Take-home Exam in Advanced Programming

Deadline: Friday, November 9, 16:00

Version 1.0a

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2018. 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 two 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 `handin.zip`, archiving one directory called `handin` 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 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* 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: The `appm` package manager

Introduction

Modern software distributions often consist of a number of largely independent *packages*, which may be installed independently of each other. However, often some package (say, an application program) may depend on some sufficiently recent other package(s) (say, libraries) also being installed, and those packages may themselves have dependencies, etc. Conversely, some packages may *conflict* with specific other packages, typically because they contain incompatible files of the same name.

All available packages are stored in a *repository*, which supplies a *metadata database* that lists the packages together with their dependency constraints. A *package manager*, when asked to install a specific package, must first try to resolve all the relevant dependencies (positive and negative) pertaining to the package. Examples of such managers include APT and YUM/DNF for Linux, Homebrew for macOS, or Chocolatey for Windows.

In this task you will be implementing part of a simple package manager, `appm`, focusing on the dependency resolver ("solver" for short). A small `appm` database could look like this (available as `examples/intro.db` in the handout):

```
package {
  name foo;
  version 2.3;
  description "The foo application";
  requires bar >= 1.0
}

package {
  name bar;
  version 1.0;
  description "The bar library"
}
```

```
package {
  name bar;
  version 2.1;
  description "The bar library, new API";
  conflicts baz < 3.4, baz >= 5.0.3
}

package {
  name baz;
  version 6.1.2;
}
```

Given such a database, and a user request to install a particular package (say, `foo`), the solver attempts to construct a complete list of package names and corresponding versions to be installed. (For simplicity, we will assume that no packages have already been installed prior to the request, so the solver is starting from a completely blank slate.)

The solver should satisfy a number of correctness criteria, which we may group into three categories:

Consistency Any solution constructed by the solver must be well formed:

a. All packages (with the indicated versions) are actually available in the database.

b. Any package name may only occur once in the list; in particular, it is not possible to install two different versions of the same package simultaneously.

c. The package requested by the user is in the list.

d. For any package in the list, all the packages it requires are also in the list.

e. For any package p in the list, all other packages p' in the list satisfy the version constraints of p. That is, if p requires p', then p' is one of the acceptable versions; and if p conflicts with p', then p' is not one of the conflicting versions.

The order of packages in the list is *not* significant.

Quality Often, there will be several different solutions satisfying all the above requirements. In such cases, the following additional criteria apply:

f. Only actually required packages should be installed. That is, it should not be possible to remove any package from the list and still have a consistent one.

g. Newer versions of all packages are preferred. That is, it should not be possible to replace any package on the list with a newer version, and still have a consistent list.

Note that even with these additional requirements, solutions are not necessarily unique. In such cases, it doesn't matter which one is found.

Effectivity Finally, the solver itself should satisfy some functionality requirements:

h. For any well-formed database and request, the solver should terminate with either a solution satisfying the above criteria, or a correct determination that no such solution is possible (e.g., due to a dependency not being available in the repository, or two dependencies being in conflict with each other.) In the latter case, it is *not* a requirement that a specific cause of the failure be explicitly identified.

i. The efficiency of the solver is in general not a concern, so even a worst-case running time exponential in the size of the database is acceptable. (For extra credit, and Turing Award, you may try to do better.) However, the presence of *irrelevant* packages (i.e., those which are not direct or indirect dependencies of the initial request) in the database should not *drastically* affect the running time. That is, the solver should still be usable, even with tens or hundreds of irrelevant database entries added.)

Thus, with the database above, a user request to install package `foo` would succeed, and install `foo` (version 2.3) and `bar` (version 2.1). Had the specification of `foo` also included the clause "`requires baz`", the solution would instead have been to install `foo` 2.3, `baz` 6.1.2, and `bar` 1.0

Implementing `appm`

Versions

In `appm`, a *version* is a sequence of one or more natural numbers (each written with one or more decimal digits), separated by periods ("`.`"), for example "`16`" or "`4.7.2`". There is no

limit on the length of the sequence, but each number must be in the range 0 through 999 999. Initial zeros in a number are not significant; that is, the version "`4.0.1`" is considered exactly the same as "`04.00.001`" (but different from both "`4.1`" and "`4.0.1.0`").

