Project 2: Rackette

CS17 Fall '25 | Brown University

Due: 11:00 PM, November 7, 2025

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 CS17'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 in the form of strings that can be printed (i.e., shown to the user, not 'sent to a printer'). 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. We'll use italics to introduce new-and-as-yet-undefined terms, as we've done here.
2. *Parse* 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*. Each abstract program piece will correspond to
   - a definition to be added to the top level environment, or
   - an expression.
3. *Process* the resulting internal representation, one abstract-program-piece at a time.

[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!

(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 convert that value into a string and print it.

(b) If a piece is a definition, process it by changing[2] 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. You can see how we did this by looking at `Read.re`.

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.

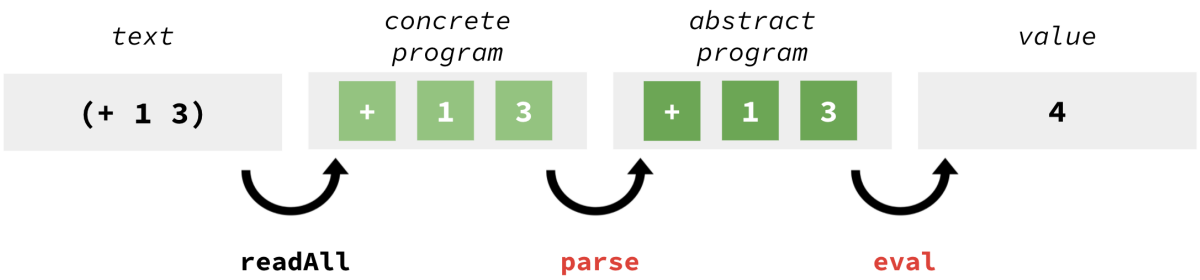


Figure 1: The read, parse, and evaluation process of interpretation. The steps in red are what you will be responsible for implementing. There is a more in-depth version of this diagram in section 5.4.

2 Rackette Roadmap

This project may seem overwhelming at first. However this document, alongside the Rackettecita homework, design check problems, gear up, and Rackette Guide provide a comprehensive map to the project. If you carefully use these resources, you will be very prepared to tackle Rackette.

Here's a suggested list of steps to start off the project:

1. Read this handout in full.
   - This document is very long, not because it's filled with many things you have to do, but instead because it's intended to guide you through the project step by step.
2. Complete the design check tasks.
   - You may want to reference the Rackettecita homework for a better conceptual understanding.
3. Read the Rackette Guide document and work your way through it.
   - The Rackette Guide provides a suggested road map for the actual coding aspect of Rackette, and other useful information.
4. If you'd like, print out files you find yourself needing to reference frequently.
   - Rackette Guide

[2]This is an informal description. We're not actually changing a defined value!

- `Types.re`
- Provided input-output pairs for parsing, processing, and evaluation
- The Appendix of this handout (Section 11), which functions as an implementation checklist

3 The "read" procedure

The `read` procedure that we've provided reads in a `rawProgram` (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)`. You've already encountered this in the Rackettecita homework.

