Homework 9: Rackette-cita
Due: 10:00 PM, Nov 7, 2017

Contents

1 Introduction 2

2 Read 4
   2.1 Read in Racket 4
   2.2 Read in OCaml 5

3 Parser 6
   3.1 Rackette-cita 7
   3.2 Error Checking 8

4 Arithmetic Evaluator 8

5 Print 9

6 REPL 10

Objectives

By the end of this homework, you will know:

- how read works

By the end of this homework, you’ll be able to:

- write a parser for a small subset of the Racket language
- write an evaluator for a small subset of the Racket language
- print the outcome of reading, parsing, and evaluating some Racket expressions

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 Parser, 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 (e.g., parser.ml corresponds to Parser):

- README.txt
- CS17setup.ml
- read.ml
- parser.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. If you are using a departmental linux system, all your solution files should reside in your ~/course/cs017/homeworks/hw09 directory.

For this and every assignment, you should also have a README.txt file whose first line contains only your CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:

jfh@cs.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 zip your hw09 directory into a file hw09.zip.

Hand in this zip file using the method you learned in the first lab: visit https://tinyurl.com/cs0170-handin to get started.

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 expr)
  (cond
    [(number? expr) ... ]
    [(boolean? expr) ... ]
    [(string? expr) ... ]
    ...
    [(and
      (list? expr)
      (not (empty? expr))
      (equal? (first expr) 'if)
      (not (empty? (rest expr))))
      (not (empty? (rest (rest expr))))
      (not (empty? (rest (rest (rest expr))))))
      (empty? (rest (rest (rest (rest expr))))))
    ... ]
    ... ))
```

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?

As discussed in class, you can achieve exactly that by breaking down the interpretation process into two phases:

1. The first phase attempts to parse the program—that is, translate it into an abstract program (i.e., an internal representation), producing an error if it finds the program to be invalid.

2. The second phase processes the successfully parsed, and hence syntactically valid, abstract program, and, if there’s an expression, 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.

\(^1\)Take CS173 to learn why this isn’t the only possibility.
In this homework, there is no environment. We are dealing with a language so simple that we don’t need one. For Rackette, however, an environment will be necessary.

For this homework, you will write, in OCaml, an interepreter for Rackette-cita, a tiny subset of Rackette. More specifically, you will define an abstract program type, and then your parser will output an abstract program that will serve as input to your Rackette-cita evaluator, which will then produce a value.

Recall from class that “syntax” is the collection of rules that say what’s OK in a language. We’ll define three different sets of rules — concrete syntax, abstract syntax, and value syntax. We will say that things that satisfy the rules of concrete syntax are “concrete programs”. For example, a string representing a program like (+ 3 a) can be transformed into something closely related, which reflects the rules of concrete syntax. What it is transformed into is a concrete program like this:

\[
\text{List } [\text{Symbol } "+" ; \text{Number } 3; \text{Symbol } "a"]
\]

We’ll later transform this into some abstract program, which we’ll then process (either as a definition or as an expression). The result of evaluating an expression will be something called a “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, parse outputs an abstract program 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 of the parse depends on your choice of abstract programs.

Next, eval transforms this parse into its value, represented as a value: e.g., VInt 35 (again, 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 the parser. The first question is: what should your parser consume? A string? No. It cannot consume a string because both Racket and OCaml would try to evaluate a string before it could be parsed!

So, instead of operating on strings, your parser 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 your parser will consume the concrete program and produce an abstract program.
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 (list 1 2 3), (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, ‘list, ‘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, (list ‘list 1 2 3), (list ‘if ‘true 1 0), and (list ‘+ 17 18).

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]]

---

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

```ocaml
# 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:

```
<expression> := <number> | <string> | <list>
<number> := <digit>+
<symbol> := any sequence of non-whitespace, non-paren characters that is not a number
<list> := ( <expression>* )
```

where that last thing means “a left paren, any number of expressions, and a right paren”, and we’ve simplified the description of a symbol so that it’s in English rather than BNF, and I’ve skipped over the part about what separates things, i.e., “Is 22ab the number 22 followed by the symbol ab, or is it a symbol 22ab?” (Answer: the latter.) The rules for separation are, roughly, that whitespace and punctuation (like parens) separate things.

