

Project 4

Due November 14, 2008

11:59:59pm

Updates

- 11/3. Renamed `class` method to `class_of`, since methods can't have same name as keywords.

Introduction

In this project, you will write an interpreter for a new language called Rube, which is a simple object-oriented programming language with a syntax similar to Ruby. To get started, we've supplied you with a parser for translating Rube source code into abstract syntax trees. Your job is to write a series of OCaml functions for executing programs in AST form.

This is a long write-up mostly because we need to describe precisely what Rube programs do, and because we explain this both in English and in math (as operational semantics). But the actual amount of code you'll need to write for the basic interpreter is remarkably small—in fact, if you understand the operational semantics, they essentially tell you exactly what to write in OCaml for your interpreter.

What to Submit

We've left a `p4.tar.gz` file in the usual place. This time, this directory contains several files:

<code>.submit</code>	The usual submit file
<code>Makefile</code>	A file for building this project
<code>OCamlMakefile</code>	A helper for <code>Makefile</code>
<code>ast.mli</code>	A signature describing Rube abstract syntax trees
<code>lexer.mll</code>	A lexer for tokenizing Rube programs
<code>parser.mly</code>	A parser for parsing Rube programs (uses the lexer)
<code>rube.ml</code>	The file you need to edit
<code>ri.ru</code>	Some sample Rube programs

To build the project, `cd` into the directory and type `gmake`. This will build an executable `rube` that simply reads in a Rube program from standard input, parses it into an abstract syntax tree, *unparses* the AST to standard output, and then evaluates the program and prints the result. (“Unparsing” is the process of going from an AST to a textual representation.) For example, if you build `rube` and then type `./rube < r1.ru`, you should see

```
% ./rube < r1.ru
"Hello, world!\n".print()
```

Evaluates to:

Implement me!

The only file you need to submit is `rube.ml`. You may not modify any of the other files; we will overwrite the other files with fresh copies when we grade your projects. For grading purposes, we will not use the parser—we will invoke the functions that you write in `rube.ml` directly from within OCaml. However, you will most likely find that being able to parse source programs will make it easier for you to test your interpreter. *Warning:* If you thought OCaml's parser error messages were bad, you haven't seen anything yet! The Rube parser is bare bones, and contains no error recovery whatsoever. So if you make a typo in a Rube program, you'll just get a parse error message. If you get really stuck with this, just build up your Rube programs inside of OCaml, rather than using the parser.

<i>prog</i>	::=	<i>expr</i>	Rube program
<i>expr</i>	::=	<i>n</i>	Integers
		nil	Nil
		" <i>str</i> "	String
		<i>id</i>	Local variable
		@ <i>id</i>	Field
		if <i>expr</i> then <i>expr</i> else <i>expr</i> end	Conditional
		<i>expr</i> ; <i>expr</i>	Sequencing
		<i>id</i> = <i>expr</i>	Local variable write
		@ <i>id</i> = <i>expr</i>	Field write
		<i>expr</i> . <i>id</i> (<i>expr</i> , ..., <i>expr</i>)	Method invocation
		class <i>method</i> ... <i>method</i> end	Anonymous class
<i>method</i>	::=	def <i>id</i> (<i>id</i> , ..., <i>id</i>) <i>expr</i> end	

Figure 1: Rube syntax

Rube Syntax

The formal syntax for Rube programs is shown in Figure 1. A Rube program *prog* is made up of a single (but possibly quite complicated) expression. To execute a program, we evaluate the expression to yield the result of the program. For example, the program in `r1.ru` calls the `print` method of the string `Hello, world!\n`, which causes the string to be displayed and then returns `nil`.

In Rube, as in Ruby, everything is an object, including integers *n*, the null value `nil`, and strings `"str"`. Local variables are identifiers *id*, which are made up of upper and lower case letters or symbols (including `+`, `-`, `*`, `/`, `_`, `!`, and `?`). 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 anything except `nil`, and the false branch otherwise. Rube includes sequencing of expressions, assignments to local variables, and method invocation with the usual syntax.

