Take-home exam
AP

Exam number 66

November 9, 2018

The `appm` package manager

Utility functions

Version ordering It turns out that ordering versions as described in the assignment can be accomplished by simply deriving the `Ord` typeclass from `Version` and `VNum`. By doing this, I tell the compiler to use the default ordering on the contained `VNum` list. Each of the version numbers in the list will, by default, be compared by first comparing the number, and then the string. Each element of the two lists are compared, in order, until two version numbers numbers aren't equal, in which case the inequality determines the result of the comparison. Thus, the default ordering accomplishes what I set out to do with the `Version` type.

Merging dependencies The `merge` function works by computing the intersections of allowed version ranges for constraints on the same package. More specifically, when merging two constraint lists c_1 and c_2, I build a new constraint list c' from scratch. Each element of c_1 and c_2 is added to c' iteratively. Each time, I perform a linear traversal of c'; if I stumble upon a constraint on the same package along the way, I merge the two individual constraints by computing the intersection of allowed versions and setting the flag, indicating whether the package is required or a conflict, to `True` if at least one of the flags was set beforehand. If any intersection yields an empty range, there is no result.

By building the new constraint list from scratch, I make sure that the resulting constraint list is fully reduced; if I simply merged each constraint of c_2 into c_1 and assumed c_1 itself to be fully reduced, I could potentially encounter some problems where this is not the case, i.e. when checking equivalence between dependencies of two different versions of the same package. The assumption may be reasonable since I merge them in the parser in an order that guarantees a fully reduced constraint list, but a more robust behaviour is in my opinion preferable.

Other I have added my own function `satisfies` that is shared among multiple modules. It simply checks that a given solution satisfies a given list of constraints.

Parsing `appm` databases

I have chosen to use the Parsec parser combinator library since I have worked with it before. Furthermore, it is well documented and widely used. It generates reasonable error messages when parsing, and I like the overall layout of the combinator as a whole.

Only few intricacies occured when implementing the parser, in particular parsing and generating packages. Given the Parsec parser combinator, the rest has been more or less trivial; most of the requirements are addressed by simply adding a case or two to each parser.

Parsing and generating packages The task is to parse a package in the database, but also to make sure that said record is well-formed given the criteria from the

assignment. I achieve this by defining a custom data type, `Clause`, which denotes each type of valid clause in a package. When parsing a package, I parse all the clauses and generate a `Clause` list of these. I then check that these clauses collectively satisfy the constraints on the number of each type of clause. If all is as expected, I traverse the `Clause` list and accumulate a package. The dependendcies are generated by merging (using the `merge` function described in the previous section) each occurring constraint with the already accumulated dependencies in the package. If I'm not able to merge some constraints along the way, I throw an error.

I have chosen to do it this way since it eases the sanity check of a package and allows the user to define clauses in any arbitrary order. Furthermore, it is based on a simple, linear traversal of all clauses, which can be accomplished with our favourite bulk operators (`map`, `foldl`, etc.). All in all it seems to be the simplest and most effective solution for the given task.

Other stuff worth mentioning To make keywords case insensitive I have chosen to simply parse each char of a string, convert it to lower case if applicable, generate a new string from these, and then match this string on the predefined keyword.

In order to correctly escape double quotes within a string, I parse arbitrary characters until I encounter a pair of double quotes. When this happens, I put one double quote in the string.

To parse dependencies, I first parse either the `"requires"`- or the `"conflicts"` keyword. Which of these is parsed determines the boolean flag in the resulting constraint. Then, the way the actual range is generated depends on this flag; if the flag is set, the resulting range is the *intersection* of the two inequalities. If not, the range is the *complement* of the two inequialities. That is,

`requires bar >= 0, bar < 7` :	versions 0-7 are *required*
`conflicts bar < 0, bar >= 7` :	versions 0-7 are *allowed*

Note that the inequalities are still inferred when no version numbers are specified explicitly. That is, `conflicts foo` sets an empty allowed range for `foo`, and `requires foo` sets the range to go from the smallest to the largest supported version.

Correctness The parser satisfies all criteria listed in the assignment; as we will see in the section dedicated to testing, it parses some hard coded databases as well as automatically generated ones correctly.

Solving `appm` constraints

Normalizing the database Since the assignment doesn't clarify this, I make the assumption that no package in the database be equivalent to any other package in the database after normalizing. That is, if two equivalent packages are discovered, I discard one of them.

The idea when normalizing a database is to group the packages into lists of packages with the same name and version. For each of these lists, I check if all packages in this list are equivalent to each other. If they are, I take an arbitrary package from the group (since they are equivalent) and put it in the resulting database. If not, I return an error message. I finally sort the database such that packages of the same name are adjacent in order from newest to oldest versions. This can make the job easier for the solver; if well implemented, assuming the order of packages allows us to avoid sorting in the solver.

Solving constraints At first I implemented the solver by defining an appropriate `Solver` monad to keep track of the found soulutions. It worked by keeping the database as a constant resource and a list of solutions as a 'mutable' state. Everytime a solution was discovered, it was appended onto the implicit state of the monad. I chose not to keep this implementation, however, because it produced some substantial overhead given how relatively small the task is.

The actual `solve` function is implemented recursively. Given some constraints c and a partial solution s, it checks whether s satisfies c, in which case the solution is returned as an element in a list. If not, it groups all required packages from the database (that aren't already in s) by name and computes the cartesian product of all groups — this gives us all subsets of packages that may satisfy the constraints of the partial solution. Filtering out the packages that either aren't required or are already in the solution guarantees that no package will occur twice in the solution, and that we don't waste time solving constraints for unnecessary packages. `solve` then tries to merge each subset individually into s. Each that succeeds will be added to s, yielding s' and corresponding constraint list c'. If c' is consistent, the function then solves c' on s' recursively.

Since we assume the database to be normalized, I can exploit the fact that packages with the same name are adjacent; grouping them can be done without looking through the whole database. Furthermore, the ordering makes sure that the result of the cartesian product begins with the combinations of newest versions and ends with the combinations of lowest versions. That is, to achieve a list of solutions of decreasing quality (as was one of the requirements), I don't even need to sort when solving. Thus, these assumptions can save us some (in some cases substantial) overhead and allow the implemenation to be relatively simple.

Installing packages This leads us to the installation of a package. My implementation of `install` works as follows. Given a database and a package name, the function fetches all versions of the package from the database. Again, their ordering allows us to assume the versions of the packages to be decreasing. Thus, I try to solve the newest version of the package, passing its dependencies as the initial constraints and the list containing only the name/version tuple as the inital partial solution. If the result is a non-empty list, the best solution must be the first element of said list; since my implementation doesn't alter the order of packages, our assumptions tell us that the quality of the solutions decreases, and so the best solution must necessarily be the first in the solution list. If there is no solution to the newest version, the function does the same with the version immediately

below this. This is repeated until a solution is found, or until there are no more versions left.

Testing `appm` properties

I have already reasoned about the correctness of my implementation given the assumptions about the database. To further ensure the correct behaviour of all parts of the solver, I have written a bunch of automated tests.

Black-box tests The black-box tests are hardcoded instances tested against some expected output. In particular, I have written black-box tests for version comparison, merging of constraints, the parser, and the overall solver, all of which pass. The parser tests cover all requirements from the assignment. Since hardcoding an instance for each test is demanding, there are not *too* many blackbox-tests. Instead I rely on quickcheck to automatically generate instances and check the properties of the solution. Note that the black-box tests have been split up in files corresponding to the module being tested. For instance, the black-box parser tests can be found in `appm/tests/BB/ParserTest.hs`. All test runs are still gathered in `appm/tests/BB/Main.hs`.

QuickCheck In order to actually generate well-formed instances, I have defined an automatic database generator which can be found in `appm/tests/QC/Gen.hs`. The generator for each component of the database (i.e. package names, version numbers, etc.) has been pretty straightforward to implement, however, the database as a whole has to be well-formed in order for QuickCheck to make sense. Thus, my generator works as follows. It starts by generating a list of packages with empty names and constraints. Then a list of possible names the size of the database is generated, which ensures the possibility for multiple versions of the same package in the database and additionally makes it possible to generate sensible dependencies. Each package in the package list will receive a random name from the name list, and up to four dependencies are chosen randomly from the available packages. Afterwards the generator trims (that is, removes duplicates from and afterwards sorts) the package list such that it is normalized — this is easier than using the `normalize` function since normalizing the database won't yield a database if the input isn't valid. An example of an automatically generated database in ghci (using a simple pretty-printer) can be seen in Appendix A.4.1.

I have implemented property-based tests for criteria (a), (b), (c), (d), (e), (f), and (g), along with the parser. I won't go into detail about the implementation of criteria (a) through (g), but they are implemented as dictated by the assignment. Each of them has been tested manually by tweaking things and making sure that the behaviour changes accordingly.

The property-based test for the parser works by simply 'pretty-printing' the database to a string, parsing the string, and making sure that the resulting database

is equivalent to the input database. Again, I won't go into detail about the implementation of the property-based test nor the pretty-printer.

For each of the property-based tests, QuickCheck generates a database instance. The first package from the database is then selected as the package to install. This ensures that we actually try to install a package that is in the database. It then uses `install` to find the corresponding solution, if any, and feeds it into the predicate for the corresponding property implemented in `Properties.hs`. As with the black-box tests, all property-based tests in the QuickCheck suite pass.

Assessment I have already argued some points regarding the integrity of my implementation, but I will dedicate this paragraph to a brief assessment of the tests. The databases being generated are not only well-formed, but they also represent sensible instances yielding varying solutions that are ideal when testing the given properties. This means that if a property-based test passes, we can be sure that it is not just because it is trivially true (i.e. when there is no solution). In my opinion, the black-box tests and the property-based tests (all of which pass), along with the database generator, fulfill the intended purpose of testing arbitrary implementations of the dependency solver.

Earls of Ravnica

The `district` module

I have chosen to use the `gen_statem` behaviour for my implementation. An individual district is its own state machine, and so, it makes sense to model it as such using the `gen_statem` behaviour in Erlang. The callback mode i have used is `state_functions`. That is, for each state of the district I define a callback function to handle the variaous events. This is, in my opinion, more convenient than using `handle_event_function`, since you can have a dedicated function per state.

I will try to cover only the most essential parts of my implementation, leaving the rest as something to read in the source.

Creating a district In order to create a district, and thus initialize a new state machine, I have implemented `create` by simply calling `gen_statem:start`, giving the module name, description, and the empty list as arguments. The `init` callback function sets the inital state to `under_configuration` and passes the initial state data. Since we are working with neighbours, creatures, a trigger, and a description as the state data, the data is represented by a four-tuple consisting of a `map(atom() => passage())` for the neighbours, a `map(creature_ref() => creature_stats())` for the creatures, simply a `trigger()` function for the trigger, and a string for the description. Maps make it easy to maintain a collection of pairs where each key is unique, and the lookup is easy and efficient.

Connecting districts `connect` sends a call to the given district, passing the action and the destination. Representing the connections to the neighbours as a map, it is easy see if the action is already taken; simply ask if the action already exists as a key, and if it does, return the appropriate error. Else, put it into the map with the destination as the value and return `ok`.

Activating a district This has been a bit tricky, and I have tried a few solutions before choosing to do it this way. When a district, which is under configuration, is activated, it immediately goes to the `under_activation` state and forces the next event to be an internal event where the district then activates all of its neighbours. When each activation of the neighbours has returned a value, the district checks if any activation has been impossible. If this is the case, the district itself can't be activated. In order to avoid blocking the processes, each path that is activated keeps track of a list of visited districts. Thus, everytime a district is activating its neighbours, it chooses to ignore the ones that have already been visited. This also takes care of cycles, however, even without the list of visited districts, cycles are handled automatically; `activate` terminates the chain of calls when it is already under activation.

