

Project 3

Due Thursday, October 18, 2018, 11:59:59pm

1 Introduction

In this project, you will implement RubeVM, a virtual machine for a simple bytecode language. Later in the semester, you will compile Rube (a simplified Ruby) source code into RubeVM bytecode.

2 RubeVM Fundamentals

Figure 1 defines what the RubeVM instructions are, what values they manipulate, and provides some definitions that will be useful for the operational semantics. We'll discuss the figure from top to bottom.

RubeVM is a register-based machine, where *registers* r are simply numbered starting from 0. We assume there is an unbounded number of registers. Registers contain *values* v . The first two kinds of values are integers n (e.g., 0, 1, 10, -42, etc) and strings s . Values also include *identifiers* id . When an identifier is stored in a register, we treat it as a pointer to the function of that name (more on functions later). We also use identifiers to refer to global variables (but an identifier in a register is always a function pointer). Lastly, values include *locations* ℓ , which are pointers to *tables*, which are stored in the heap. A table t is a mapping from values to values.

There are many different kinds of RubeVM *instructions* i . We'll discuss the semantics of each instruction in detail below. For now, just observe there are several groups of instructions: Basics (loading constants, moving values); arithmetic and comparisons; branching; global variable access; table manipulation; and function calls.

A RubeVM *program* P is a mapping from function names (id) to *function bodies* F , which themselves are mappings from program counters (pc) to instructions. For example, if P is a program with some function f , then $P(f)(0)$ is the first instruction in function f . Note that program counters are just natural numbers, but we write them as pc as a reminder of what they're used for.

When a program runs, it may update the *heap* H . The heap stores two kinds of mappings in it: locations ℓ are mapped to tables t , and global variable names id are mapped to values v . In other words, if H is a heap and $\ell \in \text{dom}(H)$ (ℓ is in the domain of H , i.e., there is a mapping for ℓ in H), then $H(\ell)$ will be a table. Similarly, if H is a heap and $id \in \text{dom}(H)$, then $H(id)$ will be a value. Tables are not values because they are mutable in RubeVM.

As RubeVM executes a function, that function may access a *register file* R , which is just a mapping from register numbers to values. In RubeVM, each function invocation has its own local registers that are disjoint with any other function invocation, i.e., the registers are local variables.

Finally, as RubeVM executes, there will be function calls and returns. To handle these, RubeVM maintains a *stack* S . Each stack frame has the form (id, pc, R) , where id is the name of the function, pc is the program counter, and R is the register file. The stack is a sequence of stack frames, separated with $::$, with the top-most frame on the left.

3 RubeVM Operational Semantics

We'll describe the operational semantics as a small-step semantics over machine states $\langle H, S \rangle$, where H is the heap and S is the stack. Figure 2 gives the operational semantics rules. The column "Reduction" defines judgments of the form $P \vdash \langle H, S \rangle \rightarrow \langle H', S' \rangle$, meaning that one step of execution of program P with heap H and stack S yields a new heap H' and new stack S' . Recall the current function is always on the left of the stack. Hence in the rules we almost always write the configuration pattern as $\langle H, (id, pc, R)::S \rangle$, to indicate a state with heap H , current function id , current program counter pc , current registers R , and remainder of the stack S .

