Take-home Exam in Advanced Programming

Deadline: Friday, November 8, 16:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2019. This document consists of 20 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of 2 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 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 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.

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 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 *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.

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.

Question 1: Implementing Configuraptor

Background

In *made-to-order manufacturing*, a customer can specify the desired *configuration* of a product by choosing from a variety of options for individual pieces of functionality – more than can reasonably be accommodated by selecting one of a fixed range of preconfigured product models. A prototypical example of such a situation occurs when building a desktop PC from scratch, using off-the-shelf components (case, power supply, motherboard, processor, RAM, disk, graphics card, monitor, operating system, etc.)

A complicating factor is that not all combinations of component selections are feasible, or even meaningful. For example, a powerful graphics card may require a correspondingly heavy-duty power supply; or a motherboard will only work with a specific processor family, and accommodate only a limited number of RAM modules. Also, in almost all cases, the total system price is an important constraint, precluding simultaneous top-of-the-line choices for all components.

To assist both buyers and sellers of made-to-order products, a *configurator system* is often used to keep track of the various options and the dependencies between them; then the system may be able to suggest particular configurations satisfying the customer's functionality and performance needs, as well as their budget. In this task, you will implement a simple such system, Configuraptor.

Informal description of Configuraptor

Configuraptor is a generic configurator system, not inherently tied to any particular application domain, but particularly suited for situations when products are built up from a relatively small number (typically 5–20) of discrete components. For concreteness, our examples will be mainly taken from the above-mentioned domain of PC configuration.

The two key Configuraptor concepts are *resources* and *components*. A resource represents some key characteristic of a configuration or component choice, such as available RAM or disk space, USB or video ports, or – most importantly – available funds. All resources must be explicitly declared, e.g., `resource GB-RAM-space`. Note that resource names are case-insensitive, so `gb-ram-space` or even `gB-RaM-Space` would also refer to the same resource.

A component (whether hardware or software) then provides some resources, often in exchange for others. For example, an external disk will contribute some amount of storage space, but will typically use up a free USB port; whereas an internal disk may instead use a SATA port, as well as a 3.5" drive bay; and both will naturally cost money, generally increasing with the disk size. Or some piece of software may provide a needed functionality, but require a particular operating system, as well as a set of shared libraries, to also be installed.

When talking about the resource needs of components, it is useful to distinguish between *exclusive* and *shared* uses: a component (say, a RAM module) may completely monopolize some resource (a DIMM slot), making said resource in practice unavailable for any other components. Or it may merely require some resource to be present (say, a shared library, or a sufficiently powerful graphics card), without precluding other components from utilizing it (though perhaps not at the same time). For example, we may have the following component declaration:

```
component MyExtDisk-8TB:
  provides 8000 GB-disk-space;
  requires NTFS-support;
  uses 1 USB-port;
  uses 1500 DKK.
```

That is, this particular disk provides 8000 GB of disk space, but requires the operating system to support NTFS-formatted disks. (I.e., whatever component is chosen for the OS, must include a clause "`provides NTFS-support`"). It also needs a free USB port and costs 1500 DKK.

Often, the resource needs of a component can be satisfied in alternative ways. For example, connecting a display screen may use either a free VGA or HDMI port (but not both). Conversely, a component may provide a choice between two (or more) resources, only one of which can be utilized in any particular configuration: for example, a drive bay may accomodate either one 3.5" disk (magnetic), or two 2.5" ones (SSDs), but not both at the same time. For example,

```
component Small-TrueView:
  provides 22 inch-display, 768 lines-resolution;
  uses VGA-port | HDMI-port;
  uses monitor-space, 1000 DKK.
```

Here, alternatives are separated by vertical bars "`|`", while conjunctive needs or availabilities are separated by commas "`,`". The example also shows that this particular monitor provides a 22", 768p display. However since display sizes do not in general behave additively, we don't want, say, a requirement for a 32" monitor to be satisifed by buying two smaller ones, even if that may be a cheaper solution; this is accomplished by stipulating that all monitors compete for (1 unit of) monitor space, which we must ensure is initially provided, along with sufficient funds.

All available components, with their associated resource needs and offerings are collected in a *database*. A user may then *query* said database, by describing their resource needs and budget (including available physical space, etc.), and getting an answer consisting of a suggested system configuration, satisfying all the requirements (or an indication that no such configuration is possible).

