Project 2: Rackette

7:00 PM, Nov 11, 2019

Contents

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. [1]

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 ReasonML.

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* (we'll say what these are shortly).
2. Convert this concrete program into an abstract program, i.e., an internal representation of your Rackette program as a list of ReasonML 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 an expression in the Rackette BNF. 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, process it by changing a "top-level environment" in which defined values are stored.

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.

The document is very long. That's not because it's filled with many things you have to do, but instead because it's intended to provide you a very complete map of what's needed to get the job done, and guide you through it step by step.

[1]DrRacket produces its values in a slightly different way than your program will; take CS1730 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 `concreteProgram`, which is a `list(concreteProgramPiece)`.

What is a `concreteProgramPiece`, exactly?

```
type rawProgram = string;

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

type concreteProgram = list(concreteProgramPiece);

let read = (input : rawProgram) : concreteProgramPiece => {...}
let readAll = (input : rawProgram) : concreteProgram => {...}
```

You'll be using the `readAll` 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, double-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 `concreteProgram` 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.re`, which in turn contains the procedures `read` and `readAll`. Because we have `open Read` 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. In broad strokes, Rackette is like CS17's version of Racket, but without strings, comments, or `letrec`, and only allowing integers as numbers. Since most of the programs you've written in CS17 so far don't need strings or any numbers except integers, you can use lots of your homework handins (with comments removed) as test-cases for this project. (There's a slight sticking-point: you'll need to rewrite some function-definitions to use lambdas.) The remainder of this section gives the precise details of what is and is not allowed in Rackette; descriptions of things like **`and`** and the exact form that a lambda-expression must take are typically just a repetition of what you learned about these things earlier in the semester, repeated here so that you'll have them all in one place.

3.1 Syntax

A Rackette program is a sequence of *pieces*, each piece being either a definition or an expression. See Lectures 2-4 for more information on programs, definitions, and expressions.

A *definition* contains the keyword **`define`**, followed by a name and an expression, all enclosed in parentheses. (Note that this means that you *cannot* define a function by writing `(define (f x )( + x 2));` instead, you must write `(define f (lambda (x)(+ x 2)))`.

An *expression* is any one of the several things defined by the BNF description of Rackette below. Among other things, it can be a number, a name, an if-expression, or a cond-expression.

Before we look at expressions and definitions closely, let's briefly discuss Rackette's vocabulary. This vocabulary consists of:

- Numbers (specifically, integers) and Booleans
- Parentheses: used to group expressions
- Keywords: `define, lambda, let, if, cond, and, or, empty, true, false`
- Names: all sequences of non-special characters that are not numbers, booleans or keywords

Elements of the Rackette vocabulary can be combined to form Rackette expressions (and some vocabulary elements, like `empty` or the number 17, are already expressions in and of themselves!). All Rackette expressions fall into one of the following categories:

- A constant such as an integer, `true`, `false`, or `empty`
- A name
- One of the seven types of compound Rackette expressions:
  - 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 exp1 exp2)` where `exp1` and `exp2` are also Rackette expressions.
  - An **`and`**-expression of the form `(and exp1 exp2)` where `exp1` and `exp2` are also Rackette expressions.
  - A **`cond`** expression of the form `(cond (pred expr)+)` where `pred` is a Rackette expression, and `expr` is a Rackette expression. (Note that a cond-expression must contain at least one such pair!)
  - A **`let`** expression of the form `(let ((id expr)*)body)`, where `id` is a name, and `expr` and `body` are Rackette expressions. There may be zero or more bindings (i.e., `(id expr)` pairs) in the **`let`** expression.

  - A user-defined-procedure expression of the form `(lambda (id*)body)`, where `id` is a name and `body` is a Rackette expression; we'll call these "lambda expressions."
  - A procedure-application expression of the form `(proc arg*)`, where `proc` is a Rackette expression, and each `arg` is a Rackette expression.

A bit of notation: we use x* to mean "zero or more" things of type x, and x+ to mean "one or more" x's.

