A VM program starts with a byte indicating the number of predicates P in the program. Next, there are several components:

- An unsigned integer indicating the number N of nodes to instantiate, followed by $2N$ unsigned integers corresponding to one pair of unsigned integers one for node. The first value is the node ID to use during execution and the second one the ID given by the user.
- An unsigned integer indicating the number of arguments needed to run the program.
- An unsigned integer describing the number of rules R in the program. Followed by R byte regions. Each region contains an unsigned integer, N, indicating the size of the rule and then N bytes with the string for this rule.
- An unsigned integer indicating the number S of constant strings in the program followed by S pairs containing the length of the string and the string itself.
- A byte indicating the number of code constants C and then C bytes for the types of such constants. Finally, there's an unsigned integer describing the code size for computing the constants and the code itself.
- A set of P *predicate descriptors*, with 69 bytes each.
- A set of P byte-code instructions, one for each predicate.

A predicate descriptor consists of the following fields:

- A short integer indicating the size, in bytes, of the corresponding byte-code instructions.
- 1 byte describing the predicate's properties.
- 1 byte indicating the aggregate's type, if any. The high nibble if the aggregate type and the low nibble the aggregate field.
- A byte indicating the predicate's number of fields F.
- 32 bytes with information about the fields' types. Actually, only F bytes are used, and the remaining bytes are zeroes.
- 32 bytes containing the predicate's name representing as a string. As before, unused bytes are left as zeroes.

INSTRUCTION	BYTE FORMAT	ARGS
IF	(see below)	*reg, jump_offset*
ELSE	(see below)	—
ITER	(see below)	*id, options, options arg, jump_offset, matc*
NEXT	(see below)	—
SEND	(see below)	reg_1, reg_2
REMOVE	(see below)	*reg*
OP	(see below)	val_1, val_2, val_3, *op*
NOT	(see below)	val_1, val_2
MOVE	(see below)	val_1, val_2

IF — args: *reg, jump_offset*

0	1	1	0	0	0	0	0
0	0	0	r	r	r	r	r
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j

if *reg* != 0 then process until ELSE and then jump.
if *reg* = 0 then jump to ELSE
(note: IFs may be nested)

ELSE — args: —

0	0	0	0	0	0	1	0

a marker for if blocks

ITER — args: *id, options, options arg, jump_offset, matc*

1	0	1	0	0	0	0	0
0	i	i	i	i	i	i	i
o	o	o	o	o	o	o	o
a	a	a	a	a	a	a	a
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j

iterates over all the tuples of type *id* that match according to the following *matchlist*.
after all matching facts have been processed, use *jump_offset* to jump to the next instruction

NEXT — args: —

0	0	0	0	0	0	0	1

return to iter and process next matching fact

SEND — args: reg_1, reg_2

0	0	0	0	1	0	0	0
0	0	0	r_1	r_1	r_1	r_1	r_1
0	0	0	r_2	r_2	r_2	r_2	r_2

sends the tuple in reg_1 along the path in reg_2
if $reg_1 = reg_2$ then the tuple is stored locally

REMOVE — args: *reg*

1	0	0	r	r	r	r	r

delete tuple stored in reg from database

OP — args: val_1, val_2, val_3, *op*

1	1	0	0	0	0	0	0
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2
0	0	v_3	v_3	v_3	v_3	v_3	v_3
0	0	0	o	o	o	o	o

sets $val_3 = val_1 \ op \ val_2$

NOT — args: val_1, val_2

0	0	0	0	0	1	1	1
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets $val_2 = not \ val_1$

MOVE — args: val_1, val_2

0	0	1	1	0	0	0	0
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

copies val_1 to val_2

INSTRUCTION	BYTE FORMAT	ARGS

MOVE-NIL

0	1	1	1	0	0	0	0
0	0	v	v	v	v	v	v

sets *val* to the nil list

ARGS: *val*

TEST-NIL

0	0	0	0	0	0	1	1
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

$v_2 = 1$ if v_1 is nil.
$v_2 = 0$ if v_1 is not nil.

ARGS: val_1, val_2

ALLOC

0	1	0	0	0	0	0	0
0	i	i	i	i	i	i	i
0	0	v	v	v	v	v	v

allocates a tuple of type *id* and stores it in *val*

ARGS: *id, val*

RETURN

0	0	0	0	0	0	0	0

finished processing the tuple - return

ARGS: —

CALL

0	0	1	0	0	0	0	0
0	i	i	i	i	i	i	i
0	0	0	r	r	r	r	r

call external function number *id* with *args* and store the result in *reg*

ARGS: *id, reg, args*

CONS

0	0	0	0	0	1	0	0
0	0	0	0	0	0	*t*	*t*
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2
0	0	v_3	v_3	v_3	v_3	v_3	v_3

sets $val_3 = val_1 :: val_2$
t is the list type (00 = int, 01 = float, 02 = addr)

ARGS: val_1, val_2, val_3

HEAD

0	0	0	0	0	1	0	1
0	0	0	0	0	0	*t*	*t*
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets $val_2 = head\ val_1$
t is the list type (00 = int, 01 = float, 02 = addr)

ARGS: val_1, val_2

TAIL

0	0	0	0	0	1	1	0
0	0	0	0	0	0	*t*	*t*
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets $val_2 = tail\ val_1$
t is the list type (00 = int, 01 = float, 02 = addr)

ARGS: val_1, val_2

FLOAT

0	0	0	0	1	0	0	1
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets $val_2 = (float) val_1$

ARGS: val_1, val_2

INSTRUCTION	BYTE FORMAT	ARGS

SELECT — ARGS: *size,hsize,htable,codeblocks*

0	0	0	0	1	0	1	0

this is a big instruction used to select a specific code block for a node. it is followed by a 4-byte integer indicating the *size* of the whole instruction, then a 4-byte integer indicating the size N of a simplified hash-table. N represents the number of nodes in the system for efficiency reasons. Next, there is N*4-byte integers, where each integer is the offset to a code block of the corresponding node. The offsets start after the end of the hash table. If the offset is 0, this node has no associated code block, so it should use *size* to jump to the next instruction. If the offset is positive, you should subtract one byte from it and then jump to the code block. At the end of each code block, there is a RETURN-SELECT.