What is a `concreteProgramPiece`, exactly? Here's the code that defines it.

```
type rawProgram = string;

type concreteProgramPiece =
  | NumberC(int)
  | SymbolC(string)
  | ListC(list(concreteProgramPiece));

type concreteProgram = list(concreteProgramPiece);

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

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 which 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 `NumberC` or a `SymbolC` or a `ListC`. A `NumberC` is a bit of text that looks like a number; a `SymbolC` is a bit of (non-punctuation) text that looks like anything except a number; and a `ListC` is a bunch of `NumberC`s, `SymbolC`s, or `ListC`s between matching parentheses[3].

Each subsequent step of the project can similarly be regarded as a transformation between "languages."

The next step, parsing, takes certain concrete program pieces and restructures them into *abstract* programs (while rejecting others as syntactically invalid), 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 source code for this assignment includes the file `Read.re`, which contains the procedures `read` and

[3]Notice that all the concrete program piece types have constructors ending with the letter "C", for concrete. Expressions, which form the bulk of the abstract-program representation we'll look at next, all have constructors ending in the letter "E." And values, which are what evaluation eventually produces, have constructors ending in the letter "V."

`readAll`. Because we have `open Read.Reader;` at the top of the stencil code, you'll be able to refer to these procedures in your own code.[4]

You can look through the `Read.re` code and probably make some sense of it. Here's a very brief description for the inquisitive. The `read` procedure basically converts the input string to a long sequence of characters, looks for the first non-blank character, and tries to work out what's there. If it's a digit, it tries to assemble subsequent digits to create a number; if it's a left-parenthesis, it looks for a matching right-parenthesis, and then tries to (recursively) read everything between these two. And if it's anything else, it gathers up characters up to but not including the next whitespace or parenthesis, treating those characters as a symbol.

4 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, `let*`, or `letrec`, and only allowing integers as numbers. Furthermore, `cond` expressions are placed in matching *parentheses* instead of matching brackets.[5]

Because most of the programs you've written in CS17 so far don't need strings or any numbers except integers, you can use many 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` (the Racket version) 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.

4.1 Syntax

A Rackette program is a sequence of *pieces*, each piece being either a definition or an expression. See classes from Week 1 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-expression, a name-expression, 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

[4] The `.Reader` part refers to the module `Reader` within `Read.re`.
[5] By the way, these also work in Racket — we just never told you about them. Try it out!

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 `and`-expression of the form `(and exp1 exp2)` where `exp1` and `exp2` are also Rackette expressions. (Note that in Rackette, `and` is always used with exactly two expressions; this is different from Racket.)
  - An `or`-expression of the form `(or exp1 exp2)` where `exp1` and `exp2` are also Rackette expressions. (Same note about exactly two expressions.)
  - An `if` expression of the form `(if pred yes-exp no-exp)` where `pred`, `yes-exp`, and `no-exp` 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 `lambda` expression of the form `(lambda (id*) body)`, where `id` is a name and `body` is a Rackette expression; these are used to create user-defined procedures.
  - 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 procedure-application expression of the form `(proc arg*)`, where `proc` is a Rackette expression, and each `arg` is a Rackette expression.

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

Figure 2 shows the Rackette grammar in BNF:

⟨*program*⟩ ::= ⟨*piece*⟩`+`

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

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

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

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

⟨*constant-expression*⟩ ::= `true`
| `false`
| `empty`
| ⟨*number*⟩

⟨*name-expression*⟩ ::= ⟨*name*⟩

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

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

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

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

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

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

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

Figure 2: CS17 Rackette grammar

4.1.1 Sidenote on builtins

Note that the *builtin procedures* of Rackette (which you are responsible for implementing), 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.

Your implementation of the builtins "`+`", "`-`", "`*`", and "`/`" must take in exactly two arguments, and return an error for any other number of arguments. The builtin division procedure `/` should return integers, i.e., should behave like Racket's `quotient` procedure. Your "`=`" procedure does not need to accept inputs other than numbers. But 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` 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`. This is in direct contrast to the rule for evaluating procedure-application expressions, which is why `and` and `or` are keywords, and `and`- and `or`-expressions are handled specially. You will need to implement this behavior in your Rackette implementation.

4.2 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, producing a new 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.

5 The Top Level Environment

5.1 Overview

Recall from Lab 8 that an environment is a *list* of binding-lists. For the top-level environment, this list will contain exactly one binding-list. When we talk about "adding something to the top-level environment", we mean adding it to the sole binding list in the top-level environment. When we speak of adding a list S of bindings to the current environment, we mean taking the current environment (a list of binding-lists) and extending it by including S to make a new, slightly longer, list of a binding-lists.

The initial top level environment contains a single binding list, which contains bindings of all of 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, you'll need to build an initial top level environment containing a binding-list with bindings for the builtin procedures mentioned in section 3.1.1. Every time your program is run, this top level environment should be initialized.

5.2 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 sole binding list in the top-level environment to create a new binding list in which the name `x` will be bound to the value `5`. That new binding list becomes the sole contents of a new top-level environment, which will be used for all subsequent processing.

There's one exception to this rule: if a name is defined in the top-level environment, and we process a ***definition*** *of that same name*, it's an error, and processing stops.[6]

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 a name and an expression, we first need to evaluate the expression in the top-level environment to get a value v. Then, we add the binding of the name to the value v into the top level environment's sole binding list to create a new, richer, top-level environment.

5.3 Evaluation Process

In several of the rules of evaluation, we have a top-level environment, T, and another environment, E, that we'll call the "local environment." Since E is often altered (see the "let" and "lambda" rules), it's

[6]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.

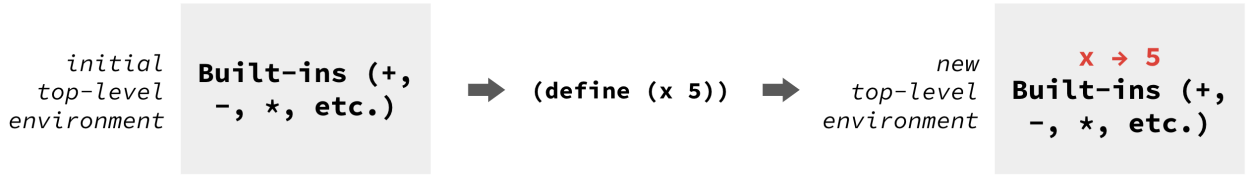


Figure 3: How definitions are added to the top-level environment in Rackette.

helpful to make "evaluation" operate on three operands: an expression, a top-level environment, and a local environment.

As we've said repeatedly, the top-level environment consists of a single binding list. The 'local environment', however, may consist of several binding lists – those added by let-expressions, lambda expressions, etc. Consider the following short program:

```
(define a 5)
(define f (lambda (x) (+ x a)))
(let ((t 4)) (f t))
```

In the final step— the evaluation of `(f t)`, the local environment contains

1. the let-binding of `t` to 4,
2. the third part of the closure representing f, which contains no bindings
3. the temporary binding of the formal argument, `x`, to the value of `t` (i.e., 4)

where each of these is in its own separate binding list.

When evaluating an expression with names, such as `(+ x 5)`, Rackette, upon reaching the name `x`, will look for a binding of `x` 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 `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 must implement this evaluation process.

5.4 Big Picture: Evaluation and Definition

Processing a program consists of processing the first program piece, and then processing the remainder of the program, which sounds very recursive, and actually is.

We handle two kinds of processing: processing definitions, and processing expressions.

Processing a definition *alters* the top-level environment; processing an expression *uses* the top-level environment and local environment for lookups, etc.

If we have one definition after another, the second definition needs to get processed using the *altered* top-level environment produced by the first definition. That suggests that our program should consume a program *and* a top-level-environment, and then somehow use, in recursive calls, a potentially *new* top-level-environment.

With this in mind, we can look at the central procedure, `process`, the main procedure, `rackette` that calls it, and see how together they implement these ideas. Here's the code:

```
let process: abstractProgram => list(value) =
  pieces => {
    let rec processHelper: (environment, abstractProgram) => list(value) =
      (tle, pieces) =>
        switch (pieces) {
        | [] => []
        | [Definition(d), ...tl] => ...
        | [Expression(e), ...tl] => ...
        };
    processHelper(initialTle, pieces);
  };