Registers	$r ::= 0 \mid 1 \mid \dots$	
Values	$v ::= n$	Integers
	$ s$	Strings
	$ id$	Identifiers
	$ \ell$	Pointers
Tables	$t ::= \{v_0 \mapsto v'_0, \dots, v_j \mapsto v'_j\}$	
Instructions	$i ::= \text{const } r, v$	Load constant v into r
	$ \text{mov } r_1, r_2$	Store r_2 in r_1
	$ \text{add } r_1, r_2, r_3$	Store $r_2 + r_3$ in r_1
	$ \text{sub } r_1, r_2, r_3$	Store $r_2 - r_3$ in r_1
	$ \text{mul } r_1, r_2, r_3$	Store $r_2 \times r_3$ in r_1
	$ \text{div } r_1, r_2, r_3$	Store $r_2 \div r_3$ in r_1
	$ \text{eq } r_1, r_2, r_3$	Store 1 in r_1 if $r_2 = r_3$, else store 0
	$ \text{lt } r_1, r_2, r_3$	Store 1 in r_1 if $r_2 < r_3$, else store 0
	$ \text{leq } r_1, r_2, r_3$	Store 1 in r_1 if $r_2 \leq r_3$, else store 0
	$ \text{is_int } r_1, r_2$	Store 1 in r_1 if r_2 is an integer, else store 0
	$ \text{is_str } r_1, r_2$	Store 1 in r_1 if r_2 is a string, else store 0
	$ \text{is_tab } r_1, r_2$	Store 1 in r_1 if r_2 is a table, else store 0
	$ \text{jmp } n$	Jump to offset n from current pc
	$ \text{if_zero } r, n$	Jump to offset n if $r = 0$, else skip
	$ \text{rd_glob } r, id$	Load global id into r
	$ \text{wr_glob } id, r$	Store r in global id
	$ \text{mk_tab } r$	Create new table and store pointer to it in r
	$ \text{rd_tab } r_1, r_2, r_3$	Store mapping for key r_3 in table r_2 into r_1
	$ \text{wr_tab } r_1, r_2, r_3$	Update table r_1 to map key r_2 to value r_3
	$ \text{has_tab } r_1, r_2, r_3$	Store 1 in r_1 if table r_2 has mapping for key r_3 , else store 0
$ \text{call } r, n_1, n_2$	Call function r , passing params in registers n_1 through n_2 (no args if $n_2 < n_1$)	
$ \text{ret } r$	Exit current function, returning r	
$ \text{halt } r$	Stops execution of the virtual machine	
Program	$P ::= \emptyset \mid P[id \mapsto F]$	Mapping from id to function body
Function body	$F ::= \emptyset \mid F[pc \mapsto i]$	Mapping from pc to instruction
Heap	$H ::= \emptyset \mid H[\ell \mapsto t] \mid H[id \mapsto v]$	Mapping from ℓ to t and id to v
Register file	$R ::= \emptyset \mid R[r \mapsto v]$	Mapping from register names to values
Stack	$S ::= \emptyset \mid (id, pc, R)::S$	

Figure 1: RubeVM syntax and runtime model.

The column “ $P(id)(pc)$ ” indicates what the instruction in the current function id at the current program counter pc must be for this rule to apply. Finally, the “Side conditions” column indicates other conditions that must be true for this rule to apply. If we wrote these rules correctly, at most one reduction rule should be possible in any state. It’s also possible no rule may apply, in which case the machine gets stuck.

Next we step through the semantics, from top to bottom.

$\text{const } r, v$ Yields a new state that is the same as the old state, except the program counter in the current stack frame has been incremented by one, and the register file has been updated so that r now maps to v . Note that this may either mean r is newly added to R , or it may be updated. Notice that in addition

	<u>Reduction</u>	<u>$P(id)(pc)$</u>	<u>Side conditions</u>
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r \mapsto v]) :: S \rangle$	const r, v	
