Due: Tuesday, October 23 at 10PM (a bit later than our original start-of-term plan)

You may work in teams of **up to 5** on this assignment.

******IMPORTANT**: Our intention is that you would only have more than 3 people in your group if your group is also your project group. If you plan to have a group of more than 3 that is not your project group, please contact us to ensure that's acceptable, either via a private post on Piazza or mail to cs311@ugrad.cs.ubc.ca. In either case, say who will be on the team (including ugrad IDs), why you are working together in this large group, and why you are not using your project team. Do so by Thursday 18 Oct at latest.*

Run handin right away to see what our automated testing produces!

Extend interpreter for the ISE (importance-sampling expression) language as defined below.

Starter File

The code you need to get started is here: assn4.rkt. This is the only code file you need to hand in. You will also need to hand in a README.txt file (as defined below).

Instructions

- Implement `begin` in `parse`
- Implement `begin`, `observe`, and `infer` in `interp`
- Fix incorrect state threading in `interp`
- Answer the theory questions in README.txt

The ISE (Importance Sampling Expression) Language

```
    ;; ISE = Importance Sampling Expression
    ;;
    ;; <ISE> ::= <number>
    ;;     | {distribution <ISE>+}
    ;;     | {uniform <number> <number>}
    ;;     | {sample <ISE>}
    ;;     | {defquery <ISE>}
    ;;     | {infer <ISE> <ISE>}
    ;;     | {begin <ISE>+}
    ;;     | {observe <ISE> <ISE>}
    ;;     | {<Binop> <ISE> <ISE>}
    ;;     | {ifelse <ISE> <ISE> <ISE>}
    ;;     | {with {<id> <ISE>} <ISE>}
    ;;     | {fun <id> <ISE>}
    ;;     | {<ISE> <ISE>} ;; function calls, any time the first symbol is not a
reserved word
    ;;     | <id>
    ;;
    ;; <Binop> ::= + | - | * | < | > | = | <= | >= | %
```

Overview

In assignment 4, we implemented RSE to manipulate distributions by constraining the acceptable values for an execution trace, completely rejecting any trace that didn't meet our constraints. In this assignment, we will extend RSE to use importance sampling to have more flexible constraints and be able to accumulate probabilities. We will do this by:

1. replacing `assert-in` with `observe`, and
2. adding **a single state** that will represent the *weight of the execution trace* of the program and carefully mutating that state at appropriate points.

Here's an example of a problem that the extended ISE can solve:

```
    ;; Roll a fair die and then a biased die, what's the chance of rolling a number <
2 both times?
    '{with {fair {uniform 1 7}}
       {with {biased {distribution 1 1 2 2 2 3 4 5 6}}
         {with {f {fun x {< x 2}}}
           {begin
             {with {result {infer 1000 {defquery {sample fair}}}}
                   {observe result f}}
             {with {result {infer 1000 {defquery {sample biased}}}}
                   {observe result f}}}}}}
```

In this example we do inference 1000 times, which yields 7820/200000 ~ 0.039, which is very close to the actual probability 1/27 ~ 0.037. (Notice that in this case, we don't care about the actual value evaluated from the program, only the likelihood. Also notice that our first `observe` in the program above and the one below is **ignored** in terms of its value. We only execute it for its side effect: a change in the likelihood state.)

We can directly extract the exact probability with this program that doesn't use infer as well:


```
'{with {fair {uniform 1 7}}
   {with {biased {distribution 1 1 2 2 2 3 4 5 6}}
     {with {f {fun x {< x 2}}}
       {begin {observe fair f}
              {observe biased f}}}}}
```


As another example, we can actually use ISE to solve the famous Monty Hall Problem. We won't write out the entire program here as it can get a little messy largely due to the lack of lists in ISE, but the basic idea is: we can have a query where we `sample` the strategies that the participant can choose from (switch or stay), `observe` the strategies that result in a win, and return the strategy sampled. Then we run `infer` on the query some number of times to get a distribution of strategies that resulted in a win, and if we look at the distribution returned, the strategy that appears most frequently in the distribution would be the answer to the problem.

State and Mutation

The new `interp` now returns 2 values using the `ValueXState` type: `(vals ISE-Value number)`, where the first value is the ISE-Value the input program evaluates to and the second value is a number representing the weight of the execution trace. The probability of the result comes from the *state*, which is updated at runtime to keep track of the current weight of the program. In essence, the *state* is threaded through the interpreter to model mutable state. Its value is normally modified via multiplication with fractions representing the proportion of lists that observe selects for (though there are some special cases where we "revert" it). The *state* represents the likelihood of seeing the results our interpreter is selecting for from our distributions. This course does not require any background in probability, so this is the extent to which we'll discuss the *state* here. If you are still curious and want a more in-depth explanation, please ask the course staff during office hours, and we'll try to muddle through! With the added state, ISE seems to be allowing mutation; however, the goal of this assignment is to simulate mutation from the program's perspective without any actual mutation in our interpreter. To do this, we want to *thread* the state through our interpreter as a parameter and return value so the interpreter always has the most updated state. The starter file already has `state` added to `interp`, but is threaded

incorrectly in some of the variants. Your task is to fix the mistakes and make sure the interpreter is always evaluating using the right state! *Hint: look at the tests if you're not sure how state should be passed around.*

