Take-home Re-Exam in Advanced Programming

Deadline: Friday, 2 February 2024 at 15:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home re-exam on the course Advanced Programming, B1-2023. This document consists of 19 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of two independent questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner. The questions each count for 50%. However, note that you must have both some non-trivial working Haskell and Erlang code to get a passing grade.

In the event of errors or ambiguities in an exam question, you are expected to state your assumptions as to the intended meaning in your report. You may ask for clarifications in the discussion forum on Absalon, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 4–8 pages, not counting appendices, presenting (at least) your solutions, reflections, and assumptions, if any. The report should contain all your source code in appendices. The report must be a PDF document.
- *The source code* should be in a .ZIP file called `code.zip`, archiving one directory called `code`, and following the structure of the handout skeleton files.

Make sure that you follow the format specifications (PDF and .ZIP). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the Digital Exam system (`eksamen.ku.dk`).

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of

your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.
- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.
- It is important that you implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some non-trivial working code in both Haskell and Erlang. (This is a *necessary*, but not *sufficient*, condition.)

Exam Fraud

This is a strictly individual exam; thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course. Submitting answers (code and/or text) you have not written entirely by yourself, or *sharing your answers with others*, is considered exam fraud.

You are allowed to ask (*not answer*) how an exam question is to be interpreted on the course discussion forum on Absalon. That is, you may ask for official clarification of what constitutes a proper solution to one of the exam problems, if this seems either underspecified or inconsistently specified in the exam text. But note that this permission does not extend to discussion of specific solution approaches or strategies, whether concrete or abstract.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. However, make sure to use proper and *specific* citations for any material from which you draw considerable inspiration – including what you may find on the Internet, such as snippets of code. Similarly, if you reuse any significant amount of code from the course assignments *that you did not develop entirely on your own*, remember to clearly identify the extent of any such code by suitable comments in the source. Remember also that AI-based coding tools (e.g., ChatGPT or Copilot) are strictly prohibited.

Also note that it is not allowed to copy any part of the exam text (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g., StackOverflow, IRC, exam banks, chatrooms, or suchlike), whether during or after the exam, without explicit permission of the author(s).

During the exam period, students are not allowed to answer questions on the discussion forum; *only teachers and teaching assistants are allowed to answer questions.* Note that, since some students may have been granted additional exam time, the prohibition against discussing the exam in public extends through **Saturday, 3 February 2024**.

Breaches of the above policy will be handled in accordance with the Faculty of Science's disciplinary procedures.

Question 1: DROLL: A dice-roll probability calculator

Many board games, notably including role-playing ones, involve an element of chance, typically expressed via dice rolls. Sometimes these rolls are described via fairly elaborate *protocols*, for example: "take the highest of 3 rolls with a 6-sided die", or "roll a die once, but you get an extra make-up roll if you rolled a 1 the first time". When designing such a protocol, it is often useful to get an idea of how likely the various possible outcomes are, or if a complicated protocol could be replaced by a simpler one (or sometimes the other way around, for dramatic effect).

Accordingly, this task is about implementing the DROLL system (inspired by Torben Mogensen's "Troll" language[1]), a tool for calculating probabilities for dice-roll protocols. It takes a textual description of a protocol, and lists the possible outcomes from performing it, together with their probabilities:

```
$ droll "highest of (3 times roll 6)"
Result distribution:
   1/216 (  0.5%): 1
   7/216 (  3.2%): 2
  19/216 (  8.8%): 3
  37/216 ( 17.1%): 4
  61/216 ( 28.2%): 5
  91/216 ( 42.1%): 6
Expected value: 4.958333
```

We see that there's only a $\frac{1}{216}$ ($=(\frac{1}{6})^3$) chance of getting a final result of 1 (because that only happens if you roll a 1 all three times), whereas the chance of getting a 6 at least once is pretty good, at $\frac{91}{216}$ ($=1-(\frac{5}{6})^3$, i.e. unless you roll a non-6 all three times). We also see that on average (i.e., if playing the game repeatedly), we would expect to get a result of almost 5.

Overview of DROLL syntax and semantics

The following is an informal presentation of the DROLL constructs and what they mean; the full formal grammar will be given later. For visual emphasis, DROLL expression snippets will generally be written like "`this`".

Numbers and bags Numeric constants are written as non-empty sequences of decimal digits, with the usual meaning, e.g, "`42`". (Leading zeros *are* permitted.) Note that all numbers in DROLL, both constant and computed, are *non-negative* integers. However, the basic data type in DROLL is actually the *bag*: an unordered collection of zero or more numbers. Bags are constructed using the comma-operator, which joins bag contents; e.g., "`(7,9,13)`" is a bag containing three numbers. The empty bag is written as "`()`". A bag with exactly one element is called a *singleton*.

