Project 5

Due ~~Tuesday, November 20~~ Friday, November 23, 2018, 11:59:59pm
No Late Submissions

Introduction

In this project, you will write a compiler for a programming language called Rube, which is a small object-oriented programming language with a syntax similar to Ruby. Your compiler will translate Rube source code into RubeVM byte code (from Project 3).

Of course, you already wrote a compiler to RubeVM in Project 4, for the Simpl language. The key difference between Rube and Simpl is that in Rube, everything is an object, and Rube has methods instead of C-like functions. In compiling Simpl, most of the language semantics mapped in a pretty straightforward way to RubeVM. In contrast, you'll have to work harder in this project to encode objects into RubeVM. (Hint: Use tables!)

Project Structure

The project skeleton code is divided up into the following files:

`Makefile`	Makefile
`lexer.mll`	Rube lexer
`parser.mly`	Rube parser
`ast.mli`	Abstract syntax tree type
`disassembler.ml`	RubeVM bytecode disassembler
`main.ml`	The main compiler logic
`r{1--4}.ru`	Small sample Rube programs

You will only change `main.ml`; you should not edit any of the other files. The file `main.ml` includes code to run the parser and then compile the input file to `rubec.out`. Right now, the generated output file always contains code that prints `Fix me!`:

```
$ make
$ ./main.byte r1.ru
$ rubevm rubec.out
Fix me!
$ ...
```

You'll need to modify the implementation of `compile_prog` in `main.ml` to perform actual compilation.

You can use your own `rubevm` from project 3, or you can use ours, which is still in

`/afs/glue.umd.edu/class/fall2018/cmsc/430/0201/public/bin/rubevm`

As you've no doubt noticed, this project writeup is fairly similar to the writuep for Simpl. However, be careful to read through the details so you understand how the languages differ. We'll try to emphasize the differences as we describe the language.

Here's the first difference: A Simpl program must have a `main()` function that's called to start the program. In contrast, in Rube, when the program is launched, it evaluates a top-level expression that lives outside of any particular method or class. (Thus, to translate Rube into RubeVM bytecode, you'll have to put the code of the top level expression into a RubeVM function `main`.)

When a Rube program finishes, the result of the top-level expression is converted to a string and then printed; more details below.

P	::=	$C^* \ E$	Rube program
C	::=	class $id < id$ begin M^* end	Class definition
M	::=	def id $(id, \ldots, id)$ E end	Method definition
E	::=	v	Values
	|	`self`	Self
	|	id	Local variable
	|	$@id$	Field
	|	`if` E `then` E `else` E `end`	Conditional
	|	`while` E `do` E `end`	While loop
	|	E; E	Sequencing
	|	$id = E$	Local variable write
	|	$@id = E$	Field write
	|	`new` $id(E, \ldots, E)$	Object creation
	|	E `instanceof` id	Object class test
	|	$E.id(E, \ldots, E)$	Method invocation
v	::=	n	Integers
	|	`nil`	Nil
	|	"*str*"	String

Figure 1: Rube syntax

Rube Syntax

The formal syntax for Rube programs is shown in Figure 1. A Rube program P consists of a sequence of class definitions followed by a single expression. To execute a program, we evaluate the expression given the set of class definitions. Every class has some superclass; there is a built-in class `Object`, and the superclass of `Object` is itself. In Rube, methods are inherited from superclasses and may be overridden; there is no overloading in Rube. In Rube, as in Ruby, everything is an object, including integers n, the null value `nil` (an instance of class `Bot`), and strings `"str"`; these three expressions are the values of the language. Local variables are identifiers *id*, which are made up of upper and lower case letters or symbols (including `+`, `-`, `*`, `/`, `_`, `!`, and `?`). The special identifier `self` refers to the current object. An identifier with an @ in front of it refers to a field. Rube also includes the conditional form `if`, which evaluates to the true branch if the guard evaluates to anything except `nil`, and the false branch otherwise. Rube includes sequencing of expressions, assignments to local variables and fields, a run-time test of the class of an object, and method invocation with the usual syntax.

Abstract syntax trees We've provided you with a parser that translates Rube source code into an abstract syntax tree. Figure 2 shows the OCaml AST data types.