System structure The CONFIGURAPTOR system is divided into four main modules:

- A *Parser*, which converts the textual representation of a database (as seen in the examples above) into an *intermediate* database, with essentially the same structure as the textual one.

- An *Elaborator*, which checks that the intermediate database is well formed, consolidates the offered and requested resources for each each component into a *resource profile*, and *desugars* alternatives into virtual resources and components, giving a *final database.*
- A *Solver* which tries to construct a satisfactory system configuration by selecting components from the final database in a goal-directed way.
- A *Main* driver program that processes command-line arguments and invokes the appropriate functionality from the other modules.

Your task

We have provided a suitable driver program. Thus, you will need to implement the other three modules, which are more precisely specified below. All are weighed roughly equally, but you should aim to solve each of them at least partially.

Question 1.1: Parser

The concrete grammar of a Configuraptor database is shown in Figure 1.

Lexical definitions

- A *name* is one or more *words*, separated by single hyphens (`-`) A word is one or more ASCII letters (`a`–`z` and `A`-`Z`) and/or digits (`0`–`9`), containing at least one letter. The total length of name (including any hyphens) may be at most 32 characters. Examples: "`AB`", "`9in-nail`"; non-examples: "`-foo`", "`bar-`", "`2-by-4`", "`ah--ha`".

  Keywords are *not* reserved, i.e., a keyword (e.g., `uses`) may in principle also be used as a name.
- A *num* is one or more ASCII digits. Its numeric value must be no more than 999999 ($10^6 - 1$). (Here, and elsewhere, don't worry about Int overflowing.)

Whitespace and comments All tokens may be surrounded by whitespace (spaces, tabs, and newlines). Whitespace is generally ignored, but may not occur inside multi-character tokens (keywords, names, or numbers). Some whitespace is necessary where a keyword, name, or number would otherwise run into an adjacent letter, digit, or hyphen.

Comments are written between curly braces, `{this is a comment}`. Any characters except curly braces may occur without restrictions inside a comment. Additionally, comments may be nested: `{a {{deeply} nested} comment}`. All (levels of) comments must be explicitly closed before the end of the input. Comments count as whitespace for token separation.

Keywords are case-insensitive. Semantically, so are resource and component names, but the parser should just pass them on as written.

$$
\begin{array}{rcl}
\mathit{Database} & ::= & \mathit{Declz} \\[6pt]
\mathit{Declz} & ::= & \epsilon \\
 & | & \mathit{Decl}\ \ \mathit{Declz} \\[6pt]
\mathit{Decl} & ::= & \text{‘}\texttt{resource}\text{’}\ \ \mathit{RNames}\ \ \text{‘}\texttt{.}\text{’} \\
 & | & \text{‘}\texttt{component}\text{’}\ \ \mathit{CName}\ \ \text{‘}\texttt{:}\text{’}\ \ \mathit{Clauses}\ \ \text{‘}\texttt{.}\text{’} \\[6pt]
\mathit{RNames} & ::= & \mathit{RName} \\
 & | & \mathit{RName}\ \ \text{‘}\texttt{,}\text{’}\ \ \mathit{RNames} \\[6pt]
\mathit{Clauses} & ::= & \mathit{Clause} \\
 & | & \mathit{Clause}\ \ \text{‘}\texttt{;}\text{’}\ \ \mathit{Clauses} \\[6pt]
\mathit{Clause} & ::= & \text{‘}\texttt{provides}\text{’}\ \ \mathit{RSpec} \\
 & | & \text{‘}\texttt{uses}\text{’}\ \ \mathit{RSpec} \\
 & | & \text{‘}\texttt{requires}\text{’}\ \ \mathit{RSpec} \\[6pt]
\mathit{RSpec} & ::= & \mathit{RName} \\
 & | & \mathit{num}\ \ \mathit{RSpec} \\
 & | & \mathit{RSpec}\ \ \text{‘}\texttt{,}\text{’}\ \ \mathit{RSpec} \\
 & | & \mathit{RSpec}\ \ \text{‘}\texttt{|}\text{’}\ \ \mathit{RSpec} \\
 & | & \text{‘}\texttt{(}\text{’}\ \ \mathit{RSpec}\ \ \text{‘}\texttt{)}\text{’} \\[6pt]
\mathit{RName} & ::= & \mathit{name} \\[6pt]
\mathit{CName} & ::= & \mathit{name}
\end{array}
$$

Figure 1: Concrete grammar of CONFIGURAPTOR databases

Disambiguation In resource specifications (*RSpec*), the implicit *scaling* operator (prefixing by a number) groups tighter than "`,`", which again groups tighter than "`|`", so that "`a | 4 b , c`" would parse as if parenthesized like "`a | ((4 b) , c)`". Further, "`,`" and "`|`" associate to the left, while scaling (necessarily) associates to the right, e.g., "`a, 3 b, 4 5 c`" parses like "`(a, 3 b), 4 (5 c)`".