If there is a path from the root district being activated to some district that can't be activated, the root district also fails. Thus, I assume that it is possible to activate a part of a territory that does not include the root being activated. On Figure 1 I have tried to illustrate my assumption about how `activate` would

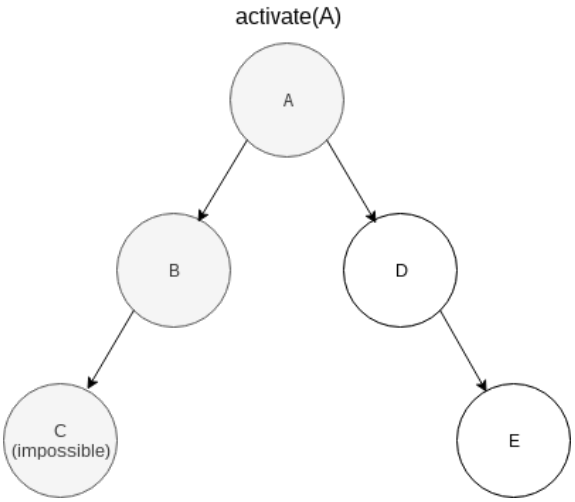

Figure 1: Activating a district A with one path to a district that can't be activated and one path without problems.

work. A, B, and (trivially) C fail to activate since they all have a path to a district that can't be activated. However, D and E are successfully activated.

Entering a district This sends a call to the district, passing the creature wanting to enter. The district simply checks to see if the reference is already a key in the creature map, and if not, applies the `entering`-trigger to the creature entering and the creatures already in the district. The result then represents the new map of creatures in the state data.

The way the well-formedness of the trigger is checked is as follows. A new process applying the trigger is spawned, which sends a message back with the result. This allows me to use the `after`-timeout when receiving the result. If there is no result within 2 seconds, the input to the trigger is simply returned. The spawned process calls the trigger and catches any exception that may occur — if this happens, the input is returned. If there is a response within 2 seconds, the check proceeds by ensuring that the result is a tuple, and that all the creature references are the same. If all is well, the new data is returned. There is no check on the use of functions inside the trigger, so it is possible for trigger to call an API function. This may cause problems if a district applies a trigger that makes a call to this very district, perhaps blocking the process.

Taking action This sends a call to the district, passing the creature reference and the action. The district makes sure that the creature is actually in the map of contained creatures and that the action exists in the map over neighbours. If so, the `leaving`-trigger is applied to the creature leaving and the creatures staying in the district, and `enter` is called on the destination. If all goes well, the creature is moved. One special case is when the action is an edge to the same district; this would, again, block everything. Thus, if the destination is the same as the current district, the `entering`-trigger is applied locally and the state data is updated accordingly.

This solution may not be the most elegant since it repeats some code from `enter`. However, given the problems that arise when handling self loops, this solution is a very simple way to avoid blocking the process.

Shutting down a district This works much in the same way as `activate`. After sending the required message to the `NextPlane` argument, the district immediately goes to the `shutting_down` state and forces an internal event that makes it shut down its neighbours. After they have all been visited, It uses the `gen_statem:stop` to actually terminate the process.

Testing `district`

I have already gone over some reasons why my implementation should behave correctly. To further ensure the behavour of my implementation, I have written some unit tests and some property-based tests.

Unit testing I have chosen to do a series of unit tests much like I did in Haskell. In case the property-based QuickCheck tests are not producing good enough instances, at least these unit tests can tell a bit about the integrity of each part of the implementation. I chose to use EUnit for this task, since it is simple and easy to use, and its error messages when tests fail are very helpful.

There are 50 unit tests which, collectively, cover all requirements from the assignment. These range from connecting districts, moving a creature from a district to another, activating territories with cycles and self loops, registering well-formed triggers and triggers returning ill-formed data and throwing exceptions, and shutting down while checking the messages afterwards. In my opinion, these tests show that the behaviour of my implementation is at least pretty close to the intended behaviour described in the assignment. The fact that all tests pass indicates that a large portion of the solution is correct. The tests can be found in `ravnica/district_bb`.

QuickChecking the properties I have written a territory generator that simply creates a random map as described in the assignment. I have also, as was the task, implemented `setup_territory`, which takes this map and creates a district for each of the keys, creates all connections from the `{atom(),key()}` list, and finally returns all district pids as the resulting territory. I won't go into detail about the concrete implementations of these. An example of a generated territory (on the form before setting up the actual districts) can be seen in Appendix B.3.1.

In order to check the neighbours of and the creatures in the districts after calling som API function, I initially used `sys:get_state`. However, I realized that this is heavily implementation-specific (how is the state data represented?), so I had to choose a much simpler, and not as flexible, approach. Thus, I have implemented property-based for three API functions:

- `activate`

- `shutdown`
- `take_action`

When testing activation, I simply make sure that if we can successully activate some district in the generated territory, then the neighbours will be active afterwards. This is done by creating a creature and moving it to the neighbours. This gives me the pid of the neighbour, and I check the state using `sys:get_state` (not relying on the state data, only the state itself). I could assume that moving the creature would be enough since it is only allowed when both the current district and the neighbour are active, but actually fetching the state is in my opinion a better indicator.

When testing the territory when shutting down, I activate some district in the generated territory, create a creature, move it to some neighbour, get the pid of said neighbour, shut down the current district, and check that both processes have terminated using `process_info`. This would indicate that a district is actually correctly shut down.

When testing actions, I simply create a creature, enter it in some district from the generated territory, move it to a neighbour through an action, and then try to move it from the same district to a neighbour through the same action. This should fail, since the creature is no longer in the inital district.

Assessment I realize that my specific property-based tests are not a sure guarantee that my implementation is perfect; they are rather simple, only testing within a very local neighbourhood in any generated territory. However, they still do show some qualities about my implementation that are required in a correct solution, so they are in my opinion far from worthless.

The unit tests, on the other hand, strongly indicate correct behaviour on various instances with intentional edge cases and errors. Here, not a single instance behaves unexpectedly as long as it is well-formed as per the assignment.

A appm listings

A.1 Source listings

A.1.1 Source code for /handin/appm/src/Defs.hs

```
module Defs where

import Data.List (intercalate)


--------
-- Types
--------

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, Ord)

newtype Version = V [VNum]
  deriving (Eq, Show, Read, Ord) -- the default ordering works perfectly

minVN, maxVN, stdVN :: Int
stdVN = 1
minVN = 0
maxVN = 1000000

minV, maxV, stdV :: Version
minV = V [VN minVN ""]  -- inclusive lower bound
maxV = V [VN maxVN ""]  -- exclusive upper bound
stdV = V [VN stdVN ""]

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)]


----------------
-- Own instances
----------------

-- define package ordering
instance Ord Pkg where
  l <= r = (name l < name r) || (name l == name r && ver l <= ver r)

-- uncomment to show database as pretty
-- remember to not derive show in this case
-- instance Show Database where
--   show = prettyDB


------------------------------
-- Pretty printer for databases
------------------------------

-- used to quickcheck the parser and better read output
-- not a pretty implementation, but the output is OK

```

```haskell
prettyDB :: Database -> String
prettyDB (DB pkgs) = (intercalate "\n\n" . map prettyPkg) pkgs

prettyPkgs :: [Pkg] -> String
prettyPkgs = intercalate "\n\n" . map prettyPkg

prettyPkg :: Pkg -> String
prettyPkg (Pkg n v des dep) =
  "package {\n" ++
  "  name " ++ prettyPName n ++ ";\n"   ++
     (if v /= stdV then "  version " ++ prettyVer v   ++ ";\n" else "") ++
     (if null des then "" else "  description \"" ++ escape des ++ "\";\n")
     ++
      prettyConstrs dep ++
  "}"

  where escape = concatMap escapeChar
        escapeChar '"' = "\"\""
        escapeChar c = [c]

prettyPName :: PName -> String
prettyPName (P s) = s

prettyVer :: Version -> String
prettyVer (V vnums) = intercalate "." . map prettyVNum $ vnums

prettyVNum :: VNum -> String
prettyVNum (VN n s) = show n ++ s

prettyConstrs :: Constrs -> String
prettyConstrs = concatMap prettyConstr

prettyConstr :: (PName,PConstr) -> String
prettyConstr (p,(b,vmin,vmax)) =
  (if b then "  requires " else "  conflicts ")  ++
  prettyPName p ++ (if b && vmin >= minV then " >= " else " < ") ++
  prettyVer vmin ++ ";\n" ++
  (if b then "  requires " else "  conflicts ")  ++
  prettyPName p ++ (if b then " < " else " >= ") ++
  prettyVer vmax ++ ";\n"
```

A.1.2 Source code for /handin/appm/src/Utils.hs

```haskell
module Utils where

import Defs

import Control.Monad (foldM)

import Data.List (sortBy)


-- merging constraint lists
merge :: Constrs -> Constrs -> Maybe Constrs
merge c1 c2 = foldM merge' [] (c1 ++ c2)

merge' :: Constrs -> (PName,PConstr) -> Maybe Constrs
merge' ((p,c):lst) (p',c') | p == p' = do
  new <-  c `intersection` c'
  return $ (p,new) : lst
merge' (c:lst) c' = (:) c <$> merge' lst c'
merge' [] c = return [c]

-- compute the intersection between two version ranges
intersection :: PConstr -> PConstr -> Maybe PConstr
intersection (b1, vmin1, vmax1) (b2, vmin2, vmax2) =
  let (vmin',vmax') = (max vmin1 vmin2, min vmax1 vmax2)
  in if vmax' <= vmin'
     then Nothing
     else return (b1 || b2, vmin', vmax')


-- check if a solution satisfies some constraints
satisfies :: Sol -> Constrs -> Bool
```

```
satisfies sol = all (sol `satisfies'`)

satisfies' :: Sol -> (PName,PConstr) -> Bool
satisfies' sol (p,(b,vmin,vmax)) | b =
  any (\(p',v) -> p' == p && (v >= vmin && v < vmax)) sol
satisfies' sol (p,(_,vmin,vmax)) =
  not $ any (\(p',v) -> p' == p && (v < vmin || v >= vmax)) sol


-- sorting packages (newer version first)
sortPkgs :: [Pkg] -> [Pkg]
sortPkgs = sortBy (flip compare)
```

