Project 4
Due 11:59:59pm Wednesday, June 11, 2012

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.

DON’T PANIC!!!

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. However, this project does require writing significantly more code than project 3, so please be sure to give yourself enough time.

Rube Syntax

The formal syntax for Rube programs is shown in Figure 1, along with the OCaml data structure for ASTs that correspond to the grammar. A Rube program $P$ is made up 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 containing no methods, 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, and strings "str". 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 anything except nil, and the false branch otherwise. Rube includes sequencing of expressions, assignments to local variables and fields, and method invocation with the usual syntax.

Notice there is a suspicious similarity between the OCaml data type and the formal grammar. Important: Don’t modify these constructors or types. Otherwise our grading scripts won’t work.

The first four 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. ENew(s) corresponds to new s. 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.)

A method meth is a tuple containing the method name, the argument names, and the method body. A class cls is a tuple containing the class name, its superclass name, and its methods. Finally, a program prog is a pair containing the list of classes and top-level expression.
$P ::= C* E$  
Rube program

$C ::= \text{class } id < id \begin{vmatrix} M \end{vmatrix} \text{ end}$  
Class definition

$M ::= \text{def } id (id, \ldots, id) E \text{ end}$  
Method definition

$E ::= n$  
Integers

\begin{vmatrix} \begin{array}{|l|}
\text{nil} & \text{Nil} \\
"str" & \text{String} \\
self & \text{Self} \\
id & \text{Local variable} \\
@id & \text{Field} \\
\text{if } E \text{ then } E \text{ else } E \text{ end} & \text{Conditional} \\
E; E & \text{Sequencing} \\
id = E & \text{Local variable write} \\
@id = E & \text{Field write} \\
\text{new } id & \text{Object creation} \\
E.id(E, \ldots, E) & \text{Method invocation}
\end{array} \end{vmatrix}$

\begin{verbatim}
type expr =
  EInt of int |
  ENil |
  ESelf |
  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 *) |
  ENew of string |
  EInvoke of expr * string * (expr list)
  (* meth name * arg name list * method body *)

type meth = string * string list * expr

(* class name * superclass name * methods *)
type cls = string * string * meth list

(* classes * top-level expression *)
type prog = cls list * expr
\end{verbatim}

Figure 1: Rube syntax
Project Structure

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

- Makefile
- OCamlMakefile: Helper for makefile
- lexer.mll: Rube lexer
- parser.mly: Rube parser
- ast.mli: Abstract syntax tree type
- rube.ml: The main program logic. You will edit this file.
- main.ml: Handle program arguments. You don’t need to edit this file.
- r{1--5}.ru: Small sample Rube programs

You will only change rube.ml; you should not edit any of the other files.

The file main.ml includes some slightly icky code that does arguments processing and dispatch to the various functions you need to write in this project. You can see the table of “commands” at the bottom of that file, which have the same names as the functions. Here is a complete list:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unparse filename</td>
<td>Parses filename and then “unparses” the AST, printing it to standard out</td>
</tr>
<tr>
<td>check_dup_params filename</td>
<td>Calls check_dup_params p, where p is the AST from parsing filename</td>
</tr>
<tr>
<td>unused filename</td>
<td>Calls unused p, where p is the AST from parsing filename; for the rest of the commands, we’ll use p implicitly in the same sense</td>
</tr>
<tr>
<td>rename filename c m x y</td>
<td>Calls rename p c m x y</td>
</tr>
<tr>
<td>superclass_of filename c</td>
<td>Calls superclass_of p c</td>
</tr>
<tr>
<td>check_dup_classes filename</td>
<td>Calls check_dup_classes p</td>
</tr>
<tr>
<td>wf_inheritance filename</td>
<td>Calls wf_inheritance p</td>
</tr>
<tr>
<td>lookup filename c m</td>
<td>Calls lookup p c m</td>
</tr>
<tr>
<td>eval_prog filename</td>
<td>Calls eval_prog p</td>
</tr>
</tbody>
</table>

For example, after building you can run:

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

To parse r1.ru and print the parsed result to standard out.

