

# Project 5

Due November 30, 2006

11:59:59pm

## Updates

- Nov 25. Minor typo fix; there was a stray “right” that should have been “left” with respect to shadowing.
- Nov 24. Typo fix—“find\_class” should have been “find\_method”.
- Nov 21. Two-day extension. The project is now due Nov 30.
- Nov 20. Fix: Shadowing in environments is now **left-to-right**. There was an inconsistency in the project writeup.
- Nov 20. Clarification: Rube programs should never contain a definition of class `Object`—the `Object` class is implicitly part of all Rube programs. (You don’t need to complain if the programmer has a definition of `Object`, but you should not assume that there is one.)

## 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. The hardest part of this project is dealing with regular expressions, which we discuss towards the end.

## What to Submit

We’ve left a `p5.tar.gz` file in the usual place. This time, this directory contains several files:

<code>.submit</code>	The usual submit file
<code>Makefile</code>	A file for building this project
<code>OCamlMakefile</code>	A helper for <code>Makefile</code>
<code>ast.mli</code>	A signature describing Rube abstract syntax trees
<code>lexer.mll</code>	A lexer for tokenizing Rube programs
<code>parser.mly</code>	A parser for parsing Rube programs (uses the lexer)
<code>nfa.mli</code>	Interface for NFAs
<code>nfa.ml</code>	Implementation for NFAs (we’ll give you this later)
<code>rube.ml</code>	<b>The file you need to edit</b>
<code>r1.ru</code>	A sample Rube program

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

<i>prog</i>	::=	<i>class...class expr</i>	
<i>class</i>	::=	<b>class</b> <i>id</i> < <i>id</i> <i>method...method</i> <b>end</b>	
<i>method</i>	::=	<b>def</b> <i>id</i> ( <i>id</i> , ..., <i>id</i> ) <i>expr</i> <b>end</b>	
<i>expr</i>	::=	<i>n</i>	Integers
		<b>true</b>	Boolean true value
		<b>false</b>	Boolean false value
		" <i>str</i> "	String
		/ <i>str</i> /	Regular expression
		<i>id</i>	Local variable
		<b>if</b> <i>expr</i> <b>then</b> <i>expr</i> <b>else</b> <i>expr</i> <b>end</b>	Conditional
		<i>expr</i> ; <i>expr</i>	Sequencing
		<b>new</b> <i>id</i>	Object creation
		<i>expr</i> . <i>id</i>	Field read
		<i>expr</i> . <i>id</i> = ( <i>expr</i> )	Field write
		<i>expr</i> . <i>id</i> ( <i>expr</i> , ..., <i>expr</i> )	Method invocation

Figure 1: Rube syntax

```
% ./rube < r1.ru
class Foo < Object {
  def mymethod()
    true
  end
}
```

```
(new Foo).mymethod()
```

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.

## Rube: Syntax and Semantics

The formal syntax for Rube programs is shown in Figure 1. A Rube program (given by the nonterminal *prog*) is made up of a list of class definitions followed by a single expression. When a program executes, the class definitions are read in, and then the expression, which may use the class definitions in the program, is evaluated. Whatever the expression evaluates to is the result of the program. For example, the Rube program in `r1.ru` defines a class `Foo` that inherits methods from `Object`, a built-in class with no methods. `Foo` has a single method, `mymethod`, that takes no arguments and evaluates to `true`. When the program is run, the expression `(new Foo).mymethod()` invokes the method, which evaluates to `true`, which is what the program returns.

Here is how a Rube program is executed:

- A class definition “`class id1 < id2 ... end`” defines a class named *id*<sub>1</sub> that inherits methods from *id*<sub>2</sub> (fields are not inherited in Rube). The body of a class consists of a list of method definitions, followed

by the keyword `end`. All Rube programs have an implicit class `Object` containing no methods, and a Rube program may not attempt to redefine the class `Object`.

- A method definition has the form “`def id(id1, ..., idn) expr end`”. Here `id` is the name of the method, and `id1` through `idn` are the names of the formal arguments to the method. The method evaluates to whatever its body `expr` evaluates to.
- To run a program, its top-level expression is evaluated. Top-level expressions may not refer to any local variables (i.e., identifiers or the special identifier `self`).

We can define 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:

$$v ::= n \mid \text{true} \mid \text{false} \mid \text{"str"} \mid /re/ \mid \{\text{class} = id_0; \text{fields} = id_1 : v_1, \dots, id_n : v_n\}$$

Values include integers  $n$ , the boolean values `true` and `false`, strings `"str"`, and regular expressions `/re/`. We’ve used a slightly different font here to emphasize the difference between program text, such as `true`, and what it evaluates to, `true`.