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  Parser

Having understood read, you are now ready to write a parser. You will do so in two steps: (i) define an abstract program corresponding to each form of concrete program; and (ii) write the actual parser to convert concrete programs into abstract programs.

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. A possible abstract program for this language would be represented by defined by the following variant type:

```plaintext
type primitive =  
  Add1  
  Sub1

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

Note that this is much simpler than the abstract_program type we’ve seen in class, which is an abstract_program_piece list. That’s because a program in this tiny language consists of exactly one expression rather than a sequence of definitions followed by an expression as in Racket.

(Note: To denote one of the two operations allowed in this language, we’re using the string “primitive” rather than “built-in”, but the idea is the same.)

Given this abstract program type, a parse procedure for this language that consumes the output of read would produce the following:

```plaintext
# parse (Number 17) ;;  
- : abstract_program = Int 17

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

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

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

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 write that out in BNF:

\[
\begin{align*}
\text{<expression>} & : = \text{<number>} \mid \text{<proc-app>}
\text{<proc-app>} & : = ( \text{<proc-name>} \text{<expression>} \text{<expression>} )
\text{<proc-name>} & : = + \mid * \mid - \mid /
\end{align*}
\]

**Task:** Define an abstract program type for the Rackette-cita subset of Racket, i.e., write a datatype for representing something that satisfies the BNF description above, analogous to the `abstract_program` defined above. You’ll want to start with something like this:

```
type primitive =
  |
  ...

type abstract_program =
  |
  ...
```

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, `parse`, which converts a concrete program into your abstract program.

Check that you can correctly parse Rackette-cita expressions like these:

- Number 17
- Symbol "silly"
- 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 your abstract program representation for these operators (and gives informative errors when the input isn’t meaningful in Rackette-cita).

**Task:** Next, you should combine `read` with your parser. Specifically, write test cases that apply `read` and then `parse` 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 abstract program type is correct. If your abstract program is incorrect, the rest of your homework will also be incorrect.

### 3.2 Error Checking

Since your parser takes input directly from the `read` procedure, which takes input from the user, your parser 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 parser: i.e., add conditions to your parser that check for invalid input. Upon encountering invalid input, use the failwith procedure to raise an error.

**Note:** In general, checking for the right number of arguments in Racket is not a task for the parser, since a procedure-application-expression can have arbitrarily many expressions in it. But in Rackette-cita, such an expression always has exactly two arguments, so you can actually detect this error during parsing. This won’t be possible with Rackette, which, like Racket, allows some procedures, like empty? to take one argument, some, like cons, to take two, etc. (You won’t, however, be required to handle procedures that take arbitrary numbers of arguments, the way that + does in Racket.)

## 4 Arithmetic Evaluator

Recall that the input to your parser is a Rackette-cita expression written as a concrete program, and that its output is that same expression as an abstract program. The next step in interpretation is to evaluate that abstract program. The result of evaluating an abstract program 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 abstract programs, and produces its value as a value.

For example:

```ocaml
# eval (parse (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:

```racket
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:

```racket
# print (arith_eval (parse (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,

```racket
# 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 `parse-and-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 `parse`, `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 `parse`, `eval`, and `print` procedures. Invoking this procedure with these arguments will allow you to run your interpreter interactively:

```
> racketteRepl parse eval print
Rackette> (+ 1 2)
3
Rackette> (* 3 4)
7
Rackette> (- 15 16)
```
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 parse eval display =
  let rd_line =
    (try Printf.printf ``s'' (display(eval(parse(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 parse eval display)
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)