Additionally, each number in the sequence may be followed by a *suffix* of 1 to 4 lowercase English letters ('a' through 'z'); e.g., "`4.3b.22ff`" is a well-formed version.

Versions are ordered lexicographically: if the initial numbers are different, they determine the order; otherwise, the two remaining sequences are compared, with an empty sequence considered less than any other. Thus, we have, e.g., that "`3.5`" is less than "`4.1`", and "`4.0.1.3`" is less than "`4.1.2`"; in particular, "`3.4.2`" is technically considered strictly less than "`3.4.2.0`".

When comparing individual numbers with suffixes, the numeric part is the most significant; e.g. "`3.4z`" is less than "`3.5a`". When the numeric parts are equal, the suffixes are ordered alphabetically, with shorter suffixes considered smaller. Thus, "`802.11`" is less than "`802.11n`", which is less than "`802.11ax`", which again is less than "`802.11bb`".

"`0`" is the smallest possible version, while all well-formed versions are strictly less than "`1000000`" (which is not itself a legal version, but may still serve as an exclusive upper bound with respect to the above ordering).

Type structure

The core `appm` types are given in the file `Defs.hs`, shown in Figure 1.

Versions are represented in the obvious way, but note that the Haskell types don't enforce the various well-formedness constraints, such as the limits on version numbers.

A *constraint* for a particular package name consists of a boolean flag indicating whether the package is *required*, together with an interval specifying the *allowed* versions of the package. For simplicity, version intervals are always semi-open, with the lower bound being inclusive, and the upper bound exclusive. Thus, for the interval to be non-empty, the upper bound must be strictly larger than the lower.

A constraint set `Constrs` is then a list of such constraints, in no particular order. Any package should be mentioned *at most once* in the list. (Not being mentioned is equivalent to being mentioned with the constraint `(False, minV, maxV)`.)

A package specification, `Pkg`, consists of the package name, version, description (in natural language), and a collection of constraints on *other* packages that must be satisfied for this package to be installable.

Finally, a solution, `Sol`, is just a list of package names and versions to be installed, in no particular order.

Question 1.1: Utility functions

The module `Utils.hs` contains utility functions relevant to more than one component of `appm`. It should contain at least the following functionality:

The type `Version` should be made an instance of class `Ord`, according to the above definition of version ordering. You may implement either `(<=)` or `compare`, since either one has a

```haskell
module Defs where

type ErrMsg = String -- for all human-readable error messages

newtype PName = P String
  deriving (Eq, Ord, Show, Read)

data VNum = VN Int String
  deriving (Eq, Show, Read)

newtype Version = V [VNum]
  deriving (Eq, Show, Read)

minV, maxV :: Version
minV = V [VN 0 ""]        -- inclusive lower bound
maxV = V [VN 1000000 ""]  -- exclusive upper bound

type PConstr = (Bool, Version, Version) -- req'd; allowed interval [lo,hi)
type Constrs = [(PName, PConstr)]

data Pkg = Pkg {name :: PName,
                ver :: Version,
                desc :: String,
                deps :: Constrs}
  deriving (Eq, Show, Read)

newtype Database = DB [Pkg]
  deriving (Eq, Show, Read)

type Sol = [(PName, Version)]
```

Figure 1: Listing of `Defs.hs`

$$
\begin{array}{rcl}
\mathit{Database} & ::= & \epsilon \\
& | & \mathit{Package}\ \ \mathit{Database} \\
\mathit{Package} & ::= & \text{`package'}\ \text{`\{'}\ \mathit{Clauses}\ \text{`\}'} \\
\mathit{Clauses} & ::= & \epsilon \\
& | & \mathit{Clause} \\
& | & \mathit{Clauses}\ \text{`;'}\ \mathit{Clauses} \\
\mathit{Clause} & ::= & \text{`name'}\ \mathit{PName} \\
& | & \text{`version'}\ \mathit{Version} \\
& | & \text{`description'}\ \mathit{String} \\
& | & \text{`requires'}\ \mathit{PList} \\
& | & \text{`conflicts'}\ \mathit{PList} \\
\mathit{PList} & ::= & \mathit{PItem} \\
& | & \mathit{PList}\ \text{`,'}\ \mathit{PItem} \\
\mathit{PItem} & ::= & \mathit{PName} \\
& | & \mathit{PName}\ \text{`>='}\ \mathit{Version} \\
& | & \mathit{PName}\ \text{`<'}\ \mathit{Version} \\
\mathit{Version} & ::= & \text{(see text)} \\
\mathit{PName} & ::= & \text{(see text)} \\
\mathit{String} & ::= & \text{(see text)}
\end{array}
$$

Figure 2: `appm` database grammar

default implementation in terms of the other.

