Exam (and Assignments) Strategies

Following are some advice on good strategies for the exam in Advanced Programming. While the advice are for the exam, most also apply to the assignments.

Before You Start

Having obtained the exam set, it is a good idea to skim the whole set and compare it to the course description.

Try to write down which learning objectives you think we want you to demonstrate in on in every question. Each question typically cover many overlapping course objectives, so focusing only one or two questions is usually **not** a winning strategy.

This approach **might** help you prioritize your time.

General

Write a handful sample programs that you want to be capable of running in the end. If writing a parser or interpreter, try to write a couple sample programs that cover as many features of the language as possible. If writing an Erlang library, write a couple programs that use the coming features of your library.

This **might** help you identify important parts of the assignment.

Stay structured, make sure to cover the course objectives and the important parts, as identified earlier.

Look for an incremental approach. Can you start with a simple implementation, or a subset of the task, and work your way towards a more full-blown implementation? Typically, you can.

If working on a task where you have to define monads, try to decompose the monad you need to define. Is it a mixture of `Maybe`/`Either`, `List`, Reader, Writer, or State monad? If you implement a partial solution, make sure to cover all the relevant parts of the monad instance.

If you need to keep track of changing state, it’s a State monad.

If you need to keep track of possible alternatives, it’s a List monad.

If you only need to read, but not write (except in local circumstances) throughout a computation, it’s a Reader monad.

If you need to write, but not read, it’s a Writer monad.

A State monad is a mixture of the Reader and Writer monad types.

For example, the `SimpleParse` parser combinator library uses a List monad to keep track of all the possible ways to parse a given `String` combined with a State monad to keep track of how much of the input that are left to be parsed.

A nice trick to identify if code based on monads are nicely modular structured: try to make a change to the monad type, now only a few places should break.

Make sure to choose adequate libraries for your tasks, if they exist. For instance, choose the parser combinator library wisely, and use OTP when relevant in Erlang. Use the library functions as indented. For instance, you should know when and why it is relevant to use `try` in `Parsec`, and likewise when and why to do use `munch` from `ReadP` and `SimpleParse`.

Clean up your code. Make it easy to verify that your solution indeed solves the task at hand. For instance, if writing a parser, your parser should closely correspond to the grammar in your report. Remove noise; such as overwhelming, explicit handling of whitespace.

Testing

Start with a nice handful of sample programs and their expected outcomes, perhaps unit tests. (You should already have some from the start. Now write some more to test the rest of the functionality.)

Make sure to test all the parts of the assignment that you’ve managed to implement.

Make sure to do both:

Positive testing, (i.e., that your solution behaves as intended in valid scenarios).

Negative testing, (i.e., that your solution adequately fails in invalid scenarios). Crashing with a run-time exception that you didn’t define yourself (e.g., undefined or inexhaustive pattern-matching) is inadequate, even in partial implementations.

Unless your are explicitly asked for it, QuickCheck can *in some circumstances* be overkill. That is, it may take more time than you are given credit for. That goes for both Haskell and Erlang. If you are pressed for time and unsure on how to use QuickCheck, you may want to start by some unit tests.

If you use QuickCheck (or any other testing method for that matter) make sure that you document that you know the weaknesses of your approach. Even when you are using QuickCheck, there might be some aspects that you are not covering. For instance, to test your parser you might write a pretty printer that renders all parentheses even those that are redundant, then you won't test that your parser handle precedence correctly.

Concurrency, and run-time behaviour in general, can be hard to test, especially using QuickCheck. It is often feasible to have some parts tested by full-blown QuickCheck, while other parts are tested using good old sample programs or unit tests.

Your tests should form a solid foundation for the assessment in your report.

Erlang tasks typically require more testing than Haskell tasks due to the weaker type-system of Erlang. Prolog tasks should be tested as well.

It should be easy for us to re-run your tests, and read your test results.

Report

Describe how you've resolved the intended and unintended ambiguities in the assignment text. Provide a documentation of your implementation. For instance, if writing a parser, describe what you've transformed the grammar (if need), and include the final grammar in your report.

Erlang tasks typically require a deeper discussion about failure scenarios and how you handle them, due to both the concurrent nature of Erlang programs, and the weaker type-system.

Discuss in how far you've solved the task. Which parts did you omit? Why? How would've you gone about those parts, had you had more time?

Assess the quality of your solution.

Your tests should form the base of your assessment.

Give an overview of how your tests relate them to the exam text.

Is your solution modular, easy to read without too deep nesting of expressions and so on?

Are you using library functions as intended?