Project 3: Rackette
7:00 PM, Nov 17, 2018

Contents

1 Introduction 1

2 The read procedure 2

3 The Language 2
   3.1 Syntax 2
      3.1.1 Sidenote on built-in procedures 3
   3.2 Semantics 4

4 The Top Level Environment 4
   4.1 Overview 4
   4.2 Adding Definitions 4
   4.3 Evaluation Process 5

5 The Assignment 5
   5.1 Data Definitions 5
   5.2 Values and Procedures 5
   5.3 Error Checking 7
   5.4 Built-in procedures 7

6 Getting Started 8
   6.1 Setting up 8
   6.2 Moving forward 9

7 Practice on Paper 10

8 Handing In 11
   8.1 Design Check 11
   8.2 Final Hand-in 12
   8.3 Grading 13

9 REPL 14
1 Introduction

When you click “Run” in DrRacket, the contents of the Definitions window are processed and something is printed in the interactions window.

In accordance with CS 17’s “no magic” slogan, this project will demystify this “processing” step. You’ll write a program that takes in another program as input and produces exactly the same result that DrRacket would.

What you’re going to do is write an interpreter. By interpret, we mean take in a Rackette program represented as a string and process the program one piece at a time to produce results. You’ll be doing this processing in OCaml.

Interpreting consists of three steps:

1. Read in the text and represent it as a concrete program, or more precisely, as a list of concrete program pieces.

2. Convert this concrete program into an abstract program, i.e., an internal representation of your Rackette program as a list of OCaml entities that we’ll call abstract program pieces, one per concrete-program-piece. Each abstract program piece will correspond to a definition to be added to the top level environment, or a Racket expression. This step is called parsing.

3. Process the resulting internal representation, one “piece” at a time.
   
   (a) If a piece is a top-level expression, evaluate it according to Rackette’s rules of evaluation, producing a different kind of entity, a value, and then converting that value into a string.
   
   (b) If a piece is a definition such as (define seven (+ 3 4)), process it by evaluating the right-hand side (the expression) to produce a value, and then append to the top-level environment the binding of the symbol to the value.

We’ve handled the first step, the conversion from a raw program string to a concrete program, for you. For this project you’ll be implementing steps 2 and 3. Much of the rest of this document deals with the particulars of these last two steps.

1DrRacket produces its values in a slightly different way than your program will; take CS173 if you’d like to learn more about this!
2 The read procedure

The read procedure we’ve provided reads in the raw text of a Rackette program and represents each “piece” as a concrete program piece, so that reading a Rackette program returns a concrete_program, which is a concrete_program_piece list.

What is a concrete_program_piece, exactly?

```
type raw_program = string ;;

type concrete_program_piece =
  | Number of int
  | Symbol of string
  | List of concrete_program_piece list ;;

type concrete_program = concrete_program_piece list ;;

let read (input: raw_program): concrete_program_piece = ...
let read_all (input: raw_program): concrete_program = ...
```

You’ll be using the read_all procedure we’ve written to convert the raw text of a Rackette program into something similar to what you saw in the Rackette-cita Homework, which you can then parse into an abstract program.

We can think of this reading process as a transformation from something that represents a program in a very open-minded language (the BNF for this language says, roughly, that a legal program is any sequence of printable characters except single quotes, back-quotes, hashmarks, vertical bars, backslashes or commas) to a slightly stricter “language”, the concrete language. This language doesn’t have a BNF, as it’s not textual, but the role of the BNF is provided by the type-definition for concrete_program above; a concrete program is a sequence of concrete program pieces, where a concrete program piece is a Number or a Symbol or a List.

Each subsequent step of the project can similarly be regarded as a transformation between languages. The next step, parsing, takes certain concrete programs and restructures them into abstract programs, and a later step takes expressions and converts them to values. Each of these subsequent things has a defined structure (which one might call a “syntax”, with the structure coming from the type definitions rather than from BNF).

The rackette.zip file in the src folder contains the file read.ml, which in turn contains the procedures read and read_all. Because we import read.ml at the top of the stencil code, you’ll be able to refer to these procedures in your own code.

3 The Language

In this section, we describe the syntax and semantics of the Rackette language.