To represent object values, we write  $\{\text{class} = id_0; \text{fields} = id_1 : v_1, \dots, id_n : v_n\}$ , which is an instance of class `id0` that has fields `id1` through `idn`, and field `idi` has value  $v_i$ . For example, `new Foo` would evaluate to  $\{\text{class} = \text{Foo}; \text{fields} = \emptyset\}$ , where by  $\emptyset$  we mean the object has no fields initially. As another example, if we write `(new Foo).f = (true)` (which updates the left-hand side of the assignment and returns it), then we would get back an object  $\{\text{class} = \text{Foo}; \text{fields} = \text{f} : \text{true}\}$ .

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  $id_1 : v_1, \dots, id_n : v_n$ , and names on the *left* shadow names on the *right*, as in lecture. 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 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 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).<sup>1</sup> Here is what the rules mean:

- The rules `INT`, `TRUE`, `FALSE`, `STR`, and `REGEXP` all say that an integer, boolean, string, or regular expression 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.) Regular expressions are like strings, except they begin and end with `/` and may not contain `/` in the middle.
- The *local variables* of a method include the parameters of the current method and `self`, which refers to 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.

Note that unlike Ruby, the only local variables are `self` and the parameters of the current method, and local variables cannot be updated. There are no local variables for the top-level expression of a program, which is evaluated outside of a method.

- 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 `true` or `false`, we return either the then or else branch. Notice that by our order-of-evaluation assumption, we evaluate the guard before either the then or else branch. If the guard does not evaluate to a boolean, these rules don’t apply, and so again your interpreter would signal an error.

---

<sup>1</sup>There’s actually a better way to specify the order of evaluation in operational semantics, but it would complicate the rules.

$$\begin{array}{c}
\text{INT} \\
\frac{}{A; n \rightarrow n} \\
\\
\text{TRUE} \\
\frac{}{A; \text{true} \rightarrow \text{true}} \\
\\
\text{FALSE} \\
\frac{}{A; \text{false} \rightarrow \text{false}} \\
\\
\text{STR} \\
\frac{}{A; \text{"str"} \rightarrow \text{"str"}} \\
\\
\text{REGEXP} \\
\frac{}{A; /str/ \rightarrow /str/} \\
\\
\text{ID} \\
\frac{id \in \text{dom}(A)}{A; id \rightarrow A(id)} \\
\\
\text{IF-T} \\
\frac{A; \text{expr}_1 \rightarrow \text{true} \quad A; \text{expr}_2 \rightarrow v}{A; \text{if } \text{expr}_1 \text{ then } \text{expr}_2 \text{ else } \text{expr}_3 \text{ end} \rightarrow v} \\
\\
\text{IF-F} \\
\frac{A; \text{expr}_1 \rightarrow \text{false} \quad A; \text{expr}_3 \rightarrow v}{A; \text{if } \text{expr}_1 \text{ then } \text{expr}_2 \text{ else } \text{expr}_3 \text{ end} \rightarrow v} \\
\\
\text{SEQ} \\
\frac{A; \text{expr}_1 \rightarrow v_1 \quad A; \text{expr}_2 \rightarrow v_2}{A; \text{expr}_1; \text{expr}_2 \rightarrow v_2} \\
\\
\text{NEW} \\
\frac{id \text{ is a valid class name}}{A; \text{new } id \rightarrow \{\text{class} = id; \text{fields} = \emptyset\}} \\
\\
\text{FIELD-R} \\
\frac{A; \text{expr} \rightarrow \{\text{class} = \dots; \text{fields} = id_1 : v_1, \dots, id_n : v_n\} \quad id_i = id}{A; \text{expr}.id \rightarrow v_i} \\
\\
\text{FIELD-W} \\
\frac{A; \text{expr}_1 \rightarrow \{\text{class} = \dots; \text{fields} = id_1 : v_1, \dots, id_n : v_n\} \quad A; \text{expr}_2 \rightarrow v \quad \text{fields} := \text{fields}[id : v]}{A; \text{expr}_1.id = (\text{expr}_2) \rightarrow v} \\
\\
\text{INVOKE} \\
\frac{A; \text{expr}_0 \rightarrow v_0 \quad v_0 = \{\text{class} = id_c; \text{fields} = \dots\} \quad A; \text{expr}_1 \rightarrow v_1 \quad \dots \quad A; \text{expr}_n \rightarrow v_n \quad \text{method}(id_c)(id_m) : \text{def } id_m(id_1, \dots, id_k) \text{ expr end} \quad k = n \quad A' = \text{self} : v_0, id_1 : v_1, \dots, id_n : v_n \quad A'; \text{expr} \rightarrow v}{A; \text{expr}_0.id_m(\text{expr}_1, \dots, \text{expr}_n) \rightarrow v}
\end{array}$$

