;
  tap_more(ok, mark_ok_p, fpmsgf("the cells are unmarked"));
  free_ok_p = btrue;
  for (f = Cells_Free; ¬null_p(f); f = cdr(f))
    for (i = 0; i < (int) fix→expect; i++)
      if (fix→safe_buf[i] ≡ f)
        free_ok_p = bfalse;
  tap_more(ok, mark_ok_p, fpmsgf("the used cells are not in the free list"));
  return ok;
}
```

**335.** ⟨ Unit test: garbage collector *sweep* 330 ⟩ +≡

```
llt_Fixture *llt_GC_Sweep_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_GC_Sweep_prepare;
  fix→destroy = llt_GC_Sweep_destroy;
  fix→act = llt_GC_Sweep_act;
  fix→test = llt_GC_Sweep_test;
  fix→skip_gc_p = btrue;
  return fix;
}
```

**336.** ⟨ Unit test: garbage collector *sweep* 330 ⟩ +≡

```
llt_Fixture *llt_GC_Sweep__Empty_Pool(void)
{
  llt_Fixture *f = llt_alloc(2);
  llt_GC_Sweep_fix(f + 0, __func__);
  llt_GC_Sweep_fix(f + 1, __func__);
  f[0].preinit_p = btrue;
  f[0].suffix = "no pool";
  f[1].suffix = "unused";
  return f;
}
```

**337.** References to the cells which make up the object are saved in *fix→safe_buf* to check that they were not put on the free list.

```
⟨ Unit test: garbage collector sweep 330 ⟩ +≡
  void llt_GC_Sweep__Used_Pool_prepare(llt_Fixture *fix)
  {
    fix→safe = cons(VOID, UNDEFINED);
    vms_push(fix→safe);
    fix→expect = llt_GC_Sweep_mark_m(vms_ref());
    ERR_OOM_P(fix→safe_buf = malloc(fix→expect));
    object_copyref(fix→safe, fix→safe_buf);
    llt_GC_Sweep_prepare(fix);
    vms_pop();
  }
```

**338.**

```
⟨ Unit test: garbage collector sweep 330 ⟩ +≡
  llt_Fixture *llt_GC_Sweep__Used_Pool(void)
  {
    llt_Fixture *f = llt_alloc(1);
    llt_GC_Sweep_fix(f + 0, __func__);
    f[0].prepare = llt_GC_Sweep__Used_Pool_prepare;
    return f;
  }
```

**339. Vectors.**

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

*vector* objects which are not live (pointed at by something in ROOTS) will have their *tag* changed to TAG_NONE and their cdr née offset changed to a pointer in the free list, as will all their contents.

Live *vector*s cell pointer, length and contents are unchanged. The offset will be reduced by the (full) size of any unused *vector*s prior to it in VECTOR.

The number of free cells in VECTOR is returned.

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

Used vectors must be pointed to from something in ROOTS. They will be pushed into VMS.

The linear nature of *vector_new* is taken advantage of to create the holes in the VECTOR buffer that *gc_vectors* must defragment.

All other aspects of the garbage collector are assumed to work correctly.

3. *What postconditions are guaranteed?*

The VM is fully reset after each test so that they begin with VECTOR in a clean state.

4. *What example inputs trigger different behaviors?*

The only things to affect the way *vector* garbage collection works is whether or not each vector is live and where they exist in memory in relation to one another, ie. whether unused vectors will leave holes in VECTOR after collection.

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

The VECTOR buffer will be packed with live/unused objects in various arrangements.

```
#define LLT_GC_VECTOR__SIZE "2718281828459"
#define LLT_GC_VECTOR__SHAPE "GNS"
```

⟨ t/gc-vector.c 339 ⟩ ≡

```
⟨ Unit test header 256 ⟩
struct llt_Fixture {
  LLT_FIXTURE_HEADER;
  const char *pattern;
  int ret_val;
  size_t safe_bufsize;
  size_t safe_size;
  cell *cell_buf;
  cell *offset_buf;
  char *safe_buf;
  size_t *size_buf;
  size_t unsafe_bufsize;
  cell *unsafe_buf;
};
⟨ Unit test body 257 ⟩
⟨ Unit test: garbage collector gc_vector 340 ⟩
llt_fixture Test_Fixtures[] = {
    llt_GC_Vector__All, Λ
};
```

**340.** These tests are highly repetetive so the *vector*s defined by the fixture are created programmatically according to the pattern in *fix→pattern* which is a simple language of `L` & `U` characters.

Each vector is created by taking a character from the pattern and `LLT_GC_VECTOR__SIZE` in turn to decide on the size of the vector and whether it is live or unused. `LLT_GC_VECTOR__SHAPE` is then cycled through to populate each *vector* with a variety of data.

Live *vector*s are pushed onto `VMS` to keep them safe from collection. Vectors which will be considered unused are pushed onto `CTS` to keep them safe from collection while the fixture is being prepared.

```
⟨ Unit test: garbage collector gc_vector 340 ⟩ ≡
    /* There are too many one-letter variables in this function which then get reused */
  void llt_GC_Vector_prepare(llt_Fixture *fix)
  {
    cell g, v;
    char buf[TEST_BUFSIZE], *p, *s, *t;
    int i, n, z;
    if (¬fix→pattern) fix→pattern = "L";
    g = NIL;
    n = SCHAR_MAX;
    s = LLT_GC_VECTOR__SIZE;
    t = LLT_GC_VECTOR__SHAPE;
    for (p = (char *) fix→pattern; *p; p++) {
      if (*s ≡ '\0') s = LLT_GC_VECTOR__SIZE;
      ⟨ Unit test part: build a "random" vector 341 ⟩
      if (*p ≡ 'L') {
        ⟨ Unit test part: serialise a live vector into the fixture 342 ⟩
        vms_push(v);
      } else
        cts_push(v);
    }
    ⟨ Unit test part: complete live vector serialisation 343 ⟩
    ⟨ Unit test part: save unused vector references 344 ⟩
    cts_reset( );
  }
```

See also sections 345, 348, 349, 350, and 351.

This code is used in section 339.

**341.** Each time a global variable is requested $g$ is decremented, cycling from NIL down to UNDEFINED. Each new number and symbol is also unique using a counter $n$ that starts high enough to create numbers not protected by *Small_Int*.

```
⟨ Unit test part: build a "random" vector 341 ⟩ ≡
  v = vector_new((z = *s++ - '0'), NIL);
  for (i = 0; i < z; i++) {
    if (*t ≡ '\0') t = LLT_GC_VECTOR__SHAPE;
    switch (*t++) {
    case 'G':
      vector_ref(v, i) = g--;
      if (g < UNDEFINED) g = NIL;
      break;
    case 'N':
      vector_ref(v, i) = int_new(n += 42);
      break;
    case 'S':
      snprintf(buf, TEST_BUFSIZE, "testing-%d", n += 42);
      vector_ref(v, i) = sym(buf);
      break;
    }
  }
```

This code is used in section 340.

**342.** The offset of a *vector* may change if there are unused *vector*s to collect so it's saved into *fix→offset_buf* instead and the live *vector*s are serialised without recording it.

```
⟨ Unit test part: serialise a live vector into the fixture 342 ⟩ ≡
  fix→safe_size++;
  fix→cell_buf = reallocarray(fix→cell_buf, fix→safe_size, sizeof(cell));
  fix→offset_buf = reallocarray(fix→offset_buf, fix→safe_size, sizeof(cell));
  fix→size_buf = reallocarray(fix→size_buf, fix→safe_size, sizeof(size_t));
  fix→cell_buf[fix→safe_size - 1] = v;
  fix→offset_buf[fix→safe_size - 1] = vector_offset(v);
  fix→safe_bufsize += fix→size_buf[fix→safe_size - 1] = object_sizeof(v);
```

This code is used in section 340.

**343.** The list of live objects saved in VMS is reversed so that the order matches that in *fix→pattern* then they are serialised sequentially into *fix→safe_buf*.

```
⟨ Unit test part: complete live vector serialisation 343 ⟩ ≡
  fix→safe_buf = calloc(fix→safe_bufsize, sizeof(char));
  VMS = list_reverse_m(VMS, btrue);
  n = 0;
  for (v = VMS; ¬null_p(v); v = cdr(v))
    n += object_copy(car(v), fix→safe_buf + n, bfalse);
```

This code is used in section 340.

**344.** Unused objects don't need to be serialised; their cell references only are saved to verify that they have been returned to the free list.

```
⟨ Unit test part: save unused vector references 344 ⟩ ≡
  fix→unsafe_bufsize = 0;
  for (v = CTS; ¬null_p(v); v = cdr(v))
    fix→unsafe_bufsize += object_sizeofref(car(v));
  fix→unsafe_buf = calloc(fix→unsafe_bufsize, sizeof(cell));
  i = 0;
  for (v = CTS; ¬null_p(v); v = cdr(v))
    i += object_copyref(car(v), fix→unsafe_buf + i);
```

This code is used in section 340.

**345.**

```
⟨ Unit test: garbage collector gc_vector 340 ⟩ +≡
  boolean llt_GC_Vector_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE], *p, *s;
    boolean ok, liveok, freeok, tagok, *freelist;
    int delta, live, serial, unused, f, i;
    cell j;
    freelist = calloc(Cells_Poolsize, sizeof(boolean));
    for (j = Cells_Free; ¬null_p(j); j = cdr(j)) freelist[j] = btrue;
    delta = live = serial = unused = 0;
    s = LLT_GC_VECTOR__SIZE;
    ok = btrue;
    for (i = 0, p = (char *) fix→pattern; *p; i++, p++) {
      if (*s ≡ '\0') s = LLT_GC_VECTOR__SIZE;
      if (*p ≡ 'L') { ⟨ Unit test part: test a live vector 346 ⟩}
      else { ⟨ Unit test part: test an unused vector 347 ⟩}
      s++;
    }
    return ok;
  }
```

**346.**