[1] T. Æ. Mogensen. "Troll, a language for specifying dice-rolls". 2009 ACM Symposium on Applied Computing, pp. 1910–1915, DOI: 10.1145/1529282.1529708

Note that bags are *not* nested (i.e., bags contain only integers, not distinct sub-bags), and may have repeated elements. For example, "`(5,(7,5))`" and "`((7,()),(5,(5)))`" represent the same bag as "`(5,5,7)`", i.e., two 5's and one 7. The *canonical* representation of a bag lists the elements in increasing order; if the bag is a singleton, the outer parentheses are omitted.

Basic operations Two bags can be compared for equality with the `is` operator, which returns (the singleton) 1 if the bags are equal, and 0 otherwise; for example "`(2,(2,(),4)) is (2,4,2)`" returns 1, and "`(2,2) is 2`" returns 0. The `is not` operator returns the opposite result, i.e. 1 if the bags are different, and 0 if they are the same.

The operator "`count` E" returns the total number of elements in the bag E; for example, "`count (2,2,4)`" evaluates to 3. Similarly, `sum` computes the sum of all elements, so "`sum (2,2,4)`" returns 8. The notation "E_1 `+` E_2" is shorthand for "`sum (`E_1`,`E_2`)`". (Note that this implies that, e.g., "`() + (1,2)`" evaluates to 3.)

Names and bindings All Droll values (i.e., bags) may be stored in *variables*. There are two ways of doing this: local and global. The local form, "`let` x `be` E_1 `in` E_2", binds x to the result of evaluating E_1 for the duration of evaluating E_2. Once E_2 has been evaluated, x is restored to its previous value (if any). For example, "`let x be 3 in (let x be 4 in x, x+5)`" evaluates to the bag $(4, 8)$.

Global bindings are made with the form "`let` x `be` E", which binds x to the result of evaluating E indefinitely (and also returns that result). For example, the expression "`let y be (let x be 4)+3 in (x,y)`" evaluates to $(4, 7)$. Any previous binding of x (whether local or global) is overwritten.

Since evaluation may have side effects, we explicitly specify that expressions are evaluated from left to right; e.g., "`((let x be 1), x+10, (let x be 2), x+20)`" evaluates to $(1, 2, 11, 22)$. We also allow explicit sequencing of expressions: "E_1`;` E_2" evaluates E_1, discards the result (but retains any global updates), and then evaluates E_2.

Accessing an unbound variable signals an error. For example, in "`(let x be 3 in x)+x`", the last use of `x` causes an error. On the other hand, "`(let x be 3); x`" returns 3.

Iteration The `times` operator allows a computation to be repeated a number of times. In "E_1 `times` E_2", E_1 must evaluate to some singleton n (otherwise an error is signaled), and then E_2 is evaluated n times and the results are collected in a bag. For example, "`(2 times 7, 3 times 5)`" evaluates to the bag $(5, 5, 5, 7, 7)$. We can also compute the arithmetic product of x and y as "`sum (`x `times` y`)`". Note that E_2 is recomputed in each iteration, and may have side effects, so that

```
let i be 0 in 5 times (let i be i+1)
```

evaluates to $(1, 2, 3, 4, 5)$, and

```
let i be 0 in 5 times (let j be i in (let i be i+1; j))
```

to $(0, 1, 2, 3, 4)$. In particular, if E_1 evaluates to 0, E_2 is not evaluated at all, and the result is just an empty bag.

Selection There are two general ways of extracting the individual elements of a bag. In the simpler one, we can ask for the maximal element in a bag: "`highest of (3,1,4,1)`" returns 4; and analogously, "`lowest of (3,1,4,1)`" returns 1. The result is always a singleton; if the bag was empty, an error is signaled.

Note that we can use this facility for comparing two numbers: for example, "`x is lowest of (x,10)`" returns 1 when $x \leq 10$ and 0 otherwise; "`x is not highest of (x,10)`" similarly tests whether $x < 10$.

The more general (and verbose) form of selection, "`take highest from x`", returns the largest element in a *named* bag x (i.e., a locally or globally bound variable), but also removes it. Thus,

```
let x be (3,1,4,1) in (take highest from x, sum x)
```

evaluates to (4,5), because x has been rebound to just (1,1,3) for the remainder of the `let`-body. Note that the form "`highest of E`" can be seen as just shorthand for "`let v be E in take highest from v`", where v is some fresh variable. "`take lowest from x`" behaves analogously.

Generalizing further, we allow the form "`take highest E from x`" as shorthand for "`E times (take highest from x)`", and similarly for the short form. For example, "`highest 2 of (3,1,4,1)`" evaluates to $(3, 4)$ (and "`lowest 2 ...`" to $(1, 1)$).

