The Rackette implementation has several stages. The first stage is translating from a string to a concrete program piece. Here are some examples:

<table>
<thead>
<tr>
<th>string</th>
<th>concrete_program_piece</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;17&quot;</td>
<td>Number 17</td>
</tr>
<tr>
<td>&quot;sq&quot;</td>
<td>Symbol &quot;sq&quot;</td>
</tr>
<tr>
<td>&quot;(sq 17)&quot;</td>
<td>List [Symbol &quot;sq&quot;; Number 17]</td>
</tr>
<tr>
<td>&quot;(define x 1)&quot;</td>
<td>List [Symbol &quot;define&quot;; Symbol &quot;x&quot;; Number 1]</td>
</tr>
<tr>
<td>&quot;(quote hi)&quot;</td>
<td>List [Symbol &quot;quote&quot;; Symbol &quot;hi&quot;]</td>
</tr>
<tr>
<td>&quot;(quote (1 2))&quot;</td>
<td>List [Symbol &quot;quote&quot;; List [Number 1; Number 2]]</td>
</tr>
</tbody>
</table>

Conversion from string (lefthand column) to concrete_program_piece (righthand column) is done by a procedure we have written for you, called read. The file read.ml provides the definitions:

```ml
type raw_program = string

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

type concrete_program = concrete_program_piece list
```

and also provides the procedures:
read: raw_program -> concrete_program_piece

and

read_all: raw_program -> concrete_program

For example, here we show read converting strings to concrete program pieces:

```
# read "17" ;;
- : concrete_program_piece = Number 17
# read "sq";;
- : concrete_program_piece = Symbol "sq"
# read "(sq 17)";;
- : concrete_program_piece = List [Symbol "sq"; Number 17]
# read "(quote (1 2))" ;;
- : concrete_program_piece = List [Symbol "quote"; List [Number 1; Number 2]]
```

So far the examples have all been of expressions. But there is another kind of concrete program piece, a definition, which binds a symbol to the value of an expression.

```
# read "(define mylist (quote (1 2)))" ;;
- : concrete_program_piece =
List
    [Symbol "define"; Symbol "mylist";
     List [Symbol "quote"; List [Number 1; Number 2]]]
```

A concrete_program is a list of concrete_program_pieces. For example, the string

```
"(define mylist (quote (1 2))) (+ 5 (car mylist))"
```

gets translated by read_all into the list

```
[
    List [Symbol "define"; Symbol "mylist";
        List [Symbol "quote"; List [Number 1; Number 2]]
    ];
    List [Symbol "+"; Number 5; List [Symbol "car"; Symbol "mylist"]]
]
```

- The first element of the output list is the concrete representation of the definition "(define mylist (quote (1 2)))" and
- the second element of the output list is the concrete representation of the expression "(+ 5 (car mylist))"
2 Parsing

What do we do with this concrete program list produced by read_all?

The next stage in the pipeline is parsing, or translating a concrete_program_piece into a form that captures the meaning of the program piece (but not the value). The procedure parse_piece does this. An abstract_program_piece is either an expression or a value. Of course, we use a variant type to distinguish them:

```plaintext
type abstract_program_piece =
  Definition of definition
  | Expression of expression
```

What is the type called definition and what is the type called expression? The former is given by a type alias:

```plaintext
type definition = identifier * expression
```

A definition can be specified by the identifier it binds and the expression whose value that identifier is bound to.

The latter is given by another variant type. This variant type has nine variants, corresponding to the nine different expression forms that Rackette must be able to handle. For example, one kind of expression is just a Rackette integer literal, such as 17. The corresponding expression data object is NumE 17.

We provide the definitions of types expression, definition, and abstract_program_piece for you, but go through these definitions carefully and make sure that you understand them!

```plaintext
type expression =
  NumE of int
  | IdentE of identifier
  | AndE of expression * expression
  | OrE of expression * expression
  ...
  | CondE of (expression * expression) list
  | QuoteE of concrete_program_piece
  | LambdaE of identifier list * expression
  | ApplicationE of expression list
```

Note that while, in reality, and and or expressions in Racket can operate on greater than two arguments, for the sake of this project we are limiting both of these expressions to operate on exactly two arguments (this is reflected in the expression type definitions for AndE and OrE).

The core of parsing is the procedure:

```plaintext
parse_expression: concrete_program_piece -> expression
```
This is where you need to know the syntax of CS17 Racket.

For the example concrete program pieces we showed earlier, we show the corresponding expressions.

<table>
<thead>
<tr>
<th>concrete program piece</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number 17</td>
<td>NumE 17</td>
</tr>
<tr>
<td>Symbol &quot;sq&quot;</td>
<td>IdentE (ID &quot;sq&quot;)</td>
</tr>
<tr>
<td>List [Symbol &quot;sq&quot;; Number 17]</td>
<td>ApplicationE [IdentE (ID &quot;sq&quot;); NumE 17]</td>
</tr>
<tr>
<td>List [Symbol &quot;quote&quot;; Symbol &quot;hi&quot;]</td>
<td>QuoteE (Symbol &quot;hi&quot;)</td>
</tr>
<tr>
<td>List [Symbol &quot;quote&quot;; List [Number 1; Number 2]]</td>
<td>QuoteE (List [Number 1; Number 2])</td>
</tr>
</tbody>
</table>

parse_expression will be your way to translate the concrete_program_piece in the lefthand column into its corresponding expression in the righthand column.