RETURN-SELECT — ARGS: *jump*

0	0	0	0	1	0	1	1

This instruction is followed by a 4-byte integer with a jump offset to the next instruction.

COLOCATED — ARGS: *n1,n2,dest*

0	0	0	0	1	1	0	0
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2
0	0	0	r	r	r	r	r

sets dest = true if nodes n1 and n2 are on the same machine
sets dest = false otherwise

DELETE — ARGS: *i,v1*

0	0	0	0	1	1	0	1
0	i	i	i	i	i	i	i
0	0	v_1	v_1	v_1	v_1	v_1	v_1

deletes the tuples of type i with the first argument as value v_1

REMOVE — ARGS: *reg*

1	0	0	0	0	0	0	0
0	0	0	r	r	r	r	r

deletes tuple reg from the database

RETURN-LINEAR

1	1	0	1	0	0	0	0

linear fact was used, execution must terminate

RETURN-DERIVED

1	1	1	1	0	0	0	0

head of rule was derived, return if some linear fact was used

INSTRUCTION	BYTE FORMAT								ARGS
RULE	0	0	0	1	0	0	0	0	*id*

rule *id* is gonna be executed

RULE DONE	0	0	0	1	0	0	0	0	

rule *id* has been matched

	0	0	0	1	0	0	1	0	
	j	j	j	j	j	j	j	j	
SAVE ORIGINAL	j	j	j	j	j	j	j	j	*jump*
	j	j	j	j	j	j	j	j	
	j	j	j	j	j	j	j	j	

save initial tuple and run the following
code until we hit a return.
the original tuple may be consumed
in the process.
if that's the case, then stop execution,
else continue by jumping to the outer block.

OP	BYTE FORMAT				
$float \neq$	0	0	0	0	0
$int \neq$	0	0	0	0	1
$float =$	0	0	0	1	0
$int =$	0	0	0	1	1
$float <$	0	0	1	0	0
$int <$	0	0	1	0	1
$float \leq$	0	0	1	1	0
$int \leq$	0	0	1	1	1
$float >$	0	1	0	0	0
$int >$	0	1	0	0	1
$float \geq$	0	1	0	1	0
$int \geq$	0	1	0	1	1
$float\%$	0	1	1	0	0
$int\%$	0	1	1	0	1
$float+$	0	1	1	1	0
$int+$	0	1	1	1	1
$float-$	1	0	0	0	0
$int-$	1	0	0	0	1
$float*$	1	0	0	1	0
$int*$	1	0	0	1	1
$float\div$	1	0	1	0	0
$int\div$	1	0	1	0	1
$addr \neq$	1	0	1	1	0
$addr =$	1	0	1	1	1
$addr >$	1	1	0	0	0
bool or	1	1	0	0	1

VALUE	BYTE FORMAT	ARGS
REG	1 r r r r r	*reg*
TUPLE	0 1 1 1 1 1 refers to the tuple currently being processed	—
HOST_ID	0 0 0 0 1 1 refers to the node currently being processed	—
NIL	0 0 0 1 0 0 the empty list	—
INT	0 0 0 0 0 1 the next 4 bytes after the current instruction are an immediate integer to which this refers	*int*
FLOAT	0 0 0 0 0 0 the next 4 bytes after the current instruction are an immediate float to which this refers	*float*
ADDR	0 0 0 1 0 1 the next 4 bytes after the current instruction are the address to which this refers	*addr*
FIELD	0 0 0 0 1 0 the next two bytes after the current instruction indicate a field of a register in the following format: X X X X f f f f X X X r r r r r with *reg* indicating a register with a tuple value and *field* indicating the tuple's field number.	*field, reg*
STRING	0 0 0 1 1 0 the next 4 bytes indicate the length of the string which are followed by the string itself	*size, content*
ARG	0 0 0 1 1 1 the next byte indicates the argument id	*id*
CONST	0 0 1 0 0 0 the next 4 bytes indicates the constant id	*const id*

ARGS	BYTE FORMAT	
VALUE	X X v v v v v v	*value*

MATCHLIST	BYTE FORMAT								
MATCHLIST	f	f	f	f	f	f	f	f	*field, marker, value*
	m	m	v	v	v	v	v	v	

requires that the tuple's field *field* match *value*
mm=11 if the match list is empty and mm=01 for the last entry in the list.

AGGREGATE	BYTE FORMAT			
none	0	0	0	0
first	0	0	0	1
int max	0	0	1	0
int min	0	0	1	1
int sum	0	1	0	0
float max	0	1	0	1
float min	0	1	1	0
float sum	0	1	1	1
int set_union	1	0	0	0
float set_union	1	0	0	1
int list sum	1	0	1	0
float list sum	1	0	1	1

TYPE	BYTE FORMAT			
int	0	0	0	0
float	0	0	0	1
addr	0	0	1	0
int list	0	0	1	1
float list	0	1	0	0
addr list	0	1	0	1
int set	0	1	1	0
float set	0	1	1	1
type	1	0	0	0
string	1	0	0	1

PROPERTY	BYTE POSITION
aggregate	1
persistent	2
linear	3
delete	4
schedule	5

NOTES:
All offsets and lengths are given in bytes.