1. VM bytecode is an array of cell pairs. The first of each pair is a fixnum encoding the opcode and a register. The second pair is a pointer or (depending on an opcode flag) a fixnum identifying another register.

Program is a "treap" largely stolen from Knuth's mmix (`mmix-sim.pdf` ss. 16–22) so that bytecode addresses can cover a large address space without taking up a lot of real space (or search time).

```
unique cell Program;      /* Program bytecode; an array. */
unique int IP, Trapped;   /* Current & waiting (if mid-trap) instruction. */
```

2.
```
#define code(I) (fix_value(proglet_ref(find_proglet(I))))
#define op(O) (code(I) & #00ff)
#define target(I) (Iregister[(code(I) & #ff00) >> 8])
#define value(I) (proglet_ref(find_proglet((I) + 1)))
#define regval(I) (fix_value(value(I)))
#define regptr(I) (Iregister[regval(I)])
void interpret(sigjmp_buf *failure)
{
  cell val;
  bool trap = false;
  while (¬trap) {
    if (IP < 0 ∨ IP % 2 ∨ null_p(find_proglet(IP)))
      trap = true, inst = OP_TRAP_DIR, val = fix(LERR_INSTRUCTION_ADDRESS);
    else if (¬(op(IP) & OP_MASK_ARG)) inst = op(IP), val = value(IP);   /* val "is" NIL. */
    else if (op(IP) & OP_MASK_DIRECT) inst = op(IP), val = value(IP);
    else if (regval(IP) ≥ 0 ∧ regval(IP) < LGCR_LENGTH) inst = op(IP), val = regptr(IP);
    else trap = true, inst = OP_TRAP_DIR, val = fix(LERR_INCOMPATIBLE);
    if (¬trap) IP += 2;
    switch (op(IP)) {⟨ Virtual Machine Opcode 3 ⟩}
  }
}
```

3.
```
#define real_trap(V) ((fix_value(val) ≥ 0 ∧ fix_value(val) ≤ LERR_LENGTH)
      ? fix_value(V) : LERR_UNIMPLEMENTED)
```

⟨ Virtual Machine Opcode 3 ⟩ ≡
```
default: val = LERR_UNIMPLEMENTED;
Trap: case OP_TRAP_DIR: case OP_TRAP_REG:
  if (¬null_p(trap_handler(real_trap(val)))) {
    trap = false;
    Trapped = IP;
    IP = fix_value(real_trap(val));
  }
  break;
```

See also sections 4 and 5.

This code is used in section 2.

4. ⟨ Virtual Machine Opcode 3 ⟩ +≡

```
case OP_JUMP_DIR: case OP_JUMP_REG:
  IP = fix_value(val);
  break;
case OP_JUMPIF_DIR: case OP_JUMPIF_REG:
  if (¬false_p(ACC)) IP = fix_value(val);
  break;
case OP_JUMPNOT_DIR: case OP_JUMPNOT_REG:
  if (false_p(ACC)) IP = fix_value(val);
  break;
case OP_HALT_DIR: case OP_HALT_REG:
  trap = true;
  break;
```

5. ⟨ Virtual Machine Opcode 3 ⟩ +≡

```
case OP_LINK_DIR: case OP_LINK_REG: failure = link_code(target(IP − 2), fix_value(val));
  if (failure_p(failure)) {
    IP −= 2;
    trap = true;
    val = failure;
    goto Trap;
  }
  break;
```

6.

```
bool bytecode_p(cell o)
{
  int i;
  if (¬array_p(o) ∨ array_length(o) ≡ 0 ∨ array_length(o) % 2) return false;
  for (i = 0; i < array_length(o); i += 2)
    if (¬fix_p(array_ref(o, i))) return false;
  return true;
}
```

7. Find a new random location in *Program* which has *length* consecutive positions available.

```
int link_random_base(int length)
{    /* assert(Program is locked); */
  assert (length > 0 ∧ ¬(length % 2));
  ...;
}
```