Randomness Finally, Droll allows for actual random choices in the protocol. The simpler form, "`roll E`", where E must evaluate to some positive singleton n, represents rolling a fair, n-sided die, i.e., it returns one of the numbers 1, 2, ..., n with equal probability. If $n = 1$, the result is always 1; $n = 0$ results in an error.

The more complex form represents a fair random draw from a bag, by also allowing the selector `any` in addition to the `highest` and `lowest` from above. For example, "`any of (4 times 42, 6 times 666)`" returns 42 with probability 40% (i.e., $\frac{2}{5}$), and 666 with probability $\frac{3}{5}$. Similarly, we may write "`let x be (5,5,7) in (take any 2 from x; x)`". Here we first throw away two random values from the initial bag (which is equivalent to just picking a single value from that bag), so the result will be 5 with probability $\frac{2}{3}$ and 7 with $\frac{1}{3}$

Finally, note that errors arising due to random choices do not necessarily preclude a meaningful probability calculation. For example "`roll (any of (4,0,6))`", will fail with probability $\frac{1}{3}$, but otherwise it is equivalent to rolling either a 4- or a 6-sided die, and Droll will give the probabilities of all 7 possible outcomes (including the failure.)

Similarly, a poorly designed protocol might at some point require drawing a number from a bag that may have become empty as a consequence of an improbable, but not impossible, sequence of previous steps. The higher-level game rules may then say that the protocol is to be run again from the start, most likely leading to a proper outcome; but of course, it is preferable for the game designer to know of this possibility, and its likelihood, ahead of time.

```
Exp  ::= MExp
      |  MExp ";" Exp
      |  "let" var "be" SExp "in" Exp

MExp ::= SExp
      |  "let" var "be" SExp

SExp ::= num
      |  var
      |  "sum" SExp
      |  "count" SExp
      |  "roll" SExp
      |  Sel "of" SExp
      |  "take" Sel "from" var
      |  SExp "+" SExp
      |  SExp "is" ["not"] SExp
      |  SExp "times" SExp
      |  "(" [CExp] ")"

CExp ::= Exp
      |  CExp "," CExp

Sel  ::= ("highest" | "lowest" | "any") [SExp]

num  ::= (see text)
var  ::= (see text)
```

Figure 1: Concrete syntax of Droll

Implementing Droll

The Droll implementation consists of two core modules, a parser and a calculator. Additionally, there is a module with shared type definitions, and a simple command-line processing module (already provided) that just invokes the parser and calculator after each other, and presents the results to the user in a readable way.

The two main modules parts are weighed equally, but you should aim to complete both, at least partially, as they demonstrate complementary skills.

Question 1.1: Parser

The concrete syntax of Droll is shown in Figure 1. Note that this grammar uses a couple of EBNF shorthands for conciseness: an element $(A_1 \mid \cdots \mid A_n)$ represents an choice between the n alternatives A_i; and $[A]$ represents that the element A is optional. The start symbol of the grammar is the nonterminal `Exp`. This grammar is supplemented with the following stipulations:

Disambiguation The operators in `SExp` have the following associativities and precedences, listed in increasing order (i.e., with the weakest-grouping first):

1. `times`: right-associative. For example, "`3 times 4 times 5`" is parsed like "`3 times (4 times 5)`", which evaluates to a bag of 12 5's. (The left-associated alternative, "`(3 times 4) times 5`", would be meaningless, because "`(3 times 4)`" does not evaluate to a singleton, and hence cannot be used as a repetition count.
2. `is` and `is not`: non-associative; that is, e.g., "`x is y is z`" is syntactically disallowed. (Both of the possible associations "`(x is y) is z`" and "`x is (y is z)`" may make sense, but since they don't mean the same thing, and neither is clearly preferable, the syntax forces the user to make an explicit choice between them.
3. `+`: left-associative. This is the conventional syntactic associativity for addition, though semantically, right-associativity would also have worked.
4. `count`, `sum`, `roll`, and `of`. These constructs having the highest precedence means that, e.g., "`roll 5 is 1`" parses as "`(roll 5) is 1`".

In `CExp`, it does not matter whether the comma operator is parsed left- or right-associatively, since the abstract syntax should be the same in either case; see below.

Concrete to abstract syntax The type representing the abstract syntax of expressions is defined in `Types.hs`:

```
data Exp =
  Cst Int
| Var VName
| Join [Exp]
| Let VName Exp (Maybe Exp)
| Seq Exp Exp
| Is Exp Exp
| Sum Exp
| Count Exp
| Times Exp Exp
| Roll Exp
| Take Sel VName
deriving (Eq, Show)

type VName = String

data Sel = Min | Max | Rand
  deriving (Eq,Show)
```

The correspondence should be mostly straightforward, with the following notes:

- Local and global `let`-bindings should parse into the `Let` constructor with and without the optional body expression, respectively.