3.1 Syntax

A Rackette program is a sequence of pieces, each piece being either a definition or an expression.
A definition is a list consisting of the keyword define, a symbol, and an expression.

An expression is something that can be evaluated, something described by the rules of evaluation. Here are the different kinds of expressions:

- An integer.
- A symbol.
- One of the following:
  - An if expression of the form (if pred yes-exp no-exp), where pred, yes-exp, and no-exp are also Rackette expressions.
  - An or expression of the form (or expl1 expl2), where expl1 and expl2 are also Rackette expressions.
  - An and expression of the form (and expl1 expl2), where expl1 and expl2 are also Rackette expressions.
  - A cond expression of the form (cond ((pred expr) (pred expr) ... ), where each occurrence of pred and expr stands for a Rackette expression. (Note that a cond-expression must contain at least one such pair!)
  - A quote expression of the form (quote datum), where datum is an integer, symbol, or a list. Note that, in accordance with the quote rule of evaluation, the datum will not be evaluated.
  - A lambda expression of the form (lambda (id1 id2 ... )body), where id1 and id2 and so on are symbols, and body is a Rackette expression.
  - A procedure-application expression of the form (proc arg1 arg2 ... ), where proc is a Rackette expression, and arg1 and arg2 and so on are Rackette expressions.

There are also things like cons and true, but these are just symbols that are bound to values (a procedure and a boolean in these two cases) in the top-level environment.

### 3.1.1 Sidenote on built-in procedures

Among the different Rackette value types will be one called a “built-in”. This is a built-in procedure. You will have to write these in OCaml. The built-in procedures are those with the following names: +, -, *, /, remainder, =, <, equal?, number?, zero?, cons, car, cdr, empty?, cons?, and not.

Although at first glance, it may seem that and and or should be built-in procedures, there is actually a subtle difference. Both and and or, unlike other built-in procedures, implement short-circuiting. This means that they don’t evaluate their second argument if they don’t have to: if the first argument of and evaluates to false, the and expression evaluates immediately to false. Similarly, if the first argument of or evaluates to true, the or expression evaluates immediately to true. You will need to achieve this behavior in your Rackette implementation.
3.2 Semantics

The semantics of Rackette are defined by the rules of processing:

1. Processing a definition adds a binding for an identifier into the top-level environment (unless the identifier is already bound there, in which case it’s an error). The next section about the top-level environment will explain this in more detail.

2. Processing a top-level expression entails evaluating the expression using the rules of evaluation to produce a value, and then producing a string that’s the printed-representation of this value.

For the full rules of evaluation, which you will need to follow while writing your evaluation program, see the appendix.

4 The Top Level Environment

In this section, we describe how the top level environment of Rackette should behave.

4.1 Overview

The initial top level environment contains bindings of all the builtin procedures. This gives the user of Rackette the ability to use addition, for instance, just as when you open up a new session of DrRacket, you’re able to use procedures like +, -, *, and /.

In Rackette, we expect you to have your own initial top level environment containing bindings for the builtin procedures mentioned in section 3.1.1. Every time your program is run, this top level environment should be initialized.

4.2 Adding Definitions

A new top level environment will be created each time the user adds a definition. For instance, the definition (define x 5) will create a binding, which will be added to the top-level environment to create a new top-level environment, which will be used for all subsequent processing, and in which the identifier x will be bound to 5.

There’s one exception to this rule: if an identifier is defined in the top-level environment, and we process a definition of that same identifier, it’s an error, and processing stops.

In your Rackette implementation, one of the procedures that you will have to implement will be add_definition. This procedure will be used to add definitions to the top-level environment, as explained above. In order to process a given definition with an identifier and an expression, we first need to evaluate the expression in the top-level environment. Then, we add the binding of the identifier to the now-evaluated expression into the top level environment to create a new, richer, top-level environment.
4.3 Evaluation Process

In several of the rules of evaluation, we have a top-level environment, $T$, and another environment, $E$, that can be called the “local environment.” Since $E$ is often altered (see the “lambda” rules), it’s helpful to make “evaluation” operate on three operands: an expression, a top-level environment, and a local environment.