Second, for the solver (and possibly also useful for the parser), a central supporting operation is the *merging* of two constraint sets:

```
merge :: Constrs -> Constrs -> Maybe Constrs
```

`merge` c_1 c_2 should conjoin the constraints, producing a new, well-formed constraint list. If the resulting set would be *inherently unsatisfiable*, i.e., contain a constraint saying that some package is both required and must have a version number belonging to an empty interval, `merge` should return `Nothing`.

Question 1.2: Parsing `appm` databases

A package database has the concrete grammar shown in Figure 2.

A package name, *PName* , can be *simple* or *general*. A simple name consists of an ASCII letter, followed by one or more letters, digits, or hyphens ("-"). However, the final character

must not be a hyphen, nor may the name contain two or more consecutive hyphens. For example, "`foo`" or "`a-bb8-c`" are valid package names, while "`bar-`", "`-baz`", or "`ab--ba`" are not. Case is significant, so "`foo`" and "`Foo`" are considered to be *different* package names.

Alternatively, a general package name is any *non-empty* quoted *String*, as specified below.

Keywords ("`package`", "`name`", etc.) are *not* case sensitive, so "`Package { nAME foo }`" is a well-formed package specification. The terminals in the grammar may be separated by arbitrary whitespace (space, tab, newline). Some whitespace is required between a keyword and an immediately following alphanumeric character or hyphen.

Comments, like in Haskell, are introduced by "`--`" (two hyphens) and last until the end of the line (or the end of the input, if that comes first). A comment counts as whitespace for the purpose of token separation.

A *String* is any sequence of ASCII characters (potentially including newlines, etc.) enclosed between double quotes (""). To include a single double-quote character in the string, it must be doubled, e.g., "`""a""b""`" is a string of 3 characters. Comments are not recognized inside strings (i.e., the sequence "`--`" is not treated specially).

When building the AST after parsing, for each package we also check some semantic well-formedness constraints that cannot easily (or at all) be expressed with a context-free grammar:

- The clause list must contain *exactly one* "`name`" clause.
- The clause list must contain *at most one* "`version`" clause. A missing version is equivalent to "`version 1`".
- The clause list must contain *at most one* "`description`" clause. A missing description is equivalent to "`description ""`".
- The clause list may contain zero or more "`requires`" and/or "`conflicts`" clauses, in any order. A clause with more than one item, e.g, "`requires foo, bar >= 5.0`" is equivalent to a sequence of one-item clauses, i.e., "`requires foo; requires bar >= 5.0`".

  Directly self-referential constraints, such as "`package {name foo; version 2; requires foo >= 1}`" are *not* allowed, as they are either trivially satisfied (like in example) or trivially unsatisfiable (like in the example, if the last clause had instead said, "`>= 3`").

  Clauses are interpreted *conjunctively* (i.e., all must be satisfied) when determining which versions of a package they allow. This may cause weaker constraints to be silently subsumed, e.g. "`requires foo >= 4, foo >= 6`" is equivalent to just "`requires foo >= 6`". Likewise, "`conflicts bar < 4; conflicts bar < 6`" is equivalent to just "`conflicts bar < 4`". Thus, all clauses together determine a single interval of allowed versions for a package.

  Note that there is an important difference between "`requires foo >= 3.14`" and `conflicts foo < 3.14`: the latter constraint is satisfied if "`foo`" is not installed at all, but the former is not.

The constraints in the requires/conflicts clauses for a single package must not be *inherently contradictory*. For example, "`requires foo >= 5; requires foo < 2`" clearly cannot be satisfied, no matter what versions of `foo` (if any) are available in the database. On the other hand "`conflicts foo < 3.4.4, foo >= 3.4.2`" is equivalent to simply "`conflicts foo`", i.e., the package is incompatible with *all* versions of `foo`.

Any violation of the above semantic requirements in a package specification should be reported as an error. You may choose to parse all remaining package specification first (which might instead fail with a syntax error in a later package), or to report semantic errors immediately.