The first four constructors should be self-explanatory. The expression `ELocRd s` represents (reading) the local variable `s`, and the expression `ELocWr(s,e)` corresponds to `s=e`, where `s` is a local variable. The expressions `EFldRd s` and `EFldWr(s,e)` represent reading and writing a field `s`. Because of the way the parser works, `s` will always begin with an @. The expression `EIf(e1,e2,e3)` corresponds to `if e1 then e2 else e3 end`. Expression `EWhile(e1, e2)` executes body `e2` as long as guard `e1` is non-`nil`, and the whole expression evaluates to `nil`. The expression `ESeq(e1,e2)` corresponds to `e1;e2`. The expression `EInstanceOf(e,s)` corresponds to `e instanceof s`. The expression `ENew(s,el)` corresponds to calling `new` on class `s` with the arguments given in `el`. The expression `EInvoke(e,s,el)` corresponds to calling method `s` of object `e` with the arguments given in `el`. (For both `ENew` and `EInvoke`, the arguments are in the same order in the list as in the program text, and may be empty.)

A method `meth` is a record containing the method name, arguments, and method body. A class `cls` is a record containing the class name, superclass, and methods. Finally, a program `rube_prog` is a record containing the list of classes and the top-level expression.

```
type expr =
    EInt of int
  | ENil
  | ESelf
  | EString of string
  | ELocRd of string      (* Read local variable *)
  | ELocWr of string * expr   (* Write local var *)
  | EFldRd of string      (* Read field *)
  | EFldWr of string * expr   (* Write field *)
  | EIf of expr * expr * expr
  | EWhile of expr * expr
  | ESeq of expr * expr
  | ENew of string * (expr list)
  | EInstanceOf of expr * string
  | EInvoke of expr * string * (expr list)
```

```
(* meth name * arg name list * method body *)
type meth = { meth_name : string;
              meth_args : string list ;
              meth_body : expr }

(* class name * superclass name * methods *)
type cls = { cls_name : string ;
             cls_super : string ;
             cls_meths : meth list }

(* classes * top-level expression *)
type rube_prog = { prog_clss : cls list ;
                   prog_main : expr }
```

Figure 2: Abstract syntax tree for Rube with type annotations

Rube Semantics

Figure 3 gives the formal, big-step operational semantics for evaluating Rube expressions (we will discuss relating these rules to compilation next). These rules show reductions of the form $P \vdash \langle A, H, E \rangle \rightarrow \langle A', H', v \rangle$, meaning that in program P, and with local variables environment A and heap H, expression E reduces to the value v, producing a new local variable assignment A' and a new heap H'. As usual, we extend the set of values v with locations ℓ, which are pointers. The program P is there so we can look up classes and methods. We've labeled the rules so we can refer to them in the discussion:

- The rules Int, Nil, and Str all say that an integer, nil, or string evaluate to the expected value, in any environment and heap, and returning the same environment and heap. In the syntax of Rube, strings begin and end with double quotes `"`, and may not contain double quotes inside them.

- Like Ruby, a local variable can be created by writing to it. The rule Id/Self says that the identifier *id* evaluates to whatever value it has in the environment A. If *id* is not bound in the environment, then this rule doesn't apply—and hence your compiled code would signal an error. Reading a local variable or `self` does not change the local variable environment or the heap. This is the same as Simpl, except that now `self` is always a valid local variable name.

- The rule Field-R says that when a field is accessed, we look up the current object `self`, which should be a location in the heap ℓ. Then we look up that location in the heap, which should be an object that contains some fields id_i. If one of those fields is the one we're looking for, we return that field's value. On the other hand, if we're trying to read field *id*, and there is no such field in `self`, then rule Field-Nil applies and returns the value `nil`. (Notice the difference between local variables and fields.) Also notice that like Ruby, only fields of `self` are accessible, and it is impossible to access a field of another object.

- The rules If-T and If-F say that to evaluate an if-then-else expression, we evaluate the guard, and depending on whether it evaluates to a non-nil value or a nil value, we evaluate the then or else branch and return that. Notice the order of evaluation here: we evaluate the guard E_1, which produces a configuration $\langle A_1, H_1, v_1 \rangle$, and then we evaluate the then or else branch with that local variable environment and heap.

- The rules While-T and While-F evaluate a while loop, using the same rule as If-* that nil is false and any other value is true. A while loop always evaluates to nil.