Here's the Rackette grammar in BNF:

⟨*program*⟩ ::= ⟨*piece*⟩+

⟨*piece*⟩ ::= ⟨*definition*⟩
  | ⟨*top-level-expression*⟩

⟨*top-level-expression*⟩ ::= ⟨*expression*⟩

⟨*definition*⟩ ::= `(define` ⟨*name*⟩ ⟨*expression*⟩ `)`

⟨*expression*⟩ ::= ⟨*constant*⟩
  | ⟨*name*⟩
  | ⟨*if-expression*⟩
  | ⟨*cond-expression*⟩
  | ⟨*lambda-expression*⟩
  | ⟨*let-expression*⟩
  | ⟨*proc-app-expression*⟩

⟨*if-expression*⟩ ::= `(if` ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩`)`

⟨*or-expression*⟩ ::= `(or` ⟨*expression*⟩ ⟨*expression*⟩`)`

⟨*and-expression*⟩ ::= `(and` ⟨*expression*⟩ ⟨*expression*⟩`)`

⟨*cond-expression*⟩ ::= `(cond (`⟨*expression*⟩ ⟨*expression*⟩`)+ )`

⟨*lambda-expression*⟩ ::= `(lambda (`⟨*name*⟩*`)` ⟨*expression*⟩`)`

⟨*let-expression*⟩ ::= `(let ((`⟨*name*⟩⟨*expression*⟩`)`*`)` ⟨*expression*⟩`)`

⟨*proc-app-expression*⟩ ::= `(`⟨*expression*⟩ ⟨*expression*⟩*`)`

Figure 1: CS 17 Rackette grammar.

Note that builtins are not included explicitly in this grammar of Rackette. This is because they are defined in the *Top Level Environment* (more on this later!).

3.1.1 Sidenote on builtins

Note that the *builtin procedures* of Rackette, namely `+`, `-`, `*`, `/`, `remainder`, `=`, `<`, `>`, `<=`, `>=`, `equal?`, `number?`, `zero?`, `cons`, `first`, `rest`, `empty?`, `cons?`, and `not`, are not part of the vocabulary; their names are treated simply as "Symbols", just like many other things. Note: "+", "-", "*", and "/" need to take in two arguments, and return an error for any other number of arguments, and that division should return integers. Your "=" procedure does not need to take inputs other than numbers however, your "equal?" procedure should.

Although at first glance, it may seem that **`and`** and **`or`** should be builtin procedures, there is actually a subtle difference. Both **`and`** and **`or`**, unlike other builtin 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 simulate 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 a name into the top-level environment (unless the name 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 name `x` will be bound to `5`.

There's one exception to this rule: if an name is defined in the top-level environment, and we process a *definition of that same name*, it's an error, and processing stops. [2]

In your Rackette implementation, one of the procedures that you will have to implement will be `addDefinition`. 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 name and an expression, we first need to evaluate the expression in the top-level environment. Then, we add the binding of the name 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 "let" and "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 names, such as `(+ x 5)`, Racket, upon reaching the name `x`, will look for a binding of `x` in $T + E$. One can do this by either forming the enviroment $T + E$ and doing a lookup, or by first trying to find a binding for `x` in E, and if this is not found, trying to find a binding for `x` in the top level environment T. If there is no binding for `x` in either environment, then evaluation terminates with an error. Note that local bindings take precedence. Thus, in the program

```
(define x 5)
(let ((x 0))
   (* x 5))
```

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.

[2] Note, however, that we may temporarily add a new binding for the name via some other means, such as using it in a `let` expression. Since we look up values in our sequence of bindings in order (from most recent to oldest), the new binding "shadows" the old one.

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 (which can also be found on the course website), you'll find the file `Rackette.re`, 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.re`, 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 hours—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, `initialTle`, and the procedures `parse`, `addDefinition`, `eval`, and `stringOfValue`.