For example, parsing the example database from the introduction should succeed with the following Haskell value of type `Database`:

```
DB [Pkg {name = P "foo", ver = V [VN 2 "",VN 3 ""],
         desc = "The foo application",
         deps = [(P "bar",(True,V [VN 1 "",VN 0 ""],V [VN 1000000 ""]))]},
    Pkg {name = P "bar", ver = V [VN 1 "",VN 0 ""],
         desc = "The bar library", deps = []},
    Pkg {name = P "bar", ver = V [VN 2 "",VN 1 ""],
         desc = "The bar library, new API",
         deps = [(P "baz",(False,V [VN 3 "",VN 4 ""],V [VN 5 "",VN 0 "",VN 3 ""]))]},
    Pkg {name = P "baz", ver = V [VN 6 "",VN 1 "",VN 2 ""], desc = "", deps = []}]
```

Your parser should export the following functions:

```
parseVersion :: String -> Either ErrMsg Version
parseDatabase :: String -> Either ErrMsg Database
```

You may use either ReadP or Parsec. (There are no restrictions on which Parsec submodules you may use, but you'll probably not find any of the previously disallowed submodules relevant for this task anyway.)

Question 1.3: Solving `appm` constraints

Normalization The first task of the dependency resolver is to check that the database is actually consistent. The parser normally ensures that each individual package is well formed and has no inherently contradictory constraints, but it does not check the database as a whole. For this, we have the function:

```
normalize :: Database -> Either String Database
```

In general, a database may have been constructed as the concatenation of the databases for several repositories, where some packages may be present in multiple repositories. Thus, `normalize` *db* should check that if *db* contains multiple specifications for a given package name and version, that their description strings are identical, and that their overall

dependencies are semantically equivalent, i.e., that they impose the same net constraints on other packages. You may assume that each individual package record is well-formed.

`normalize` should also order the output so that newer versions of a package come before older in the package list, to simplify the task of the solver. (The relative positions of packages with different names do not matter.)

Constraint solving The solver itself is structured as a backtracking search for well-formed solutions that satisfy a collection of constraints. It should export two functions:

```
solve :: Database -> Constrs -> Sol -> [Sol]

install :: Database -> PName -> Maybe Sol
```

The general function `solve` *db c s* returns all ways to extend a partial solution *s* with additional packages, so that it satisfies all the constraints in *c*. The found solutions are ordered with the best ones first.

`install` *db p* is the top-level function that tries to install the package *p*, and returns an installation list achieving this, if possible; if there's no way to correctly install *p* given the available packages and their constraints in *db*, `install` should return `Nothing`.

In the report, *briefly* explain how/why your solver satisfies properties (a)–(i) in the Introduction (or, if you know that it doesn't satisfy one or more of them, detail why not).

Question 1.4: Testing `appm` properties

Most of the correctness criteria for the solver from the Introduction are actually well suited for property-based testing. In particular, we can define a type of properties that check whether some possible output from `install` (a solution or nothing) is correct for the given input (a database and a package name):

```
type InstallProp = Database -> PName -> Maybe Sol -> Bool
```

For example, a property checking that any successful output from `install` is a *non-empty* list (a weaker version of criterion (c)) could be written as follows:

```
install_c' :: InstallProp
install_c' _db _p Nothing = True
install_c' _db _p (Just []) = False
install_c' _db _p (Just _) = True
```

First, implement (as many as you can of) properties (a), (b), (c), and (d). You may assume that your functions will only be invoked on a normalized database, but you shouldn't assume anything about the package name or the purported solution. These properties should be expressed independently of any particular solver or testing framework, and should be defined in file `tests/QC/Properties.hs`.

Second, use the above properties to actually QuickCheck your dependency resolver (i.e., the function `install` from the `Solver` module). Those tests should be in `tests/QC/Main.hs`. (If you have trouble with auto-generating suitable databases, try to at least run the tests on one or more *interesting*, hand-constructed ones.) Do consider shrinking any counter-examples you find.

It is important that your tests are meaningful for *any* (correctly typed) solver. Thus, you should only import from the `Defs`, `Utils`, and `Solver` modules, where the last one might not be your own. And in particular, you should not assume that the packages in a solution are listed in any particular order.

