

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. $\langle \text{Virtual Machine Opcode } 3 \rangle + \equiv$
case OP_JUMP_DIR: **case** OP_JUMP_REG:
 IP = *fix_value*(*val*);
 break;
case OP_JUMPIF_DIR: **case** OP_JUMPIF_REG:
 if ($\neg \text{false}_p(\text{ACC})$) 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. $\langle \text{Virtual Machine Opcode } 3 \rangle + \equiv$
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 ($\neg \text{array}_p(o) \vee \text{array_length}(o) \equiv 0 \vee \text{array_length}(o) \% 2$) **return** *false*;
 for (*i* = 0; *i* < *array_length*(*o*); *i* += 2)
 if ($\neg \text{fix}_p(\text{array_ref}(o, i))$) **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 $\wedge \neg(\text{length} \% 2)$);
 ...;
 }

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 *proklet.buffer*(*O*) ((*Oprogram* *) *segment_address*(*O*))

#define *proklet.base*(*O*) (*proklet.buffer*(*O*)-*base*)

#define *proklet.base_set_m*(*O*, *D*) (*proklet.buffer*(*O*)-*base* = (*D*))

#define *proklet.stamp*(*O*) (*proklet.buffer*(*O*)-*stamp*)

#define *proklet.stamp_set_m*(*O*, *D*) (*proklet.buffer*(*O*)-*stamp* = (*D*))

#define *proklet.free*(*O*) (*proklet.buffer*(*O*)-*free*)

#define *proklet.free_set_m*(*O*, *D*) (*proklet.buffer*(*O*)-*free* = (*D*))

#define *proklet.below*(*O*) (*proklet.buffer*(*O*)-*below*)

#define *proklet.below_set_m*(*O*, *D*) (*proklet.buffer*(*O*)-*below* = (*D*))

#define *proklet.above*(*O*) (*proklet.buffer*(*O*)-*above*)

#define *proklet.above_set_m*(*O*, *D*) (*proklet.buffer*(*O*)-*above* = (*D*))

#define *proklet.ref*(*O*, *I*) (*proklet.buffer*(*O*)-*address*[(*I*)])

#define *proklet.set_m*(*O*, *I*, *D*) (*proklet.buffer*(*O*)-*address*[(*I*)] = (*D*))

#define *proklet.has_p*(*O*, *D*) (*proklet.base*(*O*) ≤ (*D*) ∧ (*proklet.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_proklet*();