ISE Semantics

- `{begin <ISE>+}`: Evaluates the expressions in order, threading the state through the evaluation of each expression, and producing the value of the last expression and the resulting state as its own result.
- `{infer <nexpr> <qexpr>}`: Evaluates `<nexpr>` to a number *n*, and `<qexpr>` to a query *q*, and **stores the state**. Then proceed to run *q n* times, **setting state to 1 at the beginning of each run**, collect the query execution results, filter out the rejects in the results, and return a normalized distribution of the query execution results and the stored state, discounting the state changes running the thunkV might have produced. As you have probably noticed, it's very similar to the `infer` in A3, so you could definitely reuse some of your A3 code! A helper called `normalize-weighted-pairs` is given in the starter file: it takes a list of pairs, where each pair is a pair of an element of any type and a number representing the weight, and converts the list to a list of the same elements but each element is repeated according to its weight.

  Example of `normalize-weighted-pairs`:

```
(normalize-weighted-pairs (list (cons (numV 1) 0.5)
                                (cons (numV 2) 0.25)
                                (cons (numV 3) 0.25)))
>= (list (numV 1) (numV 1) (numV 2) (numV 3))
```


- `{observe <qexpr> <fexpr>}`: Evaluates `<qexpr>` to a distribution *d* and evaluates `<fexpr>` to a function *f* of type `Any -> numV`. (Here `numV` is intended to be used as Boolean). Apply *f* in turn to each element in *d* - any value for which *f* evaluates to `(numV 0)` is discarded and any value for which *f* evaluates to any other `numV` is kept, resulting in a filtered distribution. Update the state by **multiplying** it with the ratio of the number of elements remaining in the filtered distribution to the number of elements in the original distribution (i.e., the probability that a sample from this distribution passes the filtering function). **Ignore state changes produced by applying *f***. If the resulting filtered distribution is empty, **reject and set state to 0**.

Note: `observe` and `infer` can be used in two ways:

1. `observe` can produce a meaningful result in the execution trace weight (the state threaded through execution) as long as we're careful to wrap any uses of `sample` inside `infer` calls. In that case, the execution trace weight actually represents a probability that the observed conditions hold. (This may be an approximate probability if infer was used as well.)
2. `observe` can *also* be used within `infer` expressions to implement importance sampling. In this case, it impacts the 'weight' of a sample in computing the eventual distribution inferred from a collection of samples.

Our examples above only use the first approach: the state produced is meaningful and approximates the probability, whereas the value is simply discarded. We can represent the same scenario using the second approach:

```
;; Roll a fair die and then a biased die, what's the chance of rolling a number < 2
both times?
'{with {q {defquery
      {with {fair {sample {uniform 1 7}}}
            {with {biased {sample {distribution 1 1 2 2 2 3 4 5 6}}}
                  {ifelse {< fair 2}
```

```
                            {ifelse {< biased 2}
                                    1
                                    0}
                            0}}}}}
    {infer 50 q}}   ; using 50 here instead of 1000 to make it easier to show the
results
```

Using this approach we get a distribution of (0 0 0 0 0 0 0 0 0 0 0 0 0 1) and a state of 1. Here, the state is meaningless, but the distribution we get back tells us the possible outcomes and their associated probabilities.

Hint: look at the tests for examples if you find the semantics description confusing!

Parsing `begin`

In the concrete syntax, `begin` takes a list of ISE, but in the abstract syntax, the `rec-begin` variant only takes 2 fields, each field storing an ISE. We'll replace `begin` with multiple expressions by nested rec-begins each with two expressions by putting a rec-begin in the second expression. Thus, a single-expression begin like `{begin <ISE>}` is exactly equivalent to its contained expression `<ISE>`.

Here's an example of what `begin` should parse to:

```
(parse '{begin 1 2 3}) ->  (rec-begin (num 1)
                                      (rec-begin (num 2) (num 3)))
```


Reject Propagation and Dynamic Typing

In this assignment, `(rejected)` will be coming from `observe` instead of `assert-in`, but rejection handling and propagation should be the same as in assignment 3 (i.e. always propagate reject except in `infer`, where rejects are filtered out). Please note that evaluation to a `(rejected)` at any point should cause the interpreter to return with a `(rejected)` as its ISE-Value (with the state set to 0) except in `infer`, as described above.

Just like in assignment 3, if an inner expression evaluates to something that is not expected by the outer expression, the interpreter should throw an error.

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.

handin and Marking

You should turn the assignment in using the handin program (this assignment is called a4). **Submit two files: assn4.rkt and README.txt**. 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).

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