The abstract syntax of the intermediate database is defined in the module `Absyn` (omitting the `derives (Eq, Show, Read)` for all datatypes):

```haskell
type IDB = ([RName], [IComp])

data IComp = IC CName [Clause]

type Clause = (CKind, RSpec)

data CKind = CKProvides | CKUses | CKRequires

data RSpec =
    RSRes RName
  | RSNum Int RSPec
  | RSAnd RSpec RSpec
  | RSOr RSpec RSpec

type RName = String
type CName = String

type ErrMsg = String  -- general type of human-readable messages
```

API Your Parser module should export one function, for parsing a full database:

```haskell
parseString :: String -> Either ErrMsg IDB
```

The function should parse a database into an `IDB`. All resources declared in the input (whether individually or jointly with others, and no matter where) should be collected into the `[RName]`, while component declarations should be collected as `[IComp]`, with the evident correspondence between concrete and abstract syntax.

If parsing fails, the function should instead return a suitable error message (which doesn't have to be specific). Only actual syntax errors should be reported at this stage; semantic errors (e.g., using an undeclared resource in a component, or declaring the same resource twice), will be detected during elaboration.

You must use either the `ReadP` or `Parsec` parser-combinator library (as supplied with LTS-14.1). 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`.

Question 1.2: Elaborator

The abstract syntax of the final database is roughly similar to the intermediate one, but simpler:

```
type DB = ([Resource], [(CName, RProf)])

type RProf = [(Resource, (Int, Int))]

newtype Resource = R RName
  deriving (Eq, Ord, Show, Read)
```

The database gives the complete list of resources (with their canonical capitalizations), and, for each component, its *resource profile*. Such a profile says, for each relevant resource, how many units of that resource the component contributes *in net* (i.e., what it provides minus what it uses, which could be positive, negative, or zero), as well as how many units it requires to be available in the complete configuration (i.e., after summing the net contributions of all components, including its own). If *both* of those numbers are zero, the resource is omitted from the profile. The resources (in both the complete list, and in individual profiles) are always listed in (lexicographically) sorted order.

The elaborator turns lists of clauses into a single resource profile. The order of resource specifications and how they are divided into clauses doesn't matter. For example, "`uses` *A*; `provides` *B*" elaborates to the same profile as "`provides` *B*; `uses` *A*"; likewise, "`uses` *A*, *B*" means the same as "`uses` *A*; `uses` *B*"; "`uses 1` *A*" means the the same as just "`uses` *A*"; and "`uses 3 (4` *A*`)`" is equivalent to "`uses 12` *A*". Analogously for `provides` and `requires`.

For example, the specification

```
component c: provides r2, 5 r1, r3; uses 2 r1; requires 4 r3, 7 r1; uses 1 r2.
```

corresponds to the following component profile in the final database:

```
("c", [(R "r1", (3,7)), (R "r3", (1, 4))]
```

Also, scaling a resource specification is semantically equivalent to repeating it the corresponding number of times. That is, "`uses 3` *A*" behaves essentially the same as "`uses` *A*`,`*A*`,`*A*", except that the former will generally be handled more efficiently, especially for large scaling factors and/or more complicated specifications *A*.

Note that tracking net resource usages alone sometimes oversimplifies matters. For example, a USB hub (like a power strip) converts a single port into multiple ones. But if we naively specified it as

```
component USB-hub4: uses 1 USB-port; provides 4 USB-port.
```

it would behave identically to

```
component USB-hub4: provides 3 USB-port.
```

which says that, to connect a USB device to the system, we *only* need a hub, but no actual USB port on the PC. To express that we cannot simply plug a hub into itself, we need to extend either of the above specifications with a clause "`requires USB-host`", a resource that would be provided only by the computer, so that the whole USB tree can be properly rooted.