- `initialTle` produces the top-level environment with all builtin procedures as described in the Rackette grammar. (See details below.)
- `parse` consumes the output of `readAll`, namely a Rackette program represented as a `concreteProgram` (one `concreteProgramPiece` for each "piece" of the program) and produces the equivalent `abstractProgram`.
- `addDefinition` consumes an `environment` and a `definition`, and returns a new environment with all the bindings from the original environment, and the new binding from the input definition as well. If the name was already defined, this procedure should report an error and execution should terminate.
- `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).
- `stringOfValue` consumes a `value`, and converts that value into a string. For constants like numbers and booleans, `stringOfValue` should return a string representation of that constant. For a builtin procedure like +, it should return a descriptive string, something like "builtin:+". However, when called on a closure, it can return an arbitrary string that doesn't contain complete information about the closure (Racket chooses `(lambda (a1) ...)`). If at any point you don't know how to produce a string representation, evaluate a similar expression in Dr.

Racket and look at the printed representation that you get! 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), `readAll` outputs the corresponding `concreteProgram`:

```
> readAll "(define x 15) (if (= x 16) 100 -1)"
- : concreteProgram =
[List [Symbol ("define"), Symbol ("x"), Number (15)];
 List
  [Symbol("if"), List [Symbol ("="), Symbol ("x"), Number (16)], Number
      (100),
   Number (-1)]]
```

Given this `concreteProgram` as input, `parse` outputs an `abstractProgram` that can be interpreted to mean "I have 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 name `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, `stringOfValue` transforms its input from a `value` into a string: i.e., `"-1"`.

5.3 Check Expects

Unfortunately, ReasonML can't cleanly print the types we use in this project. If you write a check expect for a procedure that outputs a concrete program and your check expect is wrong, you would hope to get some helpful output that compares the expected value and the actual value. However, what ReasonML actually prints when you write a check expect doesn't look like a concrete program; it looks a lot messier. Because of this, we have provided you with a different set of check expect procedures. In `CS17SetupRackette.re`, we have provided you with the following procedures:

- `checkExpectName`
- `checkExpectDefinition`
- `checkExpectAbstractProgramPiece`
- `checkExpectAbstractProgram`
- `checkExpectConcreteProgramPiece`
- `checkExpectConcreteProgram`

- `checkExpectExpression`

The check expect procedure that you should use depends on the output of the procedure you're running. For example, if you are testing `parse`, which outputs an `abstractProgram`, you should use `checkExpectAbstractProgram`.

You can use these checkExpect procedures the same way you did in homework 7; they take in 3 arguments: the actual value (which is usually just you calling the procedure), the expected value, and some string label. For example, if you are testing some procedure `exampleProc` that outputs a `concreteProgramPiece` and you believe the output should be Number(5), you could write:

```
checkExpectConcreteProgramPiece(exampleProc(arg1, arg2), Number(5), "exampleProc
 test")
```

You may notice that we don't give you a checkExpectValue to test procedures that output a value. If you want to test something that outputs a value, you need to use your own stringOfValue procedure to turn your value into a string, and then you can just use regular checkExpect to compare that string to the expected string, since the normal checkExpect works on strings.

5.4 Error Checking

One very important function of an interpreter is to report an error on bad input (e.g., invalid Rackette 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 contain your own aggravation 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 raw program text.
- Something other than a keyword or an expression that evaluates to a procedure follows a left parenthesis.
- The user attempted to use a number, boolean, keyword, or other expression in place of a name 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. tried to use + on two booleans.

- There was a procedure application, but the procedure requires a different number of arguments than it was applied to.
- A user attempts to define an name more than once in the same let expression.

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 Rackette 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.

5.5 Builtin procedures

The rules of evaluation say that a builtin procedure is applied to a list of values. Our closure representation of a builtin is `VBuiltin(string), (list(value)=> value)`, where the string is the printed representation of the builtin, and the other component of the ordered pair is a function 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:

```
Vbuiltin ("<builtin-proc-+>", plus)
```

where `plus` is a procedure you've defined which takes a `list(value)` as input, and produces a `value` as output.