```
⟨ Unit test part: test a live vector 346 ⟩ ≡
  liveok = object_compare(fix→safe_buf + serial, fix→size_buf[live], fix→cell_buf[live], bfalse);
  tap_more(ok, liveok, fpmsgf("(L-%d) object is unchanged", live));
  liveok = vector_offset(fix→cell_buf[live]) ≡ fix→offset_buf[live] − delta;
  tap_more(ok, liveok, fpmsgf("(L-%d), object is defragmented", live));
  serial += fix→size_buf[live];
  live++;
```

This code is used in section 345.

**347.** ⟨ Unit test part: test an unused *vector* 347 ⟩ ≡
```
f = *s - '0';
delta += f ? vector_realsize(f) : 0;
tagok = freeok = btrue;
for (i = 0; i < (int) fix->unsafe_bufsize; i++) {
  j = fix->unsafe_buf[i];
  if (special_p(j) ∨ symbol_p(j) ∨ smallint_p(j)) continue;
  tagok = (tag(j) ≡ TAG_NONE) ∧ tagok;
  freeok = freelist[i] ∧ freeok;
}
tap_more(ok, tagok, fpmsgf("(U-%d) object's tag is cleared", unused));
tap_more(ok, freeok, fpmsgf("(U-%d) object is in the free list", unused));
unused++;
```
This code is used in section 345.

**348.** ⟨ Unit test: garbage collector *gc_vector* 340 ⟩ +≡
```
void llt_GC_Vector_destroy(llt_Fixture *fix)
{
  free(fix->cell_buf);
  free(fix->offset_buf);
  free(fix->safe_buf);
  free(fix->size_buf);
  free(fix->unsafe_buf);
  vm_init_imp();
}
```

**349.** ⟨ Unit test: garbage collector *gc_vector* 340 ⟩ +≡
```
void llt_GC_Vector_act(llt_Fixture *fix)
{
  fix->ret_val = gc_vectors();
}
```

**350.** ⟨ Unit test: garbage collector *gc_vector* 340 ⟩ +≡
```
llt_Fixture *llt_GC_Vector_fix(llt_Fixture *fix, const char *name)
{
  fix->name = name;
  fix->prepare = llt_GC_Vector_prepare;
  fix->destroy = llt_GC_Vector_destroy;
  fix->act = llt_GC_Vector_act;
  fix->test = llt_GC_Vector_test;
  return fix;
}
```

**351.** The tests themselves are then defined with a list of combinations of `L` & `U` that are built into the fixtures.

```
⟨ Unit test: garbage collector gc_vector 340 ⟩ +≡
  llt_Fixture *llt_GC_Vector__All(void)
  {
    static char *test_patterns[] = {"L", "LL", "LLL", "LLLU", "LLLUUU", "LLUL", "LLUUUL", "LULL",
        "LUULL", "LULUL", "LUULUL", "LUULUUL", "LLLULLLULLL", "LLLUUULLLUUULLL", "UL", "ULLL",
        "ULLLU", "UUULLL", "UUULLLUUU", "UUULLLUUULLL", "UUULLLUUULLLUUU", Λ};
    char **p;
    llt_Fixture *f;
    int c, i;
    for (c = 0, p = test_patterns; *p; c++, p++) ;
    f = llt_alloc(c);
    for (i = 0; i < c; i++) {
      llt_GC_Vector_fix(f + i, __func__);
      f[i].suffix = f[i].pattern = test_patterns[i];
    }
    return f;
  }
```

### 352. Objects.

```
#define LLT_TEST_VARIABLE "test-variable"
#define LLT_VALUE_MARCO "marco?"
#define LLT_VALUE_POLO "polo!"
#define LLT_VALUE_FISH "fish..."
```

### 353. Closures.

### 354. Environments.

Broadly speaking there are three activities that can be performed on an *environment* which need to be tested: searching, setting and lifting stack items.

⟨ t/environments.c 354 ⟩ ≡

```
⟨ Unit test header 256 ⟩
struct llt_Fixture {
  LLT_FIXTURE_HEADER;
  cell expect;  /* desired result */
  cell formals;  /* formals for env_lift_stack */
  boolean had_ex_p;  /* was an error raised? */
  int layers;  /* depth of prepared environment */
  cell layer[3];  /* prepared environment contents */
  boolean new_p;  /* seting or replacing variable */
  int null_pos;  /* where to put a NIL in formals */
  boolean proper_p;  /* create a proper list? */
  cell ret_val;  /* returned value */
  llt_copy *save_Env;  /* dump of Env */
  jmp_buf save_goto;  /* copy Goto_Error to restore */
  int save_RTSp;  /* RTSp prior to action */
  cell(*search_fn)(cell, cell);  /* env_search/env_here */
  int stack;  /* how many stack items */
  cell sym_mpf[3];  /* prepared symbol objects */
  cell sym_var[3];
  cell sym_val[3];
  boolean want_ex_p;  /* will an error be raised? */
};
⟨ Unit test body 257 ⟩
⟨ Unit test: environment objects 355 ⟩
llt_fixture Test_Fixtures[] = {
    llt_Environments__Lift_Stack, llt_Environments__Search_Multi_Masked,
    llt_Environments__Search_Multi_Simple, llt_Environments__Search_Single_Layer,
    llt_Environments__Set, Λ
};
```

**355.** ⟨ Unit test: environment objects 355 ⟩ ≡

```
void llt_Environments_prepare(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE] = {0};
  cell e[3];
  int i;
  fix→sym_mpf[0] = sym(LLT_VALUE_MARCO);
  fix→sym_mpf[1] = sym(LLT_VALUE_POLO);
  fix→sym_mpf[2] = sym(LLT_VALUE_FISH);
  ⟨ Unit test part: prepare environment layers 356 ⟩
  if (fix→stack) {⟨ Unit test part: prepare liftable stack 357 ⟩}
  bcopy(fix→save_goto, Goto_Error, sizeof(jmp_buf));
  Error_Handler = btrue;
}
```

See also sections 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, and 370.

This code is used in section 354.

**356.** All of these tests prepare an *environment* of 1-3 layers.

⟨ Unit test part: prepare *environment* layers 356 ⟩ ≡

```
vms_push(Env);
Env = e[0] = env_empty();
if (fix→layers > 1) Env = e[1] = env_extend(e[0]);
if (fix→layers > 2) Env = e[2] = env_extend(e[1]);
for (i = 0; i < fix→layers; i++)
  if (¬null_p(fix→layer[i])) env_set(e[i], fix→sym_mpf[0], fix→layer[i], btrue);
fix→save_Env = llt_copy_object(Env, btrue);
fix→save_RTSp = RTSp;
for (i = 0; i < 3; i++) {
  snprintf(buf, TEST_BUFSIZE, "test-variable-%d", i + 1);
  fix→sym_var[i] = sym(buf);
  snprintf(buf, TEST_BUFSIZE, "test-value-%d", i + 1);
  fix→sym_val[i] = sym(buf);
}
```

This code is used in section 355.

**357.** To test *env_lift_stack* the stack is seeded with up to 3 items.

⟨ Unit test part: prepare liftable stack 357 ⟩ ≡

```
rts_push(fix→sym_val[fix→stack − 1]);
if (¬fix→proper_p) fix→formals = fix→sym_var[fix→stack − 1];
else if (fix→null_pos ≡ fix→stack) fix→formals = cons(NIL, NIL);
else fix→formals = cons(fix→sym_var[fix→stack − 1], NIL);
vms_push(fix→formals);
for (i = fix→stack − 1; i > 0; i−−) {
  rts_push(fix→sym_val[i − 1]);
  if (fix→null_pos ∧ fix→null_pos ≡ i) fix→formals = cons(NIL, fix→formals);
  else fix→formals = cons(fix→sym_var[i − 1], fix→formals);
  vms_set(fix→formals);
}
```

This code is used in section 355.

**358.** ⟨ Unit test: environment objects 355 ⟩ +≡

```
void llt_Environments_destroy(llt_Fixture *fix__unused)
{
  Env = fix→save_Env→origin;
  Acc = VMS = NIL;
  free(fix→save_Env);
  bcopy(fix→save_goto, Goto_Error, sizeof(jmp_buf));
  Error_Handler = bfalse;
}
```

**359.** There is no default action or test procedures for these units.

⟨ Unit test: environment objects 355 ⟩ +≡

```
llt_Fixture *llt_Environments_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_Environments_prepare;
  fix→destroy = llt_Environments_destroy;
  fix→act = Λ;
  fix→test = Λ;
  return fix;
}
```

**360.** Searching the environment results in a variable being found and its value returned, or it's not found and `UNDEFINED` is returned. If a variable is present in more than one layer the correct variant must be returned.

This test describes a total of 14 test cases, 7 each for *env_search* and *env_here*:
* 2 tests of a single-layer *environment*, with the variable present & not.
The following cases all describe tests of multi-layered *environment*s:
* 3 tests: the variable is in the top layer, a parent layer and not present at all.
* The variable is in the top layer and in the parent layer with a different value.
* The variable is in the parent layer and *its* parent.
TODO: tests with a populated environment.

⟨ Unit test: environment objects 355 ⟩ +≡

```
void llt_Environments__Search_act(llt_Fixture *fix)
{
  fix→ret_val = fix→search_fn(Env, fix→sym_mpf[0]);
}
```

**361.** ⟨ Unit test: environment objects 355 ⟩ +≡

```
boolean llt_Environments__Search_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE] = {0};
  if (true_p(fix→expect))
    return tap_ok(fix→ret_val ≡ fix→sym_mpf[1], fpmsgf("variable␣is␣found␣&␣correct"));
  else return tap_ok(undefined_p(fix→ret_val), fpmsgf("variable␣is␣not␣found"));
}
```

**362.** Search a single layer, with and without the variable present.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  llt_Fixture *llt_Environments__Search_Single_Layer(void)
  {
    int i;
    llt_Fixture *f = llt_alloc(4);
    for (i = 0; i < 4; i++) {
      llt_Environments_fix(f + i, __func__);
      f[i].act = llt_Environments__Search_act;
      f[i].test = llt_Environments__Search_test;
      f[i].layers = 1;
    }
    for (i = 0; i < 4; i += 2) {
      f[i].search_fn = env_search;
      f[i + 1].search_fn = env_here;
    }
    f[0].suffix = "env_search: not present";
    f[1].suffix = "env_here: not present";
    f[0].layer[0] = f[1].layer[0] = NIL;
    f[0].expect = f[1].expect = FALSE;
    f[2].suffix = "env_search: present";
    f[3].suffix = "env_here: present";
    f[2].layer[0] = f[3].layer[0] = sym(LLT_VALUE_POLO);
    f[2].expect = f[3].expect = TRUE;
    return f;
  }
```

