Take-home Exam in Advanced Programming

Deadline: Friday, November 11, 16:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2016. This document consists of 15 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of 4 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner.

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 5–10 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, archiving one directory called `src` (which may contain further subdirectories).

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.

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 constitues 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 any particular 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 *specfic* academic citation for any material from which you draw considerable inspiration – including what you may find on the Internet, such as snippets of code. 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.*

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

The APMake system

APMake is a tool for automating complex build procedures for software or documents, very loosely based on GNU Make. A *makefile* consists of a collection of *rules*, each specifying how to build a set of *target* files from some *prerequisite* files, using a sequence of commands. For example, a rule could look like this:

```
foo foo.map : foo.o bar.o
        cc -o foo foo.o bar.o
        nm foo > foo.map
        strip foo
```

Here we say that, to build the executable `foo` with associated symbol table `foo.map`, we require the object files `foo.o` and `bar.o` to already be present. The *build step* then consists of first linking the two object files into `foo`, dumping the link map to `foo.map`, and finally stripping the symbol table from the executable.

In general, building a file from sources requires multiple steps, depending on which files are already present, and which must themselves be built first. For example, suppose the makefile contains two further rules:

```
foo.o : foo.c
        cc -c foo.c

bar.o : bar.s
        as -o bar.o bar.s
```

Then, in order to build `foo`, if one or more of the specified prerequisite object files do not yet exist, the build system will first compile `foo.c` to `foo.o`, and/or assemble `bar.s` to `bar.o`. (Note that the conventions and syntax for invoking the various Unix build tools are quite inconsistent, so we need to give explicit commands lines for each, not just the name of the tool.)

The above examples are all *simple* rules. A *pattern rule* gives a uniform recipe for building targets of a particular kind from prerequisites. For example, we could write a general rule for producing object files from assembler sources like this:

```
%.o : %.s
        as -o %.o %.s
```

Such a pattern rule stands for the infinite collection of rules obtained by consistently replacing all occurrences of `%` with an arbitrary string. In particular, it subsumes the special-case rule for building `bar.o` above. The `%` will often appear as the first character in a target or prerequisite filename, but in general it may be used anywhere in the name. However, we require that if *any* of the prerequisites or commands contain a `%`, then *all* of the targets must also contain at least one `%`. This is to avoid underdetermined build steps like

```
guess : %
        cat % > guess
```

where it is effectively left completely unspecified which file the system should use or build as a prerequisite for `guess`.

Note that in general there may be multiple ways of building a target file from sources. For example, if both `foo.c` and `foo.s` exist, we can build `foo.o` either by the simple rule for compiling `foo.c`, or by the pattern rule for assembling `foo.s`; both ways are considered correct. If we also had a (simple or pattern) rule for compiling C sources to assembly (using `cc -S`), it would also be possible to use that rule first to get `foo.o` from `foo.c` in two steps, generating the intermediate assembly file `foo.s` explicitly.

Many build systems have support for *incremental* building, where targets are *rebuilt* if one or more of their prerequisites has changed. For simplicity, we do not support this in APMake; that is, we conceptually only consider building from scratch.

Finally, note that sometimes a makefile may include rules potentially allowing file transformations in both directions. For example, we could have pattern rules:

```
%.pdf : %.ps
        ps2pdf %.ps %.pdf

%.ps : %.pdf
        pdftops %.pdf %.ps
```

Both are useful in different contexts, since some tools work only on Postscript files, while others require PDFs. This means that the dependency graph specified by a makefile may in general contain cycles.

Question 1: APMake parser

We give the following grammar for APMake makefiles:

$$
\begin{array}{rcl}
\mathit{Makefile} & ::= & \mathit{Rules} \\
\mathit{Rules} & ::= & \mathit{Rules}\ \ \mathit{Rule} \\
 & | & \varepsilon \\
\mathit{Rule} & ::= & \mathit{Targets}\ \texttt{':'}\ \mathit{Prereqs}\ \mathit{Commands}\ \texttt{'\textbackslash n'} \\
\mathit{Targets} & ::= & \mathit{Templates} \\
\mathit{Prereqs} & ::= & \mathit{Templates} \\
 & | & \varepsilon \\
\mathit{Commands} & ::= & \mathit{Command}\ \ \mathit{Commands} \\
 & | & \varepsilon \\
\mathit{Command} & ::= & \texttt{'\textbackslash n'}\ \texttt{'\textbackslash t'}\ \mathit{Template} \\
\mathit{Templates} & ::= & \mathit{Templates}\ \ \mathit{Template} \\
 & | & \mathit{Template} \\
\mathit{Template} & ::= & \mathit{Template}\ \ \mathit{Frag} \\
 & | & \mathit{Frag} \\
\mathit{Frag} & ::= & \mathit{Literal} \\
 & | & \mathit{Stem} \\
\mathit{Stem} & ::= & \texttt{'\%'}
\end{array}
$$

A target or prerequisite *Literal* is a nonempty sequence of characters, with each character adhering to one of the patterns below:

1. The character matches the regular expression `[a-zA-Z0-9_./-]`.
2. The character is `\`, and it is followed by `%`, `:`, ␣ (space character), or `\`.