Here are the steps you might use in writing `plus`:

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

On the other hand, you could be far more compact, and just write a simple anonymous function rather than defining `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 builtin 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 How to get this project done

6.1 Setting up

To start out, create a folder/directory either on the department filesystem (in your cs0170 directory, just as you've done for other assignments) or on your home computer if you prefer. Copy the file

`/course/cs0170/src/rackette/Rackette.zip` in the directory you'd like to work in (this folder is also available on the course website), and unzip it. You'll find five files inside:

- `Rackette.re`: a template; all your code goes in here
- `README`: your README goes in here
- `Read.re`: contains our support code (do not alter)
- `Types.re`: contains all type definitions (do not alter)
- `CS17SetupRackette.re`: a copy of the Rackette-specific teachpack (do not alter)

Once you've gotten access to these files, you're ready to get to work.

6.2 Moving forward

Here's a suggested path to completing the project which could work for you. Of course, many variations are possible as well.

- 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.*
- Import your "environment" code from the lab, and write something that creates a bogus top-level environment in which there are no bindings. Add a big comment noting that this is broken and that you need to fix it!
- Get parsing of expressions working in a really stupid way. Make every expression parse to something like the constant expression **`true`**. Use a `switch` expression, but only have a single pattern – a wildcard – and have it produce the bogus value. Put a big fat comment on this saying that it's broken and needs to be replaced.
- Get parsing of definitions working. What must a `concreteProgramPiece list` look like to be a definition? Write a pattern that matches that, and include a wildcard pattern for things that don't match it, and write a coherent error message for things that match the wildcard. For things that match your pattern, use your (not yet working) `parse` procedure to parse any expressions. That way, if you parse (**`define`** x (+ 3 1)), the result should be something like (`ID "x"`, `Bool(`**`true`**)). Why that? Because your placeholder expression-evaluator evaluates every expression to **`true`**! That's OK, though: it means that your parsing of definitions is working, or *will* work properly once parsing of expressions works.
- Get parsing of expressions working a little better: try to recognize the expression **`empty`**, for instance, and return a reasonable parse for that ... but leave everything else parsing to **`true`** still.
- Get parsing of other constant expressions — numbers, booleans — working properly. Switch your "bogus parse" answer to be something like the number -999, now that "true" is a valid parse of something you handle correctly!

- Take a break from parsing, and do something simple: write `stringOfValue`, or at least write a draft of it. You know how to make a string representation of everything except builtins and closures, but perhaps `Lists` are a little intimidating. So use a `match` expression, and handle these last three with a `failwith`. (Make a note that you need to fix this, though!)

- Now that you can parse one or two things, write a first draft of `eval`. Have it evaluate everything to some crazy number like `VNum`(-1234), but put in a comment saying that you need to fix this. Use a `switch` expression with a wildcard to generate this bogus result.

- Try to combine parsing and evaluation. See whether you can parse and evaluate a simple program that consists of just one expression. The result, of course, should be your bogus answer (e.g., -1234). Check that if you write a simple program that consists of a single definition, that it does *not* work. Why doesn't it work?

- You'd like to continue expanding `eval`, but the next logical step is to evaluate names by looking them up in some environment. Even if you write this perfectly, you should always end up with an interpreting error, because you haven't got a way to put bindings into your environment yet. So start by filling in `addDefinition` and seeing whether it works.

- Extend evaluation to handle names, by putting in a new match-case before the wildcard.

- Take a deep breath, and stop working on evaluation. Before you do so, change the wildcard in the evaluation procedure to produce a `failwith`. Why? Because you've used the bogus value to let you test the end-to-end functioning of your parser and evaluator, but now it's time to make sure that when you do something that's *not* properly handled, you know about it. Fix the wildcard in the expression parser to generate a `failwith` as well.

- Now that you've made evaluation work, at least partly — enough to test a few things — it's time to finish parsing. What remains is to parse other `concreteProgramPiece lists`. How many kinds of lists must you handle? Write a pattern for each one in your parsing procedure for expressions, and have each of these produce an informative `failwith`. Test out an if-expression, for instance, and be sure that parsing it fails, but fails by saying that it found an if-expression and that parsing of these isn't implemented yet. If it says that the failure was an un-parse-able procedure-application expression, your patterns need work!