**363.** Search a multi-layered *environment* with the variable present at the top, in the parent and not present at all.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  llt_Fixture *llt_Environments__Search_Multi_Simple(void)
  {
    int i;
    llt_Fixture *f = llt_alloc(6);
    for (i = 0; i < 6; i++) {
      llt_Environments_fix(f + i, __func__);
      f[i].act = llt_Environments__Search_act;
      f[i].test = llt_Environments__Search_test;
      f[i].layers = 3;
      f[i].layer[0] = f[i].layer[1] = f[i].layer[2] = NIL;
    }
    for (i = 0; i < 6; i += 2) {
      f[i].search_fn = env_search;
      f[i+1].search_fn = env_here;
    }
    f[0].suffix = "env_search: present in top";
    f[1].suffix = "env_here: present in top";
    f[0].expect = f[1].expect = TRUE;
    f[0].layer[2] = f[1].layer[2] = sym(LLT_VALUE_POLO);
    f[2].suffix = "env_search: present in parent";
    f[3].suffix = "env_here: present in parent";
    f[2].expect = TRUE;
    f[3].expect = FALSE;
    f[2].layer[1] = f[3].layer[1] = sym(LLT_VALUE_POLO);
    f[4].suffix = "env_search: not present";
    f[5].suffix = "env_here: not present";
    f[4].expect = f[5].expect = FALSE;
    return f;
  }
```

**364.** Search a multi-layered *environment* with the variable present in the top layer *and* parent, and the parent and its parent.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  llt_Fixture *llt_Environments__Search_Multi_Masked(void)
  {
    int i;
    llt_Fixture *f = llt_alloc(4);
    for (i = 0; i < 4; i++) {
      llt_Environments_fix(f + i, __func__);
      f[i].act = llt_Environments__Search_act;
      f[i].test = llt_Environments__Search_test;
      f[i].layers = 3;
      f[i].layer[0] = f[i].layer[1] = f[i].layer[2] = NIL;
    }
    for (i = 0; i < 4; i += 2) {
      f[i].search_fn = env_search;
      f[i+1].search_fn = env_here;
    }
    f[0].suffix = "env_search: present in top, conflict in parent";
    f[1].suffix = "env_here: present in top, conflict in parent";
    f[0].expect = f[1].expect = TRUE;
    f[0].layer[2] = f[1].layer[2] = sym(LLT_VALUE_POLO);
    f[0].layer[1] = f[1].layer[1] = sym(LLT_VALUE_FISH);
    f[2].suffix = "env_search: present in parent, conflict in ancestor";
    f[3].suffix = "env_here: present in parent, conflict in ancestor";
    f[2].expect = TRUE;
    f[3].expect = FALSE;
    f[2].layer[1] = f[3].layer[1] = sym(LLT_VALUE_POLO);
    f[2].layer[0] = f[3].layer[0] = sym(LLT_VALUE_FISH);
    return f;
  }
```

**365.** Setting a variable in an *environment* is really two mostly different processes depending on whether the variable is being created anew in the *environment* or replacing one, which must be first found and removed. Their tests are named `define` and `set`, respectively, to match the `LossLess` operators which call them.

TODO: non-empty environment.
TODO: variable in different parts of the layer (head, mid, tail).
TODO: verify that the old binding is removed.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  void llt_Environments__Set_act(llt_Fixture *fix)
  {
    fix→had_ex_p = bfalse;
    if (¬setjmp(Goto_Error)) env_set(Env, fix→sym_mpf[0], fix→sym_mpf[1], fix→new_p);
    else fix→had_ex_p = btrue;
  }
```

**366.** Regardless of the route taken there are only two possible outcomes: the variable gets set or an error is raised.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  boolean llt_Environments__Set_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE] = {0};
    boolean ok;
    cell found;
    if (fix→want_ex_p) {
      ok = tap_ok(fix→had_ex_p, fpmsgf("an␣error␣was␣raised"));
      if (fix→new_p) {
        tap_again(ok, ex_id(Acc) ≡ Sym_ERR_BOUND ∧ ex_detail(Acc) ≡ fix→sym_mpf[0],
            fpmsgf("the␣error␣is␣bound␣marco"));
      }
      else {
        tap_again(ok, ex_id(Acc) ≡ Sym_ERR_UNBOUND ∧ ex_detail(Acc) ≡ fix→sym_mpf[0],
            fpmsgf("the␣error␣is␣unbound␣marco"));
      }
    }
    else ok = tap_ok(¬fix→had_ex_p, fpmsgf("an␣error␣was␣not␣raised"));
    found = env_search(Env, fix→sym_mpf[0]);
    tap_more(ok, found ≡ fix→expect, fpmsgf("the␣variable␣has␣the␣correct␣value"));
    return ok;
  }
```

**367.** For each of define & set, there are three tests: the variable is already present, the variable is present in an ancestor, and the variable is not present.

```
⟨ Unit test: environment objects 355 ⟩ +≡
  llt_Fixture *llt_Environments__Set(void)
  {
    int i;
    llt_Fixture *f = llt_alloc(6);
    for (i = 0; i < 6; i++) {
      llt_Environments_fix(f + i, __func__);
      f[i].act = llt_Environments__Set_act;
      f[i].test = llt_Environments__Set_test;
      f[i].layers = 2;
      f[i].new_p = ¬(i % 2);
    }
    f[0].suffix = "define: already present";
    f[1].suffix = "set: already present";
    f[0].layer[1] = f[1].layer[1] = sym(LLT_VALUE_FISH);
    f[0].layer[0] = f[1].layer[0] = NIL;
    f[0].want_ex_p = btrue;
    f[0].expect = sym(LLT_VALUE_FISH);
    f[1].expect = sym(LLT_VALUE_POLO);
    f[2].suffix = "define: in an ancestor";
    f[3].suffix = "set: in an ancestor";
    f[2].layer[1] = f[3].layer[1] = NIL;
    f[2].layer[0] = f[3].layer[0] = sym(LLT_VALUE_FISH);
    f[2].expect = sym(LLT_VALUE_POLO);
    f[3].want_ex_p = btrue;
    f[3].expect = sym(LLT_VALUE_FISH);
    f[4].suffix = "define: not in the environment";
    f[5].suffix = "set: not in the environment";
    f[4].layer[1] = f[5].layer[1] = NIL;
    f[4].layer[0] = f[5].layer[0] = NIL;
    f[4].expect = sym(LLT_VALUE_POLO);
    f[5].want_ex_p = btrue;
    f[5].expect = UNDEFINED;
    return f;
  }
```

**368.** Lifting stack items into an environment has many moving parts so we return to more formally deriving its test cases.

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

Expects an environment $E_0$ and formals, which is a symbol, `NIL` or a possibly-dotted list of symbols or `NIL`. Returns an extension of that environment $E_1$.

Pops a stack item for every element of the list of formals (acting as though a dotted list were proper and a plain symbol were a list of 1 symbol). If the formal item is not `NIL` then the popped item is included in $E_1$, named by the formal. $E_0$ remains unchanged.

Allocates storage and may call garbage collection.

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

*env_lift_stack* does not validate its inputs or protect its arguments so the stack must be populated correctly and the arguments saved from garbage collection.

3. *What postconditions are guaranteed?*

An *environment* will always be returned. The stack will have been cleared of the arguments.

4. *What example inputs trigger different behaviors?*

Only the formals has any impact on operation. In particular whether it is `NIL`, a symbol or a list and, if a list, whether it contains `NIL`s or is improper.

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

A list of formals of lengths 0, 1, 2 & 3 with variants with and without `NIL`. Lengths 1 and 2 additionally test a lone symbol and an improper list.

⟨ Unit test: environment objects 355 ⟩ +≡

```
void llt_Environments__Lift_Stack_act(llt_Fixture *fix)
{
  fix->ret_val = env_lift_stack(Env, fix->formals);
}
```

**369.** ⟨ Unit test: environment objects 355 ⟩ +≡

```
boolean llt_Environments__Lift_Stack_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE] = {0};
  cell found;
  boolean ep, ok;
  ok = tap_ok(ep = environment_p(fix→ret_val), fpmsgf("the␣return␣value␣is␣an␣environment"));
  tap_again(ok, env_parent(fix→ret_val) ≡ Env, fpmsgf("the␣correct␣environment␣is␣extended"));
  tap_more(ok, object_compare(fix→save_Env→buf, fix→save_Env→size, Env, btrue),
      fpmsgf("the␣parent␣environment␣is␣unchanged"));
  tap_more(ok, RTSp ≡ fix→save_RTSp, fpmsgf("the␣stack␣is␣reset"));
  found = NIL;
  switch (fix→stack) {      /* these are all expected to fall through */
  case 3:      /* No fixtures have null_pos ≡ 3 */
    if (ep) found = env_here(fix→ret_val, fix→sym_var[2]);
    tap_more(ok, found ≡ fix→sym_val[2], fpmsgf("3rd␣argument␣is␣lifted"));
  case 2:
    if (ep) found = env_here(fix→ret_val, fix→sym_var[1]);
    if (fix→null_pos ≡ 2) tap_more(ok, undefined_p(found), fpmsgf("2nd␣argument␣is␣ignored"));
    else tap_more(ok, found ≡ fix→sym_val[1], fpmsgf("2nd␣argument␣is␣lifted"));
  case 1:
    if (ep) found = env_here(fix→ret_val, fix→sym_var[0]);
    if (fix→null_pos ≡ 1) tap_more(ok, undefined_p(found), fpmsgf("1st␣argument␣is␣ignored"));
    else tap_more(ok, found ≡ fix→sym_val[0], fpmsgf("1st␣argument␣is␣lifted"));
  }
  return ok;
}
```