- The selectors `highest`, `lowest`, and `any` in the concrete syntax are written as `Max`, `Min`, and `Rand` in the abstract one.
- `+` should desugar to `Sum` and `Join`, as previously described.
- `of` should desugar to `let` and `take`, as previously described. The `let`-binding should use the variable `_`, which cannot occur in user expressions due to to the lexical restriction on variable names, so it will not clash with anything. For example, "`highest of (1,x)`" should parse as

  ```
  Let "_" (Join [Cst 1, Var "x"]) (Just (Take Max "_"))
  ```

- Selectors with counts should desugar into iterations. For example, "`any 2 of xs`" should parse as

  ```
  Let "_" (Var "xs") (Just (Times (Cst 2) (Take Rand "_")))
  ```

  A selector without an explicit repetition count should *not* generate a redundant iteration expression, such as `Times (Cst 1) ...`
- Parentheses are used both for constructing bags and for general grouping. A single expression in parentheses should be just parse as itself; e.g., "`(2)`" should parse as `Cst 2`, not `Join [Cst 2]`. Nested bag constructions should be flattened into a single `Join`; that is, the subexpressions e_i in `Join [`e_1`,...,`e_n`]` should not themselves be `Join`-expressions, but their contents should but be spliced into the containing `Join`; and again, single-expression `Join`s should not be generated. For example, "`(5,(x,(),7))`" should be parsed as just `Join [Cst 5, Var "x", Cst 7]`. Note that this flattening should happen also if the `Join` was constructed by desugaring a `+`.

Complex tokens The following rules apply for the terminals of the grammar:

- Numbers (`num`) are non-empty sequences of decimal digits, with the usual meaning.
- Variables (`var`) are non-empty sequences of upper and lowercase ASCII letters (a–z and A–Z), decimal digits, and underscores (_), starting with a letter. Keywords (see below) are *not* allowed as variable names.
- The Droll keywords are all the letter-only tokens that occur explicitly in the grammar (`"let"`, `"be"`, etc.) Note that keywords may be written in *any case*, as in "`Roll 5`" or "`highEst OF b`". However, variable names are still case sensitive, so "`x`" and "`X`" are *different* variables.

Whitespace Tokens may be interspersed with arbitrary whitespace (spaces, tabs, and newlines), which is normally ignored. However, some non-empty whitespace must used to prevent adjacent tokens from running together. For example, "`roll5`" can only be parsed as a variable name, not as a `roll`-expression, but "`roll 5`" and "`roll(5)`" are OK.

Interface Your Parser module should export a single function

```
parseExp :: String -> Either String Exp
```

It should return either the AST resulting from a successful parse of the input string as an expression, or a suitable error message.

For implementing your parser, you may use either the ReadP or the Parsec combinator library. If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators.Parsec`); in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`. As always, don't worry about providing informative syntax-error messages if you use ReadP.

Question 1.2: Calculator

The Calculator module defines some general functionality for working probabilistic computations, in addition to the Droll-specific parts. Note that some of the types and type constructors below are defined in `Types`; you should just define the requested functions in the Calculator implementation.

Probability monad The *probability monad* is a variation of the list monad for nondeterminism, in which every possibility in the list also has an associated non-zero probability. (Outcomes with probability zero are simply not listed.) To make a proper probability distribution, the probabilities in the list must sum to 1.

To avoid issues with round-off errors, and to allow computation of exact distributions, probabilities are represented as *rational numbers* (from the standard library Base.Ratio). We thus have:

```
type Prob = Rational -- must be >=0 and <=1

newtype PD a = PD {runPD :: [(Prob, a)]} -- probs must be >0 and sum to 1
  deriving Show
```

For example, the distribution over strings in which `"Red"` has probability $\frac{1}{2}$, `"Green"` has probability $\frac{1}{3}$, `"Blue"` has $\frac{1}{6}$, and no other strings are possible, could be written as

```
myDist :: PD String
myDist = PD [(1/2,"Red"), (1/3,"Green"), (1/6,"Blue")]
```

Note that the ordering of the possibilities in the list has no significance. Also, there is no requirement that each possibility is listed only once; the probabilities for each possibility are simply added up. For example, the exact same distribution as above could also be represented by

```
myDist = PD [(1/6,"Blue"), (1/4,"Red"), (1/3,"Green"), (1/4,"Red")]
```

(This non-uniqueness of representations is why `PD` does not have a derived `Eq` instance.)

