Homework 9: Rackette-cita
Due: 11:59 PM, Nov 6, 2018

Contents

1 Introduction 2

2 Read 3
   2.1 Read in Racket ................................................. 4
   2.2 Read in OCaml .................................................. 4

3 Making Expression Trees 5
   3.1 Rackete-cita .................................................... 6
   3.2 Error Checking .................................................. 7

4 Arithmetic Evaluator 8

5 Print 8

6 REPL 9

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Make Expression Tree, Arithmetic Evaluator, and Print questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows:

- README.txt
- CS17setup.ml
- read.ml
- make_expression_tree.ml
- arithmetic.ml
- print.ml
For this assignment, all files you turn in that contain code must be OCaml files, so they must end with extension .ml. For all coding problems in this homework, you should follow the OCaml design recipe outlined in the previous homework.

For this and every future assignment, you should include the CS17setup.ml file with your hand-in. You should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the hand-in. For example:

```plaintext
README.txt:
klein@brown.edu
There’s nothing to say except that I’m turning in these files plus this README the way the instructions say that I should.
```

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files. If you choose to also store these files on department machines, all your solution files should reside in your ~/course/cs0170/homeworks/hw09 directory.

Set-Up

To start you’ll want to copy the CS17setup file and the read source file into your hw09 directory. On the department system, this would look something like this:

```bash
cp /course/cs017/src/ocaml/CS17setup.ml ~/course/cs017/homeworks/hw09
cp /course/cs017/src/hw09/read.ml ~/course/cs017/homeworks/hw09
```

Subsequently create and write the other required files corresponding to the three problems in this assignment, and make sure, whether you’re working on a department machine or your own computer, that all solution files and these first two are in the same folder. Do not modify these files in any way.

Problems

1 Introduction

We’ll call a program that consumes another program and “runs” it an interpreter. Your next project will be to write an interpreter for a subset of Racket we call Rackette. Assuming that you had some way to represent a Rackette expression in Racket, you might, as a first step in writing a part of a Rackette interpreter (in Racket), namely the evaluator, do something like this in an attempt to identify the expression as an “if expression”:

```racket
(define evaluate
  (lambda (expr)
    (cond
      [(number? expr) ... ])
```

[^1]: Take CS173 to learn why this isn’t the only possibility.
This is the right idea, but seven lines of predicates just to check if something is an `if` expression seems a bit excessive! Wouldn’t it be nice to know that `evaluate` is always applied to a syntactically valid expression?

You can achieve exactly that by breaking down the interpretation process into two phases:

1. The first phase attempts to make the program into an expression tree (i.e., an internal representation), producing an error if it finds the program to be invalid.

2. The second phase processes the expression tree (which is by definition syntactically valid) and produces a value, expressed in another internal representation.

Rackette programs will consist of zero or more definitions, followed by zero or one expressions, and processing the expression, if there is one, is called **evaluation**.

In processing Rackette, we have both the program and an environment. Definitions change the environment, and evaluating expressions involves looking things up in the environment.

In this homework, there is **no environment**. We are dealing with a language so simple that we don’t need one. As such, for this homework you need not worry about definitions. For Rackette, however, an environment will be necessary.

For this homework, you will write, in OCaml, an interpreter for Rackette-cita, a tiny subset of Rackette. We give source code for a procedure, `read`, that takes as input a string version of a Rackette-cita expression, and converts it to a concrete program. You will then write a procedure `make_expression_tree` which will convert this concrete program to an internal representation of an expression, which we will call an expression tree. Next, you will write a procedure `eval` which will take in an expression tree and output an internal representation of a value. Finally, you will write a procedure `print` which takes in this internal representation of a value and return a string version of the value.

**Example:** Consider the Rackette-cita expression `(+ 17 18)`. Given this input (as a string), `read` outputs the corresponding concrete program, namely `List [Symbol "+"; Number 17; Number 18]`. Given this concrete program as input, `make_expression_tree` outputs an expression tree that can be interpreted to mean “this expression is a summation operation of two arguments, the first of which is the number 17 and the second of which is the number 18.” The exact representation depends on what you type an `expression_tree` to be.
Next, eval transforms this expression tree into its value, represented as a value: e.g., VInt 35 (the exact representation depends on the choice of value). Finally, if we do this in the context of a read-eval-print-loop(REPL), print transforms its input from a value into a string: e.g., "35".