proklet_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. */
}
```

above: [11](#), [14](#), [16](#).

ACC: [4](#).

addr: [8](#).

address: [11](#).

array_length: [6](#), [8](#), [9](#).

array_p: [6](#).

array_ref: [6](#), [9](#).

assert: [7](#), [9](#), [14](#), [18](#).

base: [9](#), [11](#), [14](#), [15](#), [17](#), [18](#).

bc: [8](#), [9](#).

below: [11](#), [14](#), [16](#).

bytecode_p: [6](#), [8](#), [9](#).

cell: [1](#), [2](#), [6](#), [8](#), [9](#), [10](#), [11](#), [13](#), [14](#), [18](#).

cleanup: [8](#).

code: [2](#).

failure: [2](#), [5](#), [9](#), [13](#), [14](#), [15](#), [17](#).

failure_p: [5](#), [8](#).

false: [2](#), [3](#), [6](#).

false_p: [4](#).

find_proglet: [2](#), [9](#), [14](#), [18](#).

fix: [2](#), [13](#).

fix_p: [6](#).

fix_value: [2](#), [3](#), [4](#), [5](#).

found: [14](#).

free: [11](#).

i: [6](#), [9](#).

init_proglet: [12](#), [13](#), [15](#).

inst: [2](#).

interpret: [2](#).

int32_t: [10](#), [11](#).

int64_t: [14](#), [18](#).

int8_t: [11](#).

IP: [1](#), [2](#), [3](#), [4](#), [5](#).

Iregister: [2](#).

j: [9](#).

key: [14](#), [15](#), [16](#), [17](#), [18](#).

length: [7](#), [14](#), [17](#).

LERR_INCOMPATIBLE: [2](#), [8](#).

LERR_INSTRUCTION_ADDRESS: [2](#).

LERR_LENGTH: [3](#).

LERR_LINK_ADDRESS: [8](#).

LERR_NONE: [8](#).

LERR_RESERVED: [17](#).

LERR_UNIMPLEMENTED: [3](#).

LGCR_LENGTH: [2](#).

link_code: [5](#), [8](#).

link_install: [8](#), [9](#).

link_random_base: [7](#), [8](#).

link_reserve: [8](#), [14](#).

lock: [8](#).

lower: [14](#), [15](#).

more: [14](#), [17](#).

n: [9](#).

new: [14](#), [15](#), [16](#).

NIL: [2](#), [13](#), [16](#), [18](#).

null_p: [2](#), [3](#), [14](#), [15](#), [16](#), [18](#).

o: [6](#).

off: [14](#), [15](#), [17](#).

op: [2](#).

OP_HALT_DIR: [4](#), [13](#).

OP_HALT_REG: [4](#).

OP_JUMP_DIR: [4](#).

OP_JUMP_REG: [4](#).

OP_JUMPIF_DIR: [4](#).

OP_JUMPIF_REG: [4](#).

OP_JUMPNOT_DIR: [4](#).

OP_JUMPNOT_REG: [4](#).

OP_LINK_DIR: [5](#).

OP_LINK_REG: [5](#).

OP_MASK_ARG: [2](#).

OP_MASK_DIRECT: [2](#).

OP_TRAP_DIR: [2](#), [3](#).

OP_TRAP_REG: 3.
Oprogram: 11, 13.
p: 14, 18.
Priority: 10, 13, 15.
proklet_above: 11, 15, 16, 18.
proklet_above_set_m: 11, 13, 15, 16.
proklet_base: 11, 12, 15, 16, 18.
proklet_base_m: 15.
proklet_base_set_m: 11, 13.
proklet_below: 11, 15, 16, 18.
proklet_below_set_m: 11, 13, 15, 16.
proklet_buffer: 11.
proklet_free: 11, 17.
proklet_free_set_m: 11, 13, 17.
proklet_has_p: 11, 14, 18.
proklet_ref: 2, 11.
proklet_set_m: 9, 11, 13.
proklet_stamp: 11, 15.
proklet_stamp_set_m: 11, 13.
Program: 1, 7, 8, 9, 10, 12, 14, 15, 18.
 PROGRAM_LENGTH: 10, 11, 13, 17.
 PROGRAM_MASK: 9, 10, 15, 18.
 PROGRAM_START: 10, 12.
q: 14.
r: 8, 13.
real_trap: 3.
reason: 8.
Recent: 10, 12, 14, 18.
regptr: 2.
regval: 2.
segment_address: 11.
segment_new: 13.
shared: 10.
sigjmp_buf: 2, 8, 9, 13, 14.
siglongjmp: 17.
sigsetjmp: 8.
stamp: 11.
target: 2, 5.
 TODO: 15.
trap: 2, 3, 4, 5.
Trap: 3, 5.
trap_handler: 3.
Trapped: 1, 3.
true: 2, 4, 5, 6.
 UINTPTR_MAX: 13.
uintptr_t: 11.
unique: 1.
unlock: 8.
val: 2, 3, 4, 5.
value: 2.
Verror: 8.

⟨ Global 10 ⟩

⟨ Init 12 ⟩

⟨ Insert a new node into *Program* 15 ⟩ Used in section 14.

⟨ Relink nodes in *Program* to balance the tree 16 ⟩ Used in section 15.

⟨ Reserve space in *p* and **return** or insert another node 17 ⟩ Used in section 14.

⟨ Type def 11 ⟩

⟨ Virtual Machine Opcode 3, 4, 5 ⟩ Used in section 2.