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto v]) :: S \rangle$	mov r_1, r_2	$R(r_2) = v$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto n_1 + n_2]) :: S \rangle$	add r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto n_1 - n_2]) :: S \rangle$	sub r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto n_1 \times n_2]) :: S \rangle$	mul r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto n_1 \div n_2]) :: S \rangle$	div r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	eq r_1, r_2, r_3	$R(r_2) = v \wedge R(r_3) = v$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	eq r_1, r_2, r_3	$R(r_2) = v_1 \wedge R(r_3) = v_2 \wedge v_1 \neq v_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	lt r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2 \wedge n_1 < n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	lt r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2 \wedge n_1 \geq n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	leq r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2 \wedge n_1 \leq n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	leq r_1, r_2, r_3	$R(r_2) = n_1 \wedge R(r_3) = n_2 \wedge n_1 > n_2$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	is_int r_1, r_2	$\exists n. R(r_2) = n$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	is_int r_1, r_2	$\nexists n. R(r_2) = n$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	is_str r_1, r_2	$\exists s. R(r_2) = s$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	is_str r_1, r_2	$\nexists s. R(r_2) = s$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	is_tab r_1, r_2	$\exists \ell. R(r_2) = \ell$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	is_tab r_1, r_2	$\nexists \ell. R(r_2) = \ell$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1 + n, R) :: S \rangle$	jmp n	
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1 + n, R) :: S \rangle$	if_zero r, n	$R(r) = 0$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R) :: S \rangle$	if_zero r, n	$R(r) \neq 0$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r \mapsto v]) :: S \rangle$	rd_glob r, id_2	$H(id_2) = v$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H[id_2 \mapsto v], (id, pc + 1, R) :: S \rangle$	wr_glob id_2, r	$R(r) = v$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H[\ell \mapsto \{\}], (id, pc + 1, R[r \mapsto \ell]) :: S \rangle$	mk_tab r	$\ell \notin \text{dom}(H)$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto v']) :: S \rangle$	rd_tab r_1, r_2, r_3	$R(r_2) = \ell \wedge R(r_3) = v \wedge$ $H(\ell) = \{\dots, v \mapsto v', \dots\}$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H[\ell \mapsto H(\ell)[v \mapsto v']], (id, pc + 1, R) :: S \rangle$	wr_tab r_1, r_2, r_3	$R(r_1) = \ell \wedge R(r_2) = v \wedge R(r_3) = v'$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 1]) :: S \rangle$	has_tab r_1, r_2, r_3	$R(r_2) = \ell \wedge R(r_3) = v \wedge v \in \text{dom}(H(\ell))$
$P \vdash \langle H, (id, pc, R) :: S \rangle$	$\rightarrow \langle H, (id, pc + 1, R[r_1 \mapsto 0]) :: S \rangle$	has_tab r_1, r_2, r_3	$R(r_2) = \ell \wedge R(r_3) = v \wedge v \notin \text{dom}(H(\ell))$
$P \vdash \langle H, (id, pc, R_1) :: S \rangle$	$\rightarrow \langle H, (id_2, 0, R_2) :: (id, pc, R_1) :: S \rangle$	call r, n_1, n_2	$R_1(r) = id_2 \wedge$ $R_2 = [0 \mapsto R_1(n_1), \dots, n_2 - n_1 \mapsto R_1(n_2)]$
$P \vdash \langle H, (id, pc, R_1) :: (id_2, pc_2, R_2) :: S \rangle$	$\rightarrow \langle H, (id_2, pc_2 + 1, R_2[n_1 \mapsto v]) :: S \rangle$	ret r	$R_1(r) = v \wedge$ $P(id_2)(pc_2) = \text{call } r', n_1, n_2$