In your assessment, you should also briefly assess the quality of your property-based tests; that is, give an estimate of how well they fulfill the intended purpose of testing arbitrary (possibly buggy) implementations of the dependency resolver. As always, be sure to explain what your assessment is based on.

(You are welcome to formalize and QuickCheck other correctness criteria from the Introduction, and/or additional relevant properties of your own devising (including properties of the parser); but those would count towards supporting your overall code assessment, rather than specifically towards this sub-question.)

Driver program

To see your completed implementation in action, you may build the supplied `Main.hs`. This contains a command-line tool which will read and parse an `appm` database from a file, normalize it, and then try to install a named package. If successful, it'll print out the resulting installation list.

Question 2: Earls of Ravnica

This question is about implementing part of the engine for a multi-user rogue-like game taking place in the vast cityscape of Ravnica. We shall model the mazes of streets, the patchwork of grand halls, decrepit slums, ancient ruins and towering Gothic spires as connected *districts*.

Districts can be *connected* to other districts via one-directional *actions*. If a district P is connected to the district Q by at least one action we say that Q is a *neighbour* to P (but P is not a neighbour to Q unless there is a connection from Q to P). A district can be connected to itself.

A district can be in a number of main states: under configuration, active, and shutting down. In addition to these main states you may also have a number of intermediate states.

We call a set of districts a *territory*.

A *creature* can be in a district and can be moved from a district to a neighbour by taking an action. A creature consists of a pair of a reference and a map of stats. The reference should uniquely identify the creature, and each district should ensure that references are unique by disallowing creatures to enter if there is already a creature with a given reference in the district. The map of stats is used for modelling various attributes of the creature, such as strength and mental healthiness.

General comments

This question consists of two sub-questions: Question 2.1 about implementing an API for working with districts and creatures and Question 2.2 about writing QuickCheck tests against this API. Question 2.1 counts for 80% and Question 2.2 counts for 20% of this question.

In Appendix A you can find some examples of how to use the API.

Remember that is possible to make a partial implementation of the API that does not support all features. There is a section at the end of this question that suggests topics for your report, which hints at what features are considered more difficult.

Question 2.1: The `district` module

Implement an Erlang module `district` with the following API, where districts represented by process IDs, and creatures are represented by a pair of a reference and a map for stats:

- `create(Desc)` for creating a new district with a description given as a string `Desc`. Returns `{ok, District}` on success or `{error, Reason}` if some error occurred. A newly created district starts as under configuration.
- `get_description(District)` returns the description of the district. On success it returns `{ok, Desc}` or `{error, Reason}` if some error occurred. This function should work in all states.

- `connect(From, Action, To)` makes a connection from the district `From` to the district `To`, when the action `Action`, an atom, is taken. If `From` is under configuration and `Action` is not used for another connection in `From`, then the result is `ok`; otherwise the result is `{error, Reason}`.

- `activate(District)` tries to activate a district. When a district is under configuration and it is activated, it tries activate all of its neighbours. When all neighbours are either active or in the intermediate state under activation, the function returns `active`.

  If the district is already active the function returns `active`. If the district is under activation (that is, it is waiting for its neighbours to become active) the function returns `under_activation`. If the district is shutting down or it is not possible to activate all neighbours the function returns `impossible`.

- `options(District)` returns `{ok, Actions}` where `Actions` is a list of possible actions in `District`, if `District` is under configuration or active. If `District` is shutting down the function should return the atom `none`.

- `enter(District, Creature)` puts the creature `Creature` in the district `District`. Returns ok if the district is active, thus the creature is in the district; otherwise returns `{error, Reason}`. It is an error to put two creatures with the same reference in the district.

  Note that, there are no global oversight of creatures. Thus, two creatures with the same reference (thus the same creature) could be in two different districts. Likewise, a creature can be in zero districts before it is entered into a district.

- `take_action(From, CRef, Action)` moves the creature specified by `CRef` to the district `To`, if there is a connection from `From` to `To` via the action `Action`, both `From` and `To` are active, the creature is in the district `From` and the creature can be moved to `To`. Returns {`ok`, `To`} if the move succeeds, thus the creature is in `To` and no longer in `From`; otherwise returns `{error, Reason}` (and the creature stays at `From`). It is an error to put two creatures with the same reference in the same district.