A.1.3 Source code for /handin/appm/src/ParserImpl.hs

```
module ParserImpl where

-- put your parser in this file. Do not change the types of the following
-- exported functions

import Text.Parsec.Prim hiding (token)
import Text.Parsec.Char hiding (spaces)
import Text.Parsec.String
import Text.Parsec.Combinator

import Control.Monad (when, unless, foldM)

import Data.Functor (($>))
import Data.Char (toLower, toUpper)

import Defs
import Utils

-----------------------
-- Functions to export
-----------------------

parseVersion :: String -> Either ErrMsg Version
parseVersion s = case parse (spaces *> version <* eof) "" s of
  Left e -> Left $ show e
  Right v -> Right v

parseDatabase :: String -> Either ErrMsg Database
parseDatabase s = case parse (spaces *> database <* eof) "" s of
  Left e -> Left $ show e
  Right v -> Right v


------------------
-- Generic parsers
------------------

-- keywords
keyword :: String -> Parser String
keyword = token1 . mapM (\c -> char (toLower c) <|> char (toUpper c))

-- whitespace
spaces :: Parser String
spaces = many space <* skipMany cmt

spaces1 :: Parser String
spaces1 = many1 space <* skipMany cmt <|> skipMany1 cmt $> ""

-- comments
cmt :: Parser String
cmt = string "--" >> (try (manyTill anyChar (oneOf "\r\n"))
                      <|>  manyTill anyChar eof) <* spaces

-- token - ignore whitespace after
token :: Parser a -> Parser a
token p = p <* spaces

token1 :: Parser a -> Parser a
token1 p = p <* spaces1
```

```
60
61 -- symbols - strings
62 symbol :: String -> Parser String
63 symbol = token . string
64
65 symbol1 :: String -> Parser String
66 symbol1 = token1 . string
67
68 -- delimiters
69 parens :: Parser a -> Parser a
70 parens = between (symbol "(") (symbol ")")
71
72 braces :: Parser a -> Parser a
73 braces = between (symbol "{") (symbol "}")
74
75 quots :: Parser a -> Parser a
76 quots = between (char '\"') (symbol "\"")
77
78
79 -- Grammar
80
81 -- Database
82 database :: Parser Database
83 database = DB <$> many package
84
85 -- Package
86 data Clause = Name PName | Ver Version | Desc String | Deps Constrs
87
88 package :: Parser Pkg
89 package = do
90   keyword "package"
91   checkDeps =<< genPackage
92             =<< checkClauses
93             =<< braces clauses
94
95 clauses :: Parser [Clause]
96 clauses = sepEndBy clause (symbol ";")
97
98 clause :: Parser Clause
99 clause = (keyword "name"        >> Name <$> pname   )
100      <|> (keyword "version"     >> Ver  <$> (checkPkgVer =<< version) )
101      <|> (keyword "description" >> Desc <$> str     )
102      <|> (                         Deps <$> constr  )
103
104 checkPkgVer :: Version -> Parser Version
105 checkPkgVer v = do
106   unless (v < maxV) $
107     unexpected ("Version number of package (max " ++ show (maxVN - 1) ++ ")
      ")
108   return v
109
110 checkClauses :: [Clause] -> Parser [Clause]
111 checkClauses cls = do
112   when (n /= 1) $ unexpected "number of name clauses (exactly 1)"
113   when (v > 1)  $ unexpected "number of version clauses (at most 1)"
114   when (d > 1)  $ unexpected "number of description clauses (at most 1)"
115   return cls
116
117   where (n,v,d) = countClauses cls
118
119 countClauses :: [Clause] -> (Int,Int,Int)
120 countClauses = foldl count (0,0,0)
121   where
122     count (n,v,d) Name {} = (n+1,v,d)
123     count (n,v,d) Ver  {} = (n,v+1,d)
124     count (n,v,d) Desc {} = (n,v,d+1)
125     count (n,v,d) Deps {} = (n,v,d)
126
127 checkDeps :: Pkg -> Parser Pkg
128 checkDeps pkg = do
129   when (any (\(p,_) -> p == name pkg) $ deps pkg) $
130     unexpected "self-referential dependencies"
131   return pkg
132
```

```
genPackage :: [Clause] -> Parser Pkg
genPackage cls =
  let start = Pkg{name=P "",ver=stdV,desc="",deps=[]}
  in foldM applyClause start cls

applyClause :: Pkg -> Clause -> Parser Pkg
applyClause pkg (Name p ) = return $ pkg{ name = p }
applyClause pkg (Ver  v ) = return $ pkg{ ver  = v }
applyClause pkg (Desc d ) = return $ pkg{ desc = d }
applyClause pkg (Deps cs) = case deps pkg `merge` cs of
  Nothing  -> unexpected "inconsistence between constraints"
  Just cs' -> return $ pkg{ deps = cs' }


-- package names
pname :: Parser PName
pname = token $ (do
      first <- letter
      after <- concat <$> many pname'
      return $ P (first:after))
  <|> P <$> str

pname' :: Parser String
pname' = do
  c <- string "-" <|> return ""
  a <- alphaNum
  return $ c++[a]


-- versions
version :: Parser Version
version = token $ V <$> sepBy1 vnum (char '.')

vnum :: Parser VNum
vnum = do
  num <- read <$> many1 digit
  when (num < minVN || num > maxVN)
    $ unexpected $ "version number (allowed range: " ++ show minVN ++ "-"
    ++ show maxVN ++ ")"
  VN num <$> suffix

suffix :: Parser String
suffix = do
  s <- many lower
  if length s > 4
  then unexpected "number of letters in suffix (max 4)"
  else return s


-- strings
str :: Parser String
str = quots $ concat <$> many c
  where c = return <$> noneOf "\"" <|>
            try (string "\"\"" $> "\"")

-- deps
constr :: Parser Constrs
constr = do
  b  <- (keyword "requires" $> True) <|> (keyword "conflicts" $> False)
  map (\(p, vmin, vmax) -> (p, (b, vmin, vmax))) <$> bounds b

bound :: Bool -> Parser (PName, Version, Version)
bound b = do
  p <- pname
  (do lt <- (symbol "<" $> True) <|> (symbol ">=" $> False)
      v <- version
      if lt == b then return (p, minV, v)
                 else return (p, v, maxV))
    <|> return (p, minV, if b then maxV else minV)

bounds :: Bool -> Parser [(PName, Version, Version)]
bounds b = sepBy1 (bound b) (symbol ",")
```

A.1.4 Source code for /handin/appm/src/Parser.hs

```haskell
module Parser (parseVersion, parseDatabase) where

-- Do not modify this file, and do not import directly from ParserImpl,
-- elsewhere, except for white-box testing.

import ParserImpl
```

A.1.5 Source code for /handin/appm/src/SolverImpl.hs

```haskell
module SolverImpl where

-- Put your solver implementation in this file.
-- Do not change the types of the following exported functions

import Control.Monad
import Control.Arrow ((&&&)) -- just a small convenience

import Data.Maybe (mapMaybe)
import Data.List (sort, partition, groupBy)

import Defs
import Utils


--------------
-- Normalizing
--------------

-- for each package, find all packages with the same name,
-- check that they are semantically equivalent, group them
-- together, and run recursively on the rest of rest
normalize :: Database -> Either String Database
normalize (DB db) =
  let groups = groupSame db
  in if all allEquiv groups
     then return $ (DB . sortPkgs . map head) groups
     else Left "The database is not consistent"

-- equivalence between all packages in a list
allEquiv :: [Pkg] -> Bool
allEquiv (pkg:pkgs) =
  foldl (\b pkg' -> b && pkg' `equiv` pkg) True pkgs
allEquiv [] = False -- shouldn't ever happen

-- equivalence between two packages
equiv :: Pkg -> Pkg -> Bool
equiv pkg1 pkg2 =
  name pkg1 == name pkg2 &&
  desc pkg1 == desc pkg2 &&
  sort (deps pkg1) == sort (deps pkg2)


-------------------------
-- Solving and installing
-------------------------

-- solving constraints given a solution
solve :: Database -> Constrs -> Sol -> [Sol]
solve _ cs sol | sol `satisfies` cs = [sol]
solve db cs sol =
  let req     = getRequired cs sol db
      groups  = groupName req
      combos  = sequence groups
      sat     = getSats cs sol combos
  in concatMap (uncurry $ solve db) sat


-- installing:
-- fetch all versions of the wanted package and find
-- all solutions for each. then fetch the first sat-
-- isfying solution - if one exists
```

```
install :: Database -> PName -> Maybe Sol
install db p =
  let cands = getWithName p db
  in firstSol db cands

firstSol :: Database -> [Pkg] -> Maybe Sol
firstSol db (pkg:pkgs) =
  case solve db (deps pkg) $ toSol [pkg] of
    []    -> firstSol db pkgs
    sol:_ -> Just sol
firstSol _ _ = Nothing


------------------------------
-- Secondary utility functions
------------------------------

-- get required packages from the context
getRequired :: Constrs -> Sol -> Database -> [Pkg]
getRequired cs sol (DB pkgs) = filter (\pkg ->
      pkg `isRequired` cs && not (pkg `inSol` sol)
    ) pkgs

-- get all packages with given name in the database
getWithName :: PName -> Database -> [Pkg]
getWithName p (DB db) = filter (\pkg -> p == name pkg) db

-- grouping packages with same name
groupName :: [Pkg] -> [[Pkg]]
groupName = groupBy (\pkg1 pkg2 -> name pkg1 == name pkg2)

-- grouping packages with same name and version
groupSame :: [Pkg] -> [[Pkg]]
groupSame = groupBy (\pkg1 pkg2 ->
            name pkg1 == name pkg2 &&
            ver  pkg1 == ver  pkg2)

-- for a list of lists of packages, convert the lists of packages
-- that can be added consistently to a constraint/solution tuple
getSats :: Constrs -> Sol -> [[Pkg]] -> [(Constrs,Sol)]
getSats cs sol = mapMaybe (`takeIfSat` (cs,sol))

  where takeIfSat pkgs (cs,sol) | newsol <- sol ++ toSol pkgs = do
              unless (newsol `satisfies` cs) Nothing
              cs' <- (mergeAll cs . map deps) pkgs
              return (cs',newsol)

-- merge all constraints
mergeAll :: Constrs -> [Constrs] -> Maybe Constrs
mergeAll = foldM merge

-- convert a list of packages to a solution
toSol :: [Pkg] -> Sol
toSol = map (name &&& ver)

-- check if a package is required by a set of constraints
isRequired :: Pkg -> Constrs -> Bool
isRequired pkg = any (\(p,(b,_,_)) ->
              b && name pkg == p)

-- check if a package is already in the solution
inSol :: Pkg -> Sol -> Bool
inSol pkg = any (\(p,_) -> p == name pkg)
```

A.1.6 Source code for /handin/appm/src/Solver.hs

```
module Solver (normalize, solve, install) where

-- Do not modify this file, and do not import directly from SolverImpl
-- elsewhere, except for white-box testing.

import SolverImpl
```

A.1.7 Source code for `/handin/appm/src/Main.hs`

```
module Main where

import Defs
import Parser (parseDatabase)
import Solver (normalize, install)

import System.Environment (getArgs)
import Data.List (intercalate)


prettyVersion :: Version -> String
prettyVersion (V l) =
  intercalate "." [show n ++ s | VN n s <- l]

check :: String -> Either String a -> IO a
check s (Left e) = error $ s ++ ": " ++ e
check _ (Right a) = return a

main :: IO ()
main = do
  args <- getArgs
  case args of
    "-p":dbfile:b ->
      do s <- readFile dbfile
         db <- check "Parsing" $ parseDatabase s
         putStrLn . (if b == ["p"] then prettyDB else show) $ db
    "-n":dbfile:b ->
      do s <- readFile dbfile
         db <- check "Parsing" $ parseDatabase s
         db' <- check "Normalizing" $ normalize db
         putStrLn . (if b == ["p"] then prettyDB else show) $ db'
    [dbfile, pkg] ->
      do s <- readFile dbfile
         db <- check "Parsing" $ parseDatabase s
         db' <- check "Normalizing" $ normalize db
         case install db' (P pkg) of
            Nothing -> error "Cannot solve constraints"
            Just l ->
              do putStrLn "installing packages:"
                 mapM_ (\(P p,v) -> putStrLn $ p ++ " (" ++ prettyVersion v
    ++ ")") l
    _ -> error "Usage: appm DATABASE.db PACKAGE"
```