**370.** ⟨ Unit test: environment objects 355 ⟩ +≡

```
llt_Fixture *llt_Environments__Lift_Stack(void)
{
  int i;
  llt_Fixture *f = llt_alloc(11);
  for (i = 0; i < 11; i++) {
    llt_Environments_fix(f + i, __func__);
    f[i].act = llt_Environments__Lift_Stack_act;
    f[i].test = llt_Environments__Lift_Stack_test;
    f[i].layers = 1;
    f[i].layer[0] = NIL;
    f[i].proper_p = btrue;
    f[i].formals = NIL;
  }
  f[i = 0].suffix = "NIL";
  f[++i].suffix = "symbol";
  f[i].stack = 1;
  f[i].proper_p = bfalse;
  f[++i].suffix = "pair␣of␣two␣symbols";
  f[i].stack = 2;
  f[i].proper_p = bfalse;
  f[++i].suffix = "improper␣list␣of␣symbols";
  f[i].stack = 3;
  f[i].proper_p = bfalse;
  f[++i].suffix = "list␣of␣NIL";
  f[i].stack = 1;
  f[i].null_pos = 1;
  f[++i].suffix = "list␣of␣1␣symbol";
  f[i].stack = 1;
  f[++i].suffix = "list␣of␣2␣symbols";
  f[i].stack = 2;
  f[++i].suffix = "list␣of␣2␣with␣NIL␣first";
  f[i].stack = 2;
  f[i].null_pos = 1;
  f[++i].suffix = "list␣of␣2␣with␣NIL␣last";
  f[i].stack = 2;
  f[i].null_pos = 2;
  f[++i].suffix = "list␣of␣3␣symbols";
  f[i].stack = 3;
  f[++i].suffix = "list␣of␣3␣with␣a␣NIL";
  f[i].stack = 3;
  f[i].null_pos = 2;
  return f;
}
```

**371. Frames.**

**372. Lists & Pairs.**

**373. Numbers.**

**374. Symbols.**

**375. Vectors.**

**376. Interpreter.**

⟨ t/interpreter.c 376 ⟩ ≡
```
⟨ Unit test header 256 ⟩
struct llt_Fixture {
  LLT_FIXTURE_HEADER;
  ⟨ Unit test part: interpreter fixture flags 377 ⟩
  ⟨ Unit test part: interpreter fixture mutators & registers 378 ⟩
  ⟨ Unit test part: interpreter fixture state backup 379 ⟩
};
⟨ Unit test body 257 ⟩
⟨ Unit test: Interpreter 380 ⟩
llt_fixture Test_Fixtures[] = {
    llt_Interpreter__OP_CYCLE, llt_Interpreter__OP_ENV_MUTATE_M, llt_Interpreter__OP_HALT,
    llt_Interpreter__OP_JUMP, llt_Interpreter__OP_JUMP_FALSE, llt_Interpreter__OP_JUMP_TRUE,
    llt_Interpreter__OP_LOOKUP, llt_Interpreter__OP_NOOP, llt_Interpreter__OP_SNOC,
    llt_Interpreter__OP_SWAP, Λ
};
```

**377.** Flags (etc.) which instruct the testing.

⟨ Unit test part: interpreter fixture flags 377 ⟩ ≡
```
boolean custom_p;     /* whether Prog was prepared already */
boolean env_found_p;     /* whether the variable should be found */
cell env_new_p;     /* whether to set a new variable */
int extra_stack;     /* extra noise to include in the stack */
boolean had_ex_p;     /* was an error raised? */
int opcode;     /* the opcode under test */
cell set_Acc;     /* what to put in Acc */
cell sym_mpft[4];     /* prepared symbol objects */
boolean want_ex_p;     /* will an error be raised? */
```
This code is used in section 376.

**378.** Predicates indicating whether the indicated state is expected to mutate, and the new values registers are expected to contain.

⟨ Unit test part: interpreter fixture mutators & registers 378 ⟩ ≡
```
boolean mutate_Acc_p;
boolean mutate_Env_p;
boolean mutate_Fp_p;
boolean mutate_Ip_p;
boolean mutate_Prog_p;
boolean mutate_RTS_p;
boolean mutate_RTSp_p;
boolean mutate_Root_p;
boolean mutate_VMS_p;
int want_Ip;
int want_Fp;
int want_RTSp;
```
This code is used in section 376.

**379.** Copies of interpreter state immediately prior to performing each test. *backup_Env* is a pointer to the original *Env* which is saved in VMS, not a copy.

⟨ Unit test part: interpreter fixture state backup 379 ⟩ ≡

```
cell backup_Env;
llt_copy *save_Acc;
llt_copy *save_Env;
llt_copy *save_Prog;
llt_copy *save_RTS;
llt_copy *save_Root;
llt_copy *save_VMS;
int save_Fp;
jmp_buf save_goto;
int save_Ip;
int save_RTSp;
```

This code is used in section 376.

**380.** The simplest opcodes rely on being the only instruction in *Prog* immediately followed by OP_HALT. More complex tests define *Prog* in their own prepare phase. State is not copied if it's expected to mutate to save time.

⟨ Unit test: Interpreter 380 ⟩ ≡

```
void llt_Interpreter_prepare(llt_Fixture *fix){ fix→sym_mpft[0] = sym(LLT_VALUE_MARCO);
    fix→sym_mpft[1] = sym(LLT_VALUE_POLO);
    fix→sym_mpft[2] = sym(LLT_VALUE_FISH);
    fix→sym_mpft[3] = sym(LLT_TEST_VARIABLE); if (¬fix→custom_p) { Prog = vector_new(2,
        int_new(OP_HALT)); vector_ref(Prog, 0) = int_new ( fix → opcode ) ;
    Ip = 0; }
    if (¬fix→mutate_Acc_p) fix→save_Acc = llt_copy_object(Acc, bfalse);
    if (¬fix→mutate_Prog_p) fix→save_Prog = llt_copy_object(Prog, bfalse);
    if (¬fix→mutate_Env_p) fix→save_Env = llt_copy_object(Env, bfalse);
    if (¬fix→mutate_Root_p) fix→save_Root = llt_copy_object(Root, bfalse);
    if (¬fix→mutate_VMS_p) fix→save_VMS = llt_copy_object(VMS, bfalse);
    if (¬fix→mutate_RTS_p) fix→save_RTS = llt_copy_object(RTS, bfalse);
    if (¬fix→mutate_Fp_p) fix→save_Fp = Fp;
    if (¬fix→mutate_Ip_p) fix→save_Ip = Ip;
    if (¬fix→mutate_RTSp_p) fix→save_RTSp = RTSp;
    bcopy(Goto_Error, fix→save_goto, sizeof(jmp_buf));
    Error_Handler = btrue; }
```

See also sections 381, 382, 383, 384, 391, 392, 393, 395, 396, 397, 402, 403, 404, 405, 406, 407, 408, 414, 415, 416, 417, 419, 431, 432, 433, 434, 435, and 436.

This code is used in section 376.

**381.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
void llt_Interpreter_destroy(llt_Fixture *fix)
{
  free(fix→save_Acc);
  free(fix→save_Env);
  free(fix→save_Prog);
  free(fix→save_RTS);
  free(fix→save_Root);
  free(fix→save_VMS);
  VMS = RTS = NIL;
  RTSp = −1;
  RTS_Size = 0;
  bcopy(fix→save_goto, Goto_Begin, sizeof(jmp_buf));
  Error_Handler = bfalse;
}
```

**382.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
void llt_Interpreter_act(llt_Fixture *fix __unused)
{   /* TODO: use Goto_Error like the environment tests? */
  fix→had_ex_p = bfalse;
  if (¬setjmp(Goto_Begin)) interpret();
  else fix→had_ex_p = btrue;
}
```

**383.**

```
#define llt_Interpreter_test_compare(o) do
    {
      boolean ok = object_compare(fix→save_##o→buf, fix→save_##o→size, (o), bfalse);
      tap_more(all, ok, fpmsgf(#o" is unchanged"));
    }
    while (0)
```

⟨ Unit test: Interpreter 380 ⟩ +≡

```
boolean llt_Interpreter_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE];
  boolean all = btrue;
  if (¬fix→mutate_Acc_p) llt_Interpreter_test_compare(Acc);
  if (¬fix→mutate_Env_p) llt_Interpreter_test_compare(Env);
  if (¬fix→mutate_Fp_p) tap_more(all, Fp ≡ fix→save_Fp, fpmsgf("Fp is unchanged"));
  else tap_more(all, Fp ≡ fix→want_Fp, fpmsgf("Fp is changed correctly"));
  if (¬fix→mutate_Ip_p) tap_more(all, Ip ≡ fix→save_Ip, fpmsgf("Ip is unchanged"));
  else tap_more(all, Ip ≡ fix→want_Ip, fpmsgf("Ip is changed correctly"));
  if (¬fix→mutate_Prog_p) llt_Interpreter_test_compare(Prog);
  if (¬fix→mutate_RTS_p) llt_Interpreter_test_compare(RTS);
  if (¬fix→mutate_RTSp_p) tap_more(all, RTSp ≡ fix→save_RTSp, fpmsgf("RTSp is unchanged"));
  else tap_more(all, RTSp ≡ fix→want_RTSp, fpmsgf("RTSp is changed correctly"));
  if (¬fix→mutate_Root_p) llt_Interpreter_test_compare(Root);
  if (¬fix→mutate_VMS_p) llt_Interpreter_test_compare(VMS);
  if (fix→want_ex_p) tap_more(all, fix→had_ex_p, fpmsgf("an error was raised"));
  else tap_more(all, ¬fix→had_ex_p, fpmsgf("an error was not raised"));
  return all;
}
```

**384.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
llt_Fixture *llt_Interpreter_fix(llt_Fixture *fix, const char *name){ fix→name = name;
    fix→prepare = llt_Interpreter_prepare;
    fix→destroy = llt_Interpreter_destroy;
    fix→act = llt_Interpreter_act;
    fix→test = llt_Interpreter_test; fix → opcode = OP_NOOP;
    fix→mutate_Ip_p = btrue;
    fix→want_Ip = 1;
    return fix; }
```

**385.** `OP_APPLY`.

...

**386.** `OP_APPLY_TAIL`.

...

**387.** `OP_CAR`.

**388.** `OP_CDR`.

**389.** `OP_COMPILE`.

**390.** `OP_CONS`.

**391.** `OP_CYCLE`.

`OP_CYCLE` swaps the top two stack elements and advances *Ip* by 1. There must be a stack to manipulate.

⟨ Unit test: Interpreter 380 ⟩ +≡

```
void llt_Interpreter__OP_CYCLE_prepare(llt_Fixture *fix)
{
  int i;
  for (i = 0; i < fix→extra_stack; i++) rts_push(int_new(42 + 3 * i));
  rts_push(int_new(42));
  rts_push(sym("question?"));
  llt_Interpreter_prepare(fix);
}
```

**392.** If all the standard tests pass then *RTSp* was unchanged and there are two items on the stack otherwise it's not safe to consider the stack at all.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  boolean llt_Interpreter__OP_CYCLE_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE];
    boolean ok = llt_Interpreter_test(fix);
    cell top, next;
    if (RTSp ≠ fix→save_RTSp) {
      tap_fail(fpmsgf("cannot␣test␣stack␣contents"));
      tap_fail(fpmsgf("cannot␣test␣stack␣contents"));
      return bfalse;
    }
    top = rts_pop(1);
    next = rts_pop(1);
    tap_more(ok, top ≡ int_new(42), fpmsgf("the␣stack␣top␣is␣correct"));
    tap_more(ok, next ≡ sym("question?"), fpmsgf("the␣next␣stack␣item␣is␣correct"));
    return ok;
  }
```