When evaluating an expression with identifiers, such as $(+ \times 5)$, Racket, upon reaching the identifier $\times$, will look for a binding of $\times$ in $T + E$. One can do this by either forming the environment $T + E$ and doing a lookup, or by first trying to find a binding for $\times$ in $E$, and if this is not found, trying to find a binding for $\times$ in the top level environment $T$. If there is no binding for $\times$ in either environment, then evaluation terminates with an error. Note that local bindings take precedence. Thus, in the program

```
(define x 25)
((lambda (x) (+ x 0)) 0)
```

the expression evaluates to $0$ and not $25$.

Your Rackette implementation should follow this evaluation process.

5 The Assignment

This project consists of three parts: the data definitions, the main procedures, and error checking.

5.1 Data Definitions

When writing an interpreter, having comprehensive and correct data definitions becomes vitally important. After all, we need a way to encapsulate the essence of what it means for something to be “expression” or for something to be a “value”, and we need to make sure that our type-definitions adhere to the grammar of the language that we are trying to interpret. Additionally, given that we are writing an interpreter that follows environment semantics, we need type-definitions to represent the environment and its constituent bindings. For your Rackette interpreter, we have provided these data definitions for you in the support code.

In the /course/cs0170/src/rackette/rackette.zip file, you’ll find the file rackette.ml, which contains all but one of the data definitions you’ll need. You’ll have to construct your own definition for an if-expression in preparation for your design check. This file also contains skeletons for all the procedures you’re required to write. Your task is to fill in anything specified by TODO tags in rackette.ml, and to implement anything that fails with the error “not yet implemented.”

If you have any questions about the types or procedures defined in the template, please come to TA hours or visit the professor —understanding everything in the template is crucial to your successful completion of the project.

5.2 Values and Procedures

Now, let’s discuss the purpose of each value and procedure required.
Following your data definitions, write the initial top-level environment, `initial_tle`, and the procedures `parse`, `add_definition`, `eval`, and `string_of_value`.

- `initial_tle` is an OCaml identifier that must be bound to the top-level environment with Rackette bindings of the required symbols to corresponding values, including built-in procedures and also the boolean values (true and false).
- `parse` consumes the output of `read_all`, namely a Rackette program represented as a `concrete_program` (one `concrete_program_piece` for each “piece” of the program) and produces the equivalent `abstract_program`.
- `add_definition` consumes an environment and a binding, and returns a new environment with all the bindings from the original environment, and the new binding as well.
- `eval` consumes two environments—the first, the top-level environment, `T`, and the second, the local environment `E`—and an expression, and then produces the value of the expression in the environment `T + E` (`T` extended by `E`).
- `string_of_value` consumes a value, and converts that value into a string. For constants like numbers and booleans, `string_of_value` should return a string representation of that constant. For a built-in procedure like `+`, it should return a descriptive string, something like “builtin:+”. However, when called on a closure (a user-defined procedure, i.e. the value of a lambda expression), it should return a string that doesn’t need to contain complete information about the closure (Racket chooses `(lambda (a1) ...)`). If at any point you don’t know how to produce a string representation, printout a similar expression in Dr. Racket! You can model your representation after that.

The procedures above all come together in the procedure `rackette`, which reads, parses, processes, and prints a Rackette program represented as a string.

**Example.** Consider the following Rackette program:
```
(define x 15) (if (= 15 16) 100 -1)
```

Given this input (as a string), `read_all` outputs the corresponding `concrete_program`:
```
> read_all "(define x 15) (if (= x 16) 100 -1)"
- : concrete_program =
[List [Symbol "define"; Symbol "x"; Number 15];
 List [Symbol "if"; List [Symbol "="; Symbol "x"; Number 16]; Number 100;
 Number "]"]]
```

Given this `concrete_program` as input, `parse` outputs an `abstract_program` that can be interpreted to mean “A Rackette program that first has a definition that binds the name `x` to the constant 15 in the top-level environment, and then has an `if` expression, with condition `(= x 16)`, and ‘true’ expression `100` and ‘false’ expression `-1`.”

Next, `process` adds definitions present to the TLE—in this case, the identifier `x` binds to the value associated by evaluating `15` (i.e., `VNum 15`). Then `process` calls `eval` with the TLE and an empty
local environment, transforms this parse into its value, represented as a value: i.e., VNum(-1).

Finally, string_of_value transforms its input from value into a string: i.e., "-1".