   This is regarded as an escape sequence, and allows to use these otherwise significant characters as part of a literal. For instance, the string `f\%o\\o\ \:.c` is to be regarded as the literal `f%o\o :.c`.

APMake really has two syntaxes in one. First, we want to specify rules in terms of targets, prerequisites, and recipes (command sequences). Second, there are the commands themselves. Other than simple stem replacements, APMake makes no attempt to interpret the commands themselves.

Hence, a command *Literal* is more permissive. It is a sequence of arbitrary characters, until a line break (`\n`) occurs. To allow a long command to spread over multiple lines, we allow the escaping of line breaks with a `\` character. To allow the `\` character itself, it too has to be escaped, as above.

APMake is also subject to the following whitespace rules:

1. Fragments are separated by nothing.
2. Rules are separated by arbitrary whitespace, except `\t`.
3. All other tokens (in particular, targets and prerequisites) are separated by non-line-breaking whitespace (i.e., arbitrary whitespace, except `\n`).
4. The file may start and end with arbitrary whitespace.

What to implement

You should implement a module `Parser` with the following interface.

A function `parseString` for parsing a makefile given as a string:

```
parseString :: String -> Either ParseError Makefile
```

Where you decide and specify what the type `ParseError` should be, the only requirement is that it must be an instance of `Show` and `Eq`. The type `ParseError` must also be exported from the module. The handed-out skeleton code already has the exports set up correctly. Your implementation should go into `Parser/Impl.hs`.

We also provide a function `parseFile` for parsing a makefile given in a file located at a given path:

```
parseFile :: FilePath -> IO (Either ParseError Makefile)
```

Where `ParseError` is the same type as for `parseString`.

The type `Makefile` is defined in the handed out `Ast.hs`. We list this module below for quick reference. You should not change the types for the abstract syntax tree unless there is an update on Absalon telling you explicitly that you can do so.

```
module Ast where

data Frag
  = Lt String
  | St
  deriving (Eq, Show)

type Template = [Frag]

type File = String
type Command = String

type FileT = Template
type CommandT = Template

data Rule
  = Rule [FileT] [FileT] [CommandT]
  deriving (Eq, Show)

type Makefile = [Rule]
```

It is intentionally left unspecified how to parse templates with literals of length two or more characters. For instance, the template `AP` may be parsed as either `[Lt "AP"]` or `[Lt "A", Lt "P"]`. Choose a disambiguation and argue for your choice in your report.

You must use one of the three monadic parser libraries, `SimpleParse.hs`, `ReadP` or Parsec to implement your parser. You will find Haskell skeletons for the parser and abstract syntax tree on Absalon.

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`), in particular you are *disallowed* to use `Text.Parsec.Token`,

Question 2: APMake interpreter

A *build plan* for generating a file from some set of already existing files is a sequence of build steps for generating that file. A build plan is said to be *valid* if never performs a build step specified by a rule unless all of the rule's prerequisite files already exist – either initially, or as the result of some previous build step. However, a valid plan is in general allowed to contain redundant or unnecessary steps.

On the other hand, a *minimal* valid build plan is one from which no step can be removed without invalidating the plan, i.e., breaking at least one dependency. In particular,

- A minimal build plan will never invoke a build rule if *all* of the rule's targets already exist at that point in the plan. Note that, since individual rules may produce multiple targets in one logical step (like the rule for building `foo` and `foo.map` in the example above), it is allowed to unnecessarily rebuild some of the other targets, in order to generate the single missing one.
- A minimal build plan will never invoke a build rule unless at least one of that rule's targets is directly or indirectly (i.e., later in the plan) needed to satisfy a build request from the user. That is, in Haskell terms, the build is performed *lazily*.

Keep in mind that the notion of minimality is defined with respect to build steps, not individual commands. It may happen that, for a rule with multiple targets of which some already exist, any particular command in the list could be omitted. But since the build system has no understanding of what the individual commands in a rule do, it only tracks dependencies for the rule as a whole. If two targets in a rule are truly independent, it may instead be possible to split the rule into two in the makefile.