8. **#define** *lock*(*X*)
#define *unlock*(*X*)

```
Verror link_code(cell bc, int addr)
{
  Verror r;
  sigjmp_buf cleanup;
  Verror reason = LERR_NONE;
  if (addr < -1 ∨ addr % 2) return LERR_LINK_ADDRESS;
  if (¬bytecode_p(bc)) return LERR_INCOMPATIBLE;
  lock(Program);
  if (failure_p(reason = sigsetjmp(cleanup, 1))) {
    unlock(Program);
    return reason;
  }
  if (addr ≡ -1) addr = link_random_base(array_length(bc));
  link_reserve(addr, array_length(bc), &cleanup);
  unlock(Program);
  if (failure_p(reason = sigsetjmp(cleanup, 1))) {
    return reason;
  }
  link_install(bc, addr, &cleanup);
  return LERR_NONE;
}
```

9. Copy the bytecode from *bc* to the previously-reserved area of *Program* at *base*.

```
void link_install(cell bc, int base, sigjmp_buf *failure)
{
  cell n;
  int i, j;
  assert (bytecode_p(bc));
  assert (base ≥ 0 ∧ ¬(base % 2));
  i = 0;
  n = find_proglet(base);
  j = base & PROGRAM_MASK;
  while (1) {
    while (j < array_length(n)) {
      proglet_set_m(n, j, array_ref(bc, i));
      proglet_set_m(n, j + 1, array_ref(bc, i + 1));
      i += 2;
      if (i ≡ array_length(bc)) return;
      j += 2;
      base += 2;
    }
    n = find_proglet(base);
    j = 0;
  }
}
```

10. **#define** PROGRAM_START #0
#define PROGRAM_LENGTH 64
#define PROGRAM_MASK #3f

⟨ Global 10 ⟩ ≡
shared int32_t *Priority* = #f5163b17;
cell *Program*;
cell *Recent*;

11.

#define *proglet_buffer*(*O*) ((*Oprogram* ∗) *segment_address*(*O*))
#define *proglet_base*(*O*) (*proglet_buffer*(*O*)→*base*)
#define *proglet_base_set_m*(*O*, *D*) (*proglet_buffer*(*O*)→*base* = (*D*))
#define *proglet_stamp*(*O*) (*proglet_buffer*(*O*)→*stamp*)
#define *proglet_stamp_set_m*(*O*, *D*) (*proglet_buffer*(*O*)→*stamp* = (*D*))
#define *proglet_free*(*O*) (*proglet_buffer*(*O*)→*free*)
#define *proglet_free_set_m*(*O*, *D*) (*proglet_buffer*(*O*)→*free* = (*D*))
#define *proglet_below*(*O*) (*proglet_buffer*(*O*)→*below*)
#define *proglet_below_set_m*(*O*, *D*) (*proglet_buffer*(*O*)→*below* = (*D*))
#define *proglet_above*(*O*) (*proglet_buffer*(*O*)→*above*)
#define *proglet_above_set_m*(*O*, *D*) (*proglet_buffer*(*O*)→*above* = (*D*))
#define *proglet_ref*(*O*, *I*) (*proglet_buffer*(*O*)→*address*[(*I*)])
#define *proglet_set_m*(*O*, *I*, *D*) (*proglet_buffer*(*O*)→*address*[(*I*)] = (*D*))
#define *proglet_has_p*(*O*, *D*) (*proglet_base*(*O*) ≤ (*D*) ∧ (*proglet_base*(*O*) + PROGRAM_LENGTH) > (*D*))

⟨ Type def 11 ⟩ ≡
struct Oprogram {
uintptr_t *base*;
int32_t *stamp*;
cell *below*, *above*;
int8_t *free*;
cell *address*[ ];
};
typedef struct Oprogram Oprogram;

12. ⟨ Init 12 ⟩ ≡
Recent = *Program* = *init_proglet*( );
proglet_base(*Program*) = PROGRAM_START;