### 5.3 Error Checking

One very important function of an interpreter is to report an error on bad input (e.g., invalid Rackete syntax). Failure by the interpreter to report bad input could have numerous consequences, including:

- The interpreter could do something illegal while trying to process the user’s bad input, resulting in a completely uninformative error message.
- Or, even worse, an expression could evaluate without breaking, but to the wrong value. This would totally confuse the user.

You don’t need to work yourself into a tizzy rooting out all possibilities for bad input, but to limit your own irritation while programming Rackette, you should make a reasonable effort. (Good error reporting may be your very best debugging tool!) Here are some examples of reasons an interpreter might report an error:

- A user inputs an invalid concrete_program.
- Something other than a keyword or an expression that evaluates to a procedure followed a left parenthesis.
- The user attempted to use a number, boolean, keyword, or other expression in place of an identifier or a procedure.
- The first clause in an if expression did not evaluate to a boolean.
- The user attempted to apply a procedure to the wrong data types - e.g. trying to use + on two booleans.
- There was a procedure application, but the procedure requires a different number of arguments than it was applied to.

Your main objective in handling these kinds of errors is to differentiate between parse errors and evaluation errors. A parse error occurs when the user inputs something syntactically invalid, like an empty set of parentheses “()”. This program doesn’t correspond to anything in our Rackete grammar—it can’t be parsed! An evaluation error, on the other hand, results when a user inputs something syntactically valid, but semantically nonsensical, like “(2 3)”. Although this is a procedure application, according to our syntactic rules, it doesn’t make sense to apply 2 as a procedure, and we cannot reasonably return a value in this case.

### 5.4 Built-in procedures

The rules of evaluation say that a built-in procedure is applied to a list of values. Our representation of a built-in is VBuiltin of string * (value list -> value), where the string is the printed
representation of the built-in, and the other component of the ordered pair is an Ocaml procedure taking value-lists to values.

That means that as you’re setting up values to put into the initial top-level environment, you might write something like this:

\[
\text{VBuiltin } (<\text{builtin-proc-+}, \text{plus})
\]

where \text{plus} is a procedure you’ve defined which takes a value list as input, and produces a value as output.

Here are the steps you might use in writing \text{plus}:

1. Check that the input list has exactly two items in it
2. Check that both items are in fact \text{VNum}s.
3. Extract the integers from those \text{VNum}s and add them, and
4. Wrap up the resulting integer in a \text{VNum}.

On the other hand, you could be far more concise, and just write a simple anonymous procedure rather than defining \text{plus} at all. It’s your choice, and both ways work fine, as do others. By the way, you can use any string for your built-in that you like, but it should not be misleading. (It’d be bad for the string associated to + to be "<builtin: cons>", for instance.)

6 Getting Started

6.1 Setting up

To start out, copy the file /course/cs0170/src/rackette/rackette.zip in the directory you’d like to work in, and unzip it. You’ll find five files inside:

- \text{rackette.ml}: a template; all your code goes in here
- \text{README.txt}: your README goes in here
- \text{read.ml}: contains our support code (do not alter)
- \text{CS17setup.ml}: a copy of the teachpack (do not alter)
- \text{repl.ml}: support code that allows you interact with your rackette program (do not alter)

Once you’ve gotten access to these files, you’re ready to get to work.
6.2 Moving forward

After the design check, it can sometimes be daunting to know where to start with the project. To help, here is an overview of the major steps that need to be completed. Of course, there are many correct ways to approach projects, so feel free to do these in any order. For more specific information on each part, consult the rest of the project PDF, Piazza, or your friendly TA staff!

- Read the template with your partner. On alternating procedures, each of you should explain what you think the procedure does, why its input has the type it does, and why the output type is what it is. Seriously: do this before any other steps.

- Fill out the missing definition for if-expression in the rackette.ml file. Use IfE as the constructor for this type.

- Import your “environment” code from the lab to create a basic top-level environment with no bindings.

- Get parsing of expressions working. The expressions you need to implement are outlined in Section 3.1 If any piece seems intimidating, skip it for now, and leave in an informative failwith.

- Get parsing of definitions working. What must a concrete_program_piece list look like to be a definition? Use pattern matching and include a wildcard pattern for non-definitions.