**393.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
  llt_Fixture *llt_Interpreter__OP_CYCLE(void){ llt_Fixture *f = llt_alloc(2);
      llt_Interpreter_fix(f + 0, __func__);
      llt_Interpreter_fix(f + 1, __func__); f[0] . opcode = f[1] . opcode = OP_CYCLE;
      f[0].prepare = f[1].prepare = llt_Interpreter__OP_CYCLE_prepare;
      f[0].test = f[1].test = llt_Interpreter__OP_CYCLE_test;
      f[0].mutate_RTS_p = f[1].mutate_RTS_p = btrue;
      f[0].suffix = "empty␣stack";
      f[1].extra_stack = 3;
      f[1].suffix = "stack␣in␣use";
      return f; }
```

**394.** `OP_ENVIRONMENT_P.`

**395.** `OP_ENV_MUTATE_M.` 2 extra codes in Prog, Env to set in on stack. Value. Although *env_set* could fail this unit "cannot" so only two tests are needed for each boolean state of *new_p*.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  void llt_Interpreter__OP_ENV_MUTATE_M_prepare(llt_Fixture *fix)
  {
    Prog = vector_new(4, int_new(OP_HALT));
    vector_ref(Prog, 0) = int_new(OP_ENV_MUTATE_M);
    vector_ref(Prog, 1) = sym(LLT_TEST_VARIABLE);
    vector_ref(Prog, 2) = fix→env_new_p;
    Ip = 0;
    Acc = sym(LLT_VALUE_POLO);
    Tmp_Test = fix→backup_Env = env_empty( );
    rts_push(fix→backup_Env);
    if (false_p(fix→env_new_p))
      env_set(fix→backup_Env, sym(LLT_TEST_VARIABLE), sym(LLT_VALUE_FISH), FALSE);
    llt_Interpreter_prepare(fix);
  }
```

**396.** ⟨ Unit test: Interpreter 380 ⟩ +≡
```
boolean llt_Interpreter__OP_ENV_MUTATE_M_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE];
  boolean ok = llt_Interpreter_test(fix);
  cell found;
  tap_more(ok, void_p(Acc), fpmsgf("Acc is void"));
  found = env_here(fix→backup_Env, fix→sym_mpft[3]);
  tap_more(ok, found ≡ fix→sym_mpft[1], fpmsgf("the value is set"));
  return ok;
}
```

**397.** ⟨ Unit test: Interpreter 380 ⟩ +≡
```
llt_Fixture *llt_Interpreter__OP_ENV_MUTATE_M(void)
{
  llt_Fixture *f = llt_alloc(2);
  llt_Interpreter_fix(f + 0, __func__);
  llt_Interpreter_fix(f + 1, __func__);
  f[0].custom_p = f[1].custom_p = btrue;
  f[0].prepare = f[1].prepare = llt_Interpreter__OP_ENV_MUTATE_M_prepare;
  f[0].test = f[1].test = llt_Interpreter__OP_ENV_MUTATE_M_test;
  f[0].want_Ip = f[1].want_Ip = 3;
  f[0].mutate_Acc_p = f[1].mutate_Acc_p = btrue;
  f[0].mutate_RTS_p = f[1].mutate_RTS_p = btrue;
  f[0].mutate_RTSp_p = f[1].mutate_RTSp_p = btrue;
  f[0].want_RTSp = f[1].want_RTSp = -1;
  f[0].suffix = "already bound";
  f[0].env_new_p = FALSE;
  f[1].suffix = "not bound";
  f[1].env_new_p = TRUE;
  return f;
}
```

**398.** OP_ENV_QUOTE.

**399.** OP_ENV_ROOT.

**400.** OP_ENV_SET_ROOT_M.

**401.** OP_ERROR.

**402.** OP_HALT.

The only thing OP_HALT does is lower the *Running* flag to halt the VM.

⟨ Unit test: Interpreter 380 ⟩ +≡
```
llt_Fixture *llt_Interpreter__OP_HALT(void){ llt_Fixture *f = llt_alloc(1);
    llt_Interpreter_fix(f, __func__); f → opcode = OP_HALT;
    f→mutate_Ip_p = bfalse;
    return f; }
```

**403.** `OP_JUMP`. There is not much to test for `OP_JUMP`.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  void llt_Interpreter__OP_JUMP_prepare(llt_Fixture *fix)
  {
    Prog = vector_new(4, int_new(OP_HALT));
    vector_ref(Prog, 0) = int_new(OP_JUMP);
    vector_ref(Prog, 1) = int_new(3);
    Ip = 0;
    llt_Interpreter_prepare(fix);
  }
```

**404.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  llt_Fixture *llt_Interpreter__OP_JUMP(void)
  {
    llt_Fixture *f = llt_alloc(1);
    llt_Interpreter_fix(f, __func__);
    f→prepare = llt_Interpreter__OP_JUMP_prepare;
    f→want_Ip = 3;
    f→custom_p = btrue;
    return f;
  }
```

**405.** `OP_JUMP_FALSE`. Only *Ip* changes, unless `VOID` was being queried in which case *Ip* will be unchanged and an error raised.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  void llt_Interpreter__OP_JUMP_FALSE_prepare(llt_Fixture *fix){ Prog = vector_new(4,
          int_new(OP_HALT)); vector_ref(Prog, 0) = int_new ( fix → opcode ) ;
      vector_ref(Prog, 1) = int_new(3);
      Ip = 0;
      Acc = fix→set_Acc;
      llt_Interpreter_prepare(fix); }
```

**406.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  boolean llt_Interpreter__OP_JUMP_FALSE_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE];
    boolean ok = llt_Interpreter_test(fix);
    if (void_p(fix→set_Acc)) {
      tap_more(ok, exception_p(Acc) ∧ ex_id(Acc) ≡ Sym_ERR_UNEXPECTED ∧ void_p(ex_detail(Acc)),
          fpmsgf("error␣is␣unexpected␣void"));
    }
    return ok;
  }
```

**407.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
llt_Fixture *llt_Interpreter__OP_JUMP_FALSE(void){ int i;
    llt_Fixture *f = llt_alloc(4); for (i = 0; i < 4; i++) { llt_Interpreter_fix(f + i, __func__);
    f[i].prepare = llt_Interpreter__OP_JUMP_FALSE_prepare;
    f[i].test = llt_Interpreter__OP_JUMP_FALSE_test; f[i] . opcode = OP_JUMP_FALSE;
    f[i].custom_p = btrue; } f[0].suffix = "any";
    f[0].set_Acc = int_new(42);
    f[0].want_Ip = 2;
    f[1].suffix = "#t";
    f[1].set_Acc = TRUE;
    f[1].want_Ip = 2;
    f[2].suffix = "#f";
    f[2].set_Acc = FALSE;
    f[2].want_Ip = 3;
    f[3].suffix = "no value";
    f[3].set_Acc = VOID;
    f[3].want_Ip = -1;
    f[3].want_ex_p = btrue;
    f[3].mutate_Acc_p = btrue;
    return f; }
```

**408.** `OP_JUMP_TRUE`. This test mirrors that for `OP_JUMP_FALSE` except that the responses to `TRUE` and `FALSE` are inverted.

⟨ Unit test: Interpreter 380 ⟩ +≡

```
llt_Fixture *llt_Interpreter__OP_JUMP_TRUE(void){ int i;
    llt_Fixture *f = llt_Interpreter__OP_JUMP_FALSE( ); for (i = 0; i < f->max; i++) {
        f[i].name = __func__; f[i] . opcode = OP_JUMP_TRUE; } i = f[1].want_Ip;
    f[1].want_Ip = f[2].want_Ip;
    f[2].want_Ip = i;
    return f; }
```

**409.** `OP_LAMBDA`.

**410.** `OP_LIST_P`.

**411.** `OP_LIST_REVERSE`.

**412.** `OP_LIST_REVERSE_M`.

**413.** `OP_LOOKUP`. Assumes a symbol in Acc, looks for it recursively in Env. Value placed in Acc, or not found error.

Test not found vs. found.

**414.** ⟨ Unit test: Interpreter 380 ⟩ +≡