A.2 Black-box test listings

A.2.1 Source code for `/handin/appm/tests/BB/UtilTest.hs`

```
module UtilTest where

import Test.Tasty
import Test.Tasty.HUnit

import Defs
import Utils


tests :: TestTree
tests = testGroup "Util tests"
    [ testGroup "Version comparison"

      [ testCase "1.0.0 <= 1.0" $
          v1 <= v2 @?= False
      , testCase "1.0 <= 1.0.0" $
          v2 <= v1 @?= True
      , testCase "1.0.0 > 1.0" $
          v1 > v2 @?= True
      , testCase "1.0 > 1.0.0" $
          v2 > v1 @?= False
      , testCase "1.0 > 1.0.0" $
          v2 > v1 @?= False
```

```
      , testCase "1.0.0 <= 2.0" $
          v1 <= v3 @?= True
      , testCase "2.0.0 <= 2.0.0wow" $
          v4 <= v5 @?= True
      , testCase "2.0.0wow > 2.0" $
          v5 > v3 @?= True
      , testCase "1.0oh.0 > 1.0" $
          v6 > v2 @?= True
      , testCase "1.0oh.0 > 1.0.0" $
          v6 > v1 @?= True
      , testCase "3.4z > 3.5a" $
          v7 > v8 @?= False
      ]

    , testGroup "Merging constraints"
      [ testCase "4 + 4 : Nothing" $
          merge c1l c1r @?= c1Out
      , testCase "4 + 4 : Just" $
          merge c2l c2r @?= c2Out
      , testCase "4 + 4 : Nothing" $
          merge c3l c3r @?= c3Out
      , testCase "4 + 4 : Just" $
          merge c4l c4r @?= c4Out
      , testCase "4 + 4 : Nothing" $
          merge c5l c5r @?= c5Out
      ]

    ]

-------------------------------
-- Comparison between versions
-------------------------------

-- Helper function to generate versions
listToVersion :: [(Int,String)] -> Version
listToVersion = V . map (uncurry VN)

-- Test 1
v1    = listToVersion [(1,""),(0,""),(0,"")]
v2    = listToVersion [(1,""),(0,"")]
v3    = listToVersion [(2,""),(0,"")]
v4    = listToVersion [(2,""),(0,""),(0,"")]
v5    = listToVersion [(2,""),(0,""),(0,"wow")]
v6    = listToVersion [(1,""),(0,"oh"),(0,"")]
v7    = listToVersion [(3,""),(4,"z")]
v8    = listToVersion [(3,""),(5,"a")]


-------------------------------
-- Merging two constraint lists
-------------------------------

-- Test 1
c1l   = [ (P "i3-wm",  (True, V [VN 2 "",VN 0 "",VN 4 ""], V [VN 5 "",VN 2 "",VN 2 ""]))
        , (P "emacs",  (True,  V [VN 4 "",VN 1 ""], V [VN 4 "",VN 6 ""]))
        , (P "ghc",    (True, V [VN 1 "",VN 1 "",VN 25 ""], V [VN 3 "",VN 6 ""]))
        , (P "vim",    (True,  V [VN 3 "",VN 0 ""], V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c1r   = [ (P "uxterm", (True,  V [VN 7 "",VN 2 ""], V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "emacs",  (True,  V [VN 2 "",VN 3 ""], V [VN 4 "",VN 3 ""]))
        , (P "mono",   (True,  V [VN 4 "",VN 1 "",VN 1 ""], V [VN 5 "",VN 8 ""]))
        , (P "vim",    (True,  V [VN 6 "",VN 1 "",VN 3 ""], V [VN 7 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c1Out = Nothing

-- Test 2
c2l   = [ (P "i3-wm",  (True,  V [VN 2 "",VN 0 "",VN 4 ""], V [VN 5 "",VN 2 "",VN 2 ""]))
```

```
        , (P "emacs",   (True,  V [VN 4 "",VN 1 ""], V [VN 4 "",VN 6 ""]))
        , (P "ghc",     (True,  V [VN 1 "",VN 1 "",VN 25 ""], V [VN 3 "",VN 6 ""]))
        , (P "vim",     (False, V [VN 3 "",VN 0 ""], V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c2r   = [ (P "uxterm", (True,  V [VN 7 "",VN 2 ""], V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "emacs",   (False, V [VN 2 "",VN 3 ""], V [VN 4 "",VN 3 ""]))
        , (P "mono",    (True,  V [VN 4 "",VN 1 "",VN 1 ""], V [VN 5 "",VN 8 ""]))
        , (P "vim",     (True,  V [VN 5 "",VN 1 "",VN 3 ""], V [VN 7 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c2Out = Just
        [ (P "i3-wm",(True,V [VN 2 "",VN 0 "",VN 4 ""],V [VN 5 "",VN 2 "",VN 2 ""]))
        , (P "emacs",(True,V [VN 4 "",VN 1 ""],V [VN 4 "",VN 3 ""]))
        , (P "ghc",(True,V [VN 1 "",VN 1 "",VN 25 ""],V [VN 3 "",VN 6 ""]))
        , (P "vim",(True,V [VN 5 "",VN 1 "",VN 3 ""],V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
        , (P "uxterm",(True,V [VN 7 "",VN 2 ""],V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "mono",(True,V [VN 4 "",VN 1 "",VN 1 ""],V [VN 5 "",VN 8 ""]))
        ]

-- Test 3
c3l   = [ (P "i3-wm",  (True, V [VN 2 "",VN 0 "",VN 4 ""], V [VN 5 "",VN 2 "",VN 2 ""]))
        , (P "emacs",   (True,  V [VN 4 "",VN 1 ""], V [VN 4 "",VN 6 ""]))
        , (P "ghc",     (True, V [VN 1 "",VN 1 "",VN 25 ""], V [VN 3 "",VN 6 ""]))
        , (P "vim",     (False,  V [VN 2 "",VN 0 ""], V [VN 3 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c3r   = [ (P "uxterm", (True,  V [VN 7 "",VN 2 ""], V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "emacs",   (True,  V [VN 2 "",VN 3 ""], V [VN 4 "",VN 3 ""]))
        , (P "mono",    (True,  V [VN 4 "",VN 1 "",VN 1 ""], V [VN 5 "",VN 8 ""]))
        , (P "vim",     (True,  V [VN 4 "",VN 1 "",VN 3 ""], V [VN 5 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c3Out = Nothing

-- Test 4
c4l   = [ (P "i3-wm",  (True,  V [VN 2 "",VN 0 "",VN 4 ""], V [VN 5 "",VN 2 "",VN 2 ""]))
        , (P "emacs",   (False, V [VN 3 "",VN 1 ""], V [VN 4 "",VN 6 ""]))
        , (P "ghc",     (True,  V [VN 1 "",VN 1 "",VN 25 ""], V [VN 3 "",VN 6 ""]))
        , (P "vim",     (False, V [VN 3 "",VN 0 ""], V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c4r   = [ (P "uxterm", (True,  V [VN 7 "",VN 2 ""], V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "emacs",   (False, V [VN 2 "",VN 3 ""], V [VN 4 "",VN 2 ""]))
        , (P "mono",    (True,  V [VN 4 "",VN 1 "",VN 1 ""], V [VN 5 "",VN 8 ""]))
        , (P "vim",     (True,  V [VN 5 "",VN 1 "",VN 3 ""], V [VN 7 "",VN 0 "",VN 2 "",VN 0 ""]))
        ]
c4Out = Just
        [ (P "i3-wm",  (True,V [VN 2 "",VN 0 "",VN 4 ""],V [VN 5 "",VN 2 "",VN 2 ""]))
        , (P "emacs",  (False,V [VN 3 "",VN 1 ""],V [VN 4 "",VN 2 ""]))
        , (P "ghc",    (True,V [VN 1 "",VN 1 "",VN 25 ""],V [VN 3 "",VN 6 ""]))
        , (P "vim",    (True,V [VN 5 "",VN 1 "",VN 3 ""],V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
        , (P "uxterm", (True,V [VN 7 "",VN 2 ""],V [VN 9 "",VN 0 "",VN 0 ""]))
        , (P "mono",   (True,V [VN 4 "",VN 1 "",VN 1 ""],V [VN 5 "",VN 8 ""]))
```

```
        ]

-- Test 3
c5l    = [ (P "i3-wm",  (True,  V [VN 2 "",VN 0 "",VN 4 ""], V [VN 5 "",VN 2 "",VN 2 ""]))
         , (P "emacs",  (False, V [VN 1 "",VN 1 ""], V [VN 2 "",VN 6 ""]))
         , (P "ghc",    (True,  V [VN 1 "",VN 1 "",VN 25 ""], V [VN 3 "",VN 6 ""]))
         , (P "vim",    (False, V [VN 3 "",VN 0 ""], V [VN 6 "",VN 0 "",VN 2 "",VN 0 ""]))
         ]
c5r    = [ (P "uxterm", (True,  V [VN 7 "",VN 2 ""], V [VN 9 "",VN 0 "",VN 0 ""]))
         , (P "emacs",  (False, V [VN 2 "",VN 9 ""], V [VN 4 "",VN 3 ""]))
         , (P "mono",   (True,  V [VN 4 "",VN 1 "",VN 1 ""], V [VN 5 "",VN 8 ""]))
         , (P "vim",    (True,  V [VN 5 "",VN 1 "",VN 3 ""], V [VN 7 "",VN 0 "",VN 2 "",VN 0 ""]))
         ]
c5Out = Nothing
```