Note also that a minimal plan need not be *globally optimal* in any particular sense, such as containing as few steps as possible. For example, generating `foo.o` from `foo.c` in either one or two steps are both minimal, in the sense that they don't do any redundant work.

Since the dependency graph may contain cycles, it is important that the build system does not go into an infinite loop when trying to construct a build plan. For example, given the mutually recursive rules for converting between PDF and PS files, if neither `foo.ps` nor `foo.pdf` exists initially, but `foo.pdf` is needed, the build system shouldn't try to satisfy the `ps2pdf` rule's demand for `foo.ps` by attempting to in turn build that file from `foo.pdf`, if there is also a `dvips` rule for building `foo.ps` from `foo.dvi`. (Or, if there is no such rule, the system should report a failure, rather than going on forever.) In the presence of pattern rules, it may be quite hard to decide with certainty whether a particular file can be built from existing sources, so one normally sets an arbitrary limit for how complicated plans the system will consider before giving up. That is, if a given file cannot be constructed using using a chain of at most, (say) 10 rules from the existing sources, it will be considered as unbuildable.

Finally, it is important to note that the entire build plan is computed statically, before any steps are performed. APMake does not monitor the build process to check that all the promised target files are actually created by the rule's commands, nor does it attempt to detect that execution of a particular build command has failed, and either redo it (in case the failure was transient) or consider another rule for building the required targets.

Your task Given a well-formed makefile, the name of a desired target file, a function for testing whether a file initially exists, and a bound on how deep dependency chains to consider, you must construct a valid, minimal build plan for the target file, or report failure. That is, more concretely, define a Haskell function,

```
build :: Makefile -> File -> (File -> Bool) -> Int -> Maybe [Command]
```

with the requested behavior. For the bound, we say that a file can be built in 0 dependency steps if it already exists; and it can be built in $n + 1$ steps if it occurs among the targets

of a rule, all of whose prerequisites files can themselves be built in at most *n* dependency steps.

If a solution cannot be found within the given bound, your interpreter is not *required* to fail, if it can see a way to construct the target anyway, even if using a longer dependency chain. However, it is important that `build` always succeeds or fails in finite time, for any bound.

For example, given the makefile rules from above, parsed as `mf`, the call

```
build mf "foo" (`elem` ["foo.c", "bar.s"]) 2
```

should return

```
Just ["cc -c foo.c", "as -o bar.o bar.s", "cc -o foo foo.o bar.o",
      "nm foo > foo.map", "strip foo"]
```

Note that the bound is expressed in build steps, not in the number of individual commands. Here, `foo` requires only a dependency chain of length 2, because the compilation of `foo.c` and assembly of `bar.s` to object files are independent and could in principle be done in any order, or even in parallel). This means that only the final linking rule constitutes a second dependency step.

On the other hand, had we set the bound to 1 instead of 2, the above would still be a valid result, but so would `Nothing`.

Use monads as appropriate to encapsulate repetitive patterns in your code. Also, in the report, be sure to explain the design choices you made for your `build`, and how they support the goal of generating valid, minimal plans whenever they exist, while avoiding infinite loops.

As a lead-up to the definition of `build`, you are also required to define and export some specific auxiliary functions:

```
replace :: String -> Template -> String
```

This function computes the result of replacing all placeholders in the template with the specified string; for example, `replace "bar" [Lt "foo-", St, ".c"]` should return `"foo-bar.c"`.

```
match :: String -> Template -> Maybe (Maybe String)
```

This function attempts to match a string against a template, with three possible results:

- `Nothing`, if the template cannot be made to match the string by any replacement of the placeholder with some specific string. This is the case, e.g., in the sample call `match "foo.c" [St, Lt ".o"]`, or `match "foo-bar" [St, Lt "-", St]`.
- `Just Nothing`, if the template matches the string, but does not contain any placeholder, e.g., in the call `match "foo.c" [Lt "foo", Lt ".c]`,

- `Just (Just s)`, if the template matches the string by taking the placeholder to be `s`. For example, `match "foo-foo" [St, Lt "-", St]` should return `Just (Just "foo")`. Note that if the template contains at least one placeholder, there can be *at most* one way to instantiate it, to match any given string. (Why?)

You should not assume anything about the lengths of any literals in the `Template` argument, even if your parser makes some guarantee about them.

```
check :: Makefile -> Bool
```

This function checks whether all rules in the makefile are well-formed, i.e., if any of the prerequisites or commands in the rule contain a placeholder (`St`), then so do all of the targets. You may assume that `build` will only be called on makefiles that have passed this check.