One interesting feature of Rube is that classes are “first class,” meaning they are treated just like any other object—they can have methods invoked on them (like the `new` method, which creates new objects), can be passed as arguments, and can be returned as results. Classes in Rube are anonymous (like anonymous functions defined with `fun` in OCaml). The expression

```
class method ... method end
```

returns an object representing the defined class. If you want to define a class and then save it for future use, you can assign it to a variable. For example, consider the file `r3.ru`, reproduced here:

```
C = class
  def m(a, b)
    @x = a.+(b);
    @x
  end
end;

x = C.new();

x.m(1, 2)
```

This program defines a class, assigns it to the local variable `C`, instantiates it by calling `C.new()`, and then

invokes a method of the resulting object. Note that, unlike in Ruby, identifiers are all treated the same regardless of capitalization. (In Ruby, capitalized identifiers are constants that cannot be changed.)

Pretty neat, huh! The equivalent Ruby code actually does the same thing—when you write `class C ... end` in Ruby, you’re assigning to `C`, and you can freely copy that class around the program. For example, if `C` and `D` are both assigned to classes, in Rube and Ruby you can write `X = if p then C else D; y = X.new` to make `y` an instance of either `C` or `D`, depending on the value of `p`.

We can define precisely how a Rube program executes by giving a formal operational semantics for it. Just like in the class lectures, the first thing we need to do to define a semantics is to explain what programs may reduce to. In our semantics, programs will reduce to values v given by the following grammar:

$$v ::= n \mid \text{nil} \mid \text{"str"} \mid \{method_1 \dots method_n\} \mid [\text{class} = v_0; \text{fields} = @id_1 : v_1, \dots, @id_n : v_n]$$

Values include integers n , the null value `nil`, and strings `"str"`. We’ve used a slightly different font here to emphasize the difference between program text, such as `nil`, and what it evaluates to, `nil`.

We represent classes by special *class objects* $\{method_1 \dots method_n\}$, where the $method_i$ are the methods defined in the class. Finally, to represent object values, we write $[\text{class} = v_0; \text{fields} = @id_1 : v_1, \dots, @id_n : v_n]$, which is an instance of class v_0 and which has fields $@id_1$ through $@id_n$, and field $@id_i$ has value v_i . For example, in the code above, class `C` evaluates to $\{\text{def } m(a, b) \ @x = a.+(b); \ @x \ \text{end}\}$ and `x` evaluates to $[\text{class} = v_C; \text{fields} = \emptyset]$ where v_C is what `C` evaluates to and by \emptyset we mean the object has no fields initially. After calling `x.m(1,2)`, the variable `x` would evaluate to $[\text{class} = v_C; \text{fields} = @x : 3]$.

To define our semantics, we also need to define environments A , which map local variable names to values. In our semantics, A will be a list $id_1 : v_1, \dots, id_n : v_n$, and names on the *left* shadow names on the *right*. In other words, if id appears more than once in an environment A , then $A(id)$ is defined to be the left-most value id is bound to.

Figure 2 gives the formal operational semantics for evaluating Rube expressions. These rules show reductions of the form $A; expr \rightarrow v$, meaning that given local variable bindings A , the expression $expr$ reduces to the value v . We’ve labeled the rules to make them easier to discuss.

There’s an important convention in these rules: When there are multiple hypotheses (things above the line) in a rule, the hypotheses are evaluated in order from **left-to-right** and **top-to-bottom**. We need to specify the order because of side effects (writing to fields).¹ Here is what the rules mean, in English. You can do this project successfully even if you don’t understand the formal description (i.e., just going by the text below). However, recall from class that the formal description is pretty darn close to an actual OCaml implementation, so if you spend some time trying to understand the rules, it may make this project easier for you.

- The rules `INT`, `NIL`, and `STR` all say that an integer, `nil`, or string evaluate to the expected value, in any environment. In the syntax of Rube, strings begin and end with double quotes `"`, and may not contain double quotes inside them. (Escapes are not handled.)
- The *local variables* of a method include the parameters of the current method, local variables that have been previously assigned to, and `self`, which refers to the object whose method is being invoked. The rule `ID` 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 interpreter would signal an error.
- The rule `FIELD-R` says that when a field is accessed, we look up the current object `self`, which 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.