Alternatives between resources (`|`) are handled by introducing *virtual* resources and components. A clause (or conjunct in a clause) "`provides` (A `|` B)" can be seen as a conceptual abbreviation for "`provides pAorB`", where the new virtual resource `pAorB` is only used by two virtual components that in turn provide the actual resources A and B:

```
component AorB1: uses pAorB; provides A.
component AorB2: uses pAorB; provides B.
```

The virtual components are not physical pieces of hardware; they merely correspond to a implicit selection of which resource to pick when a choice is offered.

The names `pAorB`, `AorB1`, and `AorB2` have no actual significance, and were simply chosen mnemonically. An actual implementation of the elaborator should just generate arbitrary fresh names, not necessarily meaningful, but guaranteed not to clash with user-provided names, or with each other. Also, it wouldn't first construct explicit textual or intermediate-database representations of the virtual components, but just calculate their resource profiles directly.

Note also that A and/or B could themselves be composite specifications, possibly involving further alternatives. e.g., A could actually be the "`2` (C`|`D)"; then we would need to also introduce a virtual resource `pCorD` with virtual components to turn a `pCorD` into a C or a D, and so on.

Ideally, when there's a direct choice between more than two alternatives, e.g., "`A | B | C`", whether parenthesized as "`(A | B) | C`" or "`A | (B | C)`", the elaborator would only introduce a single virtual resource (`pAorBorC`), with one virtual component for each of the three branches, rather than using nested binary choices. Explain in your report whether and (if so) how you implemented this optimization.

Note that there is a difference between the clauses "`provides 2 A | 2 B`" and "`provides 2 (A|B)`": the former says that the component only provides either two `A`'s or two `B`'s, while the latter can also provide one of each, if that's what's needed.

Analogously, we may use alternatives to signify that a request for resources may be satisfied in different ways. For example, "`uses` A`|`B" would become "`uses uAorB`", where

```
component 1AorB: uses A; provides uAorB.
component 2AorB: uses B; provides uAorB.
```

Note how the `uses` and `provides` clauses go in the other direction here. Again, the above are not real components, but merely reflect choices in how an alternative-resource need was actually satisfied.

Finally, a clause "`requires` A `|` B" becomes "`uses rAorB`" (*not* "`requires rAorB`", so as not to leave "stray" virtual resources lying around), with

```
component 1AorB: requires A; provides rAorB.
component 2AorB: requires B; provides rAorB.
```

That is, the virtual components provide an (exclusive, ephemeral) copy of the virtual resource, as long as one of the alternative underlying resources is already available.[1]

API Your elaborator module should export the following functions:

```
lookres :: [Resource] -> RName -> Either ErrMsg Resource
```

```
elaborate :: IDB -> Either ErrMsg DB
```

`lookres rs s` should look in the duplicate-free list of resources rs for the name `s`, and either return the canonically capitalized resource, or a suitable error message.

`elaborate` pdb should return a final database corresponding to the intermediate one. This involves the following checks and transformations:

1. All clauses for each component should be suitably combined into a single, flat resource profile for that component.
2. All resource names in resource profiles should be canonicalized to the capitalization used in the resource declaration. Uses of undeclared resources should be reported with a suitable error message (as a `Left`-tagged result, not by calling `error`!). It is not a requirement that resources are declared before being used. (Indeed, an `IDB` contains no information about the relative placement of resource and component declarations in the original input string.)
3. The elaborator should check that all resources are declared exactly once. (Multiple declarations differing in case are not allowed.) Also, it is illegal to declare a component and a resource with the same name.

   Multiple declarations of a component *are* allowed, with the meaning that all clauses are put together as if they were in a single declaration. E.g,

   ```
   component Foo: provides A, 2 B; requires C.
   { possibly other declarations here }
   component Foo: provides 3 A; uses 10 D.
   ```

   should give the same result as the single declaration

   ```
   component Foo: provides 4 A, 2 B; requires C; uses 10 D.
   ```

[1]This elaboration of `requires`-clauses is a bit too simplistic to correctly express sharing of multiple units of a resource, as in "`require` n $(A|B)$" where $n > 1$. However, doing it properly would require a significantly more complicated elaboration process, keeping closer track of when shared resources are reserved and released, so just implement it as described here.

The component names in the declarations must match exactly, including case. (Otherwise, such as if the second declaration above was instead for "`component foo ...`", an error should be reported.)