let rackette: rawProgram => list(string) =
  program => List.map(stringOfValue, process(parse(readAll(program))));
```

As you can see at the bottom, `rackette` takes the text of a program, applies `readAll` to it to produce a `ConcreteProgram`, which is a list of `ConcreteProgramPiece`s, and processes this list. Doing so produces a list of values, each of which we convert to a string, producing a list of strings as the overall output.

How does `process` itself work? It uses a helper that consumes a top-level-environment and an abstract program (a list of abstract program pieces, each of which is a definition or an expression), starting it with the initial TLE and the whole abstract program. The helper's input list of pieces is either `empty`, in which case it produces an empty list of values as output, or starts with a definition (which doesn't produce any new values, but which, when processed, produces a new top-level environment that should be used for subsequent processing), or starts with an expression, which should be evaluated, and the result of that evaluation should be the first item in the output list-of-values. In each case, the helper will need to call itself to perform further processing, but the *arguments used for that further processing may change*. The precise structure needed is given by the rules of processing in the appendix.

Filling in the two ellipses in the code above may be the single most conceptually difficult part of this assignment. You'll definitely be asked about this at your design check.

Once you think that you have the correct filled-in code, walk through this `processProgram` procedure with a tiny program or two, something like
`(define x 3) (define y 2) (+ x y)`, to confirm that it'll do its job correctly.

Figure 4 helps to summarize how all the parts of this project fit together during the processing of a top-level expression.

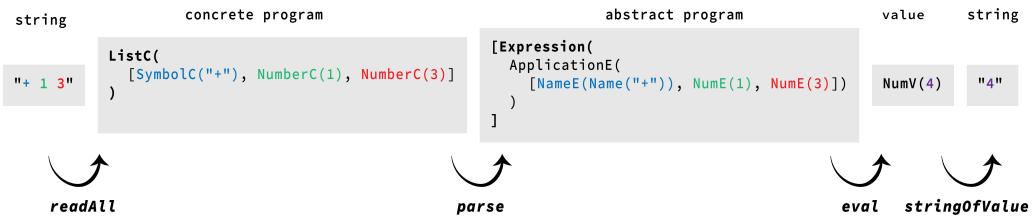


Figure 4: The pathway that Rackette takes to read, parse, evaluate, and print a top-level expression.

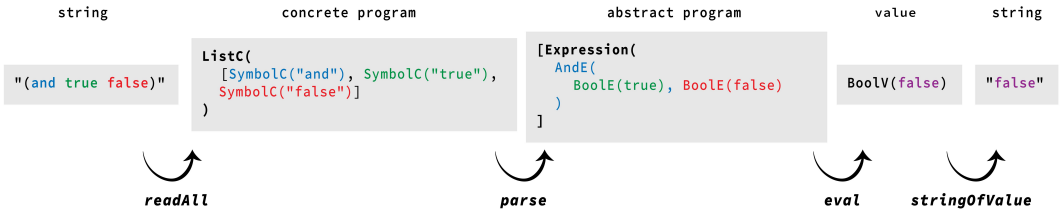


Figure 5: Another example for the pathway that Rackette takes to read, parse, evaluate, and print a top-level expression.

6 The Assignment

This project consists of five parts: the data definitions, the main procedures, creating the initial top-level environment, testing, and error checking.

6.1 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 an "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 Google Drive folder (linked on the course website) containing the source files, you'll find the file `TypesPREDESIGNCHECK.re`, which contains all of the data definitions you'll need. Your job is to write example data for each of these types. We've also left one type definition incomplete, and you'll have to fill it in for the Design Check. This will eventually be replaced with the actual `Types.re` file.[7]

You'll also find the file `Rackette.re`, which 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." In particular, the procedure `process`, which is supposed to produce a list of values, currently produces an empty list, and only processes the first piece in a program, because there's no recursive call. You'll need to replace the empty list (for both the definition- and the expression-case) with something better.

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.

6.2 Values and Procedures

Now, let's discuss the purpose of each value and procedure required.

Following your data definitions, you'll write the initial top-level environment, `initialTle`, and the procedures `parse`, `addDefinition`, `eval`, and `stringOfValue`.

[7]You will notice that many of the types incorporate records, but we have chosen not do so for the type `binding`. While using records for bindings would work, due to how you will use them, we decided, after trying it both ways, that it would be cleaner to define a binding as a name-value tuple.

- `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 a corresponding `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. But if the name was already defined in the environment, this procedure should report an error and execution should terminate.
- `eval` consumes two `environment`s—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), in which values are looked up first in E and then in T.
- `process` consumes an abstract program and produces a list of values that result from evaluating each top-level expression of the program in the appropriate environment, which it does by implementing the Rules of Processing. (Of course, along the way it also processes every definition, but this part of processing produces no output in the list of values.)
- `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) ...)`, but something like `<User-defined procedure>` would also be fine). If at any point you don't know how to produce a string representation, evaluate a similar expression in DrRacket and look at the printed representation that you get! You can model your representation after that.

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

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

Given this input (as a string), `readAll` outputs the corresponding `concreteProgram`:

```
> readAll("(define x 15) (if (= x 16) 100 -1)");
- : concreteProgram =
[ListC([SymbolC("define"), SymbolC("x"), NumberC(15)]),
 ListC([SymbolC("if"), ListC([SymbolC("="), SymbolC("x"), NumberC(16)]),
   NumberC(100), NumberC(-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` is bound to the value produced by evaluating `15` (i.e., `NumV(15)`). Then process calls `eval` with the enlarged TLE and an empty local environment, and the if-expression (`IfE`) representing `(if (= x 16) 100 -1)` to produce a value, represented as a `value`: i.e., `NumV(-1)`. Finally, `stringOfValue` transforms this `value` into a string: i.e., `"-1"`.

6.3 `initialTLE` and Builtin procedures

The rules of evaluation say that a builtin procedure is applied to a list of values. Our representation of a builtin is `BuiltinV(builtinData)` where
`type builtinData = {printedRep: string, bProc: list(value) => value),}`.
The `string` in `builtinData` 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: `BuiltinV({printedRep: "<builtin-proc-+>", bProc: 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 `NumV`s.
3. Extract the integers from those `NumV`s and add them, and
4. Wrap up the resulting integer in a `NumV`.

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!)

One last thing: all the bindings for your builtins should be in one binding list, and that binding list should be the only one in the initial top-level environment. It's easy to forget that an environment is a *list* of binding lists!