2.1 Parsing an integer literal

Let's start with a (overly) simple example. Consider the string "17". The corresponding concrete program piece is Number 7. The procedure parse_expression translates this into NumE 17, which is of type expression.

2.2 Parsing a procedure application

Because an expression contains subexpressions, parse_expression needs to be recursive.

For example, consider the expression "(car mylist)". This is a list with two elements. Each of the elements is itself a Racket expression. The corresponding concrete program piece is List [Symbol "car"; Symbol "mylist"]. How does the parse_expression procedure translate this concrete program piece? In this case, the procedure is recursively applied to

- Symbol "car"
- Symbol "mylist"

From these values of type expression the procedure must construct another expression, one that reflects the fact that the expression involves applying a procedure to arguments. The appropriate constructor is ApplicationE, and the value representing the whole expression is ApplicationE [Ident (ID "car"); Ident (ID "mylist").

2.3 Parsing a quote expression

Remember quote expressions? In Racket, the value of (quote a) is the symbol a. The rule of evaluation that applies to a list whose first element is the symbol quote is this: the value is the second element. That is, no evaluation of the second element takes place.

With that in mind, consider the string "(quote a)". The corresponding program piece is List [Symbol "quote"; Symbol "a"]). The procedure parse_expression translates this into QuoteE (Symbol "a"). How does this get parsed? The parse_expression procedure, when
applied to this concrete program piece, does not make a recursive call. Consider the part of the
definition of the expression variant type:

```
type expression =
  ...
  | QuoteE of concrete_program_piece
  ...
```

The argument to the constructor QuoteE is just the concrete program piece that was quoted.

### 3 Evaluation

The next stage in processing is actually evaluating the Rackette expressions. This is where Rackette
computations actually happen, e.g. applying Rackette procedures (built-in or user-defined). The
result of evaluation is something of type value, which is a variant type defined in the stencil
rackett.ml:

```
type value =
  | VNum of int
  | VBool of bool
  | VSymbol of string
  | VList of value list
  | VBuiltin of string * (value list -> value)
  | VClosure of identifier list * expression * environment
```

The different kinds of values are ints, Booleans, symbols, lists, and procedures. There are two kinds
of procedures, those that the Rackette user has defined, which are called closures (discussed in
Lecture on Nov. 2) and those that are implemented by OCaml procedures, which are called built-in
procedures.

The procedure eval is the core of the Rackette implementation. As part of its argument, it takes
in an expression and it outputs a value. Here are some examples:

<table>
<thead>
<tr>
<th>string</th>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;5&quot;</td>
<td>NumE 5</td>
<td>VNum 5</td>
</tr>
<tr>
<td>&quot;(or true x)&quot;</td>
<td>Or (IdentE (ID &quot;true&quot;),</td>
<td>VBool true</td>
</tr>
<tr>
<td></td>
<td>IdentE (ID &quot;x&quot;))</td>
<td></td>
</tr>
<tr>
<td>&quot;(+ 1 2)&quot;</td>
<td>ApplicationE [IdentE (ID &quot;+&quot;);</td>
<td>VNum 3</td>
</tr>
<tr>
<td></td>
<td>NumE 1; NumE 2]</td>
<td></td>
</tr>
</tbody>
</table>

Things to note:

- Even for something as simple as an int, the expression is different from the value. The
type system should help you keep things clear.

- Like parse_expression, the procedure eval must be recursive because evaluating a big
expression sometimes involves evaluating subexpressions.
• Evaluating an identifier expression (such as `IdentE (ID "true")`) involves looking up the identifier in the environment.

Actually, as discussed in the last lecture, there are two environments, the global (or top-level) environment, and the local environment. For this reason, the `eval` procedure takes a triple as argument: `(top-level environment, local environment, expression)`.

4 Built-ins

Definitions allow the user to add new bindings to the top-level environment. However, one cannot program completely from scratch. In our implementation of Rackette, a few symbols are bound in the initial top-level environment before any user definitions:

`+, -, *, /, remainder, =, <, equal?, number?, zero?, cons?, empty?, cons, car, cdr, not, true, false`

Those last two, `true` and `false`, are bound to Boolean values. The rest are bound to built-in procedures. (Unlike Racket, our Rackette addition and multiplication procedures each take exactly two arguments.)

Your code must assign to the OCaml variable `initial_tle` an environment in which all the above symbols are bound to the appropriate values (elements of type `value`). The symbol `true`, for example, is bound to `VBool true`. What about the procedures? Recall that the constructor `VBuiltIn` takes an argument of type `string * (value list -> value)`, which is a tuple type.

• The first element of the tuple is the name of the procedure, i.e. the symbol bound to it. (This is not essential to implementing Racket but it makes it possible to print a more meaningful value.)

• The second element is an OCaml procedure. It takes a list of `value`s and produces a `value`.

For example, let’s construct the value to which the symbol `+` should be bound. First let’s write the OCaml procedure:

```ocaml
let my_plus = function [VNum i; VNum j] -> VNum (i+j)
```

Now the value to which `+` is bound is `VBuiltIn (+, my_plus)`.

As part of the project, you must write an OCaml procedure for each of the built-in procedures we require. Each of these is pretty simple and doesn’t involve recursion, except `equal?`, which is a bit more complicated and uses recursion in order to test whether two lists are equal.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [link](http://cs.brown.edu/courses/csci0170/feedback).