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:

`.submit`	The usual submit file
`Makefile`	A file for building this project
`OCamlMakefile`	A helper for `Makefile`
`ast.mli`	A signature describing Rube abstract syntax trees
`lexer.mll`	A lexer for tokenizing Rube programs
`parser.mly`	A parser for parsing Rube programs (uses the lexer)
`rube.ml`	**The file you need to edit**
`r1.ru`	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.rb`, 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.

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

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

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* $\{method_1 \ldots method_n\}$, where the $method_i$ are the methods defined in the class. Finally, to represent object values, we write $[\mathsf{class} = v_0;\ \mathsf{fields} = @id_1 : v_1, \ldots, @id_n : v_n]$, 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 {`def m(a, b) @x = a.+(b); @x end`} and `x` evaluates to $[\mathsf{class} = v_C;\ \mathsf{fields} = \emptyset]$ 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 $[\mathsf{class} = v_C;\ \mathsf{fields} = @x : 3]$.

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 $id_1 : v_1, \ldots, id_n : v_n$, 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.

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