Homework 10

Introduction

In this homework, you will add a static type checking system to the Rube programming language. Recall the formal syntax for Rube programs, shown in Figure 1, but extended with type annotations, discussed below. This homework starts with roughly the same codebase that Homework 9 started from.

Homework Structure

The homework 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
`Types.sml`	The type checker—**this is where you'll write your code**
`main.sml`	The main program

You will only change `Types.sml`. You should not edit any of the other files. The file `main.sml` includes code to run the parser and then dispatch to the type checker. Right now, the "type checker" implementation just rejects the input as ill-typed, e.g., after running `make` you can run:

```
$ ./tc r1.ru
no
```

to parse `r1.ru` (from the last homework). The `no` indicates that this program does not type check. In fact, the current implementation will always print `no`, until you implement your type checker.

Rube with Type Annotations

Figure 1 gives the syntax of Rube extended with type annotations. In the revised language, classes begin with field definitions, which include types, followed by method definitions, which include type annotations on arguments and the method return (this type is to the left of the method name). We also introduce a new expression $(E : T)$ that performs a dynamic type cast to check at run time that E has type T. (But you're only implementing the type checker, so you won't actually do this dynamic check—your code will trust type casts.)

Types T are simply class names. Our system will include distinguished classes `Bot`, the class of `nil` (which can masquerade as an instance of any class) and `Object`, the root of the class hierarchy.

Finally, we give a definition of *method types* MT of the form $(T_1 \times \cdots \times T_n) \to T$, which is a method such that its ith argument has type T_i and it returns type T. Method types are *not* allowed to appear in the surface syntax, but they are handy to have during type checking.

Abstract syntax trees Figure 2 shows the SML abstract syntax tree data types for programs with type annotations. It is quite similar to the grammar for the untyped variant of the language. The only difference in the expression language is the addition of texttttECast(e,t), which represents a type cast.

A type `typ` is just a string, wrapped in the constructor `TClass`, e.g., `TClass "Bot"` is the type of `nil`. We added a constructor here to make it slightly harder to mix up strings corresponding to types with other strings. We also define `mtyp` for method types; we'll use this as the return type of a key function below.

P	::=	$C^*\ E$	Rube program
C	::=	class $id <$ id begin $F^*\ M^*$ end	Class definition
F	::=	$@id : T$	Field type
M	::=	def T id $(id : T, \ldots, id : T)$ L^* begin E end	Method definition
L	::=	$id : T$	Local variable type
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
	|	$(E : T)$	Type cast
T	::=	id	Types
MT	::=	$(T \times \cdots \times T) \to T$	Method type

Figure 1: Rube syntax (without types)

A method `meth` is a record containing the method name, return type, arguments (with types), local variable definitions, and method body. A class `cls` is a record containing the class name, superclass, fields, and methods. Finally, a program `prog` (which we renamed from `ruby_prog`) is a record containing the list of classes and the top-level expression.

Part 1: Typing Utilities

To implement type checking for Rube with type annotations, we'll need the following utility functions. You should write these functions in `Types.sml`. **Important:** We will test your code by calling these functions directly, so it's important you put them in the right file.

In the following functions, you should assume the existence of four built-in classes: `Object`, `Integer`, `String`, and `Bot`, where `Object` is the root of the inheritance hierarchy, and `Integer` and `String` extend `Object`. The class `Bot` is the bottom type, and it also has all the methods of `Object`. Figure 3 gives type signatures for built-in methods of these classes; you should assume these built-in methods exist. As in the previous homework, 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`.

1. Write a function `defined_class p c : prog -> string -> bool` that returns true if and only if class `c` is defined in program `p`. This function should return true if asked whether `Object`, `Integer`, `String`, or `Bot` are defined.

2. Write a function `no_builtin_redef : prog -> bool` that returns true if the program does not try to define classes `Object`, `String`, `Integer`, or `Bot`. (Note that we have not specified what to do for programs that redefine user classes; our test cases will never do so, and so it's up to you how to handle such cases.)