Figure 2: Rube Operational Semantics for Expressions

- The rule SEQ says that to evaluate  $expr_1; expr_2$ , we evaluate  $expr_1$  and then evaluate  $expr_2$ , whose value we return. Note that in the syntax, semicolon is a *separator*, and does not occur after the last expression. Thus, for example, `true; false` is an expression, but `true; false;` is not.
- The rule NEW says that `new id` produces a new instance of class  $id$  with an empty set of fields. We require that  $id$  is a valid class name, i.e., either `Object` or a class defined by the user. Unlike Ruby, there are no constructors in Rube.
- The rule FIELD-R says that to read a field of an expression, written  $expr.id$ , we first evaluate the expression to produce an object with fields  $id_i$ . Assuming one of those is equal to  $id$ , the field we're reading, then we return that field's value. If there is no such field, or if  $expr$  does not evaluate to an object, this rule does not apply, and your implementation would signal an error.

Note that unlike most object-oriented languages, fields of the current object may not be accessed implicitly—the user must explicitly write `self.f` to access field  $f$  of the current object.

- The rule FIELD-W says that to write a field, written  $expr_1.id = (expr_2)$ , we evaluate  $expr_1$  to an object, and then we evaluate  $expr_2$ . We then *update* the set of fields of the object to contain whatever it contained before, plus now  $id$  should map to value  $v$ —meaning if  $id$  was in the field set before, it is replaced to map to  $v$ , and if it wasn't in the field set before, then we add it. (Note that this is non-standard notation for updates, but the meaning should be clear.)

The parentheses on the right-hand side are required in the syntax, although they don't affect the semantics. They're there to make parsing a bit easier.

- Finally, the most complicated rule is for method invocation. We begin by evaluating the receiver  $expr_0$  to a value  $v_0$ , which must be an object. We then evaluate the arguments  $expr_1$  through  $expr_n$ , in order from 1 to  $n$ , to produce values. Then we perform method lookup. By  $method(id_c)(id_m)$ , we mean look up the method named  $id_m$  in class  $id_c$ .

We haven't defined this formally, but we mean the usual inherited method search: If class  $id_c$  was defined with a method  $id_m$ , then we retrieve that method. Otherwise we look in  $id_c$ 's parent class for a method  $id_m$ . We continue this process until we either find a method  $id_m$ , or we reach `Object`, which has no methods. If we reach `Object`, then we would signal an error.

Once we find a method `def id_m(id_1, ..., 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  $v_0$ , 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 in some ways than the OCaml semantics we showed you in class.

Rube also includes several built-in methods that may be invoked on the built-in types. Figure 3 gives the operational semantic rules for invoking methods on integers, booleans, and strings. (There are no methods for regular expressions.) From top to bottom:

- If  $expr_0$  and  $expr_1$  evaluate to integers  $n$  and  $m$ , respectively, then  $expr_0.+(expr_1)$  evaluates to  $n+m$ , and analogously for the methods `-`, `*`, and `/`.
- If  $expr$  is an integer  $n$ , then  $expr.to_s()$  evaluates to the string corresponding to the integer  $n$ . Analogously, `to_s` may be invoked with no arguments on `true` and `false`. Important: The `to_s` method does not put quotes around the string that's created. In your interpreter, `true.to_s()` should evaluate to the 4-character string `true`. The quotes are there in the semantics just to distinguish strings from other values.

$$\begin{array}{c}
\frac{A; \text{expr}_0 \rightarrow n \quad A; \text{expr}_1 \rightarrow m \quad \text{aop} \in \{+, -, *, /\}}{A; \text{expr}_0.\text{aop}(\text{expr}_1) \rightarrow n \text{ aop } m} \\
\\
\frac{A; \text{expr} \rightarrow n}{A; \text{expr}.\text{to\_s}() \rightarrow \text{"n"}} \\
\\
\frac{A; \text{expr} \rightarrow \text{true}}{A; \text{expr}.\text{to\_s}() \rightarrow \text{"true"}} \\
\\
\frac{A; \text{expr} \rightarrow \text{false}}{A; \text{expr}.\text{to\_s}() \rightarrow \text{"false"}} \\
\\
\frac{A; \text{expr}_0 \rightarrow \text{"str"} \quad A; \text{expr}_1 \rightarrow /re/ \quad \text{str} \in L(re)}{A; \text{expr}_0.\text{match}(\text{expr}_1) \rightarrow \text{true}} \\
\\
\frac{A; \text{expr}_0 \rightarrow \text{"str"} \quad A; \text{expr}_1 \rightarrow /re/ \quad \text{str} \notin L(re)}{A; \text{expr}_0.\text{match}(\text{expr}_1) \rightarrow \text{false}}
\end{array}$$

