Homework 9

Introduction

In this project, you will write an interpreter for a programming language called Rube, which is a small object-oriented programming language with a syntax similar to Ruby.

Project Structure

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

`Makefile`	Makefile
`Lexer.lex`	Rube lexer
`Parser.grm`	Rube parser
`Ast.sml`	Abstract syntax tree type
`Unparse.sml`	An "unparser" to print an AST
`Interp.sml`	The interpreter—**this is where you'll write your code**
`main.sml`	The main program
`r{1--4}.ru`	Small sample Rube programs

You will only change `interp.sml`; you should not need to edit any of the other files. The file `main.sml` includes code to run the parser and then invoke `Interp.run`, the function you should implement. Currently, the interpreter contains code that prints `FIX ME!`:

```
$ make
...
$ ./interp r1.ru
FIX ME!
```

You'll need to modify the implementation of `Interp.run` in `Interp.sml` to actually interpret (run) the program.

In case it's handy, we've also provided you with some printing functions in `Unparse`, which you can use to print out Rube programs, classes, methods, etc. There's an example call to `Unparse.print_prog` that's commented out in `main.sml`.

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, object creation, and method invocation with the usual syntax.

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

Figure 1: Rube syntax

```
datatype 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
  | ESeq of expr * expr
  | ENew of 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

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

The first four constructors of `expr` 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`. The expression `ESeq(e1,e2)` corresponds to `e1;e2`. The expression `ENew(s)` corresponds to `new s`. The expression `EInvoke(e,s,el)` corresponds to calling method `s` of object `e` with the arguments given in `el`. (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.

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

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

Id/Self

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

Field-R

$$\frac{A(\texttt{self}) = \ell \qquad H(\ell) = [\mathsf{class} = \mathit{id}_s;\ \mathsf{fields} = @\mathit{id}_1 : v_1, \ldots, @\mathit{id}_n : v_n]}{P \vdash \langle A, H, @\mathit{id}_i\rangle \to \langle A, H, v_i\rangle}$$

Field-Nil

$$\frac{A(\texttt{self}) = \ell \qquad H(\ell) = [\mathsf{class} = \mathit{id}_s;\ \mathsf{fields} = @\mathit{id}_1 : v_1, \ldots, @\mathit{id}_n : v_n] \qquad @\mathit{id} \notin \{@\mathit{id}_1, \ldots, @\mathit{id}_n\}}{P \vdash \langle A, H, @\mathit{id}\rangle \to \langle A, H, \texttt{nil}\rangle}$$

If-T

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

If-F

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

Seq

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

Id-W

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

Field-W

$$\frac{\begin{array}{c} P \vdash \langle A, H, E\rangle \to \langle A', H', v\rangle \\ A(\texttt{self}) = \ell \qquad H'(\ell) = [\mathsf{class} = \mathit{id}_s;\ \mathsf{fields} = F] \qquad H'' = H'[\ell \mapsto [\mathsf{class} = \mathit{id}_s;\ \mathsf{fields} = F[@\mathit{id}_f : v]]] \end{array}}{P \vdash \langle A, H, @\mathit{id}_f = E\rangle \to \langle A', H'', v\rangle}$$

New

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

Invoke

$$\frac{\begin{array}{c} P \vdash \langle A, H, E_0\rangle \to \langle A_0, H_0, \ell\rangle \qquad H_0(\ell) = [\mathsf{class} = \mathit{id}_s;\ \mathsf{fields} = \ldots] \\ P \vdash \langle A_0, H_0, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad \ldots \qquad P \vdash \langle A_{n-1}, H_{n-1}, E_n\rangle \to \langle A_n, H_n, v_n\rangle \\ \texttt{lookup\_meth}\ P\ \mathit{id}_s\ \mathit{id}_m = (\text{def}\ \mathit{id}_m\ (\mathit{id}_1, \ldots, \mathit{id}_n)\ E\ \text{end}) \qquad k = n \\ A' = \texttt{self} : \ell, \mathit{id}_1 : v_1, \ldots, \mathit{id}_k : v_k \qquad P \vdash \langle A', H_n, E\rangle \to \langle A'', H'', v\rangle \end{array}}{P \vdash \langle A, H, E_0.\mathit{id}_m(E_1, \ldots, E_n)\rangle \to \langle A_n, H'', v\rangle}$$

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

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.

- 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.

- 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. 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.

- The rule New creates a new instance of a class *id*; note that 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. 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 (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.

  In your implementation, you should call `to_s` on v and print out the resulting string when the program exits.

Errors

For grading purposes, there are three sorts of errors you should handle by raising the `Halt` exception (defined in `Ast.sml` with a specific string:

- If a program tries to call a method that does not exist, your implementation should `raise Halt "No such method"`.
- If a program tries to instantiate `Bot`, the class of `nil`, you should `raise Halt "Cannot instantiate Bot"`.
- If a program tries to instantiate a class that doesn't exist, your implementation should `raise Halt "No such class C"` where `C` is replaced by the class name.

Be sure to match the strings above *exactly* so that grading works properly. When you raise a `Halt` exception, code in `main.sml` will catch it and print the error.

For any error not on this list, your implementation may report the error in whatever way you prefer; we will not test these cases.

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 interpreter 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

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
`Bot`		*No additional methods*

Figure 4: Built-in objects and methods

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 `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.

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.

You do need to support the case of programs with classes that inherit from `Object`. But you can assume the program has no classes that inherit from `String`, `Integer`, or `Bot`.

What to Implement

You must implement the following functions in `Interp.sml`:

- `defined_class p c : rube_prog -> string -> bool`, which returns `true` if an only if class `c` is defined in program `p`.
- `lookup_meth p c m : rube_prog -> string -> string -> meth option`, which given a program `p`, a class name `c`, and a method name `m`, returns `Some m'` where `m'` is the corresponding `meth`, or `None` if there is no such method. The function `lookup_meth` should start looking in class `c` and then, if needed, recursively visit the superclasses of `c` to look for `m`.

- `run p : rube_prog -> string`, which runs a Rube program and returns the result of calling `to_s` on the value produced by the top-level expression. Code in `main.sml` will then print out the resulting string.

You will probably also want to implement some helper functions, e.g., a function to evaluation expressions.

A Big Hint

You can implement the above functions however you like. But, in interest in helping you get started, here are some definitions you are welcome to use:

