Introduction

This document contains answers to the questions asked in the AP 2025 exam text. It also contains some notes for graders on what answers we expect from students.

Questions

Question 1

The only things that have been modified are the FExp and Exp nonterminals, and it is likely that any correct solution will look very similar to fig. 1.

Some might argue that Exp1 is ambiguous because its productions have overlapping FIRST sets, but that is not true ambiguity (only ambiguous for LL parsers without backtracking), so we allow the form presented here, but it is also fine if students factor it out the same way we do for left recursion. By mistake, the "`**`" operator was left out of the exam text—it is acceptable for students to include it (if they do so correctly), but we do not require it. The grammar here does not have it.

The syntactical forms that end in an Exp (the LExp productions for "`loop`", "`let`", lambdas, etc) are in fact ambiguous, as it is not clear from the grammar how long the Exp extends. The idea is that the Exp extends as far as possible. We do not require students to have transformed the grammar for this case, but it is allowed if they do.

The implementation is very similar to the grammar. Every nonterminal corresponds to a top level definition, except for the ones with apostrophes (such as Exp2'), which are local recursive functions usually named `chain`, and the fact that the implementation supports the "`**`" operator. The longest-parse behaviour in LExp is handled correctly by the greedy behaviour of parser combinators.

Question 2

In my implementation of `Union` I filter away duplicate elements from the list representation of the set, so memory consumption will not increase when repeatedly unioning a set with itself.[1]

Question 3

In my implementation, an infinite loop will cause the `decide` function to never terminate. Adding a timeout in `Galapagos.Rules` itself is not possible, as the `rulesInteract` function must be pure, and timeouts require IO. A timeout could be added in two ways:

[1] **Correction note:** students don't have to remark on laziness here, but it is fine if they do.

```
Atom   ::=  var
         |  int
         |  bool
         |  "(" Exp ")"
         |  "meet"
         |  "ignore"
         |  "groom"
         |  "{""}"
         |  "{" Exp Exps "}"
Exps   ::=
         |  "," Exp Exps
FExp   ::=  Atom FExp'
FExp'  ::=
         |  Atom FExp'
LExp   ::=  "if" Exp "then" Exp "else" Exp
         |  "\" var "->" Exp
         |  "let" var "=" Exp "in" Exp
         |  "loop" var "=" Exp "for" var "<" Exp "do" Exp
         |  FExp

Exp3   ::=  LExp Exp3'
Exp3'  :==
         |  "*" LExp Exp3'
         |  "/" LExp Exp3'
Exp2   ::=  Exp3 Exp2'
Exp2'  :==
         |  "+" Exp3 Exp2'
         |  "-" Exp3 Exp2'
Exp1   ::=  Exp2
         |  Exp2 "without" Exp1
         |  Exp2 "union" Exp1
Exp0   ::=  Exp1 Exp0'
Exp0'  ::=
         |  "elem" Exp1 Exp0'
Exp    ::=  Exp0 Exp'
Exp'   ::=
         |  "==" Exp0 Exp'
```

Figure 1: APL grammar without left recursion, but still slightly ambiguous—in LExp the final Exps are intended to extend as far as possible. Realistically, this is also what we expect of the students.

- In `BlockAutomaton.Server`, a timeout policy could be added such that if the `rulesInteract` function does not terminate within a second, the cells involved are reset to an initial state. This is not *exactly* the same as what happens when a finch behaves illegally, but it is close.
- Otherwise, the `rulesInteract` field must be extended to be in `IO`, such that the finch meeting function could implement timeouts.

Question 4

I am reasonably confident that my solution does not have race conditions. Partly this is due the test labelled `determinism` which repeatedly runs a simulation and checks that the same outcome is observed every time.[2] But my implementation also seems like it ought to be deterministic, as each cell is represented using a single process, connected to its neighbours via four channels, and communicating with them using a completely deterministic rotating scheme, where each channel has only a single sender and receiver.

Question 5

I implement each grid cell as a persistent thread. `stepAutomaton` has to send a `CellStep` message to every thread. This could be done with a sequential loop, but this would not be fully concurrent. Instead, I construct a tree of threads that concurrently broadcast a message to multiple receivers—this requires a total of $O(n \log n)$ messages, where n is the number of threads. The actual work requires each thread to then communicate with two of its neighbours, resulting in four messages per cell. In total, for a grid of size $h \times w$, my implementation transmits $O((h \times w) \log(h \times w))$ messages.[3]

The tree-based broadcasting of `CellStep` ensures concurrent stepping, as there is no requirement that a message is sent to cell i before it can be sent to cell $i + 1$.[4]

Question 6

As memory is just the environment of the running APL expression, the simplest way to implement this would be to copy the `finchStrategy` field in its entirety. The downside of this is not just the long-running computation in an inefficiently implemented `Free` monad [5], but also that if the finch strategy has any memory stored as a set (such as the Grudger strategy), then the memory set will continue growing indefinitely to remember the actions of long-dead finches taken against the ancestors of the new finch.

[2] **Correction note:** a test is good, but it is not required and also not sufficient.

[3] **Correction note:** an asymptotic estimate is all we look for, but it is fine if students provide an exact number.

[4] **Correction note:** sequential broadcasting is correct as far as this question is concerned, but only if they realise that it inhibits concurrency.

[5] **Correction note:** We don't expect students to catch this one.

Question 7

I have tested this using probabilistic testing where I generate a configuration file with a random parameter ordering, as well as the `Config` value that is expected from parsing the configuration file. Concretely, this is represented with the type

```
data ConfigSpec = ConfigSpec Galapagos.Config String
```

defined in `Tests.hs`, and which is an instance of `Arbitrary`. The property to test is then just:

```
property $ \(ConfigSpec cfg s) ->
  parseGalapagos "input" s == Right cfg
```

A testing approach based on unit tests would be very verbose and highly brittle, and difficult to determine whether it is complete.[6]

[6] **Correction note:** but still partially correct, especially if the students realise the limitations.