- If you added enough cases to your parsing code, the wildcard should be redundant. You should be getting a "this pattern is never matched" error for that line of the parser. If you're not, you need to figure out why. If you are, then you can remove that line.

- Now actually parse an if-expression. You know that it has to contain three other expressions, right? You need to be sure that your purported if-expression really *does* contain those, and if so, you need to parse each one. Hint: recursion.

- Keep at it: parse the remaining few kinds of expressions. Perhaps **`cond`** is a little tricky, because you have to check that there's at least one test-answer pair in the cond expression. If that seems intimidating, skip it for now, and leave in the informative `failwith`.

- Now, step by step, start improving `eval`. You can already handle constants and names. What other kinds of expressions will eval have to handle? Probably . . . about seven others. Make a pattern for each one, and then handle each pattern with a `failwith`. And change the wildcard to produce a `failwith`, too, if you didn't do so already.

- If your `eval` has enough patterns, then the wildcard pattern at the bottom should, like the one in the parser, have become redundant. Check to see that you get a warning. If you do, remove the wildcard case. If not, figure out why not. (Hint: comment out the wildcard case, and ReasonML will tell you something that you've forgotten to match!)
- One by one, replace those "`failwith`" cases in `eval` with code that implements one of the rules of evaluation. If you get stuck on one, move on to the next and come back to the tricky one later.
- That initial top-level environment really *does* need to have some things defined in it. For your design check, you were supposed to flesh it out, but if you didn't, start now. Implement the `plus` procedure as described earlier, and create a `VBuiltin` with a nice "printed representation" for the addition procedure, and your `plus` procedure at its other half. The resulting thing is a nice `value`, so you can create a binding in which the name + is bound to this `VBuiltin`. And you can add this binding to an empty environment and call this your top-level environment (with a big comment saying that you need to add a lot more things. Why not list them right there, so you don't forget?).
- Now try processing a program like `(+ 1 2)` and see whether it works. Try `(define x 2)(+ x 1)`, and see whether *that* works.
- Add more builtins to the initial top-level environment. 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.
- They're probably *not* handled right, because you've got some of those earlier comments warning you about things that aren't implemented, like the printed representation of a list, or of a builtin. Fix those missing bits one by one.

The last few steps don't have a lot of detail, but that's because by this point, you should pretty much understand where you're headed. It'll be pretty exciting when you're done, we promise.

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. `"+"` ↦ a builtin-procedure that takes two numbers (represented as values) and produces their sum (represented as a value)
2. `"-"` ↦ a procedure that subtracts two numbers (in the same way as "add")
3. `"*"` ↦ a procedure that multiplies two numbers (in the same way as "add")
4. `"/"` ↦ a procedure that divides two numbers (in the same way as "add")

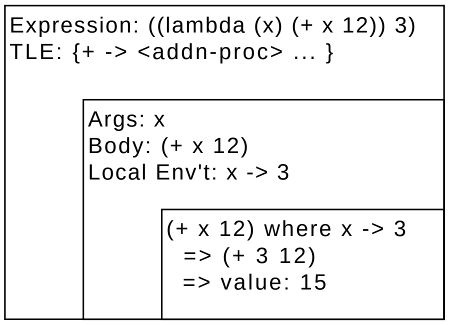
5. `"zero?"` ↦ a procedure that checks if a number is zero (in the same way as "add", i.e., operating on and producing values)

6. `" > "` ↦ 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 `(define z 2)(+ z 8)`.