¹There’s actually a better way to specify the order of evaluation in operational semantics, but it would complicate the rules.

$$\begin{array}{c}
\text{INT} \qquad \qquad \text{NIL} \qquad \qquad \text{STR} \\
\hline
A; n \rightarrow n \qquad A; \mathbf{nil} \rightarrow \mathbf{nil} \qquad A; \text{"str"} \rightarrow \text{"str"} \\
\\
\text{ID} \qquad \qquad \text{FIELD-R} \\
\frac{id \in \text{dom}(A)}{A; id \rightarrow A(id)} \qquad \frac{A(\mathbf{self}) = [\text{class} = v_0; \text{fields} = @id_1 : v_1, \dots, @id_n : v_n]}{A; @id_i \rightarrow v_i} \\
\\
\text{FIELD-NIL} \\
\frac{A(\mathbf{self}) = [\text{class} = v_0; \text{fields} = @id_1 : v_1, \dots, @id_n : v_n] \quad \forall i. @id \neq @id_i}{A; @id \rightarrow \mathbf{nil}} \\
\\
\text{IF-T} \\
\frac{A; expr_1 \rightarrow v_1 \quad v_1 \neq \mathbf{nil} \quad A; expr_2 \rightarrow v_2}{A; \mathbf{if} \ expr_1 \ \mathbf{then} \ expr_2 \ \mathbf{else} \ expr_3 \ \mathbf{end} \rightarrow v_2} \\
\\
\text{IF-F} \\
\frac{A; expr_1 \rightarrow \mathbf{nil} \quad A; expr_3 \rightarrow v_3}{A; \mathbf{if} \ expr_1 \ \mathbf{then} \ expr_2 \ \mathbf{else} \ expr_3 \ \mathbf{end} \rightarrow v_3} \\
\\
\text{SEQ} \qquad \qquad \qquad \text{ID-W} \\
\frac{A; expr_1 \rightarrow v_1 \quad A; expr_2 \rightarrow v_2}{A; expr_1; expr_2 \rightarrow v_2} \qquad \frac{A; expr \rightarrow v \quad id \neq \mathbf{self} \quad A := A[id : v]}{A; id = expr \rightarrow v} \\
\\
\text{FIELD-W} \\
\frac{A(\mathbf{self}) = [\text{class} = v_0; \text{fields} = @id_1 : v_1, \dots, @id_n : v_n] \quad A; expr \rightarrow v \quad \text{fields} := \text{fields}[@id : v]}{A; @id = expr \rightarrow v} \\
\\
\text{INVOKE} \\
\frac{A; expr_0 \rightarrow v_0 \quad v_0 = [\text{class} = v_c; \text{fields} = \dots] \quad A; expr_1 \rightarrow v_1 \quad \dots \quad A; expr_n \rightarrow v_n \quad \text{method}(v_c)(id_m) = \mathbf{def} \ id_m(id_1, \dots, id_k) \ expr \ \mathbf{end} \quad k = n \quad A' = \mathbf{self} : v_0, id_1 : v_1, \dots, id_n : v_n \quad A'; expr \rightarrow v}{A; expr_0.id_m(expr_1, \dots, expr_n) \rightarrow v} \\
\\
\text{CLASS} \\
\hline
A; \mathbf{class} \ method_1 \ \dots \ method_n \ \mathbf{end} \rightarrow \{method_1 \ \dots \ method_n\} \\
\\
\text{PROGRAM} \\
\frac{A = \mathbf{self} : [\text{class} = \{\}; \text{fields} = \emptyset] \quad A; expr \rightarrow v}{expr \Rightarrow v}
\end{array}$$

Figure 2: Rube Operational Semantics for Expressions

- 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 that by our order-of-evaluation assumption, we evaluate the guard before either the then or else branch.
- The rule SEQ says that to evaluate $expr_1; expr_2$, we evaluate $expr_1$ and then evaluate $expr_2$, whose value we return. Note that in the syntax, semicolon is a *separator*, and does not occur after the last expression. Thus, for example, $1; 2$ is an expression, but $1; 2;$ is not (and will not parse).
- The rule ID-W says that to write to a local variable id , we evaluate the $expr$ to a value v , and we then *update* the environment A so that now id is mapped to v . (In other words, if id was in A before, its value is replaced, and if it was not in A , it is added to the map A with the new value). This is non-standard notation, as it turns out, but the meaning should be clear. The rule FIELD-W is similar, except we update the object corresponding to **self** with the (possibly) new field. In both cases, assignment returns the value that was assigned. (This is in contrast to OCaml, where assignment returns the unit value.)