6.4 Check Expects

Unfortunately, ReasonML can't cleanly print the types we use in this project. If you write a `checkExpect` for a procedure that outputs a concrete program and your `checkExpect` 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 `checkExpect` doesn't look like a concrete program; it looks a lot messier. Because of this, we have provided you with a different set of `checkExpect` 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 `exampleProcedure` that outputs a `concreteProgramPiece` and you believe the output should be `NumberC(5)`, you could write:

```
checkExpectConcreteProgramPiece(exampleProcedure(argument1, argument2),
   NumberC(5), "exampleProcedure test")
```

Here are some more examples of how you might use `checkExpect`s:

```
/* sample test: parseExpression on concreteProgramPiece */
checkExpectExpression(parseExpression(SymbolC("empty")), EmptyE,
    "parse empty expression");
/* sample test: parseExpression with read */
checkExpectExpression(parseExpression(read("empty")), EmptyE,
    "read and parse empty expression");
/* sample test: parsePiece on concreteProgramPiece */
checkExpectAbstractProgramPiece(parsePiece(SymbolC("empty")),
   Expression(EmptyE), "parse piece on empty");
/* sample test: parsePiece with read */
checkExpectAbstractProgramPiece(parsePiece(read("empty")),
   Expression(EmptyE), "read and parse piece on empty");
/* sample test: evaluation with stringOfValue */
checkExpect(stringOfValue(eval(initialTle, [], EmptyE)), "empty",
    "evaluation of empty list");
/* sample test: evaluation with stringOfValue, parse, and read */
checkExpect(stringOfValue(eval(initialTle, [],
    parseExpression(read("empty")))),"empty", "evaluation of empty list
    with parse and read");
```

Note that we have included many example tests that involve calling multiple procedures on an input; for example, we both directly test `parseExpression` on an `EmptyE`, and on the output of calling `read` on the `string` "empty". You should aim to do the same in your testing, and call many combinations of procedures in order to ensure that everything you have written is working properly!

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.

In addition to providing you with `checkExpect`s, we have also written the procedure `checkError`. You are expected to write test cases for `failwith`s using `checkError`; there are more details on how to do this in the section below.

6.5 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 limit your own exasperation 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[8], and we cannot reasonably return a value in this case.

In addition to testing your procedures with correct inputs, we also ask that you test your procedures on *incorrect* inputs; that is, situations where you would expect your code to produce a `failwith`. To do so, you will use the procedure `checkError`, which is also provided for you in `CS17SetupRackette.re`. Here is an example of how to use `checkError`:

[8]More precisely, evaluating the number-expression `2` produces a value that is neither a built-in-procedure-value nor a user-defined-procedure-value.

```
checkError(() => rackette("(+ 3 false)"), "+ takes in two integers");
```

`checkError` takes in two inputs: the first is a call to your procedure, in a situation where you expect it to throw a `failwith`; the second is the `string` of text associated with your `failwith`. There is a subtlety here, however: the first input to `checkError` is actually a *procedure*, so rather than simply calling the procedure you want to error check (for example, `rackette`), you will have to input an anonymous procedure, of the form `() => rackette("input goes here")`.

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:

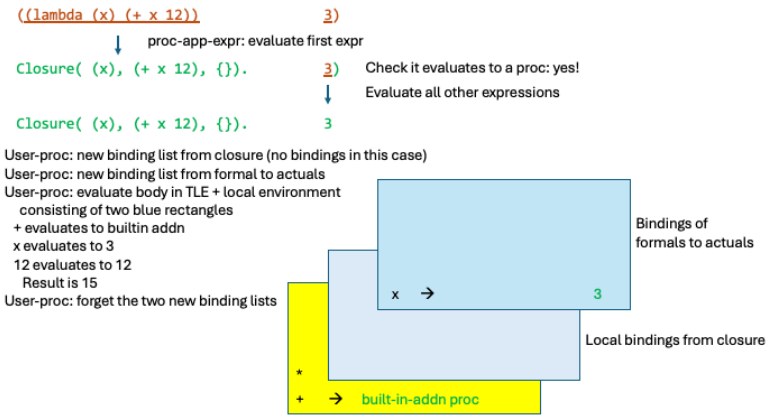
1. `"+"` $\mapsto$ a builtin-procedure that takes two numbers (represented as values) and produces their sum (represented as a value)
2. `"-"` $\mapsto$ a procedure that subtracts two numbers (in the same way as "add")
3. `"*"` $\mapsto$ a procedure that multiplies two numbers (in the same way as "add")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

While names in Rackette are represented by strings, in the following examples, we'll leave off the quotation marks, writing `+` for the thing that's bound to the built-in addition procedure, for example, and similarly for all other names.