Part 1: Warmup: Refactoring

To help get you used to working with the AST data structure, write the following recursive functions. We’ve left placeholders for these functions in rube.ml.

1. Write a function `check_dup_params : prog -> bool` that returns true if at least one of the methods defined in the program have multiple parameters with the same name; if not, your function should return false. For example, this function would return true if there were a method `def foo(x, x) ... end`. You don’t need to worry about duplicate parameter names for any other part of this project (assume they don’t occur).

2. Write a function `unused : prog -> (string * string * string) list` that returns a list of all triples (c, m, x) such that method m in class c has a parameter x that is never textually referred to in the body of the method. For example, if class C contains a method `def foo(x, y) x.+(1) end` then this method should return the triple ("C", "foo", "y"). (Your list should contain all class/method/unused variable combinations in the program; if there are none, return the empty list.)
3. Write a function \texttt{rename p c m x y : prog \to string \to string \to string \to string \to prog} that returns a new program that is the same as \texttt{p}, except method \texttt{m} of class \texttt{c} has been rewritten so that occurrences of the parameter \texttt{x} are renamed to \texttt{y}. For example, if class \texttt{C} contains a method \texttt{def foo(x, y) x.+{y} end} then \texttt{rename p "C" "foo" "x" "a"} should return a new program with a class \texttt{C} containing the method \texttt{def foo(a, y) a.+{y} end}. You don’t need to worry about name collisions, e.g., don’t worry about someone calling \texttt{rename p "C" "foo" "x" "y"} on the example program just mentioned. This function should have no effect if \texttt{c}, \texttt{m}, or \texttt{x} don’t exist in \texttt{p}. You can assume there is at most one class \texttt{c} and one method \texttt{m} of class \texttt{c} in the program.

**Part 2: Classes and Inheritance**

As you saw earlier, Rube has \textit{single inheritance}, in which each class inherits from exactly one other class. Write the following functions that recursively traverse an AST to reason about the inheritance structure of a program.

1. Write a function \texttt{check_dup_classes : prog \to bool} that returns true if at least one of class is defined twice in the program, e.g., \texttt{class C ... end ... class C ... end}. For the remainder of the project, you can assume no duplicate classes appear in the program.

2. Write a function \texttt{superclass_of p c : prog \to string \to string} that returns the superclass of class \texttt{c} in program \texttt{p}. The function should \texttt{raise Not_found} if class \texttt{c} does not exist. Be sure your function returns \texttt{Object} as the superclass of \texttt{Object}, even though \texttt{p} will not have a definition of \texttt{Object} itself. (You can assume that \texttt{p} does not define \texttt{Object}, here and throughout the project.)

3. Write a function \texttt{wf_inheritance : prog \to bool} that returns true if the inheritance hierarchy is \textit{well-formed}, meaning (a) each classes’s superclass is defined in the program, and (b) the inheritance hierarchy has no cycles in it. For example, a program containing \texttt{class A < B ... end class B < A ... end} would fail the cyclicity check.

4. Write a function \texttt{lookup p c m : prog \to string \to string \to meth} that returns the \texttt{meth} with name \texttt{m} in class \texttt{c} or, if that method is not defined, it should return the \texttt{meth} from \texttt{c}’s superclass (and so on, recursively up the class hierarchy). This function should \texttt{raise Not_found} if no such method exists in \texttt{c} or in any superclass.

**Part 3: A Rube Interpreter**

We can define precisely how a Rube program executes by giving a formal operational semantics for it, which is essentially just an interpreter written in mathematical notation. Don’t worry! We’ll also describe everything you need to know in English. However, if you take the time to understand the formal semantics rules, you’ll have a big leg up on the project—the rules are almost a transliteration of the OCaml code you’ll need to write.