- Start writing the procedures outlined in Section 5.2 namely: initial_tle, parse, add_definition, string_of_value.

- Start working on evaluation. Try to combine parsing and evaluation. See whether you can parse and evaluate a simple program that consists of just one expression. Try more complex expressions.

- See whether you can make repl work with these two test cases. This will test parsing, evaluation, and printed representations all at once.

- Work on evaluating identifiers by looking them up in some environment. Start by filling in add_definition and seeing whether it works. If your eval has enough patterns, then the wildcard pattern at the bottom should, like the one in the parser, have become redundant.

- Define the various built-in procedures in the initial top-level environment. Expand on what you started for your design check and create VBuiltins with printed representations for each procedure. The built-ins you are required to implement are as follows: +, -, *, /, remainder, =, <, equal?, number?, zero?, cons, car, cdr, empty?, cons?, and not. You must also implement the built-in boolean values (true and false). These will not be bound to VBuiltins – what should they be bound to instead?

- Test your code first with a simple program like (+ 1 2) or (define x 2) (+ x 1) to see if it works. Try more and more complex programs, all the way up to nested lambdas. Paste in your solutions to homework problems and see whether they’re handled right.
7 Practice on Paper

In this section, we include a worked example, followed by two sets of expressions. You should be able to evaluate the first set of expressions (under “Simple Evaluation”) with relative ease. Your ability to evaluate more complex expressions like those in the second set (under “Design Check Problems”) will be assessed during the design check.

For the following practice problems, you can assume that the top level environment has at least the following bindings:

1. "+" \(\mapsto\) a builtin-procedure that takes two numbers (represented as values) and produces their sum (represented as a value)
2. "-" \(\mapsto\) a procedure that subtracts two numbers (in the same way as “add”)
3. "*" \(\mapsto\) a procedure that multiplies two numbers (in the same way as “add”)
4. "zero?" \(\mapsto\) a procedure that checks if a number is zero (in the same way as “add”, i.e., operating on and producing values)
5. "> " \(\mapsto\) a procedure that checks if the first number is greater than the second number (same deal with values)

Worked Example 1 Here is a step-through of the evaluation of \((\text{define } z 2)(+ z 8)\).

- Initially, the top level environment is \{ + \mapsto addn-proc, ... \} and the local environment is empty.
- After processing the definition, the top level environment is \{ "+" \mapsto addn-proc, ..., z \mapsto VNum 2 \} The local environment is still empty.
- To evaluate \((+ z 8)\), Rackette will evaluate the identifier + by searching for a binding of it in the the top level environment extended by the local environment, and find that it is bound to a procedure that adds the next two arguments. Rackette will then do the same for z and find that it is bound, in the top-level-environment, to the value VNum 2. Rackette will evaluate the expression 8 the get the value VNum 8. And finally Rackette will apply the builtin procedure to VNum 2 and VNum 8 to get VNum 10, which is the value of the whole expression \((+ z 8)\).

Worked Example 2 Here is a diagram representing the evaluation of \((\text{define } z 2)(+ z 8)\).
The outermost box contains the initial expression and top level environment. The second box shows the evaluation of the lambda-expression, which is a closure with three parts: arguments, body, and local environment. Finally, we see the evaluation of the body using the binding from the local environment.

Keep in mind during the design check problems that the innermost, newest local environment will take precedence over all outer local environments.

We’ve been a little sloppy at one point: we wrote that the value of the identifier + was addn-proc, but it’s actually VBuiltin("+", addn-proc). We did this to save space. The same is true in the environment: when I say that + is bound to <addn-proc>, I really mean that it’s bound to VBuiltin("+", addn-proc).

**Simple Evaluation**  For practice, process the following programs in either example’s style, starting from the usual initial top-level environment.

1. (+ 3 5)
2. ((lambda (x)(/ x 10))100)

**Design Check Problems**  For the following questions, evaluate each of the expressions on paper, writing out a description as in Worked Example 1. If it helps you to draw a diagram, you may do that as well. Be sure to record the final value of the expression.