4. Alternative resource specifications are translated to plain resource profiles, by "generating" fresh virtual resources as well as corresponding components for each choice. The fresh names should be distinguished from user-provided names by starting with "#". The rest of the name can be just a unique number; it doesn't have to relate to actual resource names in the alternatives.

Note: you get points for correctly handling each of these aspects. It is strongly suggested that you prioritize the lower-numbered ones.

You should structure your elaborator using an appropriate reader–writer–state–error monad. In your report, explain how you exploit each of these components of the monad (or why not).

Question 1.3: Solver

We define a few more type synonyms:

```
type Goal = RProf

type Solution = [(CName, Int)]
```

A *goal* is simply a resource profile, i.e., a list of resources (real or virtual), for each one giving the number of units *available*, and *needed*. The *initial* goal will typically make available some amount of the "money" resource, and say that particular amounts of various functionality resources (RAM, disk, etc.) are needed. A goal is *solved* if, for each resource, the amount available is at least as large as the amount needed. Note that a solution will often provide *excess* resources: there may be some money left over; there may be more units of some resource than what was required; or the solution may provide some resources that were not asked for at all; those are all fine.

A (purported) *solution* is a list `[(CName, Int)]`, where for each entry (c, n), the integer n says how many of (real or virtual) component c to include. Each c should only occur once in the list, and all n's should be strictly positive.

Your module should export three functions:

```
combine :: RProf -> RProf -> RProf

verify :: DB -> Goal -> Solution -> Either ErrMsg RProf

solve :: DB -> Goal -> Int -> Either ErrMsg Solution
```

`combine` rp_1 rp_2 combines two resource profiles: for each resource, the net contribution is the *sum* of the two contributions, while the requirement is the *maximum* of the requirements. You may assume that rp_1 and rp_2 are well formed profiles (sorted, and no $(0, 0)$-entries), and you should ensure that the result is, too. For example,

```
combine [(R "r1", (3, 5)), (R "r3", (-2, 0)), (R "r4", (3,0))]
        [(R "r1", (2, 7)), (R "r2", (3, 4)), (R "r4", (-3,0))]
```

should return

```
[(R "r1", (5, 7)), (R "r2", (3, 4)), (R "r3", (-2, 0))]
```

`verify` *db* *g* *sol* checks whether *sol* is a well formed solution to initial goal *g* i.e., that combining *g* with the resource profiles of the components (with associated multiplicities) from *db*, as listed in the solution, results in a solved goal. If so, `verify` should return the resource profile of the complete system (including what's left over from resources provided in the initial goal); otherwise, it should return a suitable error message. You may assume that *db* and *g* are well formed, but you should not assume anything about *sol*.

`solve` *db* *g* *n* tries to solve goal *g* using as few components (including any virtual ones) from *db* as possible, and at most *n*. If this succeeds, it return a possible solution. Naturally, this solution should `verify` successfully. You may assume that *db* and *g* are well formed, and $n \geq 0$.

If the solver fails, the error message should start with either `"Impossible"`, which means that the goal cannot possibly be solved in the given database, no matter how many components we allow; or `"Exhausted"`, meaning that there is definitely no solution using *n* or fewer components, but there *may* be one with more components. After that, you may *optionally* give additional, human-readable information that might be useful in diagnosing the problem.

If a solution is found, it does not have to be the cheapest, nor provide a maximal number of excess resources; it merely has to be correct and not use more components than necessary. To find better solutions, we may re-run the solver with a tighter initial goal (a lower budget and/or higher requirements), possibly combined with a higher bound on the number of components to consider.

Efficiency is *not* a significant concern. However, your solver should not get bogged down in exploring pointless parts of the search space. You should organize it as a goal-directed search (like in logic programming), where you start from the initial goal, and consider adding *relevant* components one by one until the goal is solved. In particular, the solver should never consider adding a component (whether real or virtual) unless that component actually addresses some unsatisfied constraint in the current goal, i.e., the amount needed of some resource strictly exceeds the amount currently available, and the component contributes at least one unit of that resource.

Use monads where you deem appropriate. Justify your choice(s) in the report.

If you cannot get the solver to work in full generality, consider whether you can implement a limited version, e.g., one that only works under some assumptions about the database and/or initial goal, or one that is not guaranteed to find minimal solutions.