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

- Initially, the top level environment is [{ `+` $\mapsto$ `addn-proc, ...` }].
- After processing the definition, the TLE is [{ `z` $\mapsto$ `NumV(2)`, `+` $\mapsto$ `addn-proc,...` }] 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 (in the top-level-environment) to a builtin procedure that adds its two arguments. Rackette will then do the same for `z` and find that it is bound, in the top-level-environment, to the value `NumV(2)`. Rackette will evaluate the number-expression `8` the get the value `NumV(8)`. And finally Rackette will apply the builtin addition procedure to `NumV(2)` and `NumV(8)` to get `NumV(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)`:



This requires some explanation.

1. The black words are shorthand reminders of rules of processing/evaluation.
2. When it says "new binding list from closure", what this means is "create an environment, which will eventually consist of the two blue rectangles, where each rectangle is a binding-list. The first of these binding lists is the local environment of the closure – which happens to contain no bindings at all. This sequence of additional binding lists will be used in addition to the TLE, when we're looking up bindings.

The first line shows the Rackette expression to evaluate, underlined; the second shows what happens when we evaluate the first item in the proc-app-expression. The third shows what happens when we evaluate the other item. The remaining remarks show how to get a result from the resulting two values. The lower right shows the various environments. Each arrow corresponds to a step in the rule for evaluating procedure-application expressions. The "evaluate the body" step has been compressed into a paragraph, and the removal of the temporary bindings isn't shown.

Keep in mind during the design check problems that the innermost, newest binding list will take precedence over all outer binding lists.

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

in either the worked example 1 or worked example 2 format.

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. `(let ((x 0) (y 18)) (let ((f (lambda (a b) (+ x b ))) (x 17)) (f y x)))`
3. `(+ 3 5)`
4. `(define y 17) (let ((y 3)) (+ y 7))`

8 SRC: Accessible Design

"Digital Accessibility means that everyone can use the exact same technology as anyone else—regardless of whether they can manipulate a mouse, how much vision they have, how many colors they can see, how much they can hear, or how they process information." [9] As a programmer, it is vital that you consider accessibility in the programs you write. In this section, you'll learn about different aspects of accessibility, and think about what decisions you can make to serve accessibility.

You will submit all your answers to the Rackette SRC assignment on Gradescope in a text file named `rackette-src.txt`, and come prepared to discuss the last question with your design check TA.

Warmup Task: Web accessibility is important to ensure equal access to everyone, especially people with disabilities. It also makes websites more usable and intuitive to use. The Web Accessibility Evaluation Tool (WAVE) is a resource that identifies ways for sites to improve their accessibility.

Try out at least two websites. What kinds of errors were the most common? What surprised you about the results? Make sure to click the reference icon to see each feature's impact on accessibility. *(2-3 sentences)*

Accessibility is a crucial aspect of designing websites. But for Rackette, you're creating something very different: a programming language! Programmers are users too, and they can still benefit from a variety of features that make coding more inclusive. Read the two examples below. If you're curious about other factors that could contribute to accessibility in programming languages, see additional examples here.

(1) *Variable Names*

After Spike wrote the book "Computer Graphics: Principles and Practice, 3rd edition," his wrist hurt for the following year. Why? To generate the graphics, he was using the MATLAB computing environment, which requires complicated keystrokes. He was frequently holding down ctrl+shift+h at the same time with one hand while using the mouse with the other. Try to hold that key combination yourself!

Characters that are difficult to type make the software or programming language impossible, difficult, or strenuous to use for programmers. An example in programming is variable name conventions. The two most common naming conventions are snake_case and camelCase. The former separates words using underscores, and the latter separates words using capital letters. However, typing in either style involves using the Shift key, which poses a challenge for users with mobility-related disabilities. One solution is kebab-case, which separates words using hyphens instead. However, many languages do not accept hyphens in variable names, since they already represent the minus sign. When designing a new language, this is something to consider– does its syntax require the frequent use of characters that are difficult to type? *Note: Racket does allow hyphens in its identifiers!*

[9] The Partnership on Employment & Accessible Technology (PEAT)

(2) *ASCII vs. Unicode*

Pretty much all common programming languages have the notion of a string; however, the set of characters that can go in these strings varies. For many years, until the 2000s, most systems allowed only 128 characters, known as the ASCII Character Set. First published in the US in 1963, it includes the English alphabet in lower and upper case, Arabic numerals, and common special characters. It does not include accented characters, mathematical symbols, or symbols used in other languages. Naturally, this poses a problem for non-English speakers, as well as any individuals with non-ASCII characters in their names. As you learned in Lab last week, for former CS17 TA, Victoria Chávez '18, this meant that her last name was spelled incorrectly on every document she ever got from Brown! Fortunately, most information systems have gradually switched to Unicode, an extension of ASCII. This more inclusive version supports nearly 150,000 characters and symbols, including just about every written language.