A.2.2 Source code for /handin/appm/tests/BB/ParserTest.hs

```
module ParserTest where

import Test.Tasty
import Test.Tasty.HUnit

import Data.Either (isLeft)

import Defs
import Parser (parseDatabase)


tests :: TestTree
tests = testGroup "Parser tests"
    [ testCase "tiny" $
       parseDatabase src1 @?= Right db1
    , testCase "small" $
       parseDatabase src2 @?= Right db2
    , testCase "large" $
       parseDatabase src3 @?= Right db3
    , testCase "multiple names" $
       isLeft(parseDatabase src4) @?= True
    , testCase "implicit version range (requires)" $
       parseDatabase src5 @?= Right db5
    , testCase "implicit version range (conflicts)" $
       parseDatabase src6 @?= Right db6
    , testCase "comments" $
       parseDatabase src3cmt @?= Right db3
    , testCase "semicolon in string" $
       parseDatabase src7 @?= Right db7
    , testCase "case insensitive keywords" $
       parseDatabase src8 @?= Right db7
    , testCase "case sensitive pnames" $
       parseDatabase src9 @?= Right db9
    , testCase "no space after keyword" $
       isLeft (parseDatabase src10) @?= True
    , testCase "strings, general package names" $
       parseDatabase src11 @?= Right db11
    , testCase "name clause > 1" $
       isLeft (parseDatabase src12) @?= True
    , testCase "name clauses < 1" $
       isLeft (parseDatabase src12') @?= True
    , testCase "version clause" $
       isLeft (parseDatabase src13) @?= True
    , testCase "description clause" $
       isLeft (parseDatabase src14) @?= True
    , testCase "self-referential deps" $
       isLeft (parseDatabase src15) @?= True
    , testCase "inherently contradictory" $
       isLeft (parseDatabase src16) @?= True
    , testCase "many letters in suffix" $
       isLeft (parseDatabase src17) @?= True
```

```
    , testCase "allowed version" $
       isLeft (parseDatabase src18) @?= True
    , testCase "allowed version" $
       parseDatabase src19 @?= Right db19
    ]


-- Test 1
src1 = "package {name foo}"
db1  = DB [Pkg (P "foo") (V [VN 1 ""]) "" []]

-- Test 2
src2 =
  "package { \n\
  \  name hej; \n\
  \  version 1.3.5f; \n\
  \  description \"LOL du \"\"hej\"\" med dig\"; \n\
  \  requires bar >= 1.0; \n\
  \  conflicts bar < 4.0.9 \n\
  \}"
db2  = DB [ Pkg { name = P "hej"
                , ver = V [VN 1 "",VN 3 "",VN 5 "f"]
                , desc = "LOL du \"hej\" med dig"
                , deps = [(P "bar",(True,V [VN 4 "",VN 0 "",VN 9 ""],maxV))
    ]
            }
          ]

-- Test 3
src3 =
  "package { \n\
  \  name foo; \n\
  \  version 2.3; \n\
  \  description \"The foo application\"; \n\
  \  requires bar >= 1.0 \n\
  \} \n\n\
  \package { \n\
  \  name bar; \n\
  \  version 1.0; \n\
  \  description \"The bar library\" \n\
  \} \n\n\
  \package { \n\
  \  name bar; \n\
  \  version 2.1; \n\
  \  description \"The bar library, new API\"; \n\
  \  conflicts baz < 3.4, baz >= 5.0.3 \n\
  \} \n\n\
  \package { \n\
  \  name baz; \n\
  \  version 6.1.2; \n\
  \}"
db3  = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = [(P "bar",(True,V [VN 1 "",VN 0 ""],maxV))]
            }

          , 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 = []
            }
          ]

-- Test 4
src4 =
```

```
  "package { \n\
  \  name foo; \n\
  \  version 2.3; \n\
  \  description \"The foo application\"; \n\
  \  requires bar >= 1.0 \n\
  \} \n\n\
  \package { \n\
  \  name bar; \n\
  \  name far; \n\
  \  version 1.0; \n\
  \  description \"The bar library\" \n\
  \} \n\n\
  \package { \n\
  \  name bar; \n\
  \  version 2.1; \n\
  \  description \"The bar library, new API\"; \n\
  \  conflicts baz < 3.4, baz >= 5.0.3 \n\
  \} \n\n\
  \package { \n\
  \  name baz; \n\
  \  version 6.1.2; \n\
  \}"

-- Test 5
src5 = "package {name foo-hoo; requires hej, dav < 9.2}"
db5  = DB [Pkg (P "foo-hoo") (V [VN 1 ""]) "" [(P "hej", (True,minV,maxV))
    ,(P "dav", (True,minV,V [VN 9 "",VN 2 ""]))]]

-- Test 5
src6 = "package {name foo; conflicts hej, dav < 9.2}"
db6  = DB [Pkg (P "foo") (V [VN 1 ""]) "" [(P "hej", (False,minV,minV)),(P
    "dav", (False,V [VN 9 "",VN 2 ""],maxV))]]

-- Test 6
-- includes comment as only space after keyword
--        , comment until eof
--        , comment between packages
--        , etc.
src3cmt =
  "package { \n\
  \  name--test test\n\
  \    foo; -- this is the name \n\
  \  version 2.3; \n\
  \  description \"The foo application\"; \n\
  \  requires bar >= 1.0 \n\
  \} \n-- now i'm between packages\n\
  \package { \n\
  \  name bar; \n\
  \  version 1.0; --this is the version \n\
  \  description \"The bar library\" \n\
  \} \n\n\
  \package { \n\
  \  name bar; \n\
  \  version 2.1; \n\
  \  description \"The bar library, new API\"; \n\
  \  conflicts baz < 3.4, baz >= 5.0.3 \n\
  \} \n\n-- almost done!\n\n\
  \package { \n\
  \  name baz; \n\
  \  version 6.1.2; \n\
  \}  -- goodbye"

-- Test 7
src7 =
  "package { \n\
  \  name hej; \n\
  \  version 1.3.5f; \n\
  \  description \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  requires bar >= 1.0; \n\
  \  conflicts bar < 4.0.9 \n\
  \}"
db7  = DB [ Pkg { name = P "hej"
                , ver = V [VN 1 "",VN 3 "",VN 5 "f"]
                , desc = "LOL du; \"hej\" med dig"
```

```
                , deps = [(P "bar",(True,V [VN 4 "",VN 0 "",VN 9 ""],maxV))
    ]
              }
            ]

-- Test 8
src8 =
  "pAcKAge { \n\
  \  NAme hej; \n\
  \  VERSION 1.3.5f; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  reQUIRES bar >= 1.0; \n\
  \  coNFlicts bar < 4.0.9 \n\
  \}"

-- Test 9
src9 =
  "pAcKAge { \n\
  \  NAme hEj; \n\
  \  VERSION 1.3.5f; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  reQUIRES Bar >= 1.0; \n\
  \  coNFlicts Bar < 4.0.9 \n\
  \}"
db9  = DB [ Pkg { name = P "hEj"
                , ver = V [VN 1 "",VN 3 "",VN 5 "f"]
                , desc = "LOL du; \"hej\" med dig"
                , deps = [(P "Bar",(True,V [VN 4 "",VN 0 "",VN 9 ""],maxV))
    ]
              }
            ]

-- Test 10
src10 =
  "pAcKAge { \n\
  \  NAmehEj; \n\
  \  VERSION 1.3.5f; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  reQUIRES Bar >= 1.0; \n\
  \  coNFlicts Bar < 4.0.9 \n\
  \}"

-- Test 11
src11 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  VERSION 1.3.5f; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  reQUIRES Bar >= 1.0; \n\
  \  coNFlicts Bar < 4.0.9 \n\
  \}"
db11  = DB [ Pkg { name = P "This is a general name!!1 --- \n\"test\""
                , ver = V [VN 1 "",VN 3 "",VN 5 "f"]
                , desc = "LOL du; \"hej\" med dig"
                , deps = [(P "Bar",(True,V [VN 4 "",VN 0 "",VN 9 ""],maxV))
    ]
              }
            ]

-- Test 12
src12 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  VERSION 1.3.5f; \n\
  \}"

src12' =
  "pAcKAge { \n\
  \  VERSION 1.3.5f; \n\
  \}"

-- Test 13
src13 =
```

```
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  VERSION 1.3.5f; \n\
  \  VERSION 1.3.5f; \n\
  \}"

-- Test 14
src14 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \}"

-- Test 15
src15 =
  "pAcKAge { \n\
  \  NAme foo; \n\
  \  requires foo; \n\
  \}"

-- Test 16
src16 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  requires bar < 3; \n\
  \  requires bar >= 7.9 \n\
  \}"


-- Test 17
src17 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  version 1.2abcde.3\n\
  \}"

-- Test 18
src18 =
  "pAcKAge { \n\
  \  NAme \"This is a general name!!1 --- \n\"\"test\"\"\"; \n\
  \  dEScrIption \"LOL du; \"\"hej\"\" med dig\"; \n\
  \  version 1000000\n\
  \}"

-- Test 18
src19 =
  "pAcKAge { \n\
  \  NAme beef; \n\
  \  version 999999;\n\
  \  requires bar >= 3; \n\
  \  requires bar < 1000000 \n\
  \}"
db19  = DB [ Pkg { name = P "beef"
                 , ver = V [VN 999999 ""]
                 , desc = ""
                 , deps = [(P "bar",(True,V [VN 3 ""],maxV))]
               }
           ]
```

A.2.3 Source code for /handin/appm/tests/BB/SolverTest.hs

```
module SolverTest where

import Test.Tasty
import Test.Tasty.HUnit

import Defs
import Solver (install)


tests :: TestTree
```

```
tests = testGroup "Solver tests"
    [ testCase "tiny" $
       install db1 pname1 @?= Just [(pname1, ver1)]
    , testCase "larger" $
       install db2 pname2 @?= Just out2
    , testCase "vim old version" $
       install db3 pname3 @?= Just out3
    , testCase "vim new version" $
       install db4 pname3 @?= Just out4
    , testCase "empty database" $
       install db5 pname3 @?= Nothing
    , testCase "doesn't exist" $
       install db6 pname6 @?= Nothing
    , testCase "newer version" $
       install db7 pname7 @?= Just out7
    , testCase "conflict with newer version" $
       install db8 pname7 @?= Just out8
    ]


-- Test 1
db1 = DB [Pkg pname1 ver1 "" []]
pname1 = P "foo"
ver1 = V [VN 1 ""]


-- Test 2
db2  = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = []
           }
         , Pkg { name = P "beef", ver = V [VN 2 "",VN 0 ""]
                , desc = "The beef library - new API, same dependencies"
                , deps = [(P "baz",(True,V [VN 3 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
           }
         , Pkg { name = P "bar", ver = V [VN 2 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
           }
         , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
           }
         , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "baz",(True,V [VN 3 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
           }
         , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
           }
         ]
pname2 = P "beef"
out2 = [(pname2, V [VN 2 "", VN 0 ""]),(P "baz", V [VN 4 "", VN 1 "", VN 2
    ""]),(P "bar", V [VN 2 "", VN 1 ""])]


-- Test 3
db3  = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = [ (P "vim", (False, minV, V [VN 9 "",VN 0 ""]))
                         , (P "bar", (True, minV, maxV))]
           }
         , Pkg { name = P "bar", ver = V [VN 2 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
           }
         , Pkg { name = P "vim", ver = V [VN 11 "",VN 1 "",VN 2 ""]
                , desc = "neovim"
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
```

```
            }
          , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "vim", ver = V [VN 8 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "vim",(True,V [VN 3 "",VN 4 ""],V [VN 9 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          ]
pname3 = P "vim"
out3 = [(pname3, V [VN 8 "", VN 1 "", VN 2 ""]),(P "foo", V [VN 2 "", VN 3
    ""]),(P "bar", V [VN 2 "", VN 1 ""])]


-- Test 4
db4   = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = [ (P "vim", (False, minV, V [VN 12 "",VN 0 ""]))
                         , (P "bar", (True, minV, maxV))]
            }
          , Pkg { name = P "bar", ver = V [VN 2 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          , Pkg { name = P "vim", ver = V [VN 11 "",VN 1 "",VN 2 ""]
                , desc = "neovim"
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "vim", ver = V [VN 8 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "vim",(True,V [VN 3 "",VN 4 ""],V [VN 9 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          ]
out4= [(pname3, V [VN 11 "", VN 1 "", VN 2 ""]),(P "foo", V [VN 2 "", VN 3
    ""]),(P "bar", V [VN 2 "", VN 1 ""])]


-- Test 5
db5 = DB []

-- Test 6
db6   = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = [ (P "vim", (False, minV, V [VN 12 "",VN 0 ""]))
                         , (P "bar", (True, minV, maxV))]
            }
```

```
          , Pkg { name = P "bar", ver = V [VN 2 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          , Pkg { name = P "vim", ver = V [VN 11 "",VN 1 "",VN 2 ""]
                , desc = "neovim"
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "vim", ver = V [VN 8 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "vim",(True,V [VN 3 "",VN 4 ""],V [VN 9 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
        ]
pname6 = P "none"


-- Test 7
db7  = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
                , desc = "The foo application"
                , deps = [ (P "vim", (False, minV, V [VN 12 "",VN 0 ""]))
                         , (P "bar", (True, minV, maxV))]
            }
          , Pkg { name = P "bar", ver = V [VN 2 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          , Pkg { name = P "vim", ver = V [VN 11 "",VN 1 "",VN 2 ""]
                , desc = "neovim"
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "vim", ver = V [VN 8 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "vim",(True,V [VN 3 "",VN 4 ""],V [VN 9 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
        ]
pname7 = P "baz"
out7= [(pname7, V [VN 4 "", VN 1 "", VN 2 ""]),(P "bar", V [VN 2 "", VN 1 "
    "])]


-- Test 8
db8  = DB [ Pkg { name = P "foo", ver = V [VN 2 "",VN 3 ""]
```

```
                , desc = "The foo application"
                , deps = [ (P "vim", (False, minV, V [VN 12 "",VN 0 ""]))
                         , (P "bar", (True, minV, maxV))]
            }
          , Pkg { name = P "bar", ver = V [VN 7 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          , Pkg { name = P "vim", ver = V [VN 11 "",VN 1 "",VN 2 ""]
                , desc = "neovim"
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "baz", ver = V [VN 4 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "bar",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "vim", ver = V [VN 8 "",VN 1 "",VN 2 ""]
                , desc = ""
                , deps = [(P "foo",(True,V [VN 0 "",VN 4 ""],V [VN 5 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "beef", ver = V [VN 1 "",VN 0 ""]
                , desc = "The beef library"
                , deps = [(P "vim",(True,V [VN 3 "",VN 4 ""],V [VN 9 "",VN 0
    "",VN 3 ""]))]
            }
          , Pkg { name = P "bar", ver = V [VN 1 "",VN 1 ""]
                , desc = "The bar library, new API"
                , deps = []
            }
          ]
out8= [(pname7, V [VN 4 "", VN 1 "", VN 2 ""]),(P "bar", V [VN 1 "", VN 1 "
    "])]
```