The first step is to define a set of \textit{values} \texttt{v} that program expressions evaluate to. In our semantics, values are defined by the following grammar:

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

Values include integers \texttt{n}, the null value \texttt{nil}, and strings "\texttt{str}". We’ve used a slightly different font here to emphasize the difference between program text, such as \texttt{nil}, and what it evaluates to, \texttt{nil}. To represent object values, we write \texttt{[class = v_0; fields = @id_1 : v_1, \ldots, @id_n : v_n]}, which is an instance of class \texttt{v_0} and which has fields \texttt{@id_1} through \texttt{@id_n}, and field \texttt{@id_i} has value \texttt{v_i}. Finally, values include locations \texttt{\ell}, which are simply pointer values. Our semantics will also use a \textit{heap} that is a mapping from locations to values.

The file \texttt{rube.ml} defines a data type that’s exactly analogous to the definition of \texttt{v} above.
type value =
  VInt of int
| VNil
| VString of string
| VObject of string * (string * value) list
  (* Invariant: no field name appears twice in a VObject *)
| VLoc of int

Notice that in the implementation, a heap location is just an int, i.e., it’s an address. In the implementation, a heap will be of the following type:

type heap = (int * value) list

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 $i_1 : v_1, \ldots, i_n : v_n$, 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. This is the same convention we used for the assignment type in project 3, and we’ll use the same type:

(* Local variable environment *)
type environment = (string * value) list

Figure 2 gives the formal operational semantics for evaluating Rube expressions. These rules show reductions of the form $P \vdash (A,H,E) \rightarrow (A',H',v)$, 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 $H'$. 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 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/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 $\ell$. 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 $E_i$, which produces a configuration $(A_i,H_i,v_i)$, and then we evaluate the then or else branch with that local variable environment and heap.
Figure 2: 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. 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 $E_2$.

• 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 exactly like the assignment case in part 4 of project 3. Notice that, as in Ruby, id is added to the environment even if it wasn't there before (hence this language has implicit declarations). The rule FIELD-W is similar, except we return a new heap $H''$ that is the same as the heap $H'$ after evaluating $E$, except we update location $\ell$ to contain an object whose id 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.)

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.

• 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 $\ell$ that is not already used in the heap. Finally, we return the location $\ell$, along with a new heap $H'$ that is the same as heap $H$, except $\ell$ 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 $\ell$, 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 you implemented in part 2 of the project to find the correct method; if a method doesn’t exist, this rule won’t apply, and your implementation should signal an error.

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 $\ell$, 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 $\ell$ containing an object that is an instance of Object and contains no fields.

That’s the “core” of the semantics. Rube also includes several built-in methods that may be invoked on the built-in types. Figure 3 gives the operational semantic rules for these built-in methods. Note that these methods cannot be overridden—you should perform the behaviors specified in the figure even if some other method of the same name is defined in the class. From top to bottom:

• If $E_0$ and $E_1$ evaluate to integers $n$ and $m$, respectively, then $E_0.+(E_1)$ evaluates to $n+m$, and analogously for the methods $-, \times$, and $/$.

• If $E_0$ and $E_1$ evaluate to integers $n$ and $m$, respectively, then $E_0$.equal?($E_1$) evaluates to 1 if $n$ and $m$ are equal, and to nil otherwise.
\[ P \vdash (A, H, E_0) \rightarrow (A_1, H_1, n) \quad P \vdash (A_1, H_1, E_1) \rightarrow (A_2, H_2, m) \quad \text{aop} \in \{+, -, \ast, /\} \]
\[ P \vdash (A, H, E_0, \text{aop}(E_1)) \rightarrow (A_2, H_2, n \ \text{aop} \ m) \]
\[ P \vdash (A, H, E_0) \rightarrow (A_1, H_1, n) \quad P \vdash (A_1, H_1, E_1) \rightarrow (A_2, H_2, m) \quad n = m \]
\[ P \vdash (A, H, E_0, \text{equal?}(E_1)) \rightarrow (A_2, H_2, 1) \]
\[ P \vdash (A, H, E_0) \rightarrow (A_1, H_1, n) \quad P \vdash (A_1, H_1, E_1) \rightarrow (A_2, H_2, m) \quad n \neq m \]
\[ P \vdash (A, H, E_0, \text{equal?}(E_1)) \rightarrow (A_2, H_2, \text{nil}) \]
\[ P \vdash (A, H, E) \rightarrow (A', H', n) \]
\[ P \vdash (A, H, E, \text{to_s()} \rightarrow (A', H', \text{"n"}) \]
\[ P \vdash (A, H, E) \rightarrow (A', H', \text{nil}) \]
\[ P \vdash (A, H, E, \text{to_s()} \rightarrow (A', H', \text{"nil"}) \]
\[ P \vdash (A, H, E) \rightarrow (A', H', \text{"str"}) \quad \text{prints string to standard output} \]
\[ P \vdash (A, H, E, \text{print()} \rightarrow (A', H', \text{nil}) \]
\[ P \vdash (A, H, E) \rightarrow (A', H', \ell) \quad H'(\ell) = \text{class = id; fields = ...} \]
\[ P \vdash (A, H, E, \text{class_of()} \rightarrow (A', H', \text{"id"}) \]
\[ P \vdash (A, H, E_0) \rightarrow (A', H', \text{"str"}) \quad P \vdash (A', H', E, \text{str}(E_1, \ldots, E_k)) \rightarrow (A'', H'', v) \]
\[ P \vdash (A, H, E, \text{send}(E_0, E_1, \ldots, E_k)) \rightarrow (A'', H'', v) \]