Make the type constructor `PD` into a monad, using that a pure (i.e., effect-free) computation returns a single result with probability one, and that outcomes from a computation over a distribution are scaled by the probabilities of the input possibilities. For example,

```
do s1 <- myDist; s2 <- myDist, return $ s1 == s2
```

which represents the result of comparing two independent draws from the above distribution, should contain `True` with probability $(\frac{1}{2})^2 + (\frac{1}{3})^2 + (\frac{1}{6})^2 = \frac{7}{18}$, and `False` with the remainder, i.e., $\frac{11}{18}$. (Note that the result distribution will typically be fragmented across multiple contributions, and may need to be *normalized* for readability, as detailed below.)

There are two operations naturally associated with the probability monad:

```
unifrom :: Int -> PD Int
normalize :: Ord a => PD a -> PD a
```

The distribution `uniform` n, where $n > 0$, represents an unbiased choice between the numbers $0, 1, \ldots, n$. For example, a fair coin toss could be represented as

```
flip :: PD Bool
flip = do n <- uniform 2; return $ n == 1
```

For a distribution d over an *ordered* type a, `normalize` d represents the same distribution as d, but with the possibilities consolidated and listed in strictly increasing order. For example both of the above-listed distributions `myDist` should normalize to exactly

```
PD [(1 % 6,"Blue"),(1 % 3,"Green"),(1 % 2,"Red")]
```

(Note that `%` is the standard `show` representation of a `Rational` with a numerator and a denominator.)

Normalization is useful both for presenting the final results of a probabilistic computation to the user, and for speeding up computations over distributions, because each distinct outcome only needs to be considered once. (However, there is also a computational cost associated with a normalization, so it shouldn't be used indiscriminately.)

Finally, for those types that can be meaningfully averaged, we define a utility operation for computing the expected value of a potentially *partial* probability distribution, i.e., one that may include some failures:

```
expectation :: Fractional a => PD (Maybe a) -> (Prob, a)
```

(The class `Fractional` notably includes the type `Rational`, as well as `Double` and `Float`, but *not* `Int` or `Integer`.) The second component of the result of `expectation` d is the *weighted average* of all the non-`Nothing` possibilities in d. The first component is the probability that the computation is successful, i.e., not `Nothing`. For example, in

```
expectation $ PD [(1/2,Just 3.0), (1/4,Just 6.0), (1/4, Nothing)]
```

(i.e., the value 3.0 is twice as probable as 6.0, and there is a $\frac{1}{4}$ probability that there is no value at all), the result would be $(\frac{3}{4}, 4.0)$. If there are no `Nothing` values in the input distribution, the success probability is one; on the other hand, if `Nothing` is the only possibility, the success probability should be zero, and the expected value is not defined.

Bags The type of integer bags is straightforward:

```
newtype Bag = Bag {contents :: [Int]}  -- must be ordered
  deriving (Eq, Show, Ord)
```

Bags always have their canonical representation, so they can be compared directly for equality; and they are nominally ordered by the standard lexicographic ordering on lists. For taking the union of two bags, we define the function

```
union :: Bag -> Bag -> Bag
```

`union` can assume that the input bags are well formed, and it should return another well formed bag.

Probabilistic computations For the main probability calculator, we first define the types of values and stores:

```
type Value = Bag -- all elements must be >= 0

type Store = [(VName, Value)] -- all vnames must be distinct
```

A bag representing a Droll value can only contain non-negative integers. A store is just an association list from variable names to their current values. We require that a well formed store contains *at most one* binding for any variable (so the order of the list doesn't matter).

We also enumerate the things that can go wrong in a computation:

```
data RunError =
    Unbound VName Info
  | NonSingleton Info
  | EmptyChoice Info
  | OtherError Info -- anything else
  deriving (Show, Eq, Ord)

type Info = String -- optional additional information about the error
```

The error `Unbound` x s says that the variable x was unbound when evaluated. s may contain further information about the error, e.g. for helping track down where it occurred; its value is completely unspecified, and can be just left as `""`. Similarly, `NonSingleton` says

that a value that was required to be a singleton (e.g., the left argument to `times`) was not. `EmptyChoice` means an impossible choice, i.e., selecting (deterministically or randomly) from an empty bag, or a `roll 0`. `OtherError` is for any error situations not covered by the above. (But completely unimplemented features should simply throw a Haskell `error` when used.)

We can then define the monad of probabilistic computations with state and errors:

```
newtype Calc a = {runCalc :: Store -> PD (Either RunError (a, Store))}
```

That is, a calculation takes the current store, and returns a probability distribution over outcomes, which are either (fatal) runtime errors, or pairs of a result and a possibly updated store. Define the unit and bind functions of this monad.

The `Calc` monad also has the following associated operations:

```
fault :: RunError -> Calc a
lookVar :: VName -> Calc (Maybe Value)
setVar :: VName -> Maybe Value -> Calc ()
pick :: PD a -> Calc a
```

The operation `fault` *re* signals the runtime error *re*.