A.2.4 Source code for /handin/appm/tests/BB/Main.hs

```
module Main where

-- Put your black-box tests in this file

import Test.Tasty
import Test.Tasty.HUnit

import Defs
import qualified ParserTest as Parser
import qualified UtilTest   as Util
import qualified SolverTest as Solver

-- just a sample; feel free to replace with your own structure
tests = testGroup "Unit tests"
  [ Util.tests
  , Parser.tests
  , Solver.tests
  ]

main = defaultMain tests
```

A.3 QuickCheck listings

A.3.1 Source code for /handin/appm/tests/QC/Gen.hs

```
module Gen where

import Defs
import Test.Tasty
import Test.Tasty.QuickCheck

import Control.Monad (foldM, (<=<))
import Utils (sortPkgs, merge)

```

```
import Data.List (nub,nubBy,delete)


-- characters
lowers = ['a'..'z']
uppers = ['A'..'Z']
digits = ['1'..'9']
letters   = lowers ++ uppers
alphanums = letters ++ digits


-- package names
instance Arbitrary PName where
  arbitrary = P <$> simple

-- we only generate simple names
simple  = do
  first <- elements lowers
  after <- resize 7 $ listOf $ frequency
      [ (5, do { l <- elements letters;
                 return [l]           })
      , (2, do { l <- elements digits;
                 return [l]           })
      , (1, do { l <- elements alphanums;
                 return ['-',l]       })
      ]
  return $ first : concat after


-- versions numbers
instance Arbitrary VNum where
  arbitrary = do
    i <- sized $ \n -> choose (1,n)
    suf <- frequency [(5, return ""),(1,lowerString4)]
    return $ VN i suf

    where lowerString4 = resize 4 (listOf1 (elements lowers))


-- actual versions
instance Arbitrary Version where
  arbitrary = V <$> resize 5 (listOf1 arbitrary)


-- packages
instance Arbitrary Pkg where
  arbitrary = do
    v <- frequency [(4,arbitrary),(1,return stdV)]
    desc <- frequency [ (3, return "")
                      , (1,take <$> choose (1,15)
                                <*> listOf1 arbitraryASCIIChar) ]
    return $ Pkg (P "") v desc []
  shrink (Pkg p v des dep) = Pkg p v des <$> shrink dep


-- the database
instance Arbitrary Database where
  arbitrary = DB . sortPkgs
          <$> ( genAllDeps
            <=< trimDatabase
            <=< genNames
            <=< listOf1
           ) arbitrary
  shrink (DB db) = DB <$> shrink db

-- generate names for packages
genNames :: [Pkg] -> Gen [Pkg]
genNames pkgs = do
  validPNames <- nub <$> vectorOf (length pkgs) arbitrary
  mapM (\pkg -> do
         name' <- elements validPNames
         return $ pkg{name = name'}
       ) pkgs

```

```
-- remove duplicates
trimDatabase :: [Pkg] -> Gen [Pkg]
trimDatabase (pkg:pkgs) =
  let rest = filter (\pkg' -> name pkg /= name pkg' ||
                              ver pkg /= ver pkg'
                    ) pkgs
  in (:) pkg <$> trimDatabase rest
trimDatabase [] = return []

-- generating dependencies
genAllDeps :: [Pkg] -> Gen [Pkg]
genAllDeps pkgs =
  mapM (\pkg -> do
     cs <- pkg `genAvailDeps` pkgs
     return $ pkg{deps = cs}
  ) pkgs

genAvailDeps :: Pkg -> [Pkg] -> Gen Constrs
genAvailDeps pkg pkgs = do
      n <- choose (0,4)
      pkgs' <- take n <$> sublistOf (filter (\pkg' -> name pkg' /= name pkg
    ) pkgs)
      cs <- mapM (\pkg' -> do
          b <- arbitrary
          (vmin,vmax) <- genRange (ver pkg')
          return (name pkg',(b,vmin,vmax))
        ) pkgs'
      case mergeMultiple cs of
        Nothing     -> return []
        Just merged -> return merged

-- merge individual constraints in a constraint list
mergeMultiple :: Constrs -> Maybe Constrs
mergeMultiple [] = Just []
mergeMultiple (c:cs) = foldM (\cs' c' -> merge cs' [c']) [] cs

-- generate version range
genRange :: Version -> Gen (Version, Version)
genRange v = frequency [(4,do
      vmin <- oneof [return minV, suchThat arbitrary (\v' -> v' <= v && v'
    >= minV)]
      vmax <- if vmin == minV then vmax' else oneof [return maxV, vmax']
      return (vmin,vmax))
    , (1, suchThat arbitrary (uncurry (<)))]

    where vmax' = suchThat arbitrary (\v' -> v' > v && v' < maxV)
```

A.3.2 Source code for /handin/appm/tests/QC/Properties.hs

```
module Properties where

import Defs
import Utils
import Parser
import Solver

import Control.Monad (foldM,mapM)

import Data.List (partition, delete, sort)

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

----------------------
-- Install properties
----------------------

-- a) all packages (with the indicated versions) are actually available in
    the
-- database
install_a :: InstallProp
install_a _ _ Nothing    = True
install_a _db _ (Just sol) = all (`isInDB` _db ) sol


```

```
25 -- b) any package name may only occur once in the list; in particular, it
       is not
26 -- possible to install two different versions of the same package
27 -- simultaneously
28 install_b :: InstallProp
29 install_b _ _ Nothing = True
30 install_b _db _p (Just sol) =
31   let groups = groupByName sol
32   in not . any (\group -> length group > 1) $ groups
33
34
35 -- c) the package requested by the user is in the list
36 install_c :: InstallProp
37 install_c _ _ Nothing = True
38 install_c _db _p (Just sol) = any (\(p,_) -> _p == p) sol
39
40
41 -- d) for any package in the list, all the packages it requires are also in
       the
42 -- list
43 install_d :: InstallProp
44 install_d _ _ Nothing = True
45 install_d _db _p (Just sol) =
46   all (\pv -> case getReqs pv _db of
47           Nothing   -> False
48           Just reqs -> all (`isReqInSol` sol) reqs
49       ) sol
50
51
52 -- e) for any package in the list, all other packages in the list should
53 -- satisfy it
54 install_e _ _ Nothing = True
55 install_e _db _p (Just sol) =
56   all (\pv -> sat (delete pv sol) pv _db) sol
57
58 -- does the solution satisfy the constraints of a given package?
59 sat :: Sol -> (PName, Version) -> Database -> Bool
60 sat sol pv db = case getReqs pv db of
61   Nothing -> False
62   Just cs -> sol `satisfies` cs
63
64
65 -- f) you should not be able to remove a package without breaking
       consistency
66 install_f _ _ Nothing = True
67 install_f _db _p (Just sol) =
68   all (\pv -> fst pv == _p || -- dont remove the package we want to install
69       let sol' = delete pv sol
70       in case genConstrs _db sol' of
71         Nothing -> True
72         Just cs -> not $ sol' `satisfies` cs
73     ) sol
74
75
76 -- g) you should not be able to replace a package with a newer version
       without
77 -- breaking consistency
78 install_g _ _ Nothing = True
79 install_g _db _p (Just sol) =
80   all (\pv -> case getNewerVers pv _db of
81       []   -> True
82       pkgs ->
83         let sol' = delete pv sol
84         in all (\pv -> let sol'' = pv:sol' in
85               case genConstrs _db sol'' of
86                 Nothing -> True
87                 Just cs -> not $ sol'' `satisfies` cs ) pkgs
88       ) sol -- so sorry for this layout
89
90
91 --------------------
92 -- Parser properties
93 --------------------
94
```

```haskell
parses_db _db = case parseDatabase (prettyDB _db) of
  Right db -> db `dbEquiv` _db
  Left _ -> False

dbEquiv :: Database -> Database -> Bool
dbEquiv (DB db1) (DB db2) =
  length db1 == length db2 &&
    all (\(pkg1,pkg2) ->
            name pkg1 == name pkg2 &&
            ver pkg1 == ver pkg2   &&
            desc pkg1 == desc pkg2 &&
            sort (deps pkg1) == sort (deps pkg2)
        ) (zip (sort db1) (sort db2))


--------------------
-- Utility functions
--------------------

-- group solution tuples by name
groupByName :: [(PName,Version)] -> [[(PName,Version)]]
groupByName [] = []
groupByName ((p,v):sol) =
  let (g,r) = partition ((p ==) . fst) sol
  in ((p,v):g) : groupByName r

-- is a given package name and version in the database?
isInDB :: (PName,Version) -> Database -> Bool
isInDB (p,v) (DB db) = any (\pkg -> name pkg == p && ver pkg == v) db

-- is a given package required?
isReqInSol :: (PName,PConstr) -> Sol -> Bool
isReqInSol (p,(b, vmin, vmax)) sol | b =
  any (\(p',v) -> p' == p  && v >= vmin && v < vmax) sol
isReqInSol _ _ = True

-- get requirements of a single package
getReqs :: (PName,Version) -> Database -> Maybe Constrs
getReqs _ (DB []) = Nothing
getReqs (p,v) (DB (pkg':ps))
  | name pkg' == p && ver pkg' == v =
      Just $ deps pkg'
getReqs (p,v) (DB (_:ps)) = getReqs (p,v) (DB ps)

-- generate constraints from a solution
genConstrs :: Database -> Sol -> Maybe Constrs
genConstrs db sol =
  foldM merge [] =<< mapM (`getReqs` db) sol

-- get a list of newer versions of a given package
getNewerVers :: (PName,Version) -> Database -> [(PName,Version)]
getNewerVers _ (DB []) = []
getNewerVers (p,v) (DB (pkg':ps))
  | name pkg' == p && ver pkg' > v =
      (name pkg', ver pkg') : getNewerVers (p,v) (DB ps)
getNewerVers pv (DB (_:ps)) = getNewerVers pv (DB ps)
```

A.3.3 Source code for /handin/appm/tests/QC/Main.hs

```haskell
module Main where

import Gen

import Defs
import Properties
import Solver (install)

import Test.Tasty
import Test.Tasty.QuickCheck


----------
-- Testing
----------
```

```
16
17 prop_install_a (DB db) = let pkg = head db in install_a (DB db) (name pkg)
       (install (DB db) $ name pkg)
18 prop_install_b (DB db) = let pkg = head db in install_b (DB db) (name pkg)
       (install (DB db) $ name pkg)
19 prop_install_c (DB db) = let pkg = head db in install_c (DB db) (name pkg)
       (install (DB db) $ name pkg)
20 prop_install_d (DB db) = let pkg = head db in install_d (DB db) (name pkg)
       (install (DB db) $ name pkg)
21 prop_install_e (DB db) = let pkg = head db in install_e (DB db) (name pkg)
       (install (DB db) $ name pkg)
22 prop_install_f (DB db) = let pkg = head db in install_f (DB db) (name pkg)
       (install (DB db) $ name pkg)
23 prop_install_g (DB db) = let pkg = head db in install_g (DB db) (name pkg)
       (install (DB db) $ name pkg)
24
25 prop_parse_db  = parses_db
26
27 tests = testGroup "QC tests"
28           [ testProperty "prop a"  prop_install_a
29           , testProperty "prop b"  prop_install_b
30           , testProperty "prop c"  prop_install_c
31           , testProperty "prop d"  prop_install_d
32           , testProperty "prop e"  prop_install_e
33           , testProperty "prop f"  prop_install_f
34           , testProperty "prop g"  prop_install_g
35           , testProperty "parsing" prop_parse_db
36           ]
37
38 main = defaultMain tests
```