Figure 3: Rube Operational Semantics for Built-in Classes

- If  $\text{expr}_0$  evaluates to a string  $\text{str}$ , and  $\text{expr}_1$  evaluates to a regular expression  $\text{re}$ , then  $\text{expr}_0.\text{match}(\text{expr}_1)$  returns either `true` if  $\text{str}$  is an exact match for the regular expression  $\text{re}$ , and `false` otherwise. Here  $L(\text{re})$  is the language described by  $\text{re}$ .

## A Rube Interpreter

Your task is to write a Rube Interpreter that follows the operational semantics we just gave you. We've already written most of the infrastructure you'll need for your interpreter. We'll begin our description of the interpreter by looking at the file `rube.ml`, which is the file you'll be editing. If you scroll down to the bottom of that file, you'll find the main routine executed when you run `rube`:

```

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let ((cs, e):program) = Parser.main Lexer.token lexbuf in
  begin
    print_program (cs, e);
    print_string "\nEvaluates to:\n";
    print_value (eval_expr cs [] e)
  end

```

You can consider the first two lines to be magic: The first line binds the name `lexbuf` to the stream of tokens that `lexer.mll` extracts from standard input. The second line parses the stream of tokens to produce a `program`, which is an abstract syntax tree representing the Rube program. (We'll talk about the definition of this type shortly.) Then we print out the program using a function `print_program` that we've provided for you, print some separator text, and then print out the result of calling `eval_expr` on the program. This function, `eval_expr`, is the one you're going to have to write.

The type `program` represents abstract syntax trees of Rube programs, and this type is defined in the file `ast.mli`. Here is part of that file:

```
(* (method name, argument names (may be empty), method body) *)
type meth = string * string list * expr

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

(* A pair containing a list of classes in the program, plus
   an expression to execute to run the program *)
type program = cls list * expr
```

Here a method `meth` is represented as a tuple containing the method name, the argument names, and the method body (whose type we'll explain later). A class `cls` is a tuple with the class name, the superclass name, and the list of methods. And finally a `program` is a list of classes and an expression.

**Task 1: Write the Method Lookup Function** When your interpreter encounters a method invocation, it needs to find the method to invoke. Your first task is to write a function

```
val find_method : cls list -> string -> string -> meth
```

such that `find_method cs c m` looks up method `m` in class `c` in the list of classes `cs` and returns the method. If `c` has no method `m`, then `find_method` should look for `m` in `c`'s superclass, and it should continue up the class hierarchy until it reaches `Object`, at which point it should stop and raise the exception `Class_not_found` (which we've defined for you).  $\square$

Your Rube interpreter is going to take ASTs as input and produce values as output, just like the operational semantics. The type `value` is defined at the top of `rube.ml`:

```
type value =
  | VInt of int
  | VBool of bool
  | VString of string
  | VRegexp of string * nfa
  | VObject of string * (string * value) list ref
  (* Invariant: no field name appears twice in a VObject *)
```

The value of a Rube program can be either an integer, a boolean, a string, a regular expression (the left part of the tuple is the string representation of the regexp, and the right part is an `nfa`), or an object. The object `VObject(s, fields)` represents an instance of the class `s`. Since the fields of objects can be updated over time, `fields` is an updatable reference containing a list of string and `value` pairs. For example, when creating a new instance of class `c`, the result should be `VObject(c, ref [])`, which is an object with no fields. As another example, an instance of `c` with one field `f` that has value `true` would be `VObject(c, ref [(f, true)])`.

**Task 2: Write a Function to Convert values to Strings** Recall that the main Rube interpreter prints out the value that your program evaluates to. Your next task is to write a function

```
val value_to_string : value -> string
```

that, given a `value`, returns the corresponding `string`. More precisely:

- The integer `n` should be converted to the string representation of `n`. For example, `value_to_string (VInt 3) = "3"`. (That's the string containing the single character `3`—the quotes are not part of the string.)
- The boolean `true` should be converted to the string `true`, and `false` should be converted to `false`.