```
void llt_Interpreter__OP_LOOKUP_prepare(llt_Fixture *fix)
{
  vms_push(fix->backup_Env = Env);
  Env = env_extend(Env);
  if (fix->env_found_p) env_set(Env, sym(LLT_VALUE_MARCO), sym(LLT_VALUE_POLO), btrue);
  Acc = sym(LLT_VALUE_MARCO);
  llt_Interpreter_prepare(fix);
}
```

**415.** ⟨ Unit test: Interpreter 380 ⟩ +≡
```
void llt_Interpreter__OP_LOOKUP_destroy(llt_Fixture *fix)
{
  Env = fix→backup_Env;
  VMS = NIL;
}
```

**416.** ⟨ Unit test: Interpreter 380 ⟩ +≡
```
boolean llt_Interpreter__OP_LOOKUP_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE];
  boolean ok = llt_Interpreter_test(fix);
  if (fix→env_found_p)
    tap_more(ok, Acc = fix→sym_mpft[0], fpmsgf("Acc␣contains␣the␣looked␣up␣value"));
  else tap_more(ok,
      exception_p(Acc) ∧ ex_id(Acc) ≡ Sym_ERR_UNBOUND ∧ ex_detail(Acc) ≡ fix→sym_mpft[0],
      fpmsgf("error␣is␣unbound␣marco?"));
  return ok;
}
```

**417.** ⟨ Unit test: Interpreter 380 ⟩ +≡
```
llt_Fixture *llt_Interpreter__OP_LOOKUP(void){ llt_Fixture *f = llt_alloc(2);
    llt_Interpreter_fix(f + 0, __func__);
    llt_Interpreter_fix(f + 1, __func__); f[0] . opcode = f[1] . opcode = OP_LOOKUP;
    f[0].prepare = f[1].prepare = llt_Interpreter__OP_LOOKUP_prepare;
    f[0].test = f[1].test = llt_Interpreter__OP_LOOKUP_test;
    f[0].destroy = f[1].destroy = llt_Interpreter__OP_LOOKUP_destroy;
    f[0].mutate_RTS_p = f[1].mutate_RTS_p = btrue;
    f[0].mutate_Acc_p = f[1].mutate_Acc_p = btrue;
    f[0].suffix = "bound";
    f[0].env_found_p = btrue;
    f[1].suffix = "unbound";
    f[1].want_ex_p = btrue;
    f[1].want_Ip = −1;
    return f; }
```

**418.** `OP_NIL`.

**419.** `OP_NOOP`.
The `OP_NOOP` opcode has the same effect as `OP_HALT`, ie. none, without halting the VM.

⟨ Unit test: Interpreter 380 ⟩ +≡
```
llt_Fixture *llt_Interpreter__OP_NOOP(void)
{
  llt_Fixture *f = llt_alloc(1);
  llt_Interpreter_fix(f, __func__);
  return f;
}
```

**420.** `OP_NULL_P`.

**421.** `OP_PAIR_P`.

**422.** OP_PEEK.

**423.** OP_POP.

**424.** OP_PUSH.

**425.** OP_QUOTE.

**426.** OP_RETURN.
...

**427.** OP_RUN.
...

**428.** OP_RUN_THERE.
...

**429.** OP_SET_CAR_M.

**430.** OP_SET_CDR_M.

**431.** OP_SNOC. This opcode decomposes a pair in the accumulator, placing the *car* on the stack.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  void llt_Interpreter__OP_SNOC_prepare(llt_Fixture *fix)
  {
    Acc = cons(sym(LLT_VALUE_FISH), int_new(42));
    llt_Interpreter_prepare(fix);
  }
```

**432.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  boolean llt_Interpreter__OP_SNOC_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE];
    boolean ok = llt_Interpreter_test(fix);
    tap_more(ok, Acc ≡ fix→sym_mpft[2], fpmsgf("The␣car␣is␣in␣Acc"));
    tap_more(ok, RTSp ≡ fix→want_RTSp ∧ rts_pop(1) ≡ int_new(42), fpmsgf("The␣cdr␣is␣in␣RTS"));
    return ok;
  }
```

**433.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  llt_Fixture *llt_Interpreter__OP_SNOC(void){ llt_Fixture *f = llt_alloc(1);
      llt_Interpreter_fix(f, __func__); f[0] . opcode = OP_SNOC;
      f[0].prepare = llt_Interpreter__OP_SNOC_prepare;
      f[0].test = llt_Interpreter__OP_SNOC_test;
      f[0].mutate_Acc_p = btrue;
      f[0].mutate_RTS_p = btrue;
      f[0].mutate_RTSp_p = btrue;
      return f; }
```

**434.** OP_SWAP. This opcode is similar to OP_CYCLE except for what gets cycled.

```
⟨ Unit test: Interpreter 380 ⟩ +≡
  void llt_Interpreter__OP_SWAP_prepare(llt_Fixture *fix)
  {
    int i;
    for (i = 0; i < fix→extra_stack; i++) rts_push(int_new(42 + 3 * i));
    rts_push(int_new(42));
    Acc = sym("question?");
    llt_Interpreter_prepare(fix);
  }
```

**435.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  boolean llt_Interpreter__OP_SWAP_test(llt_Fixture *fix)
  {
    char buf[TEST_BUFSIZE];
    boolean ok = llt_Interpreter_test(fix);
    if (RTSp ≠ fix→save_RTSp) tap_fail(fpmsgf("cannot␣test␣stack␣contents"));
    else tap_more(ok, rts_pop(1) ≡ sym("question?"), fpmsgf("the␣stack␣top␣is␣correct"));
    tap_more(ok, Acc ≡ int_new(42), fpmsgf("Acc␣is␣correct"));
    return ok;
  }
```

**436.**
```
⟨ Unit test: Interpreter 380 ⟩ +≡
  llt_Fixture *llt_Interpreter__OP_SWAP(void){ llt_Fixture *f = llt_alloc(2);
      llt_Interpreter_fix(f + 0, __func__);
      llt_Interpreter_fix(f + 1, __func__); f[0] . opcode = f[1] . opcode = OP_SWAP;
      f[0].prepare = f[1].prepare = llt_Interpreter__OP_SWAP_prepare;
      f[0].test = f[1].test = llt_Interpreter__OP_SWAP_test;
      f[0].mutate_Acc_p = f[1].mutate_Acc_p = btrue;
      f[0].mutate_RTS_p = f[1].mutate_RTS_p = btrue;
      f[0].suffix = "empty␣stack";
      f[1].extra_stack = 3;
      f[1].suffix = "stack␣in␣use";
      return f; }
```

**437.** OP_SYNTAX.

**438.** OP_VOV.

**439. Compiler.**

The compiler generates bytecode from s-expressions, or raises a syntax or arity error. These tests verify that bytecode is generated when it should be not that the generated bytecode correctly implements the operator in question. This validation is performed by later tests of the integration between the compiler and the interpreter.

Each test fixture (if the compilation is expected to succeed) includes a C-string representation of the bytecode it is expected to generate which is compared with the bytecode's written representation.

```
⟨ t/compiler.c 439 ⟩ ≡
  ⟨ Unit test header 256 ⟩
  struct llt_Fixture {
    LLT_FIXTURE_HEADER;
    cell ret_val;
    cell src_val;
    char *src_exp;
    boolean had_ex_p;
    char *want;
    cell want_ex;
    cell save_Acc;
  };
  ⟨ Unit test body 257 ⟩
  ⟨ Unit test: Compiler 440 ⟩
  llt_fixture Test_Fixtures[] = {
      llt_Compiler__Eval, Λ
  };
```

**440.** ⟨ Unit test: Compiler 440 ⟩ ≡

```
void llt_Compiler_prepare(llt_Fixture *fix)
{
  Tmp_Test = fix→src_val = read_cstring(fix→src_exp);
  Error_Handler = btrue;
}
```

See also sections 441, 442, 443, 444, 445, 446, and 447.

This code is used in section 439.

**441.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
void llt_Compiler_destroy(llt_Fixture *fix__unused)
{
  Tmp_Test = NIL;
  Error_Handler = bfalse;
}
```

**442.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
void llt_Compiler_act(llt_Fixture *fix)
{
  fix→save_Acc = Acc;
  fix→had_ex_p = bfalse;
  if (¬setjmp(Goto_Error)) fix→ret_val = compile(fix→src_val);
  else fix→had_ex_p = btrue;
}
```

**443.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
char *llt_Compiler_decompile(cell);
boolean llt_Compiler_compare_bytecode(cell bc, char *want, boolean prepared_p)
{
  char *g, *w;
  ssize_t len;
  int i;
  boolean r = btrue;
  if (¬vector_p(bc)) return bfalse;
  ERR_OOM_P(g = calloc(TEST_BUFSIZE, sizeof(char)));
  if (write_bytecode(bc, g, TEST_BUFSIZE, 0) < 0) {
    free(g);
    return bfalse;
  }
  len = (ssize_t) strlen(want);
  if (prepared_p) w = want;
  else {
    ERR_OOM_P(w = malloc(len + 1));
    w[0] = '{';
    for (i = 1; i < len - 1; i++) w[i] = want[i];
    w[i] = '}';
  }
  if (len ≠ (ssize_t) strlen(g)) r = bfalse;
  else
    for (i = 0; i < len; i++)
      if (g[i] ≠ w[i]) {
        r = bfalse;
        break;
      }
  free(g);
  if (¬prepared_p) free(w);
  return r;
}
```

**444.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
boolean llt_Compiler_test(llt_Fixture *fix)
{
  char buf[TEST_BUFSIZE] = {0};
  boolean ok, match;
  if (fix→want ≡ Λ) {
    ok = tap_ok(fix→had_ex_p, fpmsgf("an␣error␣was␣raised"));
    tap_more(ok, ex_id(Acc) ≡ fix→want_ex, fpmsgf("the␣error␣type␣is␣correct"));
  }
  else {
    ok = tap_ok(¬fix→had_ex_p, fpmsgf("an␣error␣was␣not␣raised"));
    tap_more(ok, null_p(CTS), fpmsgf("the␣compiler␣stack␣is␣clear"));
    tap_more(ok, Acc ≡ fix→save_Acc, fpmsgf("Acc␣is␣unchanged"));
    match = llt_Compiler_compare_bytecode(fix→ret_val, fix→want, bfalse);
    tap_more(ok, match, fpmsgf("the␣correct␣bytecode␣is␣generated"));
  }
  return ok;
}
```

**445.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
llt_Fixture *llt_Compiler_fix(llt_Fixture *fix, const char *name)
{
  fix→name = name;
  fix→prepare = llt_Compiler_prepare;
  fix→destroy = llt_Compiler_destroy;
  fix→act = llt_Compiler_act;
  fix→test = llt_Compiler_test;
  fix→want = Λ;
  fix→want_ex = NIL;
  return fix;
}
```

**446.** *compile_eval*.

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

A list who's first expression is the symbol **eval** is passed as an argument to *compile*, which will pass control to *compile_eval*. An error is raised if other than one or two more expressions is in the list, otherwise it's compiled and the bytecode returned. *compile* takes no action to preserve its argument from the garbage collector. The argument values are not validated at this compile-time.

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

A C-string representing the expression to be compiled is read in. The majority if the VM state is ignored.

3. *What postconditions are guaranteed?*

The tests that expect an error to be raised will see that error in *Acc* and *compile* will not return. `CTS` may be changed but it can be ignored.

When compilation was a success the compilation result will be returned and `CTS` will be empty.

4. *What example inputs trigger different behaviors?*

The number of arguments to **eval** and their form (specifically, what each compiles to).

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

Four tests of 0, 1, 2 and 3 constant-value arguments validate that the VM is unchanged when compilation is a success or otherwise changed correctly.

The one-expression case is repeated three times with different types of expression to validate that the evaluating expression compiles.

These tests are duplicated again for the two-expression case; each test three times with the same types of different expression in the second position for a total of 9 two-expression tests.