A.4 Other

A.4.1 Automatically generated database

```
1  *Gen> generate (arbitrary :: Gen Database)
2  package {
3    name w2u-9-b;
4    version 1u.5i.1.2r;
5    description "B";
6  }
7
8  package {
9    name vkr;
10   version 3.4.5.1;
11   requires lj >= 0;
12   requires lj < 5;
13 }
14
15 package {
16   name vkr;
17 }
18
19 package {
20   name tB7i1nw;
21   conflicts lj < 4;
22   conflicts lj >= 4.4.2.2;
23   conflicts q < 0;
24   conflicts q >= 4.5.3.5;
25   conflicts vkr < 0;
26   conflicts vkr >= 3c.1.5.2;
27 }
28
29 package {
30   name sMx5Fn;
31   requires kGLSyl >= 0;
32   requires kGLSyl < 3un;
33 }
34
35 package {
36   name q;
37   version 3.3.5.5.2;
```

```
  requires lj >= 2hq.1;
  requires lj < 1000000;
}

package {
  name q;
  version 1.3uo.4.3.4;
  description "?9g
!q<%;2fv";
}

package {
  name lj;
  version 4.3;
}

package {
  name kGLSyl;
}

package {
  name di;
  version 5.2dxd.3.1.1;
  description " kG^j^";
  requires vkr >= 0;
  requires vkr < 3.1.3;
  conflicts sMx5Fn < 1.5.1.4;
  conflicts sMx5Fn >= 5.5hn;
  conflicts tB7i1nw < 0;
  conflicts tB7i1nw >= 3mx.3yram;
}
```

B Ravnica listings

B.1 Source listings

B.1.1 Source code for /handin/ravnica/district.erl

```
-module(district).
-behaviour(gen_statem).

% API exports
-export([create/1,
         get_description/1,
         connect/3,
         activate/1,
         options/1,
         enter/2,
         take_action/3,
         shutdown/2,
         trigger/2]).

% Callback exports
-export([init/1, callback_mode/0, code_change/4, terminate/3]).

% State exports
-export([under_configuration/3, under_activation/3, active/3, shutting_down/3]).

% types
-type passage() :: pid().
-type creature_ref() :: reference().
-type creature_stats() :: map().
-type creature() :: {creature_ref(), creature_stats()}.
-type trigger() :: fun((entering | leaving, creature(), [creature()]) -> {creature(), [creature()]}).


%%%===================================================================
%%% API (State must be a neighbours/creatures/trigger/description tuple)
%%%===================================================================

-spec create(string()) -> {ok, passage()} | {error, any()}.
create(Desc) -> gen_statem:start(?MODULE, Desc, []).

-spec get_description(passage()) -> {ok, string()} | {error, any()}.
get_description(District) -> gen_statem:call(District, get_description).

-spec connect(passage(), atom(), passage()) -> ok | {error, any()}.
connect(From, Action, To) -> gen_statem:call(From, {connect, Action, To}).

-spec activate(passage()) -> active | under_activation | impossible.
activate(District) -> activate1(District, []).
activate1(District, Visited) -> gen_statem:call(District, {activate, Visited}).

-spec options(passage()) -> {ok, [atom()]} | none.
options(District) -> gen_statem:call(District, options).

-spec enter(passage(), creature()) -> ok | {error, any()}.
enter(District, Creature) -> gen_statem:call(District, {enter, Creature}).

-spec take_action(passage(), creature_ref(), atom()) -> {ok, passage()} | {error, any()}.
take_action(From, CRef, Action) -> gen_statem:call(From, {take_action, CRef, Action}).

-spec shutdown(passage(), pid()) -> ok.
shutdown(District, NextPlane) -> shutdown1(District, NextPlane, []).
shutdown1(District, NextPlane, Visited) ->
  gen_statem:call(District, {shutdown, NextPlane, Visited}),
  gen_statem:stop(District).

```

```
-spec trigger(passage(), trigger()) -> ok | {error, any()} | not_supported.
trigger(District, Trigger) -> gen_statem:call(District, {trigger, Trigger}).



%%%==========================================================================
%%% Callback functions
%%%==========================================================================

callback_mode() -> state_functions.

init(Desc) -> {ok, under_configuration, {#{},#{},fun (_,C,Cs) -> {C,Cs} end,Desc}}.

terminate(_Reason, _State, _Data) -> void.

code_change(_Vsn, State, Data, _Extra) -> {ok, State, Data}.



%%%==========================================================================
%%% State functions
%%%==========================================================================

% Under configuration

% simply return the description
under_configuration({call, From}, get_description, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, D}}]};

% insert the (atom,To) pair into the map over neighbours if possible
under_configuration({call, From}, {connect, Action, To}, {N,C,T,D}) ->
  case maps:is_key(Action, N) of
    true  -> {keep_state, {N,C,T,D}, [{reply, From, {error, action_already_exists}}]};
    false -> {keep_state, {maps:put(Action, To, N),C,T,D}, [{reply, From, ok}]}
  end;

% activate each neighbour and wait for a response
% if any response is impossible, don't activate
under_configuration({call, From}, {activate, Visited}, Data) ->
  {next_state, under_activation, Data, [{next_event, internal, {From, activate,[self()|Visited]}}]};

% simply return the keys of the (atom,To) pairs in the map over neighbours
under_configuration({call, From}, options, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, maps:keys(N)}}]};

% replace the current trigger in the test data
under_configuration({call, From}, {trigger, Trigger}, {N,C,_,D}) ->
  {keep_state, {N,C,Trigger,D}, [{reply, From, ok}]};

% call the shut_down helper function, see below
under_configuration({call, From}, {shutdown, NextPlane, Visited}, {N,C,T,D}) ->
  NextPlane ! {shutting_down, self(), maps:to_list(C)},
  {next_state, shutting_down, {N,C,T,D}, [{next_event, internal, {From, shutdown, NextPlane, [self()|Visited]}}]};

% handle other events generically
under_configuration({call, From}, _, Data) ->
  {keep_state, Data, [{reply, From, {error, not_valid}}]};
under_configuration(_, _, _) ->
  keep_state_and_data.



% Under activation

% simply return the state
under_activation({call, From}, {activate,_}, Data) ->
  {keep_state, Data, [{reply, From, under_activation}]};
```

```
% activate neighbours that haven't been visited
under_activation(internal, {From, activate, Visited}, {N,C,T,D}) ->
  Ns = lists:filter(fun (To) -> not(lists:member(To,Visited)) end, maps:values(N)),
  Act = lists:map(fun (To) -> activate1(To,Visited) end, Ns),
  case lists:any(fun (A) -> A == impossible end, Act) of
    true  -> {next_state, under_configuration, {N,C,T,D}, [{reply, From, impossible}]};
    false -> {next_state, active, {N,C,T,D}, [{reply, From, active}]}
  end;

% same as in under_configuration
under_activation({call, From}, options, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, maps:keys(N)}}]};

% handle other events generically
under_activation({call, From}, _, Data) ->
  {keep_state, Data, [{reply, From, {error, not_valid}}]};
under_activation(_, _, _) ->
  keep_state_and_data.



% Active

% same as in under_configuration
active({call, From}, get_description, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, D}}]};

% if already active, simply return this atom
active({call, From}, {activate,_}, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, active}]};

% same as in under_configuration
active({call, From}, options, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, maps:keys(N)}}]};

% if the creature is not in the district, insert it and apply the trigger
active({call, From}, {enter, {CRef,CStat}}, {N,C,T,D}) ->
  case maps:is_key(CRef, C) of
    true  -> {keep_state, {N,C,T,D}, [{reply, From, {error, ref_already_exists}}]};
    false ->
      {{CRef,CStat1},Cs1} = apply_trigger(T,entering,{CRef,CStat},maps:to_list(C)),
      {keep_state, {N,maps:put(CRef,CStat1,maps:from_list(Cs1)),T,D}, [{reply, From, ok}]}
  end;

% check if the given action and the given creature exist
% if so, apply the trigger and move it if possible
% if it's a connection to the same district, apply the trigger locally
% and return as to avoid blocking
active({call, From}, {take_action, CRef, Action}, {N,C,T,D}) ->
  case C of
    #{CRef := CStat} ->
      case N of
        #{Action := To} ->
          {C1,Cs1} = apply_trigger(T,leaving,{CRef,CStat},maps:to_list(maps:remove(CRef, C))),
          case To == self() of
            true -> % going back to the same district, simply apply the trigger locally
              {{CRef,CStat2},Cs2} = apply_trigger(T,entering,C1,Cs1),
              {keep_state, {N,maps:put(CRef, CStat2, maps:from_list(Cs2)),T,D}, [{reply, From, {ok, To}}]};
            false -> % else, it will be handled when entering the other district
              case enter(To, C1) of
                {error, Reason} -> {keep_state, {N,C,T,D}, [{reply, From, {error, Reason}}]};
                ok              -> {keep_state, {N,maps:from_list(Cs1),T,D}, [{reply, From, {ok, To}}]}
              end
```

```
          end;
        #{} -> {keep_state, {N,C,T,D}, [{reply, From, {error,
    no_such_action}}]}
      end;
    #{}  -> {keep_state, {N,C,T,D}, [{reply, From, {error, no_such_creature
    }}]}
  end;

% same as in under_configuration
active({call, From}, {shutdown, NextPlane, Visited}, {N,C,T,D}) ->
  NextPlane ! {shutting_down, self(), maps:to_list(C)},
  {next_state, shutting_down, {N,C,T,D}, [{next_event, internal, {From,
    shutdown, NextPlane, [self()|Visited]}}]};

% handle other events generically
active({call, From}, _, Data) ->
  {keep_state, Data, [{reply, From, {error, not_valid}}]};
active(_, _, _) ->
  keep_state_and_data.

% applying the trigger: spawn a process and await a response
% if not well-formed, return the input data
apply_trigger(T, Event, {CRef,CStat}, Cs) ->
  Me = self(),
  Pid = spawn(fun () -> Me ! {self(), check_trigger(T,Event,{CRef,CStat},Cs
    )} end),
  receive
    {Pid,{{CRef,CStat1},Cs1}} ->
      case same_creatures(Cs1,Cs) of
        true  -> {{CRef,CStat1},Cs1};
        false -> {{CRef,CStat},Cs}
      end;
    {Pid, _} -> {{CRef,CStat},Cs}
  after
    2000 -> {{CRef,CStat}, Cs}
  end.

% checking the trigger
check_trigger(T,Event,C,Cs) ->
  try T(Event,C,Cs) of
    Res -> Res
  catch
    _:_ -> error
  end.

% check if the creatures are the same
same_creatures(Cs1, Cs2) ->
  case is_list(Cs1) and is_list(Cs1) of
    true ->
      Cs11 = lists:map(fun ({Ref,_}) -> Ref;
                           (Other) -> Other
                       end, Cs1),
      Cs22 = lists:map(fun ({Ref,_}) -> Ref;
                           (Other) -> Other
                       end, Cs2),
      lists:sort(Cs11) == lists:sort(Cs22);
    false -> false
  end.



% shutting down

% same as in under_configuration
shutting_down({call, From}, get_description, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, {ok, D}}]};

% impossible to active district that is shutting down
shutting_down({call, From}, {activate,_}, Data) ->
  {keep_state, Data, [{reply, From, impossible}]};

% no options when shutting down
shutting_down({call, From}, options, {N,C,T,D}) ->
  {keep_state, {N,C,T,D}, [{reply, From, none}]};
```

```
262
263 % already shutting down, so whatever
264 shutting_down({call, From}, {shutdown, _, _}, Data) ->
265   {keep_state, Data, [{reply, From, ok}]};
266 % shut down neighbours if they haven't already been stopped
267 shutting_down(internal, {From, shutdown, NextPlane, Visited}, {N,C,T,D}) ->
268   Ns = lists:filter(fun (To) -> not(lists:member(To,Visited)) end, maps:
      values(N)),
269   lists:foreach(fun (To) ->
270                     case process_info(To) of
271                       undefined -> ok;
272                       _ -> shutdown1(To,NextPlane,Visited)
273                     end
274                 end, Ns),
275   {keep_state, {N,C,T,D}, [{reply, From, ok}]};
276
277 % handle other events generically
278 shutting_down({call, From}, _, Data) ->
279   {keep_state, Data, [{reply, From, {error, not_valid}}]};
280 shutting_down(_, _, _) ->
281   keep_state_and_data.
```