Note: In past years, CS 17 students have completed the Rackette project in Racket itself. (Isn’t that cool?) There are advantages to writing an interpreter for Rackette in Racket (e.g., the built-in read procedure), and advantages to writing an interpreter for Rackette in OCaml (e.g., variant types and pattern matching). If you want, you can write an interpreter for Rackette (or Rackette-cita) in Racket as well. By doing so, you will observe the tradeoffs in language choice for yourself.

2 Read

Let’s think for a minute about make_expression_tree. The first question is: what should make_expression_tree consume? A string? No. It cannot consume a string because both Racket and OCaml would try to evaluate a string immediately, before we could see if it’s some other type of expression.

So, instead of operating on strings, make_expression_tree will operate on the output of something called the read phase, which consumes a string representing a Rackette-cita program and produces something we call a concrete program. Then make_expression_tree will consume the concrete program and produce an expression tree.

2.1 Read in Racket

Racket has a built-in read procedure. Let’s start out with a brief introduction to this procedure. Start up DrRacket for the following experiment.

Type (read) in the interactive (bottom) window. In the box that appears, type 17 and hit “Enter.” Observe the output. If you ask Racket to “read” any number or string, it produces that number or string.

Now try typing (read) yet again, but this time enter an identifier like seventeen. Observe the output. What is produced is a symbol, that is a sequence of contiguous characters prefixed by a single-quote, '\'.

Play around with read a little more. Input compound expressions like (if true 1 0), and (+ 17 18), and observe what read produces.

We hope that you can infer from these examples what read does. It produces a non-homogeneous list, which it constructs by scanning its input recursively, beginning a new list with every left parenthesis, ending that list with every right parenthesis, and converting everything in between to a token (e.g., 17, '+', '*', 'seventeen', 'if', and so on).

Informally, we refer to the non-homogeneous lists of tokens produced by read as concrete programs. Examples of concrete_program include: 17, 18, 'true', 'false', '+', '-', 'seventeen', 'eighteen', '(1 2 3)', '(if true 1 0)', and '(+ 17 18).

---

²Professor Krishnamurthi calls read a “crown jewel” of Racket. To find out why, take his programming languages course, CSCI 0173.
2.2 Read in OCaml

When writing an interpreter for a subset of the Racket language in Racket itself you can use Racket’s built-in `read` procedure. But you are writing an interpreter for a subset of the Racket language in another language, namely OCaml. Don’t worry. We have written `read` in OCaml for you. You’ll be using a basic version of `read` for this homework, and a slightly fancier one for the Rackette project.

Recall that the output of the `read` procedure in Racket was called a concrete program; the same applies to OCaml. Here is the recursive variant type we use to represent concrete programs in OCaml (in this HW; again, it’s slightly fancier in Rackette):

```ocaml
type concrete_program =
  | Number of int
  | Symbol of string
  | List of concrete_program list
```

Examples include:

```
Number 17
Symbol "silly"
List [Symbol "+" ; Number 17 ; Number 18]
List [Symbol "+" ; Number 17 ; List [Symbol "+" ; Number 18 ; Number 19]]
```

Open a terminal, Change to the directory containing your `read.ml` file, and run `ocaml`. Now type `#use "read.ml" ;;` and hit “Enter.”

**Note:** Do not redefine `concrete_program` in your handin file, as it is already defined in `read.ml`. If you redefine it, you’ll get cryptic errors that will take a long time to debug.

Now spend a few minutes playing with `read` in OCaml by typing `read` in the interactive shell followed by a sample Rackette-cita expression *expressed as a string*.