`lookVar` *x* looks up the current value of *x* in the store, and returns `Just` *v* if *x* is bound to *v*, and `Nothing` otherwise. (It does *not* signal an error in the latter case.). Conversely, `setVar` *x* `(Just` *v*`)` binds or rebinds *x* to *v*, and `setVar` *x* `Nothing` removes any binding for *x*. (If there was no such binding, it does nothing.)

Finally, `pick` *d* draws a value from the probability distribution *d*, exploiting that `Calc` is itself built on top of the probability monad.

As usual, your remaining code should only use the above operations for working with `Calc`-typed computations, rather than constructing or inspecting them directly.

Evaluator Finally, the `Calculator` module exports two functions specifically for evaluating Droll expressions:

```
eval :: Exp -> Calc Value
evaluate :: Exp -> PD (Either RunError Value)
```

`eval` *e* evaluates a (sub)expression *e* in the current store, whereas `evaluate` *e* evaluates a complete *e* in the initial (empty) store and returns the final distribution of results, which should be normalized. `evaluate` is allowed to use `runCalc`, in addition to the monad operations.

Getting started

As usual, aim to have some nontrivial working code for both modules, starting with the simpler parts, and adding more features later. Build your test suite along with the code, both to guide development, and to catch regressions.

For the Parser, start with parsing the core language of numbers, variables, bag construction, local bindings, and the simple infix and prefix operators. Unless you can quickly see how to handle them, you may want to wait with adding support for features such as global bindings, join-flattening, case-insensitive keywords, `of`- and `take`-desugaring, and exact operator precedences and associativities until after you have some basic parts of the calculator working.

For the Calculator, you may likewise want to leave `Take` for last. (However, if you cannot get the probabilistic aspects to work at all, a reasonable fall-back strategy may be to at least fully implement the deterministic parts of the language, including extracting highest/lowest values from bags.)

Finally, keep in mind that there is no requirement or expectation that the parts of the language covered by your parser should coincide with that of your calculator.

General instructions

All your code should be put into the provided skeleton files under `code/droll/...`, as indicated. In particular, the actual code for each module *Mod* should go into `src/`*Mod*`Impl.hs`. Do not modify `src/Types.hs`, which contains the common type definitions presented above; nor should you modify the type signatures of the exported functions in the APIs. Doing so will likely break our automated tests, which may affect your grade. Be sure that your codebase builds with the provided `app/Main.hs` using `stack build`; if any parts of your code contain syntax or type errors, be sure to comment them out for the submission.

Your *tests* should be runnable with `stack test`. A black-box test stub is provided in `tests/BlackBox.hs`; if you want to perform any additional white-box testing, those tests should be put in a separate `tests/suite1/WhiteBox.hs`.

In your *report*, you should describe your main design and implementation choices for each module in a separate (sub)section. Focus on the nontrivial, significant decisions you made, and justify them if you expect that they might be controversial. Low-level, technical details about the code should normally be given as comments in the source itself.

For the *assessment*, we recommend that you use the same (sub)headings as in the weekly Haskell assignments (Completeness, Correctness, Efficiency, Robustness, Maintainability, Other). In particular, some discussion of efficiency may be relevant for both the Parser and the Calculator. Feel free to skip aspects about which you have nothing significant to say. You may assess each module separately, or make a joint assessment in which you assess each aspect for both modules consecutively.

Question 2: Concurrent Turtles

The task in this question is to create a module, in Erlang, for controlling a bale of turtles on a big canvas, where each turtle has a small pen on its tail. When the pen is down, and the turtle moves, it draws on the canvas.

General comments

This question consists of two sub-questions: Question 2.1 about implementing a module for working with canvases and turtles, and Question 2.2 about writing QuickCheck tests against this API. Question 2.1 counts for 70% and Question 2.2 counts for 30% of this question.

In Appendix A you can find two examples that show how to use the API.

Remember that it is possible to make a partial implementation of the API that does not support all features. If there are functions that you don't implement, then leave them to return the atom `not_implemented`.

There is a section at the end of this question, on page 17, that specifies the expected topics for your report.

Terminology

A *picture* is represented as a list of line segments:

```
-type position() :: {integer(), integer()}.
-type line_seg() :: {position(), position()}.
-type picture()  :: [line_seg()].
```

The order of positions in segments does not matter, nor does the order of segments in a picture. Thus two pictures are equal if they contains the same segments.

Note however, that two pictures may be visually similar, but are not equal, if one of the pictures contains duplicate segments, for instance. Or two segments that line up may appear as one long segment.

A *turtle* has a position, a boolean specifying whether the pen is down, and an angle in degrees specifying which direction it is pointing. Degrees are one of the integers 0, 90, 180, or 270, where zero points east and rotation is counterclockwise so north is 90. A turtle can be *alive* or *dead*. It is unspecified exactly what dead means, as it depends on your design. It is an error to call API functions with dead turtles, and it is not something you explicitly need to check for. However, your solution should ensure that if you communicate with dead turtles, that should not interfere with the specified API.

