Homework 8: Rackette-cita
Due: 10:59 PM, Oct 30, 2019

Contents

1 Introduction 3

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

3 Parse (20 Points) 7
   3.1 Rackette-cita 8
   3.2 Error Checking 9

4 Arithmetic Evaluator (20 Points) 9

5 Print (6 Points) 9

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 handing in tasks for Parse, Arithmetic Evaluator, and Print questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. We sent out source code for this assignment in an email: Types.re, CS17SetupRackette-cita.re,
and Read.re. These files should also be turned in with your submission. Turn in the following files for this assignment:

- Types.re
- CS17SetupRackettecita.re
- Read.re
- Parse.re
- Arithmetic.re
- Print.re
- README.txt

For this assignment, all files you turn in that contain code must be Reason files, so they must end with extension .re and must all start with a Capital letter. For all coding problems in this homework, you should follow the Reason design recipe outlined on the website.

For this and every future assignment, 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 handin. For example:

README.txt:
B01234567
There’s nothing to say except that I’m turning in some 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.

Set-Up

To start, navigate to the directory containing your homework assignments on your local computer and run the following command, replacing "homework08" with whatever you would like to call your homework directory for this homework. You only need to run this command once for each homework/project:

bsb -init homework08 -theme basic-reason

Copy Types.re, CS17SetupRackettecita.re and Read.re (all provided in the email) into your homework08/src directory. DO NOT EDIT THESE. The files that you write for this assignment should also reside in the homework08/src directory. For each file that requires functions from another file, open the required files at the top using:

open <FileName>;

At the top of Parse.re, you will need to include:

open Types;
open CS17SetupRackettecita;
open Read;
At the top of Arithmetic.re, you will need the same three open statements from Parse.re, plus:
open Parse;
At the top of Print.re, you will need the same four open statements from Arithmetic.re, plus:
open Arithmetic;

To compile and run your code, from the homework08/src directory run

  npm run build
  node <file-name-to-run>.bs.js

Check Expects

For this homework you will be using three different types of check expects, depending on the part of the problem. For parse, use checkExpectParse which takes in two abstractPrograms (the output type of the parse function) and a string (message). The call to it will look like:

  checkExpectParse(parse(<proc-args>), <expected-output>, <message>)

For eval, the procedure is called checkExpectEval and operates the same, but the arguments are two values and a string. For print, use checkExpect with arguments of two 'a (in this case they will be strings, the output of print) and a string (message).

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”:

```
(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))))))]
```

1Take 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 \texttt{if} expression seems a bit excessive! Wouldn’t it be nice to know that \texttt{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 \texttt{parse} the program—that is, translate it into an \texttt{abstract program} (i.e., an internal representation), producing an error if it finds the program to be invalid.

2. The second phase \texttt{processes} the successfully parsed, and hence syntactically valid, abstract program, and, if there’s an expression, produces a \texttt{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 \texttt{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 \texttt{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 Reason, an interpreter for Rackette-cita, a tiny subset of Rackette. More specifically, you will write a parser which will take in a concrete program and output an abstract program that will serve as input to your Rackette-cita evaluator, which will then produce a value. Check out the \texttt{Types.re} file to see the types we’ll be dealing with.

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 \texttt{( + 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:

\begin{verbatim}
List([Symbol("+"), Number(3), Symbol("a")]);
\end{verbatim}

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.”

\textbf{Example:} Consider the Rackette-cita expression \texttt{( + 17 18 )}. Given this input (as a string), \texttt{read} outputs the corresponding concrete program, namely \texttt{List([Symbol("+"), Number(17), Number(18)])}. Given this concrete program as input, \texttt{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.”

Next, \texttt{eval} transforms this parse into its value, represented as a value: e.g., \texttt{VInt(35)} (again, the exact representation depends on the choice of value). \texttt{print} transforms its input from a value into a string: e.g., \texttt{"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 Reason (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 Reason 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 in the Advanced Student language 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 concreteProgram 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 Reason

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

\[^2\]Professor Krishnamurthi calls read a “crown jewel” of Racket. To find out why, take his programming languages course, CSCI 0173.
another language, namely Reason. Don’t worry. We have written read in Reason 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 Reason. Here is the recursive variant type we use to represent concrete programs in Reason (in this HW; again, it’s slightly fancier in Rackette):

```
type concreteProgram =
  | Number(int)
  | Symbol(string)
  | List(list(concreteProgram));
```

Examples:

```
Number(17)
Symbol("silly")
List([Symbol("+"), Number(17), Number(18)]);
List([Symbol("+"), Number(17), List([Symbol("*"), Number(18), Number(19)])])
```

As you’re familiar with in Racket, when you evaluate a procedure, Racket prints out the result of the evaluation. However, when a procedure is evaluated in Reason, the result is not printed to the user (though it’s still computed!).

Enter Sketch.sh. Sketch does the kind of printing we’re familiar with from Racket, making it a great choice for Rackette-cita.

To use the read procedure in Sketch:

- In your browser, navigate to sketch.sh.
- Open a new sketch.
- Copy all of the code from the Types.re file we provided and paste it into your sketch.
- Click the <>+ icon at the bottom of the sketch to create a new block of code.
- Copy all of the code from the Read.re file we provided and paste it into this new block. Delete the first line that says open Types;.
- Click the <>+ icon at the bottom.
- In this new block, write read("17") and hit enter. This should output:
  ```
  let read: string => concreteProgram = <fun>;
  ```

Now spend a few minutes playing with read in Reason by typing read in Sketch followed by a sample Rackette-cita expression expressed as a string.

Note: Do not redefine any types in your handin file, as they are already defined in Types.re. If you redefine them, you’ll get cryptic errors that will take a long time to debug. However, you still must write data definitions for all types.

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

where the last line 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 concreteProgram is exactly that.

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

3 Parse (20 Points)

Having understood read, you are now ready to write a parser which will 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 abstractProgram =
    | Int(int)
    | Prim(primitive, abstractProgram);

(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));
- : abstractProgram = Int(17)

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

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

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

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:

```
<expression> := <number> | <proc-app>
<proc-app> := ( <proc-name> <expression> <expression> )
<proc-name> := + | * | - | /
```

Task: Write the procedure, parse, which converts a concrete program into an abstract program. Check that you can correctly parse Rackette-cita expressions like these:

```plaintext
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 the 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.

### 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:

```
# 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 (20 Points)

Recall that the input to your parser is a Rackette-cita expression written as a concrete program and its output is the abstract program representation of that expression. 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:

```
type value = VInt(int)
```
Task: Write a procedure `eval` that evaluates abstract programs, and produces its value as a value. For example:

```scheme
# 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 `switch` expression, which handles each of your primitives, like addition or subtraction, differently.)

5 Print (6 Points)

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

```scheme
# print(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,

```scheme
# 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.

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).