Closures and Probabilistic Programming

Due: Sunday, October 7th

You may work in teams of 2 on this assignment. Both team members may submit their code to run our test suite, but only one team member should make a final submission for grading. See the handin section for details and try it out right away to see what our automated testing produces!

Write an interpreter for the RSE (rejection-sampling expression) language as defined below.

Starter File

You will need two code files to get started: assn3.rkt and assn3-library.rkt. **You must only submit the assn3.rkt code file to hand in** (unless you are attempting the bonus). You will also need to hand in a README.txt file (as defined below).

Instructions

- **Implement (which of course includes testing!) the interpreter cases for `distribution`, `app`, `defquery`, `infer`, `assert-in`**, whose desired functionality is described below.
- **Answer the theory questions**, found in the README file.

The RSE (Rejection Sampling Expression) Language

```
;;<RSE>::=<number>
;;|{distribution<RSE>+}
;;|{uniform<number><number>}
;;|{sample<RSE>}
;;|{assert-in<RSE><RSE>}
;;|{defquery<RSE>}
;;|{infer<RSE><RSE>}
;;|{<Binop><RSE><RSE>}
;;|{ifelse<RSE><RSE><RSE>}
;;|{with{<id><RSE>}<RSE>}
;;|{fun<id><RSE>}
;;|{<RSE><RSE>};;functioncalls,anytimethefirstexpressionisnotareservedsymbol
;;|<id>
;;
;;<Binop>::=+|-|*|/|<|>|=|<=|>=
```

For conventional language features, RSE features with-bindings, first class functions, and conditional expressions. We use numbers in place of booleans, with 0 as false and all other numeric values as true.

Instead of substitution semantics, we will use *environments*, which are a more efficient way of binding values to names.

Additionally, to support first-class functions and probabilistic programming, RSE values may be distributions (lists of other values), numbers, closures, queries, or a special `rejected` constant.

The RSE language has eager evaluation and subexpressions are evaluated from left to right.

Probabilistic Features

In assignment 2, we saw a way to create and manipulate distributions. In this assignment, we will use those distributions in a much more powerful way.

However, we will impose a stronger restriction on distributions: **Unlike in Assignment 2, in this assignment we will *NOT* accept empty distributions.** (You will notice that the EBNF changed from a `{distribution <RSE>*}` to a `{distribution <RSE>+}`)

Probabilistic programming allows us to determine the probability that a process had some initial value, given some constraints on the result of the process. For example, we might want to know the probability, when we roll a pair of dice, that either die shows a 1, given that their sum is 4. In this sense, it allows us to work backwards, creating a distribution of input values from constraints on the output.

We have some language features that you will implement to support this:

- `sample` (provided for you, and which you should **not** change): if D is a distribution, then `{sample D}` evaluates to a randomly chosen value from D.
- `distribution`: similar to those in A2, except that:
  - distributions may now contain arbitrary expressions, instead of just numbers.
  - distributions cannot be empty: Any attempt to produce an empty distribution must signal an error as soon as possible.
- `defquery`: much like `fun`, `{defquery E}` wraps the expression E without evaluating it, so that we can perform inference on it later.
- `assert-in`: this lets us express constraints in the body of our queries. If `B` is some condition and `E` is an expression, `{assert-in B E}` evaluates to the value of E if B is true, and evaluates to a special value `rejected` if B is false.
- `infer`: `{infer nexp qexp}` evaluates `nexp` to a number `n` (non-numeric, non-rejected values are errors) and `qexp` to a query value `q` (non-query, non-rejected values are errors), and runs `q` `n` times. The list of results are then compiled into a distribution, with any `rejected` values filtered out. If this filtered list is empty, then `infer` should produce an error instead.

The method of inference described above is known as *rejection sampling*, thus the name of our language. Because `sample` produces values randomly, running a query many times may give different results, The distribution produced by `infer` describes the *conditional probability* of the query: that is, the distribution of values given that all its `assert-in` coniditons are true.

For example, when we roll two dice, we can use the following inference to determine the distribution of values on the first dice, given that both rolls add up to a value greater than or equal to 7:

```
{with{d{uniform17}}
{infer1000{defquery
{with{x{sampled}}
{with{y{sampled}}
{assert-in{>=7{+xy}}
x}}}}}}
```

Dynamic Typing and Rejection Propagation

Because there is more than one value type, you must perform dynamic typing checks on the results of recursive calls to your interpreter, in order to ensure the values operations are performed on are of the correct type.

There is one special case for this: We say that `rejected` propagates: whenever any input to any operation evaluates to `rejected`, the operation as a whole does. For example, `{+ 3 x}` should evaluate to rejected if `x` evaluates to rejected. Note however that:

- Functions and queries delay evaluation, so `fun` and `defquery` should never evaluate to rejected until their bodies are evaluated

- `rejected` values from queries run by `infer` are not propagated, but filtered out. This is what makes inference work.

While these features may seem complicated, their implementation is not as complex as one might think, and may end up looking *very similar* to other language features.

README.txt and Theory Questions

The README.txt has some theory questions for you to answer, along with places to enter your student information. Please fill out the fields as specified and answer the questions. **Do ONLY use the given README.txt file as template and do not modify the header section except for filling out the TODOs!**

handin and Marking

You should turn the assignment in using the handin program (this assignment is called a3). **Submit two files: assn3.rkt and README.txt** (plus, if you choose to try the bonus, one more file: bonus.rkt). Also include your name(s) in each file.

A large percentage of your mark will be based on the automated tests that run during `handin`. The rest will be split between the answers to your theory questions and possibly a check of your code (eg. for readability and code style, among other things).

If you're working in a team, both team members may submit their code to run our test suite, but only one team member should make a final submission for grading. **You can invalidate the submission that should be ignored by including a file named IGNORE.txt.** Please be aware that we will choose a random submission and may further deduct marks if both of your team members submit a final solution for grading.

Just a reminder, late assignments are not accepted, and (basically) no excuses will be entertained. So, handin your assignments early and often!

Bonuses

See the Bonus section on the website for more info. Please handin any bonus attempts in a separate file: bonus.rkt!

- **(1 Bonus Point):** Pseudo-random number generators tend to be "stateful": using some initial seed, their internal state is set. Every time a number is generated, they change. In this way they are not truly random: the numbers generated are entirely determined by the state, and each call to the generator changes the state.

  Give and very clearly explain an example of an RSE program that, when evaluated using both eager and lazy evaluation, could produce different results, even if the same random number generator and seed are used.

- **(1 Bonus Point):** The `uniform` distribution feature is implemented as syntactic sugar, and it only accepts numbers, not arbitrary expressions. Modify the implementation of `uniform` so that it accepts arbitrary expressions and it is still syntactic sugar (i.e. Without any changes to any `define-type` nor to the `interp` function, though you are allowed to change other helper functions in the file besides the `parse` function).