A *canvas* has a set of turtles, and each turtle belongs to exactly one canvas. The canvas keeps track of which turtles are alive and dead, and what each turtle has drawn.

A turtle only draws on the canvas if the pen is down and the turtle moves. Thus, line segments of length zero are not possible to draw.

Your code should be able to handle concurrent canvases, and the canvases should be completely independent.

For visualising pictures you can use the function `svg:write/2` from the `svg` module found in the skeleton .zip file. The function takes a filename and a `Picture`, and writes an SVG image corresponding to the picture. The SVG file can then, for instance, be displayed by your web-browser.

Question 2.1: The `turtletalk` Module

Implement a module `turtletalk` for working with canvases and turtles. The client API of the module can be divided into a canvas-related part and a turtle-related part.

The turtle-related part of the API is the following functions, where *T* is always a turtle ID:

- A function `forward(`*T*`, `*N*`)` to move a turtle forward. For example, if the turtle is at position `{`*X*`, `*Y*`}` with angle 90 and it is moved forward *N* paces, then the new position is {*X*, *Y* + *N*}, and likewise for the other angles. Note that *N* must be an integer greater than or equal to zero, and the function should check this.

  Note that a call `forward(`*T*`, 0)`, while legal, will *not* generate a line segment.

- The functions `anti(`*T*`, `*D*`)` and `clock(`*T*`, `*D*`)`; where `anti` pivots the turtle *D* degrees anticlockwise, and `clock` pivots the turtle *D* degrees clockwise. Note that *D* must be one of the integers 0, 90, 180, or 270, and the functions should check this.

- A function `setpen(`*T*`, `*P*`)`, where *P* is either the atom `up` or the atom `down`, for setting the state of the turtle's pen.

- A function `clone(`*T*`, `*N*`)` for creating *N* new turtles. The new turtles belong to the same canvas as the original turtle, they start in the same position, have the same angle and have the pen in the same position. But they have not drawn anything on the canvas. Note that *N* must be an integer greater than or equal to zero, and the function should check this.

  On success the function should return `{ok, `*TL*`}` where *TL* is a list of turtle IDs for the *new* turtles; otherwise the function should return `{error, Reason}` (you decide `Reason`) and the turtle should die.

- A function `position(`*T*`)` for getting the position of a turtle. The function should return `{ok, `*P*`}` if it succeeds, where *P* is the position.

- The functions `forward`, `anti`, `clock`, and `setpen` should all return the atom `ok` for correct input. If the functions are called with wrong arguments the functions should return a pair `{error, Reason}` (you decide `Reason`) and the turtle should die.

The canvas-related part of the API is the following functions, where *C* is a canvas ID:

- A function `new_canvas()` for starting a new canvas process. The function should return `{ok, `*C*`}` if it succeeds.

- A function `blast(`C`)` for stopping a canvas process and all its turtles. After the function is called, there should be no processes alive stemming from the canvas C.
- A function `new_turtle(`C`)` for starting a new turtle process, with position `{0,0}`, the pen up and angle 0. The function should return `{ok,` T`}` if it succeeds.
- A function `picture(`C`)` for getting the picture painted by all living turtles on the canvas. The final picture is the concatenation (in any order) of the individual pictures painted by the turtles. The function should return `{ok,` P`}` if it succeeds.
- A function `turtles(`C`)` for getting the turtle IDs of all living turtles and a count of dead turtles. Thus, on success the function should return `{ok, {`LA`,` N`}}` where LA is the list of live turtles and N is the count of dead turtles.
- A function `graveyard(`C`)` for getting the picture painted by all dead turtles who have been on the canvas; the pictures by currently livings turtles should not be included. The final picture is the concatenation of the individual pictures painted by the turtles.
- The framework must be robust at least in the following senses: the picture drawn by a turtle should never get lost as long as the canvas is alive (even if the turtle dies), if one of the turtle processes fails (and thus dies), the canvas process should detect the failure, but the canvas (and all other turtles) should continue. No turtle processes should survive if their canvas process stops for one reason or another.

Question 2.2: Testing `turtletalk`

Make a module `test_turtletalk` that uses `eqc` QuickCheck for testing a `turtletalk` module. We evaluate your tests with various versions of the `turtletalk` module that contain different planted bugs and check that your tests find the planted bugs. Thus, your tests should only rely on the API described in the exam text.

Your `test_turtletalk` module should contain the following functions:

- A QuickCheck generator `turtle_cmd/0` that generates a *turtle command*. A turtle command is a pair consisting of the name (as an atom) of one of the functions for the turtle API and valid arguments (given as a list) for that for that function, except for the turtle ID.