General advice For all parts, efficiency is secondary to correctness, but your programs shouldn't be gratuitously inefficient. Make sure that you have tested your solutions, and that your testing is automated so that we can easily run your tests and verify your results.

If you cannot solve the problem in full generality, try scaling it down to something simpler. For example, your interpreter might not support pattern rules, or not necessarily generate minimal scripts. Be very explicit about what limitations or simplifications you impose, and why you believe that you have at least fully solved the simplified problem.

The Pirate system

The following two questions is to make two essential parts of Pirate, an alternative to the build-system Ninja.

Every build system needs to invoke commands that build something. Question 3 is about making a library that takes care of handling concurrent command invocations.

Question 4 is about orchestrating the execution of build plans (called treasure-maps in the Pirate system).

It should be mostly possible to solve the two questions independently, as there is only a single API function linking them together. However, you are allowed to use the library from Question 3 to solve Question 4, and vice versa, if you think that makes sense.

Question 3: Robust Commands

This question is about making a library, `gen_command`, for handling the generic parts of making robust commands, that can be used in a Pirate server, for instance.

Often when we invoke certain commands we want to keep track of, and limit, the number of concurrent invocations. Maybe because we know that the command is resource intensive, linking big C++ projects for instance, or because of external demands, maybe we only have licence to run a limited number of concurrent instances of the command.

Similarly, part of making a robust command is to handle cleaning up after commands that are aborted.

The `gen_command` module should handle the generic parts of managing concurrent invocations and structuring the cleaning up procedure, while leaving the implementation of a command to a callback module.

The `gen_command` module should export the following API:

- `start(Mod, Args)` for starting a command server with callback module `Mod`. Returns `{ok, ServerRef}` on success or `{error, Reason}` if some error occurred. The argument `Args` is passed on to the callback function `initialise`.
- `invoke(ServerRef, Args, From)` for invoking the command. If less than the limit of concurrent invocation of the command is running, a new concurrent evaluation is started; otherwise the invocation is queued until it is possible to run it. Returns `{Ref, CID}`. When the command is completed it will send a message of the form `{Ref, Reply}` to `From`, where `Reply` is a pair where the first component is either `success` or `failure` and the second component is the atom `aborted` if the command was aborted (even if it was queued and never started) or result of the command (successful or not). If `handle_invocation/2` throws an exception of any kind the second component is a pair of the exception kind and the exception.
- `avast(ServerRef, CID)` for aborting a command invocation.
- `ahoy(ServerRef, CID)` for querying the state of a command invocation. Returns one of the atoms `queued`, `running` or `completed`.
- `furl(ServerRef)` for shutting down a command server. All ongoing (running and queued) invocations will be aborted. Returns `ok` when all ongoing commands are finished and `handle_furl/1` from the callback module has returned.

A callback module for `gen_command` should define the following callbacks:

- `initialise(Args)` for computing the initial state and for specifying how many concurrent invocations are allowed, should return `{ok, State :: term(), Limit}`. Where `Limit` is either an integer or the atom `infinity`, where `infinity` means that there are no limit on how many concurrent invocations can be started.

- `handle_invocation(Args :: term(), State :: term ())` for handling an invocation. Should return `{reply, Reply :: term()}`, return `blimey`, may throw an exception or may exit. Returning `blimey` should have the same effect as if `avast/2` was called on the invocation.
- `carren(State :: term(), Args :: term())` is called after an invocation with arguments `Args` has been aborted. Used for cleaning up.
- `handle_furl(State :: term())` is called as part of `furl/1` after all invocations has been aborted. Used for cleaning up any global state. Should return `ok`.

Document which properties your module provides (and under which assumptions). Remember to detail in your report how you have tested these properties. In general, as always, remember to test your solution and include your test in the hand-in.

Question 4: Concurrent Plans

This question is about making Pirate, an alternative to the build-system Ninja. Pirate make it possible to start a *mission* according to a *treasure-map* that describes what tasks need to done to complete a mission.

A mission can be succeed with a result, or can fail with a result. That is, the result of a mission will always be a tuple where the first component is one of the atoms `success` or `failure`.

Implement an Erlang module `pirate` with the following API, where `SubMap` is always a treasure-map and `SubMaps` is a list of treasure-maps:

- `go_on_account()` for starting a Pirate server.

  Returns `{ok, Pirate}` on success or `{error, Reason}` if some error occurred.

- `register_simple(Pirate, Cmd, Fun)` is for registering the function `Fun` as the command `Cmd`, an atom, at the Pirate server. It is assumed that it is safe to terminate a process running `Fun` at any point, and that there are not limits on how many concurrent instances we can run. Likewise you *may* assume that `Fun` will always succeed with a result `R`, thus the result of invoking `Cmd` should be `{success, R}`.