- `shutdown(District, NextPlane)` shuts down a district. Before a district is shut down it sends a message `{shutting_down, District, Creatures}` to the pid `NextPlane`, where `Creatures` is a list of creatures currently in the district (even if there are no creatures in the district). Then it shuts down all its neighbours. When all neighbours are either shut down or is shutting down it returns `ok` and the process terminates.

- `trigger(District, Trigger)` for registering the trigger `Trigger` on `District`. The trigger is called when a creature is entering or leaving the district.

  The trigger is called with three arguments: `Event`, `Creature` and `Creatures`. Where `Event` is one of the atoms `entering` or `leaving`, `Creature` is the creature entering or leaving and `Creatures` is a list of all creatures in the district (excluding the creature entering or leaving). The trigger should return a pair `{Creature1, Creatures1}` where `Creature1` is the same creature as `Creature` (that is, they have the same

reference but might have different stats) and similar for `Creatures1` with respect to `Creatures`.

The district should monitor that the trigger is well-behaved: that it returns a result within two seconds, that the result has the right form and that it does not throw exceptions or exits. If a trigger is not well-behaved, then the result of it should be ignored. If the trigger is well-behaved then `Creature1` and `Creatures1` is used. Thus, when a creature, `C`, is moved from district `P` to district `Q`, and `P` has a trigger `T` associated where `T(leaving, C, Cs)` returns `{C1, Cs1}` then `C1` is entering `Q` (and `Cs1` stays at `P`).

A district can only have one trigger, so if this function is called more than once then only the last trigger is associated with the district.

If your implementation supports trigger this function should return `ok` if `District` is under configuration; otherwise it should return `{error, Reason}`. If your implementation does not support triggers this function should always return `not_supported`.

Question 2.2: QuickCheck `district`

Make a module `district_qc` that uses QuickCheck for testing a `district` module. We will test your `district_qc` module with our special version of the `district` module, so your tests should only rely on the API described in the exam text.

Your module should at least export:

- A data generator `territory/0` that generates Erlang maps on the form:

  ```
  map( integer() => [{atom(), integer()}] )
  ```

  That is, a map with integers as a keys and a list of pairs of atoms and integers as values.

- A function `setup_territory(Map)` that takes a map on the form returned by the generator `territory/0` and sets up a district for each key in the map with neighbours as specified by the corresponding value. If an integer only exists as value it is set up as a district without neighbours. Returns a list of all created districts.

- A property `prop_activate/0` that is related to `district:activate/1`.

- A property `prop_take_action/0` that is related to `district:take_action/1`.

The intention is that you use the `territory/0` generator in your properties. If you don't use `territory/0` make sure that you document why in your report, and what you have done instead.

You are welcome (even encouraged) to make more properties. These properties should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of `district` (perhaps they are related to an extended API or you are testing a sub-modules of your implementation), they should start with the prefix `myprop_`, so that we know that these most likely only work with your own implementation of `district`.

Topics for your report for Question 2

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

In particular your report should document:

- If your implementation can handle territories with cycles. And if so, what your strategy is for handling cycles.
- If your implementation requires special extra communication between districts than what is described in the API.
- If your implementation supports triggers or not. If you support triggers, have you made any assumptions about triggers. For instance, are triggers allowed to call other functions including functions from `district`, can triggers return results of the wrong form, can triggers throw exceptions (which kinds) and so on.

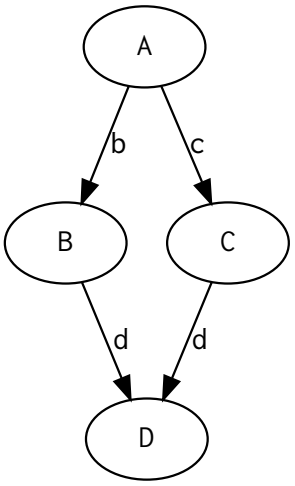
Appendix A: Example use of `district`

Appendix A.1: Small example

The following is a small example demonstrating how to use the `district` API. It simulates a love story between Alice and Bob, where Bob gets separated from Alice, but finds his true love in the end.

The territory contains no cycles and uses no triggers.