13. **cell** *init_proglet*(**sigjmp_buf** ∗*failure*)

```
{
  cell r;
  r = segment_new(sizeof(Oprogram), PROGRAM_LENGTH, sizeof(cell), 0, failure);
  proglet_base_set_m(r, UINTPTR_MAX);
  proglet_stamp_set_m(r, Priority);
  Priority += #9d3779b9;      /* mmix-sim.pdf ss. 17 */
  proglet_free_set_m(r, 0);
  proglet_below_set_m(r, NIL);
  proglet_above_set_m(r, NIL);
  for (i = 0; i < PROGRAM_LENGTH; i += 0) {
    proglet_set_m(r, i, fix(OP_HALT_DIR));
    proglet_set_m(r, i + 1, NIL);
  }
  return r;
}
```

14. Reserve *length* consecutive locations in *Program* beginning at *base*, inserting new proglet(s) as necessary.

```
void link_reserve(int base, int length, sigjmp_buf *failure);
{
  int64_t key;
  int off;
  bool lower;
  cell above, below, new, p, q;      /*assert(Program is locked); */
  assert (base ≥ 0 ∧ ¬(base % 2));
  assert (length > 0 ∧ ¬(length % 2));
  if (proglet_has_p(Recent, base)) p = Recent;
  else {
  more: p = find_proglet(base);
    if (¬null_p(p)) goto found;
    ⟨ Insert a new node into Program 15 ⟩
  found: Recent = p;
    ⟨ Reserve space in p and return or insert another node 17 ⟩
  }
}
```

15. Look for a node in *Program* prior to where *base* would be, or the most recently inserted one.

```
⟨ Insert a new node into Program 15 ⟩ ≡
  key = base & ~(PROGRAM_MASK);
  off = base & PROGRAM_MASK;
  for (p = q = Program; ; p = q) {
    lower = key < proglet_base(p);
    if (lower) q = proglet_below(p);
    else q = proglet_above(p);
    if (null_p(q) ∨ proglet_stamp(q) ≥ Priority) break;
  }
  init_proglet(failure);      /* TODO: GC loses all pointers! */
  proglet_base_m(new, key);
  if (lower) proglet_below_set_m(p, new);
  else proglet_above_set_m(p, new);
  ⟨ Relink nodes in Program to balance the tree 16 ⟩
```

This code is used in section 14.

16.

```
⟨ Relink nodes in Program to balance the tree 16 ⟩ ≡
  p = q;
  above = below = q = new;
  while (¬null_p(p)) {
    if (key < proglet_base(p)) {
      proglet_above_set_m(above, p);
      above = p;
      p = proglet_below(p);
    }
    else {
      proglet_below_set_m(below, p);
      below = p;
      p = proglet_above(p);
    }
  }
  proglet_above_set_m(above, NIL);
  proglet_below_set_m(below, NIL);
  p = q;
```

This code is used in section 15.

17.

```
⟨ Reserve space in p and return or insert another node 17 ⟩ ≡
  if (proglet_free(p) > off) siglongjmp (*failure, LERR_RESERVED);
  if (off + length ≤ PROGRAM_LENGTH) {
    proglet_free_set_m(p, off + length);
    return;
  }
  length -= (PROGRAM_LENGTH - proglet_free(p));
  proglet_free_set_m(p, PROGRAM_LENGTH);
  base = key + PROGRAM_LENGTH;
  goto more;
```

This code is used in section 14.

18. Return the array object within *Program* which holds the address in *base*.

```
cell find_proglet(int base)
{
  cell p;
  int64_t key;
  assert (base ≥ 0 ∧ ¬(base % 2));
  if (proglet_has_p(Recent, base)) return Recent;
  key = base & ∼(PROGRAM_MASK);
  for (p = Program; ¬null_p(p); ) {
    if (proglet_base(p) ≡ key) return p;
    if (key < proglet_base(p)) p = proglet_below(p);
    else p = proglet_above(p);
  }
  return p;      /* May be NIL. */
}
```