- A string should be converted to itself.
- A regular expression should be converted to the string representation of the regular expression in between /'s. In other words, `value_to_string (VRegex(r, _))` should be a /, followed by `r`, followed by a /. You should ignore the `nfa` part of the regular expression when converting them to strings.
- Finally, an object `VObject(c, ref [(f1,v1); ...; (fn,vn)])` should be converted to a string of the form “`{c f1=v1, ..., fn=vn}`”. If there are no fields in the object, then the `}` should appear immediately to the right of `c`. Otherwise, there should be one space after `c`, and the string should contain the name of each field, followed by `=`, followed by the string value of the field (computed via a recursive call). Fields should be separated by a comma followed by a space before the next field. There should be no comma or space to the right of the right-most field.

Hint: The code to unparse expressions should help you quite a bit for this part. In that code we used OCaml's `Printf` module. You're welcome to use that, though it's a bit complex, so you may want to stick to the ordinary string conversion functions (e.g., `string_of_bool` etc). □

Finally we get to the interesting part: running a Rube program. Here is the definition of type `expr` from `ast.mli`:

```
type expr =
  EInt of int
| EBool of bool
| EString of string
| ERegexp of string
| ELocal of string
| EIf of expr * expr * expr
| ESeq of expr * expr
| ENew of string
| ERead of expr * string
| EWrite of expr * string * expr
| EInvoke of expr * string * (expr list)
```

Notice there is a suspicious similarity between these expressions and the grammar for Rube we gave you in Figure 1. The first four constructors should be self-explanatory. The expression `ELocal s` represents the local variable `s`. Notice in our abstract syntax tree, we use strings for the names of local variables. 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 `ERead(e,s)` corresponds to `e.s`, and the expression `EWrite(e1,s,e2)` corresponds to `e1.s=(e2)`. Lastly, `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.)

**Task 3: Write a Function to Evaluate Rube Expressions** Your next task is to write a function

```
val eval_expr : cls list -> (string * value) list -> expr -> value
```

that evaluates a Rube expression. From left-to-right, the arguments are: The list of classes in the program; the local variable environment in which to evaluate the expression; and the expression itself. The result of this function is a Rube `value`. For now, ignore the cases for regular expressions; we'll get to those below.

Your function should raise the exception `Eval_error`, which we've defined, if it detects an error during evaluation (such as calling a method with the wrong number of arguments, or using an integer in the guard of an if expression).

You'll use the list of classes to look up methods when you get to a method invocation. Your interpreter will need to handle specially the cases for invoking methods on integers, booleans, and strings. You'll use the environment to look up the value of a local variable when you encounter one. The environment is empty

when evaluating the top-level expression in a program. When you're inside of a method, the environment should contain a binding for `self`, the current object, and for each of the formal arguments. You'll need to create this environment yourself when you encounter a method call. In the environment, identifier bindings earlier in the list should override identifier bindings later in the list.  $\square$

## Regular Expressions in Rube

For the last part of this project, you'll extend the Rube interpreter to handle regular expressions. Recall that Rube strings have a `match` method to check whether they are in the language defined by a regular expression. The type of regular expressions given to you by the parser is `ERegexp` of `string`, where the string is the textual representation of the regular expression that the user passes in. In order to implement the `match` function, you'll need to convert these regular expressions into NFAs, and then you can check if the string is accepted by the NFA. You can reuse your solution to project 4 for this part or, if you prefer, you can use our solution, which we will give you shortly. When we grade your solution to this part of the project, we'll use our NFA code.

**Task 4: Parse regular expressions** In project 4, you wrote a function to convert a `regexp` AST datatype into an NFA. The Rube interpreter, however, only gives you a string representation of regular expressions. Your next task is to write a function

```
val string_to_regexp : string -> Nfa.regexp
```

Your function should produce an instance of the `regexp` datatype from project 4 (here put in interface `nfa.mli`), given a string.

$r ::=$	$\varepsilon$	Empty regexp
	<b>a</b>	Single character
	$r r$	Union
	$rr$	Concatenation
	$r^*$	Kleene Closure
	$(r)$	Parentheses

This grammar is ambiguous, and you should use the following conventions to disambiguate it:

- Union `|` is right-associative and has the lowest precedence.
- Concatenation is right-associative and has higher precedence than union.
- Kleene star has the highest precedence

For example, you should parse `ab|c` as `(ab)|c`, and you should parse `abc*` as `ab(c*)`.  $\square$

Now that you can convert strings to `regexps`, you can add regular expressions to your interpreter.

**Task 5: Rube Regular Expressions** Your final task is to add code to your interpreter to handle regular expressions. Specifically, a regular expression `ERegexp s` should evaluate to the value `VRegexp(s,n)`, where `n` is the `nfa` for `s` you get by first converting the string to a `regexp`, and then converting the `regexp` to an `nfa`. Then implement the `match` method of strings, using `Nfa.accept`  $\square$

## 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.