Int

$$\frac{}{P \vdash \langle A, H, n \rangle \to \langle A, H, \mathsf{n} \rangle}$$

Nil

$$\frac{}{P \vdash \langle A, H, \texttt{nil} \rangle \to \langle A, H, \mathsf{nil} \rangle}$$

Str

$$\frac{}{P \vdash \langle A, H, \texttt{"}str\texttt{"} \rangle \to \langle A, H, \text{``}\mathsf{str}\text{''} \rangle}$$

Id/Self

$$\frac{id \in dom(A)}{P \vdash \langle A, H, id \rangle \to \langle A, H, A(id) \rangle}$$

Field-R

$$\frac{A(\texttt{self}) = \ell \qquad H(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_n : v_n]}{P \vdash \langle A, H, @id_i \rangle \to \langle A, H, v_i \rangle}$$

Field-Nil

$$\frac{A(\texttt{self}) = \ell \qquad H(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_n : v_n] \qquad @id \notin \{@id_1, \ldots, @id_n\}}{P \vdash \langle A, H, @id \rangle \to \langle A, H, \texttt{nil} \rangle}$$

If-T

$$\frac{P \vdash \langle A, H, E_1 \rangle \to \langle A_1, H_1, v_1 \rangle \qquad v_1 \neq \texttt{nil} \qquad P \vdash \langle A_1, H_1, E_2 \rangle \to \langle A_2, H_2, v_2 \rangle}{P \vdash \langle A, H, \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ end} \rangle \to \langle A_2, H_2, v_2 \rangle}$$

If-F

$$\frac{P \vdash \langle A, H, E_1 \rangle \to \langle A_1, H_1, \mathsf{nil} \rangle \qquad P \vdash \langle A_1, H_1, E_3 \rangle \to \langle A_3, H_3, v_3 \rangle}{P \vdash \langle A, H, \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ end} \rangle \to \langle A_3, H_3, v_3 \rangle}$$

While-T

$$\frac{P \vdash \langle A, H, E_1 \rangle \to \langle A_1, H_1, v_1 \rangle \qquad v_1 \neq \mathsf{nil} \qquad P \vdash \langle A_1, H_1, E_2; \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rangle \to \langle A_2, H_2, v_2 \rangle}{P \vdash \langle A, H, \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rangle \to \langle A_2, H_2, v_2 \rangle}$$

While-F

$$\frac{P \vdash \langle A, H, E_1 \rangle \to \langle A_1, H_1, \mathsf{nil} \rangle}{P \vdash \langle A, H, \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rangle \to \langle A_1, H_1, \mathsf{nil} \rangle}$$

Seq

$$\frac{P \vdash \langle A, H, E_1 \rangle \to \langle A_1, H_1, v_1 \rangle \qquad P \vdash \langle A_1, H_1, E_2 \rangle \to \langle A_2, H_2, v_2 \rangle}{P \vdash \langle A, H, (E_1;\ E_2) \rangle \to \langle A_2, H_2, v_2 \rangle}$$

Id-W

$$\frac{P \vdash \langle A, H, E \rangle \to \langle A', H', v \rangle \qquad id \neq \texttt{self} \qquad A'' = A'[id \mapsto v]}{P \vdash \langle A, H, id = E \rangle \to \langle A'', H', v \rangle}$$

Field-W

$$\frac{P \vdash \langle A, H, E \rangle \to \langle A', H', v \rangle \qquad A(\texttt{self}) = \ell \qquad H'(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = F] \qquad H'' = H'[\ell \mapsto [\mathsf{class} = id_s;\ \mathsf{fields} = F[@id_f : v]]]}{P \vdash \langle A, H, @id_f = E \rangle \to \langle A', H'', v \rangle}$$

InstanceOf-T

$$\frac{P \vdash \langle A, H, E \rangle \to \langle A', H', \ell \rangle \qquad H'(\ell)(\texttt{class}) = id}{P \vdash \langle A, H, E \texttt{ instanceof } id \rangle \to \langle A', H', 1 \rangle}$$

InstanceOf-F

$$\frac{P \vdash \langle A, H, E \rangle \to \langle A', H', \ell \rangle \qquad H'(\ell)(\texttt{class}) \neq id}{P \vdash \langle A, H, E \texttt{ instanceof } id \rangle \to \langle A', H', \texttt{nil} \rangle}$$

New-NoInit

$$\frac{id \in P \qquad id \neq \texttt{Bot} \qquad \ell \notin dom(H) \qquad H' = H[\ell \mapsto [\mathsf{class} = id;\ \mathsf{fields} = \emptyset]] \qquad \texttt{lookup\_meth}\ P\ id\ \texttt{initialize} = \emptyset}{P \vdash \langle A, H, \texttt{new } id() \rangle \to \langle A, H', \ell \rangle}$$