3. Write a function `lookup_meth p c m : prog -> string -> string -> mtyp` that returns the type (an `mtyp`) of method `m` in class `c` or, if that method is not defined, it should return the type from `c`'s superclass (and so on, recursively up the class hierarchy). This function should `raise Not_found` if

```
datatype expr =
    EInt of int
  | ENil
  | ESelf
  | EString of string
  | ELocRd of string      (* Read a local variable *)
  | ELocWr of string * expr   (* Write local variable *)
  | EFldRd of string      (* Read a field *)
  | EFldWr of string * expr   (* Write a field *)
  | EIf of expr * expr * expr
  | ESeq of expr * expr
  | ENew of string
  | EInvoke of expr * string * expr list
  | ECast of expr * typ
```

```
and typ = TClass of string

type mtyp = (typ list) * typ

type meth = { meth_name : string,
              meth_ret: typ,
              meth_args : (string * typ) list,
              meth_locals : (string * typ) list,
              meth_body : expr }

type cls = { cls_name : string,
             cls_super : string,
             cls_fields : (string * typ) list,
             cls_meths : meth list }

type prog = { prog_clss : cls list,
              prog_main : expr }
```

Figure 2: Abstract syntax tree for Rube with type annotations

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 3: Built-in objects and methods

no such method exists in `c` or in any superclass. The types of built-in methods in `Object`, `String`, `Integer`, and `Bot` **should be returned** by this function. Also, if a class inherits a method from a superclass, the inherited method's type should be returned. (Note that we have not specified any particular behavior if the same method is defined twice within one class; our test cases will never do so.)

4. Write a function `lookup_field p c f : prog -> string -> string -> typ` that returns the type of field `f` in class `c`. As with `lookup_meth`, it should recursively explore superclasses to find field definitions if necessary. This function should `raise Not_found` if `f` is not defined in `c` or in any of its superclasses. (Note that we have not specified any particular behavior if the same field is defined twice within one class; our test cases will never do so.)

Part 2: Subtyping

Since our language includes subclassing, we will need to define a subtyping relationship as part of type checking. Figure 4 shows the subtyping rules for this language. In words, the rules are as follows:

- Bot states that the bottom type `Bot` is a subtype of any other type.

$$\textsc{Bot}\ \frac{}{P \vdash \texttt{Bot} \leq T} \qquad \textsc{Obj}\ \frac{}{P \vdash T \leq \texttt{Object}} \qquad \textsc{Class}\ \frac{(\text{class } id_1 < id_2 \text{ begin } \ldots \text{ end}) \in P}{P \vdash id_1 \leq id_2}$$

$$\textsc{Trans}\ \frac{P \vdash id_1 \leq id_2 \qquad P \vdash id_2 \leq id_3}{P \vdash id_1 \leq id_3} \qquad \textsc{Refl}\ \frac{}{P \vdash id \leq id}$$

Figure 4: Subtyping

- Obj states that any type is a subtype of `Object`.
- Class says that id_1 is a subtype of id_2 if id_1 is a subclass of id_2 in the program text.
- Trans says that subtyping of classes is transitive, and Refl says that subtyping is reflexive.

Write a function `is_subtype p t1 t2 : prog -> typ -> typ -> bool` that returns true if and only if type `t1` is a subtype of `t2` according to this definition. Put `is_subtype` in `Types.sml`.