Main program

This is provided in the handout. Usage is:

```
configuraptor (@DB.cr | %DB.hcr) ... (-n NUM | -p NUM RES | -r NUM RES) ...
```

An argument `@`*DB*`.cr` reads in the database in file *DB*`.cr` (the extension `.cr` can actually be anything). Similarly `%`*DB*`.hcr` reads the database from an already parsed Haskell value of type IDB. The contents of all databases are concatenated (resource and component lists separately) between parsing and elaboration.

An argument `-n` *NUM* (where *NUM* is a non-negative integer) sets the maximum number of components to consider for the solver. (If more than one `-n` is given, the *last* one takes effect; if none are present, the maximum defaults to 10.)

An argument `-p` *NUM* *RES* says that, initially, *NUM* units of *RES* are available. *NUM* should be non-negative, and *RES* a resource declared in one of the databases. Analogously, `-r` *NUM* *RES* says that the initial requirement is for the specified number of units of a resource. Both `-p` and `-r` may be repeated, for specifying multiple resources.

Using multiple databases, you can have general specifications of all components in a fixed database and a detailed specification of the goal requirements in another, e.g. `goal.cr` could contain:

```
resource system.

component DreamPC:
  provides system; requires 16 GB-RAM, 27 inch-display, Linux, ... .
```

Then you can invoke the main program as

```
configuraptor @components.cr @goal.cr -p 10000 DKK -r 1 system -n 20
```

You may use the command-line program for informal experimentation or integration testing, but you should still do proper, automated testing of your individual modules. You are not expected to test the main program, nor to employ it it in your tests. (In fact, you are free to ignore it entirely if you don't find it useful.)

General instructions

You should use the skeleton files provided. *Do not* modify the types in the abstract syntax or APIs, as this will likely prevent your solution from working with our automated tests.

For each of the three modules, there is both a file `Module.hs`, which exports only the required API, and `ModuleImpl.hs`, which exports everything. You should place your code in the latter only. Then, if you want to to do white-box testing of some of your internal functions, you may import them from the relevant implementation modules.

Your implementation modules should not import directly from each other. If you have a non-trivial type definition or function that you want to share between two or more modules, place it in `Utils.hs` and import it from both.

You should not modify the `package.yaml` file, except possibly in the testing section. That is, your main code should import only from the allowed packages. For *testing*, you may use anything from LTS-14.1 (e.g., a different testing framework) if necessary. As usual, your tests should be reproducible by running `stack test`. If any special instructions are needed, give them in the report.

We provide some sample database files in the `examples/` directory (both `.cr` and `.hcr` versions). Note that these *do not* exercise all the relevant functionality of the parser, elaborator, and solver, so you should also test with your own (probably smaller, and targeted) sample databases. Do consider whether any parts of the system would be particularly suitable for property-based testing, and if relevant, implement some such tests as well.

Question 2: Shared Optimism

The task in this question is to implement a server that maintains a shared state in the form of a key/value map and can execute operations on this shared state using a simplified version of what is called *optimistic concurrency control.*

General comments

This question consists of two sub-questions: Question 2.1 about implementing an API for starting an optimistic server and for working with operations, 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. Note that Question 2.2 can be solved with a simple partial (or even without) implementation of the `optimistic` module from Question 2.1.

In Appendix A you can find an example on 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 19, that suggests topics for your report.

Terminology

An *optimistic server* maintains a shared state in the form of a *key/value* map. An *operation* is a function that reads a *view* of the state, and computes a result and a potential *change* to the shared state. A view is a sub-map of the state. Similarly a change to a state is a map with the insertions or updates to be performed. Thus, we use the following Erlang representation of views, changes, and operations:

```
-type view() :: map().
-type change() :: map().
-type operation() :: fun((view()) -> {any(), change()}).
```

The *read set* of an operation is the keys of its view, and the *write set* is the keys of its change. The *dependency set* is the union of the read and write sets. Two operations are in *conflict* if one of their write sets overlaps the other's dependency set. We define the *dirty set* of an operation to be the keys that are written by other concurrent operations.

We want to execute operations *concurrently* but *isolated.* That is, if we have two concurrent operations, op_1 and op_2, that are not in conflict, then the end state of running those two operations should be the same as running them sequentially in some order.

To perform an operation on the shared state, you must first start the operation (which creates an intermediate map containing the view of the shared state), and then, when the operation has completed, *commit* the changes to the shared state. An operation can either be successfully committed (and updating the state of the server with the changes) or it can be *aborted* (and the state is not updated).