1. (((lambda (x y)((lambda (y)(+ x y))x))17 18)
2. (((lambda (x y)((lambda (x)(+ x y))x))17 18)
3. (((lambda (x y)((lambda (x)(+ x y))y))17 18)
4. (define fact (lambda (x)(if (zero? x) 1 (* x (fact (- x 1))))))(fact 3)
5. (define x 3)((lambda (y)(+ x y))5)
6. (define x 3)((lambda (x)(+ x x))5)

8  Handing In

8.1 Design Check

Design checks will be held on Nov. 6-8, 2018. We will send out an email detailing how to sign up for design checks, so please check your inbox periodically, and sign up as soon as possible.

**Reminder:** You are required to pair program for this CS 17 project. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check until you have one. You must work with a different partner on each project.

You and your partner should do the following to prepare for your design check:
- Fill in the definition for if-expression in the template code.

- Practice evaluating Rackette expressions by hand. Specifically, write out your evaluations for the “Practice on Paper” section of this assignment and bring them to your Design Check. Although you don’t need to follow this format exactly, be sure that it is sufficiently clear what happens at each step of the problem.

- Write your initial top level environment (i.e. fill in initial_tle in the template) with at least two, but preferably all the built-in procedures. (The first one is the hardest.)

- Write out the rules of evaluation for and expressions and for or expressions.

- Give a few examples of each defined data type in the rackette.ml file.

- For each variety of Rackette expression, be prepared to describe a strategy for evaluating it (i.e., converting it from an abstract_program_piece to a value).

Before your design check, upload your answers to these questions to Gradescope. You must upload your files before your design check, even if that slot is before the official Gradescope deadline. Only one partner should hand in the files.

During the design check, you will discuss your solutions to each of the questions above with a TA. Please come prepared! The better prepared you are, the more productive the design check will be, and if you come away from the design check with a good understanding of how to proceed, you should do well on the project. Please also come on time—arriving late to your design check may result in a deduction.

### 8.2 Final Hand-in

The final hand-in is due by 7:00 PM, Nov 17, 2018. Hand in your files via Gradescope. Only one person in your group should turn in the project.

For the final hand-in, you are required to hand in five files:

- README.txt,

- rackette.ml, which should contain all your work, starting from the template file we provided, and leaving all the template code (especially the type signatures) unchanged, except for the parts labeled “TODO” or function-bodies that currently say things like failwith "eval is not yet implemented", which should be replaced by actual implementations.

- read.ml, which you should not alter,

- CS17setup.ml, which you should not alter,

- repl.ml, which you should not alter, and which many of you may not use at all, but should still hand in.

All your code should be fully commented.

In the ‘README’ file, you should provide:
• instructions describing how a user would interact with your program (what would they input? what would they expect as output?)

• an overview of how all the pieces of your program fit together (when a user provides an input, what series of procedures are called in order to produce an output?)

• a short description of any possible bugs or problems with your program (rest assured, you will lose fewer points for documented bugs than undocumented ones)

• a list of the people with whom you collaborated

• a description of any extra features you chose to implement

8.3 Grading

The design check counts for 20% of your grade. Specifically,

• Define the IfE: 1 points

• Explain and complete the commenting for all the types in the template: 4 points

• Provide examples of each type, including IfE: 3 points

• Complete your initial top-level environment: 4 points

• Explain your strategy for evaluating the parsed Rackette expressions: 4 points

• Complete the practice problems: 4 points

Functionality counts for 80% of your grade. Specifically, you will be graded on your program’s correct implementation of the following:

• Your top level environment: 8 points

• Your parse procedure: 22 points

• Your add_definition procedure: 5 points

• Your eval procedure: 40 points

• Your string_of_value procedure: 5 points

Partial functionality merits partial credit.

You can lose up to 20% of your grade for bad style. Remember to follow the design recipe; that is, include type signatures, specifications, and test cases for all procedures you write. And add comments to any code which would otherwise be unclear.
9  REPL

This part of the project is strictly for fun. You can ignore this section and still get full credit for Rackette. 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 REPL, given working implementations of parse, eval, and print. Code that accomplishes this is available in /course/cs017/src/rackette/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:

```plaintext
> racketteRepl parse eval print
Rackette> (+ 3 4)
7
Rackette> (if true 7 666)
7
Rackette> (and (number? 4) (zero? 4))
false
Rackette> (define add3 (lambda (x) (+ x 3))) (add3 4)
7
```

If you want to try this for yourself, go to your rackette directory and run ocaml repl.ml.