Part 3: Type Checking

Finally, you must write a function `tc_prog p : prog -> unit` that returns successfully if and only if `p` type checks, and otherwise it raises an exception. Put this function in `Types.sml`. Figure 5 gives the static type checking rules that you will translate into code. Here A is a *type environment*, which as discussed in class is an associative list mapping local variables to their types. Most of the rules have the form $P; A \vdash E : T$, meaning that in program P with environment A, expression E has type T. We'll explain the other kinds of rules as we encounter them. 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 have the obvious types.
- The *local variables* of a method include the parameters of the current method, locals defined at the top of the method, and `self`, which refers to the object whose method is being invoked. The rules Self and Id say that `self` or the identifier *id* has whatever type is assigned to it in the environment A. If `self` or *id* is not bound in the environment, then this rule doesn't apply—and hence your type checker would signal an error.
- The rule Field-R says that when a field is accessed, it has whatever type we get by looking it up in the program according to the `lookup_field` function you already wrote, finding the current class by looking up the type of `self`. Notice that unlike in untyped Rube, it is an error to refer to fields that have not been pre-defined. Also notice that like Ruby, only fields of `self` are accessible, and it is impossible to access a field of another object.
- The rule If says that to type a conditional, the three sub-expressions must all be well-typed, and both branches must have the same types, which is the type of the `if`. (Programmers could insert a type cast to ensure this holds. It would also be possible to have a more general rule for `if`.)
- The rule Seq says that the type of E_1; E_2 is the type of E_2. Notice that this rule requires E_1 to be well typed, but it doesn't matter what that type is.
- The rule Id-W says that a write to a local variable is well-typed if the type of the right-hand side of the assignment is a subtype of the variable type. Notice that unlike untyped Rube, it is an error to write to a variable that hasn't been defined as either a parameter or local. Notice also that it is an error to

$$\textsc{Int}\ \frac{}{P; A \vdash n : \texttt{Integer}} \qquad \textsc{Nil}\ \frac{}{P; A \vdash \texttt{nil} : \texttt{Bot}} \qquad \textsc{Str}\ \frac{}{P; A \vdash \texttt{"}str\texttt{"} : \texttt{String}} \qquad \textsc{Self}\ \frac{}{P; A \vdash \texttt{self} : A(\texttt{self})}$$

$$\textsc{Id}\ \frac{id \in dom(A)}{P; A \vdash id : A(id)} \qquad \textsc{Field-R}\ \frac{A(\texttt{self}) = id_{\texttt{self}}}{P; A \vdash @id : (\texttt{lookup\_field}\ P\ id_{\texttt{self}}\ @id)}$$

$$\textsc{If}\ \frac{P; A \vdash E_1 : T \qquad P; A \vdash E_2 : T' \qquad P; A \vdash E_3 : T'}{P; A \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \texttt{end} : T'}$$

$$\textsc{Seq}\ \frac{P; A \vdash E_1 : T_1 \quad P; A \vdash E_2 : T_2}{P; A \vdash (E_1;\ E_2) : T_2} \qquad \textsc{Id-W}\ \frac{P; A \vdash id : T \qquad P; A \vdash E : T' \qquad P \vdash T' \leq T}{P; A \vdash id = E : T}$$

$$\textsc{Field-W}\ \frac{P; A \vdash @id : T \qquad P; A \vdash E : T' \qquad P \vdash T' \leq T}{P; A \vdash @id = E : T} \qquad \textsc{New}\ \frac{\texttt{defined\_class}\ P\ id}{P; A \vdash \texttt{new}\ id : id} \qquad \textsc{Cast}\ \frac{P; A \vdash E : T'}{P; A \vdash (E : T) : T}$$

$$\textsc{Invoke}\ \frac{\begin{array}{c}P; A \vdash E_0 : T_0' \quad \cdots \quad P; A \vdash E_n : T_n' \\ (\texttt{lookup\_meth}\ P\ T_0'\ id_m) = (T_1 \times \cdots T_k) \to T \\ k = n \qquad T_1' \leq T_1 \quad \cdots \quad T_n' \leq T_n\end{array}}{P; A \vdash E_0.id_m(E_1, \ldots, E_n) : T}$$

$$\textsc{Method}\ \frac{P; (L^*, id_1 : T_1, \ldots, id_n : T_n, \texttt{self} : id_c) \vdash E : T' \qquad P \vdash T' \leq T}{P; id_c \vdash \text{def}\ T\ id\ (id_1 : T_1, \ldots, id_n : T_n)\ L^*\ \text{begin}\ E\ \text{end}}$$