  For instance, five samples from this generator could be:

  ```
  {setpen, [up]}
  {clone, [7]}
  {position, []}
  {forward, [10]}
  {setpen, [down]}
  ```

- A QuickCheck generator `turtle_cmds/0` that generates a list of turtle commands. The generated lists should have zero to twenty elements.

- A function `cmds_to_picture(Cmds)` that takes a list of turtle commands, and returns the picture generated by executing these commands in sequence (first to last) by a turtle on a blank canvas.
- A QuickCheck property `prop_no_empty/0` that checks that a list of turtle commands (as generated by `turtle_cmds/0`) will generate a picture (via `cmds_to_picture/1`) that contains no empty line segments.
- A `test_all/0` function that runs all your tests that only depends on the specified `turtletalk` API. These tests should involve the required properties in this module, but should also test aspects and functionality not covered by the required properties.
- We also evaluate your tests on your own implementation; for that you should export a `test_everything/0` function (that could just call `test_all/0`).

You may want to put your tests in multiple files (especially if you use both `eqc` and `eunit`, as they both define a `?LET` macro, for instance). If you use multiple files, they must all start with the prefix `test_`.

You are welcome (even encouraged) to make more QuickCheck properties than those explicitly required. Properties that only depend on the specified `turtletalk` API should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of the `turtletalk` library (perhaps they are related to an extended API or you are testing sub-modules of your implementation), they should start with the prefix `myprop_`, so that we know that these properties most likely only work with your implementation of `turtletalk`.

Topics for your report for Question 2

You should clearly document if you have implemented all parts of the question. Likewise, remember to detail how you have tested your module. In general, as always, remember to test your solution and include your tests in the hand-in.

Your report should also document:

- What erroneous behaviours your implementation can handle, and how you have tested that.
- What is the architecture for your code: How many processes do you have and what are their roles. How you deal with the management of helper processes (if any). Including your strategy for not keeping processes alive longer than needed. How you have tested this.
- The quality or limitations of your tests. Especially those explicitly required in Question 2.2. Explain how you have measured this quality. For example by using the QuickCheck aggregation functions; or how much of the API or code paths are covered by your tests.

Appendix A: Example use of `turtletalk`

Appendix A.1: `square.erl`

The `square` module demonstrates how to use the `turtletalk` API for drawing a square. The picture returned by `square:draw/0` should be equivalent to the picture:

```
[{{0,42}, {42,42}},
 {{42,0}, {42,42}},
 {{0,0},  {0,42}},
 {{0,0},  {42,0}}]
```

See the terminology section on page 14 for the definition of equality for pictures.

```
-module(square).
-export([draw/0, draw_to/1]).

draw() ->
    {ok, C} = turtletalk:new_canvas(),
    {ok, T} = turtletalk:new_turtle(C),
    turtletalk:setpen(T, down),
    turtletalk:forward(T, 42),
    turtletalk:anti(T, 90),
    turtletalk:forward(T, 42),
    turtletalk:anti(T, 90),
    turtletalk:forward(T, 42),
    turtletalk:anti(T, 90),
    turtletalk:forward(T, 42),
    ok = turtletalk:anti(T, 90),
    {ok, Pic} = turtletalk:picture(C),
    Pic.

draw_to(File) ->
    Pic = draw(),
    svg:write(File, Pic).
```

Appendix A.2: `multi.erl`

The `multi` module demonstrate how to use the `turtletalk` API to work with multiple canvases and turtles. The `multi:draw/0` function returns two pictures, both of a square.

```
-module(multi).
-export([draw/0, draw_to/2]).

draw() ->
    {ok, C1} = turtletalk:new_canvas(),
    {ok, T1} = turtletalk:new_turtle(C1),

    {ok, C2} = turtletalk:new_canvas(),
    {ok, T2} = turtletalk:new_turtle(C2),

    Pics =
        lists:map(
          fun({C, T, S}) ->
                  turtletalk:setpen(T, down),
                  {ok, [TC]} = turtletalk:clone(T, 1),
                  turtletalk:forward(T, S),
                  turtletalk:anti(T, 90),
                  turtletalk:forward(T, S),

                  turtletalk:anti(TC, 90),
                  turtletalk:forward(TC, S),
                  turtletalk:clock(TC, 90),
                  turtletalk:forward(TC, S),

                  {ok, Pic} = turtletalk:picture(C),
                  turtletalk:blast(C),
                  Pic
          end, [{C1, T1, 50}, {C2, T2, 75}]),
    Pics.

draw_to(File1, File2) ->
    [Pic1, Pic2] = draw(),
    svg:write(File1, Pic1),
    svg:write(File2, Pic2).
```