10  Appendix: The Rules of Evaluation

The rules of evaluation describe how to assign values to expressions. Before we list the rules, we need a few definitions.

A binding is a pair consisting of an identifier and a value, which we represent with an arrow notation, so that, for example, \( x \mapsto 17 \) is a representation of a binding.

An environment is a set of bindings. For example, \( \{ x \mapsto 17, y \mapsto 18 \} \) is an environment. Note: there are many ways to represent such a set; we’ll use a sequence of bindings, with the lookup rule being that you lookup an identifier by looking at the bindings in order, and using the first one that’s a binding for that identifier.

A closure is a representation of a user-defined procedure. Our version of a closure is a triple consisting of

1. a sequence of identifiers, called the formal arguments
2. an expression, called the body, and
3. a birth environment, which we’ll describe further presently.

There’s one more kind of value in Rackette, a built-in procedure. For instance, the identifier + will be bound to a built-in procedure in the top level environment before any processing of any input even starts.

Our representation of a built-in procedure is a pair consisting of
1. a string, used as the printed representation of the procedure. (For instance, the addition procedure might have the string `<builtin proc:+>` as its first component.)

2. an Ocaml procedure that takes a list of values and produces a value. We’ll later describe what a value is, and what the procedure for a built-in looks like.

Here are the rules of evaluation for expressions:

- The value of a number in $T + E$ is the VNum representing that same number.
- The value of a lambda expression in $T + E$ is a closure composed of the lambda expression’s formal arguments, its body (an expression), and $E$ (its birth environment).
- **if** expressions: To evaluate `(if pred yes-exp no-exp) in $T + E$,
  1. Evaluate `pred` in $T + E$. Call its value `b`. If `b` is not a boolean, evaluation terminates in an error.
  2. If `b` is true, the value of the **if** expression is the value of the `yes-exp` expression in $T + E$.
  3. If `b` is false, the value of the **if** expression is the value of the `no-exp` expression in $T + E.

  **Note:** Although the entire **if** expression is parsed, only one of the `yes-exp` or `no-exp` expressions is ever evaluated.

- **cond** expressions: To evaluate `(cond ((pred expr)*) in $T + E$,
  1. Consider the first pair in the cond expressions. Let `p` be the value of `pred` in $T + E$.
  2. If `p` is not a boolean, evaluation terminates with an error.
  3. Otherwise, if the `p` is true, evaluate `expr` in $T + E$ to get a value `v`. Then `v` is the value of the cond-expression.
  4. If `p` is false, go to the next (pred expr) pair, and repeat the steps above.
  5. If the `pred` from every pair in the cond-expression evaluates to false, evaluation terminates with an error.

  (You might observe that the rule above could easily be implemented with recursion. Also: there’s no else in Rackette.)

- **quote** expressions: The result of evaluating `(quote datum)` is just the second item itself. (In Rackette, evaluation will involve converting a concrete_program_piece into a value).

- procedure-application expressions: To evaluate `(proc arg1 arg2 ...) in $T + E$,
  1. Evaluate `proc` to get a value `p`. If its value is neither a built-in nor a closure, evaluation terminates with an error. Otherwise, evaluate all of the arguments in $T + E$, obtaining a list of values, `actuals`.
  2. If `p` is a built-in procedure, then apply the built-in procedure’s associated function to `actuals` to get a result `v`; `v` is then the value of the procedure-application expression.
  3. If `p` is a closure, `c`, with environment $E'$, then
    (a) let `L` be an environment in which the formal arguments of the closure `c` are bound to the corresponding actual arguments, `actuals`. If the number of formals is not the same as the number of actuals, evaluation terminates in an error.
(b) Let $E'' = E' + L$.

(c) Evaluate the body of the closure $c$ in $T + E''$ to get a value $v$.

(d) The value of the anonymous procedure application is $v$.

Missing from the list above are the exact rules for or-expressions and and-expressions. You get to write those as clearly as you can, as part of the design check.

### 11 Extra Features

If you finish your project early, but want to keep working on Rackette, there are lots of additional features you can implement. Note that these are strictly for fun: they do not count towards your grade, and they certainly do not make up for missing features in the actual assignment. Email the TA list for some cool ideas!

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)