Examples include:

```
# read "17" ;;
- : concrete_program = Number 17

# read "silly" ;;
- : concrete_program = Symbol "silly"

# read "(+ 17 18)" ;;
- : concrete_program = List [Symbol "+" ; Number 17 ; Number 18]

# read "(+ 17 (* 18 19))" ;;
- : concrete_program = List [Symbol "+" ; Number 17 ; List [Symbol "*" ; Number 18 ; Number 19]]

# read "(+ 15 16)" ;;
Exception: Failure "Syntax Error".
```

You might be wondering “OK, I see what `read` is producing, but what’s the actual “concrete program”? What are the rules?”
The answer is roughly this:

A concrete program is a number, symbol, or list. A number is one or more digits. A symbol is any sequence of non-whitespace, non-paren characters that is not a number. A list is a list of zero or more numbers, symbols, or lists.

The good news here is that you really don’t need to know these rules, you just need to know the datatype that we use to represent something that satisfies these rules, and the type-definition given for concrete_program is exactly that.

Note: Our reader does not handle comments. Experiment to verify this.

3 Making Expression Trees

Having understood read, you are now ready to write a procedure to convert a concrete program into an expression tree. You will do so in two steps: (i) define a type for expression tree that corresponds to the possible forms of a concrete program; and (ii) write the a procedure make_expression_tree to convert concrete programs into expression trees.

To illustrate what you need to do, we’ll consider a still smaller subset of Racket in which there are only integers and the operations add1 and sub1.

```plaintext
type primitive =
  | Add1
  | Sub1

type expression_tree =
  | Int of int
  | Prim of primitive * expression_tree
```

Given this, make_expression_tree consumess the output of read and would produce the following:

```plaintext
# make_expression_tree (Number 17) ;;
- : expression_tree = Int 17

# make_expression_tree (List [Symbol "sub1" : Number 2]) ;;
- : expression_tree = Prim (Sub1, Int 2)

# make_expression_tree (List [Symbol "add1" : List[Symbol "sub1" : Number 2]]) ;;
- : expression_tree = Prim (Add1, Prim (Sub1, Int 2))
```

Note that the output of this make_expression_tree procedure includes everything needed to evaluate the input.

3.1 Rackette-cita

You are now on your way to writing an interpreter for the slightly larger language Rackette-cita. Rackette-cita includes (only) the following Rackette expressions (and no definitions, so every program is just a single expression):
• Numbers—specifically, integers

• Two-argument expressions involving numbers and the arithmetic operators +, *, -, /

The / operator denotes integer division, in which the remainder is discarded, so that (/ 8 3) evaluates to 2.

Let’s define this more generally. An expression is a number or a procedure application. A procedure application consists of a procedure name, an expression, and another expression (all in parentheses). A procedure name is either +, *, -, or /.

Task: Define a type expression_tree for the Rackette-cita subset of Racket, i.e., write a datatype for representing something that satisfies the description above. You’ll want to start with something like this:

```
type primitive = ...
type expression_tree = ...
```

and fill in each ellipsis, using the example from the tiny language above to guide you a little.

Task: Now that you know how you’re going to represent Rackette-cita expressions internally, write the procedure, make_expression_tree, which converts a concrete program into an expression tree.

Check that you can correctly make expression trees from Rackette-cita concrete syntax like this:

- Number 17
- List [Symbol "+"; Number 17; Number 18]
- List [Symbol "+"; Number 17; List [Symbol ";"; Number 18; Number 19]]

Include these test cases (and others) in your handin files.

Hint: It might be useful to write a helper procedure that converts strings like "+" and "*" into primitives for these operators (and gives informative errors when the input isn’t meaningful in Rackette-cita).

Task: Next, you should combine read with your make_expression_tree procedure. Specifically, write test cases that apply read and then make_expression_tree to inputs like these:

```
"17"
"(- 16 15)"
"(* (+ 31 32) 22)"
```

Include these test cases (and others) in your handin files.

Note: We strongly recommend that you go to TA hours to verify that your expression tree type is correct. If your expression tree type is incorrect, the rest of your homework will also be incorrect.

3.2 Error Checking

Since your procedure takes input directly from the read procedure, which takes input from the user, your make_expression_tree procedure should display a useful error message if the user makes a
mistake. This is something we’ve not done before. Now we are. For example, a user might try to apply + to only one argument. In such a case, it would be helpful for the user to see a message like, "Operator expects two arguments."

To implement this, you can make use of failwith, which works like this:

```ocaml
# failwith "+ expected two ints as input." ;;
Exception: failure "+ expected two ints as input."
```

However, you are not expected to test for these errors.