New-Init

$$\frac{\begin{array}{c} id \in P \qquad id \neq \texttt{Bot} \qquad A_0 = A \qquad \ell \notin dom(H) \\ H_0 = H[\ell \mapsto [\mathsf{class} = id;\ \mathsf{fields} = \emptyset]] \qquad P \vdash \langle A_0, H_0, E_1 \rangle \to \langle A_1, H_1, v_1 \rangle \qquad \ldots \qquad P \vdash \langle A_{n-1}, H_{n-1}, E_n \rangle \to \langle A_n, H_n, v_n \rangle \\ \texttt{lookup\_meth}\ P\ id\ \texttt{initialize} = (\text{def } \texttt{initialize}\ (id_1, \ldots, id_n)\ E \text{ end}) \qquad k = n \\ A' = \texttt{self} : \ell, id_1 : v_1, \ldots, id_k : v_k \qquad P \vdash \langle A', H_n, E \rangle \to \langle A'', H'', v \rangle \end{array}}{P \vdash \langle A, H, \texttt{new } id(E_1, \ldots, E_n) \rangle \to \langle A_n, H'', \ell \rangle}$$

Invoke

$$\frac{\begin{array}{c} P \vdash \langle A, H, E_0 \rangle \to \langle A_0, H_0, \ell \rangle \\ H_0(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = \ldots] \qquad P \vdash \langle A_0, H_0, E_1 \rangle \to \langle A_1, H_1, v_1 \rangle \qquad \ldots \qquad P \vdash \langle A_{n-1}, H_{n-1}, E_n \rangle \to \langle A_n, H_n, v_n \rangle \\ \texttt{lookup\_meth}\ P\ id_s\ id_m = (\text{def } id_m\ (id_1, \ldots, id_n)\ E \text{ end}) \qquad k = n \\ A' = \texttt{self} : \ell, id_1 : v_1, \ldots, id_k : v_k \qquad P \vdash \langle A', H_n, E \rangle \to \langle A'', H'', v \rangle \end{array}}{P \vdash \langle A, H, E_0.id_m(E_1, \ldots, E_n) \rangle \to \langle A_n, H'', v \rangle}$$

Program

$$\frac{P = C^*\ E \qquad A = \texttt{self} : \ell \qquad H = \ell : [\mathsf{class} = \texttt{Object};\ \mathsf{fields} = \emptyset] \qquad P \vdash \langle A, H, E \rangle \to \langle A', H', v \rangle}{\vdash P \Rightarrow v}$$

Figure 3: Rube Operational Semantics for Expressions

- The rule Seq says that to evaluate $E_1;\ E_2$, we evaluate E_1 and then evaluate E_2, whose value we return. This is the same as Simpl.

- The rule Id-W says that to write to a local variable *id*, we evaluate the E to a value v, and we return a configuration with a new environment A'' that is the same as A', except now *id* is bound to v. This is the same as Simpl. Notice that our semantics forbid updating the local variable `self` (since there's no good reason to do that, and if we allowed that, it would let users change fields of other objects). The parser forbids this syntactically.

- The rule Field-W is similar; notice that we can create new fields by writing to them. We return a new heap H'' that is the same as the heap H' after evaluating E, except we update location ℓ to contain an object whose id_f field is v. Here, F stands for the original set of fields, and $F[id_f : v]$ stands for F except id_f is now mapped to v; this notation either updates the previous mapping (if one existed) or adds a new mapping to F. In both cases, assignment returns the value that was assigned. (This is in contrast to OCaml, where assignment returns the unit value.)

- The rules InstanceOf-T and InstanceOf-F say that to evaluate an instanceof test, we evaluate the expression E and return 1 if the resulting object's class exactly matches *id* and `nil` otherwise. (Note that this is slightly different than Java, where `instanceof` returns true if the left-hand side is an instance of either the right-hand side or a subclass of the right-hand side.)

- The rules New-NoInit and New-Init create a new instance of a class *id* via the `new` keyword. The first rule describes the case where the class definition does not contain an `initialize` method. The second rule is for classes that define a custom initialization method, which can take zero or more arguments. According to both rules, making a new instance of `Bot`, the class of `nil`, is not allowed. First we check to make sure that *id* is a class that's actually defined in the program (we write this check as $id \in P$). Then we find a fresh location ℓ that is not already used in the heap.