Figure 2: Operational Semantics

to letting registers be initialized with integers and strings, this also lets registers be initialized with a function pointer. Actually calling a function is a two-step process: The name is loaded into a register with `const`, and then that register is used as an operand to the `call` instruction (discussed below).

Also notice this rule technically lets locations be loaded into registers (like casting an integer to a pointer in C), but well-behaved programs will never use this behavior. In fact, the assembler we give you forbids this syntactically.

Almost all the remaining instructions increment the program counter by one, so we will leave that out of the subsequent discussion unless there is a special case.

- mov** r_1, r_2 Stores the value v associated with r_2 in the current register file in the register r_1 .
- add** r_1, r_2, r_3 Updates r_1 to contain the sum of the integers in r_2 and r_3 . Notice no behavior for this instruction is defined if r_2 or r_3 does not contain an integer.
- sub** , **mul** , **div** Similar to **add** .
- eq** r_1, r_2, r_3 Stores 1 in r_1 if r_2 and r_3 both contain the same value, or 0 if they contain different values. Notice this instruction can compare any values (integers, strings, identifiers, or locations).
- lt** , **leq** Similar to **eq** , but only allows integer comparisons.
- is_int** r_1, r_2 stores 1 in r_1 if r_2 contains an integer or 0 otherwise.
- is_str** r_1, r_2 stores 1 in r_1 if r_2 contains a string or 0 otherwise.
- is_tab** r_1, r_2 stores 1 in r_1 if r_2 contains a table or 0 otherwise.
- jmp** n Unconditionally updates the program counter to the previous program counter plus $n + 1$. Since n is an integer, it may be negative to jump to an earlier instruction in the function. (The extra plus one here seems to be standard on VMs and CPUs.) Jumping outside the pc range of the current function is an error.
- if_zero** r, n Adds $n + 1$ to the previous program counter if r contains zero, otherwise “falls through” to the next instruction by adding one to the program counter. Note that it is allowed for $R(r)$ to not be an integer—that case will always fall through, however.
- rd_glob** r, id_2 Stores the value v mapped to id_2 in the heap into r .
- wr_glob** id_2, r Stores the contents of r into the mapping for id_2 in the heap.
- mk_tab** r Generates a fresh location ℓ that is not already mapped in the heap; updates the heap to map ℓ to an empty table; and updates r to contain ℓ .
- rd_tab** r_1, r_2, r_3 Looks up the value v stored in r_3 in the table whose location ℓ is stored in r_2 , and stores the corresponding value v' into r_1 . Note this rule is undefined if v is not a key in the table.
- wr_tab** r_1, r_2, r_3 Updates the table whose location ℓ is indicated by r_1 so that the value v stored in r_2 is mapped to the value v' stored in r_3 . This may cause either a new mapping to be added if there was none previously, or the previous mapping to be replaced.
- has_tab** r_1, r_2, r_3 Writes 1 into r_1 if the table whose location ℓ is indicated by r_2 has a mapping for key r_3 . Otherwise writes 0 into r_1 .
- call** r, n_1, n_2 The register r must contain an id_2 that names a function in the program. Note there is no check here that id_2 must be a valid function name, but if it is not, the semantics will get stuck in the next step. A new register file R_2 is created and initialized with the contents of the current stack frame’s registers n_1 through n_2 , offset so $R(n_1)$ is in register 0, $R(n_1 + 1)$ is in register 1, etc. If $n_2 < n_1$, the new register file R_2 is empty by definition. Then a new stack frame is pushed onto the stack, with the callee function name id_2 , the initial program counter 0, and the initial register file R_2 . Thus, in the next step execution will continue inside of id_2 .
- ret** r Pops the current stack frame off the stack, incrementing the caller stack frame’s program counter by one. Also copies the value v in register r to the register n_1 in the caller’s stack frame, where n_1 is the initial register indicated in the call instruction that created the current stack frame.
- halt** r Stops execution of the virtual machine (hence there is no semantics rule for it). This will turn out to be useful for your compiler. In the implementation, the contents of the register passed **halt** is printed (but this is not part of the formal semantics).

```

type reg = [ 'L.Reg of int ]
type value = [ 'L.Int of int | 'L.Str of string | 'L.Id of string | 'L.Loc of int ]
type id = [ 'L.Id of string ]

type instr =
  | L.const of reg * value (* dst, src *)
  | L.mov of reg * reg (* dst, src *)
  | L.add of reg * reg * reg (* dst, src1, src2 *)
  | L.sub of reg * reg * reg (* dst, src1, src2 *)
  | L.mul of reg * reg * reg (* dst, src1, src2 *)
  | L.div of reg * reg * reg (* dst, src1, src2 *)
  | L.eq of reg * reg * reg (* dst, src1, src2 *)
  | L.lt of reg * reg * reg (* dst, src1, src2 *)
  | L.leq of reg * reg * reg (* dst, src1, src2 *)
  | L.is_int of reg * reg (* dst, src *)
  | L.is_str of reg * reg (* dst, src *)
  | L.is_tab of reg * reg (* dst, src *)
  | L.jump of int (* offset *)
  | L.if_zero of reg * int (* src, offset *)
  | L.rd_glob of reg * id (* dst, src *)
  | L.wr_glob of id * reg (* dst, src *)
  | L.mk_tab of reg (* dst *)
  | L.rd_tab of reg * reg * reg (* dst, tab, key *)
  | L.wr_tab of reg * reg * reg (* tab, key, value *)
  | L.has_tab of reg * reg * reg (* dst, tab, key *)
  | L.call of reg * int * int (* dst, first, last *)
  | L.ret of reg (* src *)

type fn = instr array
type prog = (string, fn) Hashtbl.t
type heap = (value, value) Hashtbl.t
type regs = (int, value) Hashtbl.t
type stack = (string * int * regs) list
type config = heap * stack

```

Figure 3: Operational Semantics API in OCaml

In addition, if any instruction tries to access a register that doesn't exist (i.e., that has not yet been written), that is undefined and should cause the virtual machine to raise an exception.

Finally, to run a program from scratch, we start executing the `main` function of the program, with the program counter at 0, an empty register file, and an empty heap. Thus, the initial configuration for program P is

$$\langle \emptyset, (\text{main}, 0, \emptyset) :: \emptyset \rangle$$

The program exits when `main` returns. That is, when a program reaches a configuration

$$\langle H, (\text{main}, pc, R) :: \emptyset \rangle$$

and $P(\text{main})(pc) = \text{ret } r$. The value returned from the program is $R(r)$.

(Note it is an error if `main`, or any other function, runs out of instructions before returning.)

4 Core Implementation in OCaml

Your job is to implement the operational semantics in OCaml. Figure 3 shows a translation of (most of) Figure 1 into OCaml. Registers, values, and identifiers are given by types `reg`, `value`, and `id`. Notice there is

overlap between `value` and `id`. Also notice we have arbitrarily decided to represent locations as integers, and that we have not given you a representation of tables—that choice is up to you. When you do add tables to the set of values, *do not* modify the other constructors. For debugging purposes, you will probably want to add the capability of printing out table values; you’ll want to modify `Disassembler.value` to do that (just follow the pattern).