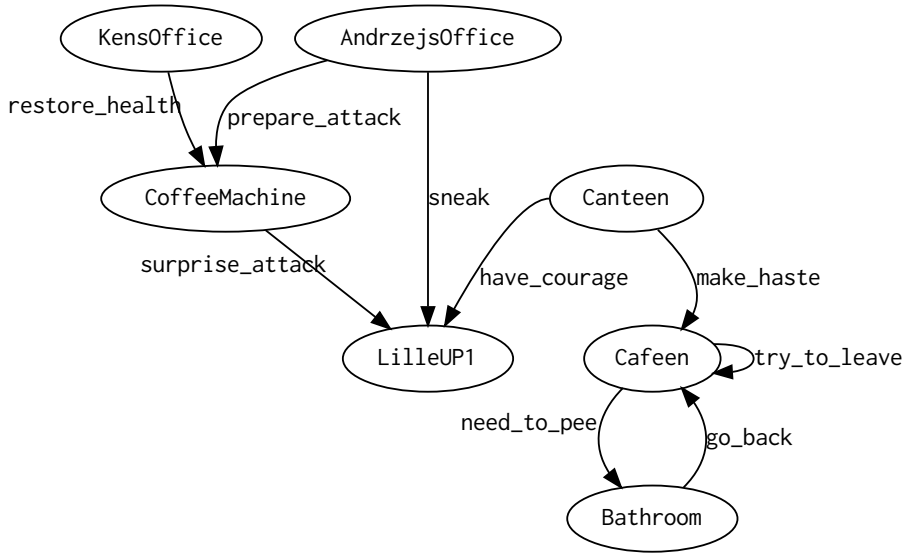
```
-module(the_diamond_path).
-export([a_love_story/0]).

a_love_story() ->
  % Defining a dimond-shaped territory.
  {ok, A} = district:create("A"),
  {ok, B} = district:create("B"),
  {ok, C} = district:create("C"),
  {ok, D} = district:create("D"),
  district:connect(A, b, B),
  district:connect(A, c, C),
  district:connect(B, d, D),
  district:connect(C, d, D),

  % Activating the districts.
  % Since there is a path from A to every other district, this will suffice:
  district:activate(A),

  % Two players without stats.
  {BobRef, _}   = Bob   = {make_ref(), #{}},
  {AliceRef, _} = Alice = {make_ref(), #{}},

  % Bob and Alice entered the same district:
  district:enter(A, Bob),
  district:enter(A, Alice),

  % But on that day, they choose to follow different paths.
  district:take_action(A, BobRef, b),
  district:take_action(A, AliceRef, c),

  % But fortunately, there is no way to get lost in the diamond path.
  district:take_action(B, BobRef, d),
  district:take_action(B, AliceRef, d),
  A.

% THE END
```



Appendix A.2: Involved example

The following is a larger example demonstrating how to use the `district` API. It tries to simulate a day at DIKU with as much accuracy as possible.