- `register_command(Pirate, Cmd, Mod, Args)` is for starting a command server with the callback module `Mod` and arguments `Args` and registering it as the command `Cmd`, an atom, at the Pirate server.

  Command servers are described in Question 3.

  Note: This question is awarded full points even if the function `register_command/3` is implemented as a function that returns `undefined` if you haven't implemented Question 3.

- `mission(Pirate, TreasureMap)` for starting a mission according to the treasure-map `TreasureMap`.

  Returns `{ok, MissionID}` on success or `{error, Reason}` if some error occurred.

  A treasure-map describe the different sub-missions of a mission, where a sub-mission is also described by a treasure-map. That is, a treasure-map is of the following form:

  - `{cmd, Cmd, Arg}` is the simplest mission. It just consists of invoking the registered command `Cmd` with the argument `Arg`. The result of the command is the result of mission.
  - `{line, SubMaps}` is for running the sub-missions in `SubMaps` in sequence. If a sub-mission fails, then the rest of sub-missions a skipped and the mission is failed. The result of the mission is the result of the last sub-mission.
  - `{swarm, SubMaps}` is for running the sub-missions in `SubMaps` concurrently. If any sub-mission succeeds, then the whole mission succeeds and the result is one the results of the successful sub-missions. The mission is not complete until all sub-missions are completed.

  - `{blood_brothers, SubMaps}` is for running the sub-missions in `SubMaps` concurrently. If any sub-mission fails then all other sub-missions are aborted and the whole mission is considered a failure with the result of the failing sub-mission. If all sub-missions succeeds then the whole mission succeeds and the result is one the results of the successful sub-missions. The mission is not complete until all sub-missions are completed.
  - `{false_flag, SubMap}` is for turning failure into success. If the sub-mission, described by `SubMap`, fails with the result `R`, then the mission succeeds with the result `R`.
  - `{warp, SubMap, FunMission}` runs the sub-mission described by `SubMap`, if the sub-mission succeeds with the result `R`, then `{success, R}` is given as argument to `FunMission`; otherwise `{failure, FR}` is given as argument to `FunMission`. `FunMission` is a function that returns a treasure-map as result. The result of the mission is the result of running the mission described by the result of `FunMission`. If `FunMission` throws an exception the mission fails with the result `not_a_fun_map`.
  - `{tick_tack, SubMap, Limit}` starts a time-triggered bomb that will go off if the mission described by `SubMap` is not completed in `Limit` hundreds of a second (that is, a limit of 100 corresponds to one second). If the bomb explodes then the sub-mission is aborted, and the result is a failure with the result `timeout`; otherwise the result of the mission is the result of the sub-mission.

  Appendix A gives some examples of treasure-maps.

- `ahoy(MissionRef)` queries the status of a mission. It returns:
  - `{failure, aborted}` if the mission was aborted.
  - `{failure, FailureRes}` if the mission failed with `FailureRes`.
  - `{success, Result}` is the mission succeeded with `Result`.
  - `{ongoing, NumSub}` if the mission is still ongoing, and there are currently `NumSub` ongoing submissions (including sub-sub-missions and so on).

Document which properties your module provides (and under which assumptions). Likewise you should document if you have implemented all parts of the question, for instance, is all form of treasure-maps supported. Remember to detail in your report how you have tested your module. In general, as always, remember to test your solution and include your test in the hand-in.

Appendix A: Example treasure-maps

Appendix A.1: Simple treasure-map

The following treasure-map show how to run two commands in sequence:

```
{line, [ {cmd, write_to_file, ["hello.txt", "Arr!\n"]}
       , {cmd, write_to_file, ["hello.txt",
                               "Dead men tell no tales\n"]}
       ]}
```

Where `write_to_file` is assumed to be a command used for writing text to a file.

Appendix A.2: Involved treasure-map

The following treasure-map is an example of a more involved treasure-map:

```
{blood_brothers,
 [ {cmd, print, "Yo, ho, ho, and a bottle of rum"}
 , {tick_tack,
      {warp, {cmd, read_file, ["hello.txt"]},
             fun ({success, Content}) ->
                     {cmd, write_to_file,
                             ["olleh.txt",
                              lists:reverse(Content)]};
                 (_) ->
                     {cmd, write_to_file,
                             ["olleh.txt",
                              "blimey"]}
             end},
    500}
 ]}
```

Where `write_to_file` is assumed to be a command used for writing text to a file, `read_file` reads the content of a file and `print` is used for printing something to the screen.