Project 4

Due 11:59:59pm Thu, May 10, 2012

Updates

- Apr 30. Moved `print()` method from Object to String.
- Apr 27. Added a missing case to `assembler.ml/build_constants` to handle constant arguments to `Cmp`.

Introduction

In this project, you will write a compiler that translates Rube source code (from project 2) into Lua bytecode. To avoid dependencies on project 2, we will compile the original version of Rube, without any static type annotations; we will also make a few simplifications in the Rube semantics to make a compiler a bit easier to write.

Project Structure

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

`Makefile`	Makefile
`OCamlMakefile`	Helper for makefile
`lexer.mll`	Rube lexer
`parser.mly`	Rube parser
`ast.mli`	Abstract syntax tree type
`assembler.ml{i}`	Lua bytecode assembler
`rubec.ml`	The main compiler logic
`r{1--4}.ru`	Small sample Rube programs

You will only change `rubec.ml`; you should not edit any of the other files. The file `rubec.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
$ ./rubec r1.ru
$ lua rubec.out
Fix me!
$ ...
```

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

Recall that in Rube, the program executes by running the top-level expression. To make it easier to debug and grade your compiled programs, the programs you generate should take that expression, turn it into a string, and print it out; more details below.

Rube Syntax and Semantics

As a reminder, the formal syntax for Rube programs is shown in Figure 1. Figure 2 gives the formal operational semantics for evaluating Rube expressions (we will discuss relating these rules to comilation 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,

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	::=	n	Integers
	|	`nil`	Nil
	|	`"`str`"`	String
	|	`self`	Self
	|	id	Local variable
	|	@id	Field
	|	`if` E `then` E `else` E `end`	Conditional
	|	E; E	Sequencing
	|	$id = E$	Local variable write
	|	@$id = E$	Field write
	|	`new` id	Object creation
	|	$E.id(E, \ldots, E)$	Method invocation

Figure 1: Rube syntax (without types)

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'. 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. (Escapes are not handled.)

- The *local variables* of a method include only the paramters of the current method as wel as `self`, which refers to the object whose method is being invoked. (Note that unlike Ruby, a local variable cannot 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 interpreter would signal an error. Reading a local variable or `self` does not change the local variable environment or the heap.

- 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 rule Seq says that to evaluate E_1; E_2, we evaluate E_1 and then evaluate E_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). Notice again the order of evaluation between E_1 and E_2.

$$\textsc{Int}\ \frac{}{P \vdash \langle A, H, n\rangle \to \langle A, H, \mathsf{n}\rangle} \qquad \textsc{Nil}\ \frac{}{P \vdash \langle A, H, \texttt{nil}\rangle \to \langle A, H, \mathsf{nil}\rangle} \qquad \textsc{Str}\ \frac{}{P \vdash \langle A, H, \texttt{"}str\texttt{"}\rangle \to \langle A, H, \text{“}\mathsf{str}\text{”}\rangle}$$

$$\textsc{Id/Self}\ \frac{id \in dom(A)}{P \vdash \langle A, H, id\rangle \to \langle A, H, A(id)\rangle} \qquad \textsc{Field-R}\ \frac{A(\texttt{self}) = \ell \qquad H(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_n : v_n]}{P \vdash \langle A, H, @id_i\rangle \to \langle A, H, v_i\rangle}$$

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

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

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

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

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

$$\textsc{Field-W}\ \frac{\begin{array}{c}P \vdash \langle A, H, E\rangle \to \langle A', H', v\rangle \qquad A(\texttt{self}) = \ell \qquad H'(\ell) = [\mathsf{class} = id_s;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_n : v_n] \\ H'' = H'[\ell \mapsto [\mathsf{class} = id_s;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_i : v, \ldots, @id_n : v_n]]\end{array}}{P \vdash \langle A, H, @id_i = E\rangle \to \langle A', H'', v\rangle}$$

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

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

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

Figure 2: Rube Operational Semantics for Expressions

- 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. As mentioned above, it's not possible to create a new local variable by writing to it, so *id* must already exist in A' in order to change it.

  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, except 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 am object whose id_i field is v. In both cases, assignment returns the value that was assigned. (This is in contrast to OCaml, where assignment returns the unit value.)

- Next, the rule NEW creates a new instance of a class *id*. 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. Finally, 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. (Notice that there are no constructors in Rube.)

- The most complicated 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. 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. Notice that this is different than in Project 2, in which the top-level expression could not access fields.

Built-in methods

In addition to the core language, recall that Rube also includes several built-in methods that may be invoked on the built-in types; the type signatures for these methods are given in Figure 3, 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. If you follow the design decisions in the next section, you can achieve this using Lua's built-in equality test. However, this method should be overridden for `String` and `Integer` to do a deep equality test of the string and integer values, respectively. In these last two cases, your methods should always return false if the object `self` is being compared to is not a `String` or `Integer`, respectively.

Class	Method type	
`Object`	`equal?` : (`Object`) → `Object`	equality check
	`to_s` : () → `String`	convert to string
`String`	`+` : (`String`) → `String`	string concatenation
	`print` : () → `bot`	print to standard out
`Integer`	`+` : (`Integer`) → `Integer`	addition
	`-` : (`Integer`) → `Integer`	subtraction
	`*` : (`Integer`) → `Integer`	multiplication
	`/` : (`Integer`) → `Integer`	division

Figure 3: Built-in objects and methods

- 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`, and for `Integer` to return a `String` containing the textual representation of the integer. You can use Lua's string library[1] to perform the conversion.
- The `print` method prints a string to standard out, as-is. (E.g., do not add any additional newlines.) You can call Lua's `io.write` function to perform printing. Your `print` method should return `nil`.
- The `+` method on `String`s performs string concatenation; you can use the `Concat` bytecode instruction to do this. Your method should halt execution with an error if passed a non-`String` as an argument.
- The `+`, `-`, `*`, and `/` methods perform integer arithmetic, using the appropriate Lua bytecode instructions. Your method should halt execution with an error if passed a non-`Integer` as an argument.

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, just as if the program called v`.to_s().print()`.

This is a fairly complex project, with a lot of interlocking parts. We suggest that you try implementing language features in approximately the following order: `self`, as a pointer to a table for fields only; ints, not as an object, but as a primitive value that can be stored in a field and will be printed at the end of execution; field reads and writes; sequencing; and then conditionals. Then you can move on to objects, method calls, and built-in methods.

It's up to you how to design the run-time representation of Rube data. We will *only* test your compiler by compiling Rube code and seeing what running it under Lua prints out. However, we have the following suggestions (which you may disregard) on how you might set some things up:

- You could implement objects using the standard Lua approach mentioned on the class slides. But it may be simpler to implement 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 teh object. We've chosen `#vtable` as the vtable key in the hash because no Rube field name can begin with `#`. By default, keys that are not in tables in Lua correspond to Lua's nil value. So if you represent Rube nil as Lua nil, you conveniently will get the special non-existent field behavior of Field-R for free.
- You'll create vtables by first creating one Lua 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

[1] `http://lua-users.org/wiki/StringLibraryTutorial`

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 Lua function corresponding to a Rube method will take a `self` argument.

- There's some example code commented out in `rubec.ml` that shows how to create and call a function in Lua bytecode. There are two steps: First, you create a `function_unit` data structure for the function, and add it to the list of `child_functions` for the code for the top-level expression. Second, to get a function, you use the `Closure` bytecode instruction, which takes as its second argument the index of the function in the `child_functions` list.

- Thus, at a high-level, your top-level `function_unit` will look like:

```
{instructions = [prolog code for setting up environment and vtables;
                 instructions for top-level expression;
                 code for printing out final value of top-level expr];
 num_params = 0;
 child_functions = [...function_units for each method...]
}
```

- Don't worry about shadowing among parameters names and `self`. We won't test that.

- Local variables (just `self` and parameters) will need to be assigned to registers. You don't need to worry about spilling registers to memory—you can assume enough Lua registers exist.

- To handle built-in methods, you'll want to generate a standard set of vtables for `Object`, `Integer`, and `String`, containing the built-in methods. Thus, integers and strings will be pointers to objects, i.e., tables. You could use a field name such as `#contents` to store the actual primitive Lua integer or string, 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.

- Bytecode for comparisons in Lua is a bit confusing. There's some example use of the `Cmp` instruction commented out in the `compile_prog` function of `rubec.ml` you can look at. Note that the No-Frills Lua guide mentions "Due to the way code is generated and the way the virtual machine works, a JMP instruction is always expected to follow an EQ, LT or LE." We haven't tested whether this is actually the case, but it seems hard to generate code any other way, and this may be a restriction on the instruction set.

- You'll most likely want to use (only) the following instructions in your compiler (there are a few more available in `assembler.mli`, but you need not use them):

Load_{Const,Nil,Bool}	Jump	Cmp	Move
Arith	Return	Concat	Call
{Load,Set,Make}_Table	{Get,Set}_Global	Closure	

  If there are additional instructions you want to use beyond the ones available, let us know and we'll add support for them, or you can (it's pretty straightforward).

- It's quite possible to create a malformed Lua bytecode file with the assembler. For example, you could forget to add a `Return` at the end of a function. If you make such a mistake, you'll be given the cryptic error message `lua: rubec.out: bad code in precompiled chunk`. Thus, we suggest that you try to make small changes in the output of your compiler, so that you will know exactly where the mistake lies when you get such an error message.

Extra Credit

Already finished the project? Too much free time on your hands? Bored and looking for something to do? I will give up to 20% extra credit on this project for extending your compiler to also perform **optimization**. Here are the rules for extra credit:

- I will only consider extra credit if you get at least 75% of the points from the regular grading. So don't work on optimization at the expense of core functionality. I will also only look at extra credit submissions that come with on-time projects.
- Your optimizer should only be enabled if `-O` is passed on the command line to `rubec`. Thus, we will perform regular grading of your project with optimization disabled.
- You are welcome to incorporate your dataflow analysis from project 3 into your optimizer. You are not required to do so, however.
- You must write up, in good quality English prose, a thorough description of the optimizations you implemented. You must also perform and document an experiment showing whether your optimizations actually improve performance. That is, you should come up with one or more Rube inputs; compile them with and without optimization; and then report the time they take to run under the two scenarios. To follow good experimental procedure, you should run each program at least 5 times and report the median and standard deviation. (You should really run as many times as necessary so that the median you report is clearly representative, but I'd guess 5 runs is enough.) You should also report key characteristics of the machine you ran the experiment on, e.g., CPU and memory size.
- Include your experimental inputs and writeup in your submission. I should be able to easily reproduce your experiment(s) from what you submit. (You might have multiple experiments if you want to measure different optimizations separately.)
- Recall that Lua has a JIT compiler, so even if you implement what you think is an optimization, it may make no difference in practice. You can still get partial extra credit even in this case, but you'll get the most extra credit if your optimizations truly improve performance. Thus, you might think about optimizations that a standard JIT might have trouble doing.

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.