The territory contains cycles and uses multiple triggers.




```
% Example contributed by Joachim and Mathias
-module(a_day_at_diku).
-export([run_world/0]).

make_drunker({CreateRef, Stats}) ->
    #{sobriety := CurSobriety} = Stats,
    {CreateRef, Stats#{sobriety := CurSobriety - 1}}.

make_sober({CreateRef, Stats}) ->
    #{sobriety := CurSobriety} = Stats,
    {CreateRef, Stats#{sobriety := CurSobriety + 1}}.

cheers(_, Creature, Creatures) ->
    io:format("Cheeeeers!~n"),
    {make_drunker(Creature), lists:map(fun make_drunker/1, Creatures)}.

rest_a_bit(entering, Creature, Creatures) ->
    {make_sober(Creature), Creatures};
rest_a_bit(leaving, Creature, Creatures) ->
    {Creature, Creatures}.

andrzejs_office(entering, {CreatureRef, Stats}, Creatures) ->
    io:format("You get lost in Andrzejs stacks of papers, lose 1 sanity!~n"),
    #{sanity := CurSanity} = Stats,
    {{CreatureRef, Stats#{sanity := CurSanity - 1}}, Creatures};
```

```
andrzejs_office(leaving, Creature, Creatures) ->
    {Creature, Creatures}.

lille_up1(entering, {CreatureRef, Stats}, Creatures, KenRef, AndrzejRef) ->
    CreatureRefs = lists:map(fun({Ref, _Stats}) -> Ref end, Creatures),
    KenPresent = lists:member(KenRef, CreatureRefs),
    AndrzejPresent = lists:member(AndrzejRef, CreatureRefs),
    if KenPresent and AndrzejPresent ->
        {{CreatureRef, Stats#{stunned => true}}, Creatures};
       true ->
        {{CreatureRef, Stats}, Creatures}
    end;
lille_up1(leaving, _Creature, _Creatures, _KenRef, _AndrzejRef) ->
    % This is misbehaving, thus the trigger has no effect
    ok.

generate_territory() ->
    {ok, KensOffice} = district:create("Ken's office"),
    {ok, AndrzejsOffice} = district:create("Andrzej's office"),
    {ok, CoffeeMachine} =
        district:create("The Coffee Machine at the end of the APL hallway"),
    {ok, Canteen} =
        district:create("The Canteen at the top floor of the DIKU building"),
    {ok, Cafeen} = district:create("The student bar, \"Caféen?\""),
    {ok, Bathroom} = district:create("The bathroom at the student bar"),
    {ok, LilleUP1} =
        district:create("The smaller auditorium at the DIKU building"),

    ok = district:connect(KensOffice, restore_health, CoffeeMachine),
    ok = district:connect(AndrzejsOffice, prepare_attack, CoffeeMachine),

    % Andrzej sometimes skips his coffee
    ok = district:connect(AndrzejsOffice, sneak, LilleUP1),

    ok = district:connect(CoffeeMachine, surprise_attack, LilleUP1),
    ok = district:connect(Canteen, make_haste, Cafeen),
    ok = district:connect(Canteen, have_courage, LilleUP1),

    ok = district:connect(Cafeen, try_to_leave, Cafeen),
    ok = district:connect(Cafeen, need_to_pee, Bathroom),
    ok = district:connect(Bathroom, go_back, Cafeen),

    % Places to spawn or place advanced triggers
    [KensOffice, AndrzejsOffice, CoffeeMachine, Canteen, Bathroom,
     Cafeen, LilleUP1].

place_triggers(KenRef, AndrzejRef, AndrzejsOffice, Cafeen,
```

```
            Bathroom, LilleUP1) ->
    district:trigger(AndrzejsOffice, fun andrzejs_office/3),
    district:trigger(Cafeen, fun cheers/3),
    district:trigger(Bathroom, fun rest_a_bit/3),
    district:trigger(LilleUP1,
           fun (Event, Creature, Creatures) ->
              lille_up1(Event, Creature, Creatures, KenRef, AndrzejRef)
           end),
    ok.

run_world() ->
    KenRef = make_ref(),
    AndrzejRef = make_ref(),

    KenStats = #{hp => 100, sanity => 7.4},
    AndrzejStats = #{hp => 100, sanity => 80, mana => 100},

    [KensOffice, AndrzejsOffice, CoffeeMachine, Canteen, Bathroom,
     Cafeen, LilleUP1] = generate_territory(),

    place_triggers(KenRef, AndrzejRef, AndrzejsOffice, Cafeen,
                   Bathroom, LilleUP1),

    % Activate the initial nodes. The rest will follow
    active = district:activate(KensOffice),
    active = district:activate(AndrzejsOffice),
    active = district:activate(Canteen),

    Ken = {KenRef, KenStats},
    Andrzej = {AndrzejRef, AndrzejStats},

    StudentRefs = lists:map(fun (_) -> make_ref() end, lists:seq(1, 100)),
    StudentStats = #{hp => 10, sobriety => 50, sanity => 15},

    PrebenRef = make_ref(),
    PrebenStats = #{hp => 1, sobriety => 150, sanity => 150},

    % Spawn the creatures
    ok = district:enter(KensOffice, Ken),
    ok = district:enter(AndrzejsOffice, Andrzej),
    ok = district:enter(Cafeen, {PrebenRef, PrebenStats}),
    lists:map(fun (StudentRef) ->
                      ok = district:enter(Canteen, {StudentRef, StudentStats})
              end, StudentRefs),

    ok = district:take_action(KensOffice, KenRef, restore_health),
    ok = district:take_action(AndrzejsOffice, AndrzejRef, sneak),
```

```
% That morning, Bob thought he could sneak into Lille UP1 before Andrzej,
% but he was already too late
ok = district:take_action(Canteen, hd(StudentRefs), have_courage),
ok = district:take_action(CoffeeMachine, KenRef, surprise_attack),
{KensOffice, AndrzejsOffice, Canteen}.
```