Task: Look through this guide on the differences between accessibility, usability, and inclusion. Then, for each of the two examples above, determine which aspect of inclusive design it addresses. Make sure you explain your reasoning! *(3-4 sentences)*

Task: Your turn! You have finished Rackette, and are so inspired by your success that you've decided to make the next big programming language! You've also decided to make a new development environment (like VSCode or DrRacket) in which programmers can use your language. You are hoping that millions of programmers around the world will want to use your new language/tools, so to maximize your chances of success, you'd like to make them accessible to as many people as possible.

1. Come up with another factor (e.g., names with non-ASCII characters) that could impact the accessibility of your language/tools. This should **not** be one of the two examples listed above. (Think about your own experience while programming. What features made it easier or more difficult?)
2. What kind of features (e.g. switching to Unicode) could you include in your language/tools to help accommodate this? Come up with at least two, and don't forget to explain your reasoning!
3. Say you decided to include speech-to-text and screen reader support in your development environment. What do you see as potential practical pitfalls of such a system? How can we address these pitfalls? Name at least two and how you would attempt to address them. Don't forget to explain your reasoning!

   We want you to think about the technical considerations involved in making systems truly accessible. It might not be enough to just include speech-to-text and screen reader support, you must also consider how the nature of your system (and other systems you depend on) might affect accessibility. Ideas to get you started: What about your language would make it difficult/easy to use a screen reader? Speech-to-text technology can often be frustrating to use; how would this translate into a programming context?

Be prepared to discuss your answers for all three parts of the final task with your Design Check TA!

Optional: Read this 3-paragraph-long article for examples of accessible programming languages.

Note: VSCode already has lots of accessibility features you can use. If you are interested in what options are available, read about them here! You might be surprised by how many there are! :)

9 Getting Started

We have provided a guide to help you through Rackette, which you can find here. It contains the source

code, a guide on records, instructions on how to set up Rackette, and a suggested pathway to completing it. However, it is highly advised that you focus on your design check material first and meet with your design check TA before starting on the suggested path.

10 Handing In

10.1 Design Check

Design checks will be held from October 28th to October 31st. By October 24th, you should have received an email from the HTAs containing information about your partner, as well as an email with information about how to sign up for design checks. You must sign up for a design check by **11:00pm EST on Sunday, October 26th**. We recommend getting in communication with your partner as soon as possible so that you can sign up for a design check time that works for both of you. Also, remember you must adhere to the pair programming guidelines.

You should do the following to prepare for your design check:

- Fill in the definition for if-expression in the template code (the TODO in `TypesPREDESIGNCHECK.re`).
  - We will release an updated version of `Types.re` and `CS17SetupRackette.re` after design checks that will work with the setup code. The version of `CS17SetupRackette.re` you receive before the design checks do not have support for if-expressions. Please use these files instead when they are released.
- Fill out the code for the two incomplete lines in `process`, using the rules of processing as a guide. Be sure there's a recursive call in each.
- 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 (feel free to submit a screenshot or document instead — we just want to see that you've worked through the problems!). 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. The rest should be similar to the first, and relatively painless to implement.)
- Write out the rules of evaluation for `and` expressions and for `or` expressions.
- Give a few examples of *each* defined data type in the `TypesPREDESIGNCHECK.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`).
- Complete the tasks in the SRC: Accessible Design section. You should submit your answers in a .txt file named `rackette-src.txt` and come prepared to discuss the last question with your Design Check TA. Submit SRC to the **Rackette SRC assignment** on Gradescope.
- Submit `TypesPREDESIGNCHECK.re` with your completed if-expression and examples, `Rackette.re` with at least two built-ins in the top level environment, and your answers to the "Practice on Paper" to the **Rackette Design Check** assignment on Gradescope at least one hour before your Design Check.

10.2 Final Handin

The final handin is due by **Friday, November 7th, at 11pm EST** on Gradescope. Only one person in your group should turn in the project, and add the other partner to the Gradescope submission.

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.
- `Types.re`, which should be the version released after design checks.
- `Read.re`, which you should not alter.
- `CS17SetupRackette.re`, which you should not alter and should be the version released after design checks.