There are three ways that an operation, *op*, can be aborted:

- The operation is explicitly aborted.
- The operation fails (throws an exception, exits, generate an error, or return a value of the wrong type).
- Another operation, *op′*, that conflicts with *op* is executed concurrently with *op*, and *op′* commits its updated state before *op*. In other words, the intersection of the dependency set and the dirty set is non-empty.

  Multiple operations can be executed concurrently. However, if the keys in an operation's dependency set are updated, the operation is operating on a now obsolete state and should be stopped. Queries of the result on such an operation must be answered with `aborted`.

Question 2.1: The `optimistic` module

Implement an Erlang module `optimistic` with the following API, where `S` denotes a process ID of an optimistic server and `OR` denotes a reference to an operation (you decide the representation). All functions are allowed to return `{error, Reason}` if some unspeficied error occurred.

- `start(State)` for starting an optimistic server with `State` as the initial shared state. `State` is an Erlang `map`. Returns `{ok, S}` on success.
- `stop(S)` for stopping an optimistic server; this includes aborting all ongoing operations. Returns `{ok, State}` after all operations have been aborted, where `State` is the current shared state of `S`.
- `reset(S, State)` for resetting the shared state of the optimistic server to `State`; this includes aborting all ongoing operations. Returns `ok`.
- `delete(S, Keys)` for deleting `Keys` from the shared state; this includes updating the dirty set of ongoing operations (which may cause them to abort). Where `Keys` is a list of keys, and non-existing keys are ignored. Returns `ok`.
- `operation(S, Reads, OFun)` for starting a new operation on `S`. Returns `{ok, OR}`. Where `Reads` is a list of keys to be read from the shared state and `OFun` is a function that takes a `View` corresponding to `Reads` (non-existing keys are ignored) and returns a pair `{Res, Change}`.

  The server executes `OFun(View)` with a view of the shared state corresponding to `Reads`. If `OFun` fails, the operation is aborted.

  The keys in `Reads` and `Change` are added to the dependency set of `OR` and the keys in `Change` are added to the write set of `OR`.

  Remember that it is up to you to decide how to represent `OR`.

- `commit(OR)` for committing an operation. Waits for the operation to complete. Updates the shared state of `S` with the changes from the operation, if `OR` is not aborted and the intersection between the dirty set and the dependency set is empty. The dirty set of other ongoing operations are updated with the write set of `OR` (which may cause them to be aborted). Return either `{ok, Res}` if the operation returned `{Res, Change}`, or `aborted` if the operation is aborted.

  Note that `commit/1` is idempotent, meaning that multiple application of `commit/1` to the same argument should give the same result (as long as the optimistic server that `OR` originates from is running).

- `abort(OR)` for aborting the operation `OR`. Returns `aborted` if `OR` is not committed or if it is already aborted; otherwise returns `too_late`.

Your library must be robust against erroneous operations. In particular, your solution should be able to deal with slow or non-terminating operation functions.

Concurrent execution of operations

Operations can only affect other running operations and the shared state when they are committed. To exploit this inherent concurrency, each operation should run in its own process. That is, when an optimistic server is asked to start an operation, it should spawn a new helper operation-process for maintaining the intermediate state of the operation.

Note that your implementation should stop all unused processes and should not keep operation-processes around longer than necessary. Also, processes waiting for answers from aborted operations should be answered with `aborted` as quickly as possible. Explain in your report how you solve these challenges.

How to get started

Try to start by implementing a version of the `optimistic` library that only runs a single operation at a time, perhaps even without spawning a separate operation-process. When this works, you can extend it to handle concurrent operations.

Question 2.2: Testing `optimistic`

Make a module `test_optimistic` that uses `eqc` QuickCheck for testing an `optimistic` module. We evaluate your tests with various versions of the `optimistic` module that contains 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. In this question, we assume that all values in key/value maps are integers, we don't assume anything about the keys in general.

Your `test_optimistic` module should contain the following functions:

- `mkopr(Opr, Args)` a helper function that generates an operation function from the atom `Opr` and the arguments `Args`. You decide which atoms are valid for `Opr` and what `Args` should be. We will not call this function directly, only through symbolic calls generated by your generators.
- A QuickCheck generator `good_opr/0` that generates an operation in the form of a tuple with three elements: the first element is a list of keys corresponding to the read set, the second element is a list of keys corresponding to the write set, and the third element is a symbolic call which can be evaluated to an operation function that works on a view corresponding to the keys in the read set.