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). If a user tries to write to **self**, your implementation should signal an error.

- The most complicated rule is for method invocation. We begin by evaluating the receiver $expr_0$ to a value v_0 , which must be an object. We then evaluate the arguments $expr_1$ through $expr_n$, in order from 1 to n , to produce values. Then we perform method lookup. By $method(v_c)(id_m)$, we mean look up the method named id_m in class v_c . Thus here v_c must be some class object $\{\dots\}$, and if it is not, your interpreter would halt and signal an error.

Once we find a method **def** $id_m(id_1, \dots, id_k)$ 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. Finally, we make a new environment A' in which **self** is bound to the receiver object v_0 , 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.

Note that in Rube, unlike in Ruby, there is no inheritance of methods, and no subclassing. Also 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 in some ways than the OCaml semantics we showed you in class.

- Rule CLASS says that a class definition is evaluated to the corresponding class object, which just records the methods that were defined.
- Finally, rule PROGRAM explains how to evaluate a Rube program. We evaluate the expression $expr$ that makes up the program, starting in an environment A where **self** is the only variable in scope, and it is bound to an object whose class is the “empty” class containing no methods (written $\{\}$), and with no fields. (Notice that there’s no **Object** class in this system—since we have no class hierarchy, we don’t need a class for the root of the hierarchy.)

That’s the “core” of the semantics. Rube also includes several built-in methods that may be invoked on the built-in types, including the method **new**, which can be invoked on classes to create new instances of them. Figure 3 gives the operational semantic rules for these built-in methods. From top to bottom:

- If $expr_0$ and $expr_1$ evaluate to integers n and m , respectively, then $expr_0.(expr_1)$ evaluates to $n+m$, and analogously for the methods $-$, $*$, and $/$.

$$\begin{array}{c}
\frac{A; \text{expr}_0 \rightarrow n \quad A; \text{expr}_1 \rightarrow m \quad \text{aop} \in \{+, -, *, /\}}{A; \text{expr}_0.\text{aop}(\text{expr}_1) \rightarrow n \text{ aop } m} \\
\\
\frac{A; \text{expr} \rightarrow n}{A; \text{expr}.\text{to_s}() \rightarrow \text{"n"}} \\
\\
\frac{A; \text{expr} \rightarrow \text{nil}}{A; \text{expr}.\text{to_s}() \rightarrow \text{"nil"}} \\
\\
\frac{A; \text{expr} \rightarrow \text{"str"} \quad \text{prints string to standard output}}{A; \text{expr}.\text{print}() \rightarrow \text{nil}} \\
\\
\frac{A; \text{expr} \rightarrow \{\text{method}_1 \dots \text{method}_n\}}{A; \text{expr}.\text{new}() \rightarrow [\text{class} = \{\text{method}_1 \dots \text{method}_n\}; \text{fields} = \emptyset]} \\
\\
\frac{A; \text{expr} \rightarrow [\text{class} = v_0; \text{fields} = \dots]}{A; \text{expr}.\text{class_of}() \rightarrow v_0} \\
\\
\frac{A; \text{expr}_0 \rightarrow \text{"str"} \quad A; \text{expr}.\text{str}(\text{expr}_1, \dots, \text{expr}_k) \rightarrow v}{A; \text{expr}.\text{send}(\text{expr}_0, \text{expr}_1, \dots, \text{expr}_k) \rightarrow v}
\end{array}$$