TODO: explain these macros.

```
#define CAT2(a, b)  a /**/ b
#define CAT3(a, b, c)  a /**/ b /**/ c
#define CAT4(a, b, c, d)  a /**/ b /**/ c /**/ d
#define LLTCC_EVAL_FIRST_COMPLEX(xc, xa)  CAT3(CAT3("␣OP_QUOTE␣(", xa, ")␣OP_PUSH"),
    CAT3("␣OP_QUOTE␣", xc, "␣OP_LOOKUP"), "␣OP_CONS␣OP_COMPILE␣OP_RUN")
#define LLTCC_EVAL_FIRST_LOOKUP(x)  CAT3("␣OP_QUOTE␣", x, "␣OP_LOOKUP")
#define LLTCC_EVAL_FIRST_QUOTE(x)  CAT2("␣OP_QUOTE␣", x)
#define LLTCC_EVAL_SECOND_COMPLEX(xc, xa)  CAT2(LLTCC_EVAL_FIRST_COMPLEX(xc, xa), "␣OP_PUSH")
#define LLTCC_EVAL_SECOND_LOOKUP(x)  CAT2(LLTCC_EVAL_FIRST_LOOKUP(x), "␣OP_PUSH")
#define LLTCC_EVAL_SECOND_QUOTE(x)  CAT2(LLTCC_EVAL_FIRST_QUOTE(x), "␣OP_PUSH")
#define LLTCC_EVAL_VALIDATE(x)
    CAT3("␣OP_ENVIRONMENT_P␣OP_JUMP_TRUE␣", x, "␣OP_QUOTE␣unexpected␣OP_ERROR")
#define LLTCC_EVAL_ONEARG()  "␣OP_COMPILE␣OP_RUN␣OP_RETURN␣"
#define LLTCC_EVAL_TWOARG()  "␣OP_COMPILE␣OP_RUN_THERE␣OP_RETURN␣"
```

⟨ Unit test: Compiler 440 ⟩ +≡

```
void llt_Compiler__Eval_prepare(llt_Fixture *fix)
{
  llt_Compiler_prepare(fix);
  car(fix→src_val) = env_search(Root, sym("eval"));
}
```

**447.** ⟨ Unit test: Compiler 440 ⟩ +≡

```
llt_Fixture *llt_Compiler__Eval(void)
{
  int i;
  llt_Fixture *f = llt_alloc(16);
  for (i = 0; i < 16; i++) {
    llt_Compiler_fix(f + i, __func__);
    f[i].prepare = llt_Compiler__Eval_prepare;
  }
  i = -1;
  ⟨ Unit test part: compiler/eval fixtures 448 ⟩
  return f;
}
```

**448.** Constant-value arguments validate arity validation.

⟨ Unit test part: compiler/**eval** fixtures 448 ⟩ ≡

```
f[++i].src_exp = "(eval)";
f[i].suffix = "eval";
f[i].want_ex = Sym_ERR_ARITY_SYNTAX;
f[++i].src_exp = "(eval␣42)";
f[i].suffix = "eval␣x";
f[i].want = CAT2(LLTCC_EVAL_FIRST_QUOTE("42"),LLTCC_EVAL_ONEARG());
f[++i].src_exp = "(eval␣4␣2)";
f[i].suffix = "eval␣x␣x";
f[i].want = CAT4(LLTCC_EVAL_SECOND_QUOTE("2"),LLTCC_EVAL_VALIDATE("9"),
    LLTCC_EVAL_FIRST_QUOTE("4"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣4␣2␣?)";
f[i].suffix = "eval␣x␣x␣x";
f[i].want_ex = Sym_ERR_ARITY_SYNTAX;
```

See also sections 449, 450, 451, and 452.

This code is used in section 447.

**449.** Validating that a single expression is compiled.

⟨ Unit test part: compiler/**eval** fixtures 448 ⟩ +≡

```
f[++i].src_exp = "(eval␣42)";
f[i].suffix = "eval␣<constant>";
f[i].want = CAT2(LLTCC_EVAL_FIRST_QUOTE("42"),LLTCC_EVAL_ONEARG());
f[++i].src_exp = "(eval␣marco?)";
f[i].suffix = "eval␣<symbol>";
f[i].want = CAT2(LLTCC_EVAL_FIRST_LOOKUP("marco?"),LLTCC_EVAL_ONEARG());
f[++i].src_exp = "(eval␣(build␣an␣expression))";
f[i].suffix = "eval␣<complex␣expression>";
f[i].want = CAT2(LLTCC_EVAL_FIRST_COMPLEX("build","an␣expression"),LLTCC_EVAL_ONEARG());
```

**450.** Two expressions where the first is constant.

⟨ Unit test part: compiler/**eval** fixtures 448 ⟩ +≡

```
f[++i].src_exp = "(eval␣42␣24)";
f[i].suffix = "eval␣<constant>␣<constant>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_QUOTE("24"),LLTCC_EVAL_VALIDATE("9"),
    LLTCC_EVAL_FIRST_QUOTE("42"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣42␣marco?)";
f[i].suffix = "eval␣<constant>␣<symbol>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_LOOKUP("marco?"),LLTCC_EVAL_VALIDATE("10"),
    LLTCC_EVAL_FIRST_QUOTE("42"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣42␣(get␣an␣environment))";
f[i].suffix = "eval␣<constant>␣<complex␣expression>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_COMPLEX("get","an␣environment"),
    LLTCC_EVAL_VALIDATE("16"),LLTCC_EVAL_FIRST_QUOTE("42"),LLTCC_EVAL_TWOARG());
```

**451.** Two expressions where the first is a symbol.

⟨ Unit test part: compiler/**eval** fixtures 448 ⟩ +≡

```
f[++i].src_exp = "(eval␣marco?␣24)";
f[i].suffix = "eval␣<symbol>␣<constant>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_QUOTE("24"),LLTCC_EVAL_VALIDATE("9"),
    LLTCC_EVAL_FIRST_LOOKUP("marco?"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣marco?␣polo!)";
f[i].suffix = "eval␣<symbol>␣<symbol>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_LOOKUP("polo!"),LLTCC_EVAL_VALIDATE("10"),
    LLTCC_EVAL_FIRST_LOOKUP("marco?"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣marco?␣(a␣new␣environment))";
f[i].suffix = "eval␣<symbol>␣<complex␣expression>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_COMPLEX("a","new␣environment"),
    LLTCC_EVAL_VALIDATE("16"),LLTCC_EVAL_FIRST_LOOKUP("marco?"),LLTCC_EVAL_TWOARG());
```

**452.** Two expressions where the first is a complex expression.

⟨ Unit test part: compiler/**eval** fixtures 448 ⟩ +≡

```
f[++i].src_exp = "(eval␣(get␣an␣expression)␣24)";
f[i].suffix = "eval␣<complex␣expression>␣<constant>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_QUOTE("24"),LLTCC_EVAL_VALIDATE("9"),
    LLTCC_EVAL_FIRST_COMPLEX("get","an␣expression"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣(get␣another␣expression)␣marco?)";
f[i].suffix = "eval␣<complex␣expression>␣<symbol>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_LOOKUP("marco?"),LLTCC_EVAL_VALIDATE("10"),
    LLTCC_EVAL_FIRST_COMPLEX("get","another␣expression"),LLTCC_EVAL_TWOARG());
f[++i].src_exp = "(eval␣(once␣more)␣(this␣time␣with␣feeling))";
f[i].suffix = "eval␣<complex␣expression>␣<complex␣expression>";
f[i].want = CAT4(LLTCC_EVAL_SECOND_COMPLEX("this","time␣with␣feeling"),
    LLTCC_EVAL_VALIDATE("16"),LLTCC_EVAL_FIRST_COMPLEX("once","more"),LLTCC_EVAL_TWOARG());
```

**453. I/O.**

**454. 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 454 ⟩ ≡
  ⟨ Old test executable wrapper 243 ⟩
  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 455 ⟩
    ⟨ Test integrating car 456 ⟩
    ⟨ Test integrating cdr 457 ⟩
    ⟨ Test integrating null? 458 ⟩
    ⟨ Test integrating pair? 461 ⟩
    ⟨ Test integrating set-car! 464 ⟩
    ⟨ Test integrating set-cdr! 465 ⟩
  }
```

**455.** 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 455 ⟩ ≡
  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 454.

**456.**

```
⟨ Test integrating car 456 ⟩ ≡
  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 454.

**457.** ⟨ Test integrating cdr 457 ⟩ ≡
```
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 454.

**458.** ⟨ Test integrating null? 458 ⟩ ≡
```
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 459 and 460.

This code is used in section 454.

**459.** ⟨ Test integrating null? 458 ⟩ +≡
```
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);
```

**460.** ⟨ Test integrating null? 458 ⟩ +≡
```
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);
```

**461.** ⟨ Test integrating pair? 461 ⟩ ≡
```
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 462 and 463.

This code is used in section 454.

**462.** ⟨ Test integrating pair? 461 ⟩ +≡
```
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);
```

**463.** ⟨ Test integrating pair? 461 ⟩ +≡
```
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);
```

**464.** 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! 464 ⟩ ≡
```
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 454.

**465.** ⟨ Test integrating set-cdr! 465 ⟩ ≡
```
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 454.

**466. 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 466 ⟩ ≡
  ⟨ Old test executable wrapper 243 ⟩
  void test_main(void)
  {
    cell t, m, p;
    char *prefix;
    char msg[TEST_BUFSIZE] = {0};
    ⟨ Test integrating eval 467 ⟩
  }
```

See also section 472.

**467.** 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 467 ⟩ ≡
  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 468, 469, and 470.

This code is used in section 466.

**468.** 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 467 ⟩ +≡
  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);
```

**469.** 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 467 ⟩ +≡
  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);
```

**470.** **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 467 ⟩ +≡
  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)"));
```

**471.** 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);
```

**472.**

```
⟨ t/eval.c 466 ⟩ +≡
  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 473 ⟩
    ⟨ Test the inner environment when testing eval 474 ⟩
  }
```

**473.**

```
⟨ Test the outer environment when testing eval 473 ⟩ ≡
  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 472.

**474.** ⟨ Test the inner environment when testing **eval** 474 ⟩ ≡

```
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 472.