B.1.2 Source code for /handin/ravnica/the_diamond_path.erl

```
1  -module(the_diamond_path).
2  -export([a_love_story/0]).
3
4  a_love_story() ->
5    % Defining a dimond-shaped territory.
6    {ok, A} = district:create("A"),
7    {ok, B} = district:create("B"),
8    {ok, C} = district:create("C"),
9    {ok, D} = district:create("D"),
10   district:connect(A, b, B),
11   district:connect(A, c, C),
12   district:connect(B, d, D),
13   district:connect(C, d, D),
14
15   % Activating the districts.
16   % Since there is a path from A to every other district, this will suffice
       :
17   district:activate(A),
18
19   % Two players without stats.
20   {BobRef, _}   = Bob   = {make_ref(), #{}},
21   {AliceRef, _} = Alice = {make_ref(), #{}},
22
23   % Bob and Alice entered the same district:
24   district:enter(A, Bob),
25   district:enter(A, Alice),
26
27   % But on that day, they choose to follow different paths.
28   district:take_action(A, BobRef, b),
29   district:take_action(A, AliceRef, c),
30
31   % But fortunately, there is no way to get lost in the diamond path.
32   district:take_action(B, BobRef, d),
33   district:take_action(C, AliceRef, d),  % <-------------- | changed in ver.
       1.0.1 |
34   A.
35
36 % THE END
```

B.1.3 Source code for /handin/ravnica/a_day_at_diku.erl

```
1 % Example contributed by Joachim and Mathias
2 -module(a_day_at_diku).
3 -export([run_world/0]).
4
5
6 make_drunker({CreateRef, Stats}) ->
7     #{sobriety := CurSobriety} = Stats,
8     {CreateRef, Stats#{sobriety := CurSobriety - 1}}.
```

```
9
10 make_sober({CreateRef, Stats}) ->
11     #{sobriety := CurSobriety} = Stats,
12     {CreateRef, Stats#{sobriety := CurSobriety + 1}}.
13
14 cheers(_, Creature, Creatures) ->
15     io:format("Cheeeeers!~n"),
16     {make_drunker(Creature), lists:map(fun make_drunker/1, Creatures)}.
17
18 rest_a_bit(entering, Creature, Creatures) ->
19     {make_sober(Creature), Creatures};
20 rest_a_bit(leaving, Creature, Creatures) ->
21     {Creature, Creatures}.
22
23 andrzejs_office(entering, {CreatureRef, Stats}, Creatures) ->
24     io:format("You get lost in Andrzejs stacks of papers, lose 1 sanity!~n"
       ),
25     #{sanity := CurSanity} = Stats,
26     {{CreatureRef, Stats#{sanity := CurSanity - 1}}, Creatures};
27 andrzejs_office(leaving, Creature, Creatures) ->
28     {Creature, Creatures}.
29
30 lille_up1(entering, {CreatureRef, Stats}, Creatures, KenRef, AndrzejRef) ->
31     CreatureRefs = lists:map(fun({Ref, _Stats}) -> Ref end, Creatures),
32     KenPresent = lists:member(KenRef, CreatureRefs),
33     AndrzejPresent = lists:member(AndrzejRef, CreatureRefs),
34     if KenPresent and AndrzejPresent ->
35         {{CreatureRef, Stats#{stunned => true}}, Creatures};
36        true ->
37         {{CreatureRef, Stats}, Creatures}
38     end;
39 lille_up1(leaving, _Creature, _Creatures, _KenRef, _AndrzejRef) ->
40     % This is misbehaving, thus the trigger has no effect
41     ok.
42
43 generate_territory() ->
44     {ok, KensOffice} = district:create("Ken's office"),
45     {ok, AndrzejsOffice} = district:create("Andrzej's office"),
46     {ok, CoffeeMachine} =
47         district:create("The Coffee Machine at the end of the PLTC hallway"
       ),
48     {ok, Canteen} =
49         district:create("The Canteen at the top floor of the DIKU building"
       ),
50     {ok, Cafeen} = district:create("The student bar, \"Cafeen?\""),
51     {ok, Bathroom} = district:create("The bathroom at the student bar"),
52     {ok, LilleUP1} =
53         district:create("The smaller auditorium at the DIKU building"),
54
55     ok = district:connect(KensOffice, restore_health, CoffeeMachine),
56     ok = district:connect(AndrzejsOffice, prepare_attack, CoffeeMachine),
57
58     % Andrzej sometimes skips his coffee
59     ok = district:connect(AndrzejsOffice, sneak, LilleUP1),
60
61     ok = district:connect(CoffeeMachine, surprise_attack, LilleUP1),
62     ok = district:connect(Canteen, make_haste, Cafeen),
63     ok = district:connect(Canteen, have_courage, LilleUP1),
64
65     ok = district:connect(Cafeen, try_to_leave, Cafeen),
66     ok = district:connect(Cafeen, need_to_pee, Bathroom),
67     ok = district:connect(Bathroom, go_back, Cafeen),
68
69     % Places to spawn or place advanced triggers
70     [KensOffice, AndrzejsOffice, CoffeeMachine, Canteen, Bathroom,
71      Cafeen, LilleUP1].
72
73 place_triggers(KenRef, AndrzejRef, AndrzejsOffice, Cafeen,
74                Bathroom, LilleUP1) ->
75     district:trigger(AndrzejsOffice, fun andrzejs_office/3),
76     district:trigger(Cafeen, fun cheers/3),
77     district:trigger(Bathroom, fun rest_a_bit/3),
78     district:trigger(LilleUP1,
79             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),



    % =====| Following two lines changed in ver. 1.0.1 | =====
    {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
    % =====| Following two lines changed in ver. 1.0.1 | =====
    {ok, _} = district:take_action(Canteen, hd(StudentRefs), have_courage),
    {ok, _} = district:take_action(CoffeeMachine, KenRef, surprise_attack),

    {KensOffice, AndrzejsOffice, Canteen}.
```

B.2 QuickCheck listings

B.2.1 Source code for /handin/ravnica/district_qc.erl

```
-module(district_qc).

-export([territory/0, setup_territory/1]).
-export([prop_activate/0, prop_shutdown/0, prop_take_action/0]).

-include_lib("eqc/include/eqc.hrl").


%%%===================================================================
%%% Generators
%%%===================================================================

atom()      -> elements([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,x,y,z])
    .
```

```
key()       -> ?SIZED(Size,choose(0,Size)).
creature() -> {make_ref(), #{}}.

% Generating the map
territory() ->
  ?LET(L, list({key(),list({atom(),key()})}),
       make_nonexisting_neighbours(maps:from_list(L))).

% Make sure that all neighbours exist
make_nonexisting_neighbours(Map) ->
  NMap = maps:map(fun (_,V) -> maps:from_list(V) end, Map),
  Fun = fun (_,V,Acc) -> case maps:is_key(V,NMap) of
                           true -> Acc;
                           false -> [V|Acc]
                         end
        end,
  NotInMap = maps:fold(fun (_,V,Acc) ->
                           lists:append( Acc
                                       , maps:fold(Fun, [], V))
                       end, [], NMap),
  lists:foldl(fun (K,Acc) -> maps:put(K, #{}, Acc) end, NMap, NotInMap).


% Setting up the territory
setup_territory(Map) ->
  Ds = maps:map(fun (K,_) ->
                    {ok, District} = district:create(integer_to_list(K)),
                    District
                end, Map),
  maps:map(fun (K,V) ->
               maps:map(fun (K1,V1) ->
                            district:connect(maps:get(K,Ds),K1,maps:get(V1,
    Ds))
                        end, V)
           end, Map),
  maps:values(Ds).


%%%====================================================================
%%% Properties
%%%====================================================================

% Activation property
prop_activate() ->
    ?FORALL(T, ?SUCHTHAT(T1, territory(), maps:size(T1) > 0),
    ?LET(T1, setup_territory(T),
    ?FORALL(D, elements(T1),
    ?LET({ok,O}, district:options(D),
    ?IMPLIES(length(O) > 0,
      ?IMPLIES(district:activate(D) == active,
        ?FORALL(Act, elements(O),
        ?LET({CRef,CStat}, creature(),
        ?IMPLIES(district:enter(D,{CRef,CStat}) == ok,
          ?LET({ok,To}, district:take_action(D,CRef,Act),
          is_active(To)
    ))))))))).

% use system call to check if active
is_active(D) ->
  case sys:get_state(D) of
    {active, _} -> true;
    _ -> false
  end.


% Shutdown property
prop_shutdown() ->
    ?FORALL(T, ?SUCHTHAT(T1, territory(), maps:size(T1) > 0),
    ?LET(T1, setup_territory(T),
    ?FORALL(D, elements(T1),
    ?LET({ok,O}, district:options(D),
    ?IMPLIES(length(O) > 0,
      ?IMPLIES(district:activate(D) == active,
        ?FORALL(Act, elements(O),
```

```
        ?LET({CRef,CStat}, creature(),
        ?IMPLIES(district:enter(D,{CRef,CStat}) == ok,
          ?LET({ok,To}, district:take_action(D,CRef,Act),
          ?IMPLIES(district:shutdown(D,self()) == ok,
            (is_shut_down(D)) and (is_shut_down(To))
    ))))))))))).

is_shut_down(D) ->
  case process_info(D) of
    undefined -> true;
    _         -> false
  end.


% Take_action property
prop_take_action() ->
    ?FORALL(T, ?SUCHTHAT(T1, territory(), maps:size(T1) > 0),
    ?LET(T1, setup_territory(T),
    ?FORALL(D, elements(T1),
    ?LET({ok,O}, district:options(D),
    ?IMPLIES(length(O) > 0,
      ?IMPLIES(district:activate(D) == active,
        ?FORALL(Act, elements(O),
        ?LET({CRef,CStat}, creature(),
        ?IMPLIES(district:enter(D,{CRef,CStat}) == ok,
          ?LET({ok,To}, district:take_action(D,CRef,Act),
          ?IMPLIES(To /= D,
            district:take_action(D,CRef,Act) /= ok
    ))))))))))).
```

B.3 Other

B.3.1 Automatically generated territory

```
#{ 0 =>  #{k => 17,o => 13,y => 25},
   2 =>  #{},
   5 =>  #{},
   7 =>  #{},
   8 =>  #{g => 10,y => 13},
   9 =>  #{b => 24,
           f => 5,
           g => 26,
           i => 17,
           j => 31,
           k => 24,
           l => 23,
           o => 12,
           r => 0,
           z => 7},
   10 => #{},
   11 => #{b => 7,
           c => 10,
           g => 12,
           j => 11,
           r => 7,
           x => 18},
   12 => #{},
   13 => #{},
   14 => #{},
   17 => #{a => 8,
           e => 0,
           f => 14,
           i => 2,
           j => 2,
           l => 13,
           o => 29,
           q => 23,
           x => 20,
           y => 29},
   18 => #{},
   20 => #{v => 2},
   23 => #{},
```

```
    24 => #{},
    25 => #{},
    26 => #{},
    28 => #{p => 11,v => 17},
    29 => #{},
    31 => #{}}
```