```
type location = int
type ('a, 'b) mutable_alist = ('a * 'b ref) list ref
datatype rubevalue =
           RNIL
         | RINT of int
         | RSTR of string
         | RLOC of int
type id = string
type name = string
type clsname = string
type object = { class : clsname, fields : (id, rubevalue) mutable_alist }
type locals = (name, rubevalue) mutable_alist
type heap = (location, object) mutable_alist
val empty_A : unit → locals = fn () => ref []
val empty_H : unit → heap = fn () => ref []
type config = locals * heap * expr
type result = locals * heap * rubevalue
```

Note that you can solve this homework different ways, and some of those ways won't use exactly the above types.

More Hints

You can do the assignment without reading any of the following! But in case it is helpful, here's some more explanation.

The `run` Function

The Rube interpreter you will be writing is in many ways very similar to the interpreter you wrote in the Core ML homework two weeks ago. Consider the operational semantics for evaluating expressions of type `aexpr`:

$$\frac{}{\langle a, AInt(n)\rangle \rightarrow n} \qquad \frac{x \in dom(a)}{\langle a, AVar(x)\rangle \rightarrow a(x)} \qquad \frac{\langle a, ae_1\rangle \rightarrow n_1 \qquad \langle a, ae_2\rangle \rightarrow n_2}{\langle a, APlus(ae_1, ae_2)\rangle \rightarrow n_1 + n_2}$$

You might have implemented this by writing a function that, when given an environment and an `aexpr` (the things to the left of the arrow), produces an integer (the thing to the right of the arrow), by pattern matching on the different `aexpr`s:

```
fun aeval (a:env) (ae:aexpr): int =
   let fun eval (AInt n) = n
         | eval (AVar x) =
           (case find x a
              of (SOME n) => n
```

```
                  | NONE => raise Not_found)
        | eval (APlus (ae1, ae2)) = eval ae1 + eval ae2
  in eval ae end
```

This is a very common pattern when writing interpreters, and one we suggest emulating for this homework. For example, here's some code to get you started. (Warning, this code is not complete!)

```
fun eval p (A, H, ENil) = (A, H, RNIL)
  | eval p (A, H, EInt n) = (A, H, RINT n)
    ... define one pattern per opsem rule

fun run (p:rube_prog): string =
    let val {prog_clss=cls, prog_main=e} = p
        val A = ... define initial locals environment
        val H = ... define initial heap environment
            ... other necessary setup code
        val (A', H', v) = eval p (A, H, e)
        val s = (* convert v to a string *)
    in
      s
    end
```

Notice the type of `eval` mirrors the type of the operational semantics judgment: it takes a program and a configuration, and produces a new configuration.

Built-in Methods

Recall from above that, just like Ruby, every value is treated as an object, including the built-in integers, strings, and `nil`. However, if you implement built-ins using the Big Hint above, you'll notice that `RNIL`, `RINT`, and `RSTR` are not stored as locations that point to the heap; rather, they are stored directly.

For example, suppose we aree trying to evaluate `EInvoke(EInt 1, "+", [EInt 2])`. Your interpreter will evaluate the receiver to `RINT 1` and argument to `RINT 2`. But then how do we run dynamic dispatch on receiver `RINT 1`? We can't do the usual thing and look up the method in the program text, because it's not there.

There are many different solutions to this problem. However, here is a simple one. Inside your handling of `EInvoke`, you can add special cases for the built-ins, something like this:

```
fun eval p (A, H, EInvoke ...) =
  (* meth = method name as a string *)
  let val v_receiver = ... in      (* receiver as a value *)
  let val v_args = List.map ... in    (* args as a value list *)
  case (v_receiver, meth, v_args) of
      (RINT n, "+", [RINT m]) => RINT (n+m)        (* Handle + on integers *)
    | (RINT n, "+", _) => error                    (* Int + with any other args is an error *)
    | (RSTR s, "+", [RSTR s']) => RSTR (s ^ s')    (* Handle + on strings *)
    | (RSTR s, "+", _) => error                    (* String + on any other args is an error *)
    | ...
    | (_, _, _) => (* general dynamic dispatch case *)
```

Note that the one downside here is that it's not very extensible: the code for built-ins is mixed into the code for `eval`. That's completely fine for this homework. But if you're looking for a challenge, consider whether you can do better!

What to Submit

Submit your edited `Interp.sml`.