  For instance, five samples from this generator could be:

  ```
  {[d], [d], {call,test_optimistic,mkopr,[incr,d]}}
  {[c,d], [c,d], {call,test_optimistic,mkopr,[swap,{c,d}]}}
  {[f], [f], {call,test_optimistic,mkopr,[incr,f]}}
  {[e,a], [e,a], {call,test_optimistic,mkopr,[swap,{e,a}]}}
  {[a,d,f], [a,d,f],
   {call,test_optimistic,mkopr,
         [seq,
          [{call,test_optimistic,mkopr,[swap,{d,f}]},
           {call,test_optimistic,mkopr,[incr,a]}]]}}
  ```

  Again, remember that the arguments to `mkopr/2` are up to you to decide. The given samples are just for inspiration.
- A parameterised QuickCheck property `prop_opr_accurate(OprGen)` that takes a generator that generates tuples in the same form as `good_opr/0`. That is tuples with three elements: the first element is a list of keys corresponding to the read set, the second element is a list of keys corresponding to the write set, and the third element is a symbolic call which can be evaluated to an operation function that works on a view corresponding to the keys in the read set. Hence `good_opr/0` could be an argument to test this property with, but we might test it with other generators as well.

  The property should check that a generated tuple `{Reads, Writes, SymOpr}` is *accurate*. That is, `SymOpr` can be evaluated to a function, `Fun`, and when `Fun` is applied on a view, corresponding to `Reads`, it gives a result of the right form: a pair where the second element is a change that corresponds to `Writes`.

Remember that Fun may fail (raise an exception of any class or return a result of the wrong form), and your property should be robust against that. You may assume that Fun terminates though.

- A property `prop_server_commit/0` that checks that starting an operation and then committing it on an optimistic server, gives the right result and that the shared state is updated correctly. This property should test the following functions from the `optimistic` module: `start/1`, `operation/3`, `commit/1`, and perhaps also `stop/1`.

  In your report, make sure that you document whether your implementation of this property deals with operations that might fail or not.

- A property `prop_isolated/0` that checks that running two non-conflicting operations concurrently gives the same result as running them sequentially in some order.

- A `test_all/0` function that runs all your tests that only depends on the specified `optimistic` 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 `optimistic` API should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of the `optimistic` 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 `optimistic`.

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.
- How you deal with the management of helper operation-processes. Including your strategy for aborting helper processes as early as possible and 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.

Appendix A: Example use of optimistic

The following example demonstrates how to use the optimistic API. The function incr/2 increments the value for a given key. The function updates/3 spawns two processes that both increment the value for a given key N times. The given keys can either be the same key, conflicting_updates/1, in which case some operations are aborted (with high probability); or the keys can be different, nonconflicting_updates/1, in which case none of the operations should be aborted. The functions conflicting_updates/1 and nonconflicting_updates/1 both return a tuple with four elements; you should think about the relation of these four numbers.

```erlang
-module(counter).
-export([conflicting_updates/1, nonconflicting_updates/1]).

incr(S, C) ->
    {ok, OR} = optimistic:operation(S, [C],
                              fun (View) ->
                                      V = maps:get(C, View),
                                      {success, #{C => V+1}}
                              end),
    optimistic:commit(OR).

updates(K1, K2, N) ->
    {ok, S} = optimistic:start(#{K1 => 0, K2 => 0}),
    Me = self(),
    P1 = spawn(fun() -> Me ! {self(),
                              lists:map(fun(_) -> incr(S, K1) end,
                                        lists:seq(1, N))}
               end),
    P2 = spawn(fun() -> Me ! {self(),
                              lists:map(fun(_) -> incr(S, K2) end,
                                        lists:seq(1, N))}
               end),
    L2 = receive {P2, Res2} -> Res2 end,
    L1 = receive {P1, Res1} -> Res1 end,
    {ok, #{K1 := C1, K2 := C2}} = optimistic:stop(S),
    {C1, length([ R || {ok, R} <- L1]),
     C2, length([ R || {ok, R} <- L2])}.

conflicting_updates(N) ->
    updates(crash, crash, N).

nonconflicting_updates(N) ->
    updates(x, y, N).
```