Figure 3: Rube Operational Semantics for Built-in Classes

- If *expr* evaluates to an integer *n*, then *expr.to_s()* evaluates to the string corresponding to the integer *n*. Analogously, *to_s* may be invoked with no arguments on *nil*. Important: The *to_s* method does not put quotes around the string that's created. In your interpreter, *nil.to_s()* should evaluate to the 3-character string *nil*. The quotes are there in the semantics just to distinguish strings from other values.
- If *expr* evaluates to a string, then *expr.print()* prints the string to standard out and returns *nil*.
- If *expr* evaluates to a class, then *expr.new()* returns an object whose class is the one whose *new* method was invoked, and with no fields. Notice that there are no constructors in Rube.
- If *expr* evaluates to an object, then invoking the *class_of* method on it returns the object's class. Note that even though we treat integers, *nil*, and strings as objects, we have no rule for invoking their *class_of* method. (We could add that, but it would be kind of tedious.)
- To invoke *send* method on an expression (i.e., a reflective method call), we evaluate the first argument *send* to a string; then we turn that string into a method identifier, and perform a normal call on the receiver object with the remaining arguments. (This is just like the *send* method in Ruby.)

A Rube Interpreter

Your task is to write a Rube Interpreter that follows the operational semantics we just gave you. We've already written most of the infrastructure you'll need for your interpreter. We'll begin our description of the interpreter by looking at the file `rube.ml`, which is the file you'll be editing. If you scroll down to the bottom of that file, you'll find the main routine executed when you run `rube`:

```

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let (e:expr) = Parser.main Lexer.token lexbuf in
  begin
    print_program e;
    print_string "\nEvaluates to:\n";
    print_value (eval_expr (ref []) e)
  end

```

You can consider the first two lines to be magic: The first line binds the name `lexbuf` to the stream of tokens that `lexer.mll` extracts from standard input. The second line parses the stream of tokens to produce an `expr`, which is an abstract syntax tree representing the Rube program. (We'll talk about the definition of this type shortly.) Then we print out the program using a function `print_program` that we've provided for you, print some separator text, and then print out the result of calling `eval_expr` on the program. This function, `eval_expr`, is the main one you're going to have to write.

The type `expr` represents abstract syntax trees of Rube programs, and this type is defined in the file `ast.mli`. Here is part of that file:

```

type expr =
  EInt of int
  | ENil
  | EString of string
  | ELocal of string      (* Read a local variable *)
  | EField of string     (* Read a field *)
  | EIf of expr * expr * expr
  | ESeq of expr * expr
  | EWrite of string * expr (* Write a local variable *)
  | EWriteField of string * expr (* Write a field *)
  | EInvoke of expr * string * (expr list)
  | EClass of meth list

(* (method name, argument names (may be empty), method body) *)
and meth = string * string list * expr

```

Notice there is a suspicious similarity between these expressions and the grammar for Rube we gave you in Figure 1. **Important:** Don't modify these constructors or the `meth` type. Otherwise our grading scripts won't work.

The first three constructors should be self-explanatory. The expression `ELocal s` represents (reading) the local variable `s`. Notice in our abstract syntax tree, we use strings for the names of local variables. The expression `EField s` represents reading a field `s`. Here `s` is also a string, but it will happen to be the case that because of the way the parser works, it will always begin with an `@`.

The expression `EIf(e1,e2,e3)` corresponds to `if e1 then e2 else e3 end`. The expression `ESeq(e1,e2)` corresponds to `e1;e2`. The expression `EWrite(s,e)` corresponds to `s=e`, where `s` is a local variable, and the expression `EWriteField(s,e)` corresponds to `s=e` when `s` is a field. `EInvoke(e,s,e1)` corresponds to calling method `s` of object `e` with the arguments given in `e1`. (The arguments are in the same order in the list as in the program text, and may be empty.) Lastly, `EClass` represents an (anonymous) class definition, which is simply a list of methods.

A method `meth` is a tuple containing the method name, the argument names, and the method body. Here the `and` keyword in front of the definition of type `meth` makes it mutually recursive with the type definition of `expr`. (Notice that `expr` refers to `meth`, and `meth` refers to `expr`.) You can use this trick to write mutually recursive functions as well; look at the output functions towards the top of `rube.ml` for other examples of this.