  In New-NoInit, we return the location ℓ, along with a new heap H' that is the same as heap H, except ℓ maps to a fresh instance of *id* with no initialized fields.

  In New-Init, we evaluate the arguments to `new` (as described below in method invocation) before evaluating the body of the class's `initialize` method. As with New-NoInit, we return the location ℓ, along with a new heap, but the heap in this case is the result of evaluating both the arguments and the initializer's body.

- The Invoke rule is for method invocation. We begin by evaluating the receiver E_0 to location ℓ, which must map to an object in the heap. We then evaluate the arguments E_1 through E_n, in order from 1 to n, to produce values. (Notice here the "threading" of the location variable environment and heap through the evaluation of E_1 through E_n.) Next, we use the *lookup* function to find the correct method.

  Once we find a method `def` $id_m(id_1, \ldots, id_k)$ with the right name, id_m, we ensure that it takes the right number of arguments—if it doesn't, again we would signal an error in the implementation (though this is not one of the errors we will test; see below). Finally, we make a new environment A' in which `self` is bound to the receiver object ℓ, and each of the formal arguments id_i is bound to the actual arguments v_i. Recall that in the environment, shadowing is left-to-right, so that if *id* appears twice in the environment, it is considered bound to the leftmost occurrence. We evaluate the body of the method in this new environment A', and whatever is returned is the value of the method invocation.

  Notice that Rube has no nested scopes. Thus when you call a method, the environment A' you evaluate the method body in is not connected to the environment A from the caller. This makes these semantics simpler than a language with closures.

- Finally, rule Program explains how to evaluate a Rube program. We evaluate the expression E of the program, starting in an environment A where `self` is the only variable in scope, and it is bound to a location ℓ containing an object that is an instance of `Object` and contains no fields.

Class	Method type	
Object	equal? : (Object) → Object	equality check
	to_s : () → String	convert to string
	print : () → Bot	print to standard out
String	+ : (String) → String	string concatenation
	length : () → Integer	string length
Integer	+ : (Integer) → Integer	addition
	- : (Integer) → Integer	subtraction
	* : (Integer) → Integer	multiplication
	/ : (Integer) → Integer	division
Map	find: (Object) -> Object	return value mapped to key
	insert : (Object, Object) -> Bot	add key-value mapping
	has : (Object) -> Object	check whether key is in map
	iter : (Object) -> Bot	iterate through mapping
Bot		*No additional methods*

Figure 4: Built-in objects and methods

In your implementation, you should call `to_s` on v and print out the resulting string when the program exits (by returning it from main in the output RubeVM program).

Errors

For grading purposes, here are particular ways you should handle certain errors:

- If a program tries to call a method that does not exist, your implementation should print the string `halt: No such method` and then exit immediately.
- If a program tries to instantiate `Bot`, the class of `nil`, your implementation should print `halt: Cannot instantiate Bot` and then exit immediately
- For any error not on this list, your implementation may report the error in whatever way you prefer; we will not test these cases.

As with Simpl, you should report all errors at run time rather than at compile time, even if you can detect them at compile time. (Note there is only one space after `halt:` above despite the way it looks.)

Built-in methods

Notice that one thing Rube lacks is semantics for useful things like basic arithmetic on integers. That is because everything in Rube is an object, and hence arithmetic is encoded just as methods with certain particular names. Rube also includes several other built-in methods and classes.Your compiler should behave as if the built-in classes exist at the start of the program, with the appropriate methods defined. The type signatures for the built-in methods are given in Figure 4, and their semantics is as follows:

- The `equal?` method of `Object` should compare the argument to `self` using pointer equality, and should return `nil` if the two objects are not equal, and the `Integer 1` if they are equal. However, this method should be overridden for `String` and `Integer` to do a deep equality test of the string and integer values, respectively, returning `1` for equality and `nil` for disequality. In these last two cases, your methods should always return `nil` if the object `self` is being compared to is not a `String` or `Integer`, respectively. The `equal?` method of `Map` is simply inherited from `Object`, so it should do pointer equality.