**475. Conditional Integration.** Before testing conditional interaction with *environment*s 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 475 ⟩ ≡
  ⟨ Old test executable wrapper 243 ⟩
  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 476 ⟩
    ⟨ Test integrating if 480 ⟩
  }
```

**476.** 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 476 ⟩ ≡
  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 477, 478, and 479.

This code is used in section 475.

**477.** `(if #f 'marco?)` ⇒ `VOID`:

```
⟨ Sanity test if's syntax 476 ⟩ +≡
  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);
```

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

⟨ Sanity test **if**'s syntax 476 ⟩ +≡
```
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);
```

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

⟨ Sanity test **if**'s syntax 476 ⟩ +≡
```
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);
```

**480.** 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** 480 ⟩ ≡
```
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 481, 482, and 483.

This code is used in section 475.

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

```
⟨ Test integrating if 480 ⟩ +≡
  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)"));
```

**482.**

```
⟨ Test integrating if 480 ⟩ +≡
  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)"));
```

**483.** 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 480 ⟩ +≡
  Root = Tmp_Test;
```

**484. 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 484 ⟩ ≡
  ⟨ Old test executable wrapper 243 ⟩
  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 485 ⟩
    ⟨ Test entering an applicative closure 486 ⟩
    ⟨ Applicative test passing an applicative 487 ⟩
    ⟨ Applicative test passing an operative 488 ⟩
    ⟨ Applicative test returning an applicative 489 ⟩
    ⟨ Applicative test returning an operative 490 ⟩
  }
```

**485.** 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** 485 ⟩ ≡

```
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 484.

**486.** 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 486 ⟩ ≡

```
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 484.

**487.** 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* 487 ⟩ ≡

```
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 484.

**488.** 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 *environment*s 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* 488 ⟩ ≡

```
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 484.

**489.** 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* 489 ⟩ ≡

```
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 484.

**490.** 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 *environment*s 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* 490 ⟩ ≡

```
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 484.

**491. 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 491 ⟩ ≡
  ⟨ Old test executable wrapper 243 ⟩
  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 492 ⟩
    ⟨ Test entering an operative closure 493 ⟩
    ⟨ Operative test passing an applicative 494 ⟩
    ⟨ Operative test passing an operative 495 ⟩
    ⟨ Operative test returning an applicative 496 ⟩
    ⟨ Operative test returning an operative 497 ⟩
  }
```

**492.**

```
#define TEST_OB "(vov␣((E␣vov/env)))"
#define TEST_OB_PRINT "(vov␣((E␣vov/env))␣...)"
⟨ Test calling vov 492 ⟩ ≡
  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 491.

**493.** 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 493 ⟩ ≡

```
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 491.

**494.** 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* 494 ⟩ ≡

```
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 491.

**495.** 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* 495 ⟩ ≡

```
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 491.

**496.** 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* 496 ⟩ ≡

```
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 491.

**497.** 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* 497 ⟩ ≡

```
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 491.

**498. 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 498 ⟩ ≡

```
⟨ Old test executable wrapper 243 ⟩
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);
}
```

### 499. TODO.

⟨ List of opcode primitives 499 ⟩ ≡ /* 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 231 ⟩
#**endif**

See also section 505.

This code is used in section 91.

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

```
⟨ repl.c 500 ⟩ ≡
#include "lossless.h"
  int main(int argc, char **argv__unused)
  {
    char wbuf[BUFFER_SEGMENT] = {0};
    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, wbuf, BUFFER_SEGMENT, 0);
        printf("%s\n", wbuf);
      }
    }
    if (Interrupt) printf("Interrupted");
    return EXIT_SUCCESS;
  }
```

**501. Association Lists.**

```
⟨ Function declarations 8 ⟩ +≡
  cell assoc_member(cell, cell);
  cell assoc_content(cell, cell);
  cell assoc_value(cell, cell);
```

**502.**
```
  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);
  }
```

**503. Misc.**

**504.** **#define** *synquote_new*(*o*) *atom*(*Sym_SYNTAX_QUOTE*, (*o*), FORMAT_SYNTAX)
/* */

**505.** ⟨ List of opcode primitives 499 ⟩ +≡
{"symbol?", *compile_symbol_p*} ,

**506.** **void** *compile_symbol_p*(**cell** *op*, **cell** *args*, **boolean** *tail_p__unused*)
{
*arity*(*op*, *args*, 1, 0);
*compile_expression*(*cts_pop*( ), 0);
*emitop*(OP_SYMBOL_P);
}

**507.** ⟨ Opcode implementations 10 ⟩ +≡
**case** OP_SYMBOL_P:
*Acc* = *symbol_p*(*Acc*) ? TRUE : FALSE;
*skip*(1);
**break**;

## 508. Index.

⟨ Applicative test passing an *applicative* 487 ⟩ Used in section 484.
⟨ Applicative test passing an *operative* 488 ⟩ Used in section 484.
⟨ Applicative test returning an *applicative* 489 ⟩ Used in section 484.
⟨ Applicative test returning an *operative* 490 ⟩ Used in section 484.
⟨ Compile a combiner 176 ⟩ Used in section 174.
⟨ Compile an atom 175 ⟩ Used in section 174.
⟨ Compile applicative combiner 189 ⟩ Used in section 176.
⟨ Compile native combiner 178 ⟩ Used in section 176.
⟨ Compile operative combiner 198 ⟩ Used in section 176.
⟨ Compile unknown combiner 179 ⟩ Used in section 176.
⟨ Compile unquote-splicing 213, 214, 215, 216 ⟩ Used in section 212.
⟨ Complex definitions & macros 144, 145, 249 ⟩ Used in sections 1 and 2.
⟨ Evaluate optional arguments into a *list* 193 ⟩ Used in section 189.
⟨ Evaluate required arguments onto the stack 192 ⟩ Used in section 189.
⟨ Externalised global variables 7, 13, 20, 26, 31, 42, 48, 57, 67, 79, 92, 94, 102, 113, 147, 166, 181, 221, 224, 246 ⟩ Used in section 1.
⟨ Function declarations 8, 14, 22, 27, 36, 43, 51, 59, 70, 73, 81, 88, 97, 104, 109, 115, 135, 167, 206, 227, 240, 244, 471, 501 ⟩ Used in sections 1 and 2.
⟨ Global initialisation 3, 33, 69, 80, 103, 114, 182, 195 ⟩ Cited in section 97. Used in section 98.
⟨ Global variables 6, 12, 19, 25, 30, 47, 56, 66, 78, 91, 93, 101, 112, 148, 165, 180, 194, 220, 223, 245 ⟩ Used in section 2.
⟨ Handle terminable 'forms' during *list* construction 129 ⟩ Used in section 128.
⟨ List of opcode primitives 499, 505 ⟩ Used in section 91.
⟨ Look for optional arguments 191 ⟩ Used in section 189.
⟨ Look for required arguments 190 ⟩ Used in section 189.
⟨ Mutate if bound 85 ⟩ Used in section 84.
⟨ Mutate if unbound 86 ⟩ Used in section 84.
⟨ Old test executable wrapper 243 ⟩ Used in sections 454, 466, 475, 484, 491, and 498.
⟨ Opcode implementations 10, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 162, 163, 164, 507 ⟩ Used in section 110.
⟨ Operative test passing an *applicative* 494 ⟩ Used in section 491.
⟨ Operative test passing an *operative* 495 ⟩ Used in section 491.
⟨ Operative test returning an *applicative* 496 ⟩ Used in section 491.
⟨ Operative test returning an *operative* 497 ⟩ Used in section 491.
⟨ Pre-initialise *Small_Int* & other gc-sensitive buffers 15, 23, 28, 34, 50, 60, 68, 96, 169, 226 ⟩ Used in section 98.
⟨ Process lambda formals 188 ⟩ Used in section 187.
⟨ Protected Globals 21, 32, 49, 58, 95, 168, 225 ⟩ Used in section 41.
⟨ Quasiquote a pair/list 209 ⟩ Used in section 208.
⟨ Quasiquote syntax 210, 211, 212 ⟩ Used in section 208.
⟨ Read bytes until an invalid or terminating character 134 ⟩ Used in section 132.
⟨ Read dotted pair 130 ⟩ Used in section 129.
⟨ Read the first two bytes to check for a number 133 ⟩ Used in section 132.
⟨ Reader forms 121, 123, 124, 125, 126, 127 ⟩ Used in section 120.
⟨ Repeat the fixture with garbage collection 261 ⟩ Used in section 259.
⟨ Sanity test **if**'s syntax 476, 477, 478, 479 ⟩ Used in section 475.
⟨ Scan operative informals 197 ⟩ Used in section 196.
⟨ Search *Root* for syntactic combiners 177 ⟩ Used in section 176.
⟨ System headers 4 ⟩ Used in sections 1, 2, and 222.
⟨ Test calling **lambda** 485 ⟩ Used in section 484.
⟨ Test calling **vov** 492 ⟩ Used in section 491.
⟨ Test entering an *applicative* closure 486 ⟩ Used in section 484.
⟨ Test entering an *operative* closure 493 ⟩ Used in section 491.
⟨ Test integrating car 456 ⟩ Used in section 454.

⟨ Unit test: garbage collector *sweep* 330, 331, 332, 333, 334, 335, 336, 337, 338 ⟩ Used in section 329.
⟨ Unit test: grow heap pool 265, 266, 267, 268, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282 ⟩ Used in section 264.
⟨ Unit test: grow vector pool 284, 285, 286, 287, 290, 291, 292, 293, 294, 295 ⟩ Used in section 283.
⟨ Unmark all vectors 46 ⟩ Used in section 45.
⟨ Walk through the splicing list 217, 218 ⟩ Used in section 216.
⟨ lossless.h 1 ⟩
⟨ repl.c 500 ⟩
⟨ t/cell-heap.c 264 ⟩
⟨ t/compiler.c 439 ⟩
⟨ t/environments.c 354 ⟩
⟨ t/eval.c 466, 472 ⟩
⟨ t/exception.c 498 ⟩
⟨ t/gc-mark.c 296 ⟩
⟨ t/gc-sweep.c 329 ⟩
⟨ t/gc-vector.c 339 ⟩
⟨ t/if.c 475 ⟩
⟨ t/interpreter.c 376 ⟩
⟨ t/lambda.c 484 ⟩
⟨ t/llalloc.c 222 ⟩
⟨ t/llt.h 254 ⟩
⟨ t/lltest.c 219 ⟩
⟨ t/pair.c 454 ⟩
⟨ t/sanity.c 253 ⟩
⟨ t/vector-heap.c 283 ⟩
⟨ t/vov.c 491 ⟩

# LossLess Programming Environment