**Task:** Add error checking to your make_expression_tree procedure: i.e., add conditions to your procedure that check for invalid input. Upon encountering invalid input, use the failwith procedure to raise an error.

**Note:** For this homework, we don’t expect you to test the errors you throw. However, you should make sure that invalid inputs do have informative failwiths as described above.

## 4 Arithmetic Evaluator

Recall that the input to make_expression_tree is a Rackette-cita program written as a concrete program, and that its output is of type expression_tree. The next step in interpretation is to evaluate that expression tree. The result of evaluating an expression is a value. For our simple arithmetic evaluator, the only possible values are integers, so the value is pretty trivial:

```ocaml
type value = VInt of int
```

**Task:** Write a procedure eval that evaluates expression trees, and produces its value as a value.

For example:

```ocaml
# eval (make_expression_tree (read "(* (+ (- 6 3) (/ 9 3)) 0)")) ;;
- : value = VInt 0
```

**Hint:** It might be useful to write a helper procedure that evaluates applications of a particular primitive to two values. This procedure should consume a primitive and produce a procedure that applies the input primitive to two values, and produces yet another value. (It will involve a large match expression, which handles each of your primitives, like addition or subtraction, differently.)

**Hint:** To add together two integers, you’re probably going to want to apply OCaml’s + function. There’s a slight problem: in OCaml, + is an infix operator, but you may find yourself wanting a prefix operator for addition. It turns out that there’s a special piece of OCaml syntax for this: putting parens around an infix operator converts it to a prefix operator, so that

```ocaml
3 + 5;;

(+) 3 5;;
```

both produce the value 8. In fact, you can write this:
let
  proc = (+)
in
proc 3 5;;

and get the value 8 as well.

5 Print

Task: To conclude, write the procedure print that converts values back into a string, as follows:

# print (eval (make_expression_tree (read "(* (+ (- 1 2) (/ 3 1)) 5)"))) ;;
- : string = "10"

Hint: Feel free to use the built-in procedure string_of_int, which consumes an integer and produces that integer as a string. For example,

# string_of_int 17 ;;
- : string = "17"

Note: To test print, try something like print (VInt 3);;. The point here is to illustrate that you don’t need to apply print only to the results of eval, but can instead test it directly like this.

Note: Make sure to write a couple test cases for the combination of all procedures.

6 REPL

This part of the homework is strictly for fun. You can ignore this section and still get full credit for Rackette-cita. But it’s pretty cool (and, did we mention, fun?), so we recommend you give it a try.

Although it is beyond the scope of CS 17, it is fairly easy to build a Rackette (and hence, Rackette-cita) REPL, given working implementations of make_expression_tree, eval, and print. Code that accomplishes this is available in /course/cs0170/src/hw09/repl.ml.

The REPL procedure is called rackette_repl, and its arguments are your make_expression_tree, eval, and print procedures. Invoking this procedure with these arguments will allow you to run your interpreter interactively:

> racketteRepl make_expression_tree eval print
Rackette> (+ 1 2)
3
Rackette> (* 3 4)
7
Rackette> (- 15 16)
-1
Rackette> (/ 17 18)
0
If you have time, take a quick look at the racketteRepl code (reproduced below). Note that there is no base case. Instead, every call to this procedure leads to another recursive call. This means that, once called, the procedure will continue its evaluation until you force it to halt.

There are two particular parts of this code that you are unlikely to be familiar with:

- `Printf.printf`: This procedure prints a string to the standard output (e.g., the terminal).
- `try ... with`: This construct prints any error messages your interpreter encounters without halting the REPL.

```ocaml
let rec racketteRepl = function (make_expression_tree, eval, display) ->
  let rd_line =
    (try Printf.printf ``s'' (display(eval(make_expression_tree(read(
      read_line()))))))
 Printf.printf ``Rackette > '' ;
  match rd_line with
  | e -> (match e with
    | Failure(str) -> Printf.printf ``Error: \%s\n'' str
    | _ -> Printf.printf ``Error: \%s\n'' ``Other exception failure
      '' ));
  (racketteRepl (make_expression_tree, eval, display))
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)