Homework Assignment 7

Any automatically graded answer may be manually graded by the instructor. Submissions are expected to only use functions taught in the course. If a submission uses a disallowed function, that exercise can get zero points. Excluding promises, *all functions that mutate values are disallowed* (mutable functions usually have a `!` in their name).

The interpreter

1. Reimplement functions `d:eval-exp` and `d:eval-term` from Homework Assignment 5 according to question 7, question 8, and question 9. Implement the following **effectful** functions:
   (a) `(env-put e x v)`: given a heap `m` return as a new state `(environment-put m e x v)` and as a result `(d:void)`.
   (b) `(env-push e x v)`: given a heap `m` return `(environment-push m e x v)`
   (c) `(env-get e x)`: given a heap `m` return the same state and as a result `(environment-get m e x)`.
   Feel free to use the solution of Homework Assignment 5 as the basis of your implementation.

Handling multiple arguments

Recall the currying and the uncurrying functions. The objective of this exercise is to perform currying as a code-transformation step.

2. Function `break-lambda` takes a list of parameters `p` and a curried term `t`. If the list of parameters is empty, then the return should be a lambda with one parameter named `_`, and the body `t`. If the list of parameters is nonempty, then the return should be a curryed lambda with the same number of parameters and a body `t`.

3. Function `break-apply` takes a curried expression `ef` and a list of curried expressions `ea`. If the argument list is empty, then the return should be calling function `ef` with a single argument `(d:void)`. If the argument list is nonempty, then the return is a curried function application where `ef` is the function and the arguments are `ea`.

4. Function `d:curry` takes a *term* (which may include sequences and definitions) and recursively curries any function declaration (lambda) and any function application contained in the given term, using the two functions above (**Attention:** see question 7.)

Supporting primitives

5. Implement support for the branching primitive `if` in our language according to the following formal rules. *Tip:* Typically, both rules are implemented in the same Racket-branch.

$$\frac{e_c \Downarrow_E \texttt{\#f} \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_f}\ \texttt{(E-if-f)} \qquad \frac{e_c \Downarrow_E v \qquad v \neq \texttt{\#f} \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_t}\ \texttt{(E-if-t)}$$

6. (10 points) **Extra credit.** Implement support for built-in operations according to the formal rule below, where `builtin` is a value, and f is a Racket function with contract `(-> d:value? d:value?)`. *Tip:* to ease the implementation, consider extending the branch for function application.

$$\frac{e_f \Downarrow_E (\texttt{builtin}\ f) \quad \blacktriangleright \quad e_a \Downarrow_E v_a}{(e_f\ e_a) \Downarrow_E f(v_a)}\ \texttt{(E-app-b)}$$

Manually graded questions

7. **Manually graded.**
   (a) Ensure `d:curry` calls parameter `break-lambda-impl` and parameter `break-apply-impl` instead of directly calling functions `break-lambda` and `break-apply`.
   (b) Ensure `d:eval-exp` calls parameter `d:eval-term-impl` instead of directly calling function `d:eval-term`.
   (c) Ensure `d:eval-term` calls parameter `d:eval-exp-impl` instead of directly calling function `d:eval-exp`.
8. **Manually graded.** Rewrite `d:eval-exp` and `d:eval-term` to be monadic. Your solution should be using the `do`-notation, `eff-bind`, and `eff-pure`.
9. **Manually graded.** *Use pattern matching instead of boolean branching and avoid using accessor functions.* Ensure your solution **only** uses `match` (your solution should **not** use `cond` nor `if`). Additionally, your solution must **not** use functions to retrieve the fields of structs and instead pattern match its contents (*e.g.*, do not use `d:lambda-params1`).