Task 1: Write a Function to Convert values to Strings Your Rube interpreter is going to take ASTs as input and produce values as output, just like the operational semantics. The type `value` is defined at the top of `rube.ml`:

```
type value =
  VInt of int
  | VNil
  | VString of string
  | VObject of value * (string * value) list ref
  (* Invariant: no field name appears twice in a VObject *)
  | VClass of meth list
```

Important: Don't modify these constructors. Otherwise our grading scripts won't work.

The value of a Rube program can be either an integer, nil, a string, an object, or a class. The object `VObject(v, fields)` represents an instance of the class `v`. Since the fields of objects can be updated over time, `fields` is an updatable reference containing a list of string and `value` pairs. For example, when creating a new instance of class `v0 = VClass [...]`, the result should be `VObject(v0, ref [])`, which is an object with no fields. As another example, as instance of `v0` with one field `f` that has value `nil` would be `VObject(v0, ref [(f, VNil)])`. Class values `VClass` contain the list of methods defined in the class. We do not require that methods in classes have distinct names; if they do not, and you need to look up a method in a class, you can return any method. We will not test your code with classes that have duplicate methods.

Recall that the main Rube interpreter prints out the value that your program evaluates to. Your first task is to write a function

```
val value_to_string : value -> string
```

that, given a `value`, returns the corresponding `string`. More precisely:

- The integer `n` should be converted to the string representation of `n`. For example, `value_to_string (VInt 3) = "3"`. (That's the string containing the single character `3`—the quotes are not part of the string.)
- The `nil` value should be converted to the string `nil`.
- A string should be converted to itself.
- An object `VObject(v0, ref [(f1,v1); ...; (fn,vn)])` should be converted to a string of the form `"{v0 f1=v1, ..., fn=vn}"`. If there are no fields in the object, then the `}` should appear immediately to the right of `v0`. Otherwise, there should be one space after `v0`, and the string should contain the name of each field, followed by `=`, followed by the string value of the field (computed via a recursive call). Fields should be separated by a comma followed by a space before the next field. There should be no comma or space to the right of the right-most field.
- A class should be converted to `<class>`. (I.e., we won't actually print out the contents of the class.)

Hint: The code to unparse expressions should help you quite a bit for this part. In that code we used OCaml's `Printf` module. You're welcome to use that, though it's a bit complex, so you may want to stick to the ordinary string conversion functions (e.g., `string_of_bool` etc). □

Task 2: Write a Function to Evaluate Rube Expressions Finally we get to the core of the project: running a Rube program. Write a function

```
val eval_expr : (string * value) list ref -> expr -> value
```

that evaluates a Rube expression. From left-to-right, the arguments are: The local variable environment in which to evaluate the expression (this is updateable to allow you to write to local variables); and the expression to evaluate. The result of this function is a Rube `value`.

Your function should raise the exception `Eval_error`, which we've defined, if it detects an error during evaluation (such as calling a method with the wrong number of arguments, trying to invoke `new` on an object that is not a class, etc).

Challenge Problem

Already finished the project? Too much free time on your hands? Bored and looking for something to do? I will give out up to **16 points of extra credit** on this project for implementing some interesting and creative extension to the Rube programming language. Here are the rules for the extra credit:

- I will only consider extra credit if your project scores at least 85/100 on the secret tests for an on-time project. So do not work on the extra credit at the expense of core functionality. We will also only look at extra credit submissions that come with on-time projects.
- Your extension must not break any of the behavior specified for Rube in the project. In other words, it must be a language extension, not revision.
- The only possible extra credit scores are 0, 2, 4, 8, or 16 points. You will get 2 points for some small extension, like adding a simple new kind of built-in object with a couple of methods. You will get 4 points for a more thorough but still straightforward extension. You will get 8 points for something impressive, and 16 points for something that blows me away. I do not expect to give out many (and may give out no) 16 point bonuses.
- You must write up, in good quality English prose, a thorough description of your extension(s), including: (a) what your extension is, (b) why it is interesting, (c) working examples of using it in Rube. Include your examples as Rube code that I can run your submitted code on. Submit your description a plain text file `extra-credit.txt`, along with extra sample code, uploaded to the submit server along with your project. Separately, send Joao an email so we know to look for it.

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.