- Initially, the top level environment is { + ↦ `addn-proc`, ... } and the local environment is empty.
- After processing the definition, the top level environment is { `"+"` ↦ `addn-proc`, ..., `z` ↦ `VNum` 2 } The local environment is still empty.
- To evaluate `(+ z 8)`, Rackette will evaluate the name + 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 ((**lambda** (x)(+ x 12))3):



```
Expression: ((lambda (x) (+ x 12)) 3)
TLE: {+ -> <addn-proc> ... }

    Args: x
    Body: (+ x 12)
    Local Env't: x -> 3

        (+ x 12) where x -> 3
          => (+ 3 12)
          => value: 15
```

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 name + 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. `(let ((x 19))(+ x 32))`
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. `(let ((x 0))(let (( f (lambda (a)(* x a ))))(let ((x 1 ))(f 5))))`
5. `(let ((x 0)(y 18))(let ((f (lambda (a b)(+ x b )))(x 17))(f y x)))`
6. `(+ 3 5)`
7. `(define fact (lambda (x)(if (zero? x) 1 (* x (fact ( - x 1))))))(fact 3)`
8. `(define y 17)(let ((y 3))(+ y 7))`

8 Handing In

8.1 Design Check

Design checks will be held on Oct 29-31, 2019. 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 and your partner should do the following to prepare for your design check:

- Practice evaluating Rackette expressions by hand. Specifically, write out your evaluations for the "Practice on Paper" section of this assignment and take a picture to submit on Gradescope. Please submit this on Gradescope before your design check starts. Although you don't need to follow our example 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 `initialTle` 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 `Types.re` file.
- For each variety of Rackette expression, be prepared to describe a strategy for evaluating it (i.e., converting it from an `abstractProgramPiece` to a `value`).

8.2 Final Handin

The final handin is due by 7:00 PM, Nov 11, 2019 on Gradescope. Only one person in your group should turn in the project.

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

- `README.txt`,
- `Rackette.re`, 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.re`, which you should not alter,
- `Types.re`, which you should not alter,
- `CS17SetupRackette.re`, which you should not alter,

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,

- Be able to understand and explain each of the variant types defined in the template: 4 points
- Provide examples of each type,: 3 points
- Complete your initial top-level environment including at least two built-in procedures (ex. +,-, ...) : 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 `addDefinition` procedure: 5 points
- Your `eval` procedure: 40 points
- Your `stringOfValue` procedure: 5 points

Partial functionality gets partial credit.

You can lose up to 20% of your grade for bad style. Remember to write the I/O statement, type annotation in procedures themselves or commented out type signature, check expects, and that Design Recipes need to be written for ALL procedure you write. This includes the "rackette" procedure. Add comments to any code which would otherwise be unclear. You will not receive full credit for style if a TA is ever in doubt about what a section of your code is doing or how it works.

9 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 name 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 a name by looking at the bindings in order, and using the first one that's a binding for that name.

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

1. a sequence of names, called the *formal arguments*
2. an expression, called the *body*, and
3. an environment, which we'll describe further presently.

There's one more kind of value in Rackette, a *builtin procedure*. For instance, the name + will be bound to a builtin procedure in the top level environment before any processing of any input even starts.

Our representation of a builtin 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. a ReasonML function that takes a list of values to a value. We'll later describe what a value is, and what the function for a builtin looks like.

Now, the possible values in Rackette are constants, builtin procedures, and closures (the representation of user-defined procedures).

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 the boolean false is `VBool(false)`, and the value of the boolean true is `VBool(true)`.
- The value of a λ-expression in $T + E$ is a closure composed of the λ-expression's formal arguments, its body (an expression), and E.
- `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.)
- `let` expressions: To evaluate `(let ((id expr)*)body)` in $T + E$,
  1. Let L be a new empty environment.
  2. For each `(id expr)` pair, do the following:
     (a) Evaluate `expr` in $T + E$. Call its value v.
     (b) Add the binding $id \mapsto v$ to L.
  3. Let E' be the environment $E + L$ (i.e., with all of the `id` $\mapsto$ `expr` bindings added). The value of the `let` expression is the value of `body` in $T + E'$.

- procedure-application expressions: To evaluate `(proc arg*)` in $T + E$,
  1. Evaluate `proc` to get a value p. If its value is neither a builtin nor a closure, evaluation terminates with an error. Otherwise, evaluate all of the arguments from `arg*` in $T + E$, to a list of values, *actuals*.
  2. If p is a builtin procedure, then apply the builtin 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.

10 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!