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

10.3 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: 2 points
- Correct if-expression: 1 point
- Complete your initial top-level environment including at least two built-in procedures (ex. +,-, ...) : 4 points
- Fill in and explain `process`: 2 points
- Explain your strategy for evaluating the parsed Rackette expressions: 3 points

- Complete the practice problems: 4 points

Functionality counts for 73% of your grade. Specifically, you will be graded on your program's correct implementation of the following:

- Your initial top level environment: 8 points
- Your `parse` function: 22 points
- Your `addDefinition` function: 4 points
- Your `eval` function: 35 points
- Your `stringOfValue` function: 4 points

Partial functionality gets partial credit.

The final 7% of your grade is reserved for testing. Note that you cannot get these points back during your final grading.

Note that style is being graded deductively. Remember to write Design Recipes for ALL procedures you write, and the I/O for the four procedures that we provided for you: `rackette`, `process`, `parse`, and `parsePiece`.

Add comments to any code which would otherwise be unclear.

After your work has been graded, you will have the option to attend a virtual final grading meeting. Final grading meetings will take 30 minutes during which you and a TA will discuss how your project was graded and any mistakes you made.

As a part of your final grading, you have the opportunity to fix any functionality mistakes on your hand-in and then resubmit the project on Gradescope. This resubmission process can get you up to 50% of credit back on any functionality points that you did not earn. You can only work on the resubmission during the final grading meeting itself and up to 30 minutes after the end of your final grading meeting. If you are working in a group, all partners must attend the final grading to qualify for resubmission points. Furthermore, you must submit to the resubmission Gradescope assignment within 30 minutes of the end of your final grading meeting.

A TA will contact you with more information about final gradings after the assignment deadline has passed.

11 Appendix: The Rules of Processing and Evaluation

The *rules of processing and evaluation* describe how to handle definitions and top-level expressions, and how to assign values to expressions, respectively. Before we list the rules, we need a few definitions.

A **binding** is a pair consisting of a name and a value, which we represent with an arrow notation, so that, for example, $\{x \mapsto 17\}$ is a representation of a binding.

A **binding list** is a list of bindings. For example, $\{x \mapsto 17, y \mapsto 18\}$.

An **environment** is a list of binding lists. The lookup rule will be that to look up a name, you start at the first binding list in the environment. If you don't find it there, you move on to the next binding list. When

looking inside of one binding list, you will look through the bindings in order and use the first one that's a binding for that name (if you find it).

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).

11.1 The Rules of Processing

A Rackette program consists of zero or more definitions, followed by zero or more top-level expressions. We'll call each of these a 'piece'. To process a program, we process the pieces one after another, each step taking place in some environment.

1. To process a definition `(define name exp)` in a top-level-environment T, we
   (a) check that the name is indeed a valid Rackette name, and that it is not already bound to a value in T. If either check fails, it's an error.
   (b) evaluate the expression `exp` using the Rules of Evaluation to produce a value v. If evaluation fails for any reason, it's an error.
   (c) Create a new top-level environment T', containing all the bindings of T, but additionally a binding from the `name` to the value v, and use T' as the top-level environment for processing the next piece (i.e., the next definition or top-level expression).
2. To process an expression `exp`, in the context of some top-level-environment T, we
   (a) apply the Rules of Evaluation to `exp` to produce a value v. If evaluation fails, it's an error.
   (b) prepend the value v to the list of values produced by evaluating the remainder of the program starting with top-level-environment T.

When all definitions and top-level-expressions have been processed, the result is a list of values, each of which can then be converted to a string and printed.

11.2 The Rules of Evaluation

Here are the rules of evaluation for expressions:

Note: Pay close attention to the notation on the arguments to these expressions. Once again, x* means "*zero* or more things" of type x, and x+ means "one or more" x's.

- The value of a number in $T + E$ is the `NumV` representing that same number. The value of the boolean `false` is `BoolV(false)`, and the value of the boolean `true` is `BoolV(true)`. The value of `empty` is a `ListV` representing the empty list.
- The value of a lambda-expression in $T + E$ (where T is the top-level-enviroment, and E is the local environment) is a closure composed of the lambda-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 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 this value p is neither a builtin nor a closure, evaluation terminates with an error. Otherwise, evaluate all of the arguments from `arg*` in $T + E$, to get 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. If the number of actual arguments is not the number of arguments that the associated function expects, evaluation terminates with an error.

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. Note that E is *not* part of the environment in which the body is evaluated.
   (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.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the (mostly) anonymous feedback form here!