Figure 3: Rube Operational Semantics for Built-in Classes

- If \( E \) evaluates to an integer \( n \), then \( E, \text{to_s()} \) evaluates to the string corresponding to the integer \( n \). Analogously, \( \text{to_s()} \) may be invoked with no arguments on \( \text{nil} \). Important: The \( \text{to_s()} \) method does not put quotes around the string that’s created. In your interpreter, \( \text{nil, to_s()} \) should evaluate to the 3-character string \( \text{nil} \). The quotes are there in the semantics just to distinguish strings from other values.

- If \( E \) evaluates to a string, then \( E, \text{print()} \) prints the string to standard out and returns \( \text{nil} \).

- If \( E \) evaluates to an object, then invoking the \( \text{class_of()} \) method on it returns a string containing the object’s class. Note that even though we treat integers, \( \text{nil} \), and strings as objects, we have no rule for invoking their \( \text{class_of()} \) method. (We could add that, but it would be kind of tedious.)

- To invoke \( \text{send()} \) method on an expression (i.e., a reflective method call), we evaluate the first argument \( \text{send} \) 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 \( \text{send()} \) method in Ruby.) Notice that the order of evaluation of receiver and arguments is slightly different here. If you want to, you may change the order of evaluation for send’s receiver and arguments, if it makes your implementation easier; we won’t write any test cases that check the order of evaluation for send’s receiver and arguments.

Finally we get to the core of the project: running a Rube program. Write a function

\[ \text{val eval_prog : prog \to value} \]

that evaluates a Rube program and returns its value. More precisely, if \( P \) is the mathematical representation of the OCaml \( \text{prog} \ p \), then \( \text{eval_prog \ p} \) should return a value \( v \) that corresponds to the \( v \) in \( \vdash P \Rightarrow v \).
You’ll likely want to write another function

```ocaml
val eval_expr : prog -> environment * heap * expr -> environment * heap * value
```

that evaluates a single expression in a given program, environment, and heap, returning a new environment and heap and the resulting value. As above, assuming OCaml program \( p \), environment \( a \), heap \( h \), and expression \( e \) correspond to \( P, A, H, \) and \( E \) in the formal system, and \( P \vdash \langle A, H, E \rangle \rightarrow \langle A', H', v \rangle \), then `eval_expr p (a,h,e)` should return \( \langle a', h', v \rangle \) where \( a', h', \) and \( v \) correspond to \( A', H', \) and \( v \), respectively.

Your evaluation functions can raise any exception to signal an error during evaluation (such as calling a method with the wrong number of arguments, reading an undefined local variable, etc). We’ve supplied the exception `Eval_error` as a convenient exception to throw, but you’re not required to use it.

**Debug Help**

- We’ve provided you with several functions to help you print out the current program state in your evaluator: `print_value`, `print_env`, and `print_heap` print out a value, local variable environment, and heap, respectively. You can call these functions as `print_value stdout thing_to_print` etc.