- The `to_s` method for an arbitrary `Object` can behave however you like; we won't test this. This method should be overridden for `String` to return `self`; for `Integer` to return a `String` containing the textual representation of the integer; and for `nil` to return the `String` containing the three letters `nil`.
- The `print` method prints an object to standard out, as-is. (E.g., do not add any additional newlines.) For strings and integers, the output should be clear. For `nil`, the output should be the three letters `nil`. For `Object`, the output can be whatever you like; we won't test this. Your `print` method should return `nil`.
- The `+` method on `String`s performs string concatenation. Your method should halt execution with an error if passed a non-`String` as an argument.
- The `length` method on `String`s returns the length of the string.
- The `+`, `-`, `*`, and `/` methods perform integer arithmetic. Your method should halt execution with an error if passed a non-`Integer` as an argument.
- The `find(k)` method of `Map` takes a key `k` and returns the value mapped to `k`. Execution should halt if `k` is not in the map. The `find` method should compare keys using RubeVM's `eq` instruction.
- The `insert(k,v)` method of `Map` takes a key `k` and a value `v` and adds a mapping from `k` to `v` to the map. The `insert` method should compare keys using RubeVM's `eq` instruction.
- The `has(k)` method of `Map` takes a key `k` and returns `nil` if the key is not in the map, or some non-`nil` value (of your choice) if the key is in the map. The `has` method should compare keys using RubeVM's `eq` instruction.
- The `iter(o)` method of `Map` takes an object `o` that is assumed to have a two-argument method `call`. It then iteratively calls `o.call(k,v)` for each mapping from `k` to `v` in the table. The `iter` method should return `nil`.

Finally, any built-in class can be instantiated with `new`, except for `Bot`. We won't test the behavior of `new Object()`. Calling `new String()` should return an empty string. Calling `new Integer()` should return 0. Calling `new Map()` should return an empty map.

Compilation

As mentioned in the introduction, your compiled program should begin by executing the top-level expression, which will yield a value v. Your compiled program should then print v out by calling v`.to_s` and printing the resulting string to standard output.

The key challenge in this project is figuring out how to encode Rube objects using RubeVM concepts (recall RubeVM does not have a primitive notion of objects). The choice is up to you, but we have the following suggestions on how to set things up. (Note that we will <u>only</u> test your compiler by compiling Rube code and seeing what running it under RubeVM prints out, so you are certainly free to disregard these suggestions.)

- You can represent an object as a table `"#vtable" = vt, f1 = ..., f2 = ..., ...`, where `vt` is a reference to the virtual method table for the object, and the `fi` are the fields of the object. We've chosen `#vtable` as the vtable key in the hash because no Rube field name can begin with `#`.
- In Rube, accessing fields that have not been written is not an error, but instead it returns `nil`. However, in RubeVM it is a (fatal) error to try to read from an undefined key. So, you'll need to first check whether a key exists in a table before trying to read it.

- You'll create vtables by first creating one RubeVM function for each method in the Rube program. Then you'll assemble those into vtables, which should include each class' methods as well as all inherited methods that are not overridden. You can then store the vtables in global variables of whatever names you choose, which you'll then use to initialize the objects at a call to `new`. (If you wanted to make this even cleaner, you could make a global `classtable` table that mapped class names to their vtables.) Don't forget that each RubeVM function corresponding to a Rube method will take a `self` argument.
- Don't worry about shadowing among parameters names, locals, and `self`. We won't test that.
- As with Simpl, local variables will need to be assigned to registers.
- To handle built-in methods, you'll want to generate a standard set of vtables for `Object`, `Integer`, `String`, `Map`, and `Bot`, containing the built-in methods. Thus, integers, strings, and maps will be pointers to objects, i.e., tables. You could use a field name such as `#contents` to store the actual primitive RubeVM integer, string, or table, which will then be manipulated specially by your implementations of the built-in methods. This is terribly inefficient, but it's ok for this project.
- Yes, that's right, a `Map` will be represented by two tables: a table representing the object itself, and a table storing the contents of the map.
- It will be convenient to add a small runtime system (i.e., some utility functions your generated code can call as needed) to your compiled program. Rather than manually create the bytecode for that runtime system, it may be easier to write RubeVM source code for it; compile it into bytecode; and then use the disassembler to retrieve the right bytecode instructions.
- It's up to you how to represent `nil`, but one way to do it is to create an empty table when the program launches, and use its location as `nil`.
- You'll need to do a bit of work to use the RubeVM `ifzero` instruction to implement the Rube `if` method, since `ifzero` branches on based on whether a value is 0 or 1, whereas `if` branches based on whether the guard is `nil` or not.

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.