The middle part of the figure shows the instructions. Finally, the bottom part of the figure defines types for a function body (an array of instructions), the program (an associative list mapping function names to their bodies), the heap (a mapping from values to values), registers (a mapping from register numbers to values), the stack (a list of stack frames containing the function name, program counter, and registers), and the configuration (a pair of heap and stack).

You must implement the following functions in the file `rubevm.ml` (the first couple are just warmups, and you may or may not use them in the last two functions):

- `max_regs p f : prog → string → int` takes a program and the name of a function in the program, and returns the highest numbered register accessed within the body of the function.
- `current_inst p (h, s) : prog → config → instr` takes a program and a configuration, and returns the next instruction to be executed (i.e., the instruction that `run_inst` would execute if it were called).
- `run_inst p (h, s) : prog → config → config` takes a program and a current configuration, and executes the next instruction, yielding a new configuration. This function should raise an exception (of your choice) if the program attempts to do anything that is erroneous in the semantics, i.e., if there is no reduction rule for that case. Some example erroneous cases are adding two strings, “calling” an integer as a function, reading a non-existent key in a table, etc. This function should also raise an exception if it is asked to execute a `halt` instruction.
- `run_prog p : prog → prog_ret` runs a program, starting in the initial configuration (as defined above), until either `main` returns or the program halts. In the first case, the function should return `Reg v` where v is the value returned by `main`. In the second case, the function should return `Halt v` where v is the contents of the register passed to `halt`.

To help make it a bit easier to write RubeVM programs, we’ve provided you with an assembler. Once you have a working prototype, you can run a program from scratch as `./main.byte file`, where `file` is the name of a file containing rubevm assembly code. For an example of a program in this format, see `r1.rubevm`, which doesn’t do anything interesting but does include every possible instruction. After a program is executed, `rubevm` prints the value returned from running the program.

5 Foreign Functions

The interpreter above is theoretically useful, but programs are still limited in practical terms. For example, there’s no way to do I/O within the language. To solve this problem, you must extend your interpreter to perform special handling of certain *foreign function calls*. In RubeVM, foreign function calls look just like normal calls (they are invoked with the `call` instruction as usual), but the interpreter intercepts them and performs special operations that are written in OCaml rather than in RubeVM code. You must implement the following foreign functions:

- `print_string(s)`. Prints string `s` to standard output. This function should print `s` as-is, e.g., it does not print a newline after the string. Returns the string that was printed.
- `print_int(n)`. Prints integer `n` to standard output (again without anything else, e.g., without adding a newline). Returns the integer (not its string representation) that was printed.

- `to_s(v)` . If `v` is a string, the string is returned as-is. If `v` is an integer, converts the integer to a string and returns that string. If `v` is an identifier, returns the string `Function< id >` , where `id` is the name of the function. Note there should be no extra spaces added, e.g., if `foo` is the `id`, then `to_s` would return `Function<foo>` and not `Function <foo>` or `Function< foo >` etc.
- `to_i(v)` . If `v` is an integer, returns the integer as-is. If `v` is a string, converts the string to an integer and returns that. Your implementation may do whatever you choose if `v` is a string that contains characters other than digits and a possible initial minus sign.
- `concat(s1, s2)` . Returns the concatenation of strings `s1` and `s2` .
- `length(s)` . Returns the length of string `s` .
- `size(t)` . Returns the number of key-value pairs in table `t` . (Note `t` is really a “pointer to a table” in the internal RubeVM representation.)
- `iter(t, f, x)` . Calls function `f` (which in the semantics will be an `id`) once on each key–value pair in table `t` , passing `x` as a third argument. For example, if `k` maps to `v` in the table, the function `f` will be called as `f(k,v,x)` . The iteration order is unspecified (any order is fine). The `iter` function always returns `0` .

In all cases, your implementation must abort with an exception (whichever exception you prefer) if a foreign function is called with the wrong types of arguments. Since there are quite a few foreign functions to implement, it would probably be a good idea to design your interpreter to make it easy to add new foreign functions.

Also, if the bytecode defines a function with the same name as a foreign function, then the bytecode function takes precedence. For example, if the program defines `to_i` , then calling `to_i` should jump to the program’s function, not the foreign function.

Academic Integrity

The Campus Senate has adopted a policy asking students to include the following statement on each assignment in every course: “I pledge on my honor that I have not given or received any unauthorized assistance on this assignment.” Consequently your program is requested to contain this pledge in a comment near the top.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus—please review it at this time.