$$\textsc{Class}\ \frac{\begin{array}{c}P; id \vdash M_1 \quad \cdots \quad P; id \vdash M_n \\ id \notin \{\texttt{Object}, \texttt{Integer}, \texttt{String}, \texttt{Bot}\}\end{array}}{P \vdash \text{class}\ id < id'\ \text{begin}\ F^*\ M_1\ \cdots\ M_n\ \text{end}} \qquad \textsc{Program}\ \frac{P = C_1 \cdots C_n\ E \qquad P \vdash C_1 \quad \cdots \quad P \vdash C_n \qquad P; \cdot \vdash E : T}{\vdash P}$$

Figure 5: Static Type Rules

write to the local variable `self` (which is implicitly syntactically distinct from the non-terminal *id*), and your implementation should signal an error in this case.

- Similarly, the rule FIELD-W says that a field write is well-typed if the type of the right-hand side is a subtype of the field type. Again, unlike untyped Rube, fields must be defined with types prior to writing to them.

- The rule NEW says that a `new` expression is well-typed if the class being constructed exists in the program according to the `defined_class` function you wrote earlier. Notice that the class can appear *anywhere* in the program—it could be listed before the current class definition or after.

- The rule CAST says that an expression can be type cast to any type. The resulting type of the cast is the cast-to type. (Type casts should be checked at run-time, but since we're not modifying the interpreter for this homework we won't worry about that.)

- The rule INVOKE says that for a method invocation to be well-typed, the receiver object expression and all arguments must be well-typed. Also, the types of the arguments must be subtypes of the method type we find by looking up id_m in whatever class corresponds to the receiver object, using the `lookup_meth` function you wrote earlier.

  Notice that we don't need to do anything else here, e.g., we don't need to look inside the method body.

That's because we'll separately check that the method actually has the type the `lookup_meth` function says it has. Neat!

- The rule METHOD says that for a method definition to be well typed inside of class id_c, it must be that if we typecheck the method body with locals assigned their types as given, parameters assigned their types as given, and `self` given type id_c, the body of the method has a type T' that is a subtype of the declared method return type. Note the order here says that locals may shadow parameters, which may shadow `self`; however, we will *not* test whether you implement this shadowing.

- The rule CLASS says that for a class definition to be well typed, and all its method definitions must be well-typed. There must also be no definitions of `Object`, `Integer`, `String`, or `Bot`.

  (Note, we are leaving out one check—that a subclass's type annotations are consistent with its superclass's type annotations—to make the homework slightly shorter.)

- Finally, rule PROGRAM says that for a program to be well-typed, all of its classes must be well-typed, and its top-level expression must be well-typed under the empty environment. This is slightly different than the last homework, in which the top-level expression did have `self` bound to an object. This means that in Typed Rube, the top-level expression cannot even refer to `self` or fields!

What to Write

You'll need to write at least the following types and functions:

- `tc_prog p : prog -> unit` to type check the whole program. This function returns the unit value `()` if the program type checks. Otherwise it raises an exception (any exception).

- `type env = (string * typ) list` for representing local variable environments. Notice that you don't need mutable environments here because types for locals and fields never change (unlike their values, which could change).

- `tc_expr : prog -> env -> expr -> typ` to type check an expression. This function should return a type if the expression type checks, and otherwise it should raise an exception.

- `tc_meth : prog -> string -> meth -> unit` to type check a method against its type signature (i.e., it implements rule METHOD).

Your type checking functions can raise any exception to signal that a program is ill-typed. We've supplied the exception `Type_error` as a convenient exception to throw, but you're not required to use it. In `main.sml`, there's code that calls your `tc_prog` function and either prints `yes` or `no`, depending on whether `tc_prog` type checked the input or found a type error (i.e., raised an exception), respectively.

During development, you may wish to comment out the exception handling code in `main.sml` so that you can see exactly which exception is being raised.