FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Advanced Programming Exam

Exam number 16

2018/11/09

Contents

1 The appm Package Manager **1**

1.1 Chosen Parser Library 1

1.2 Grammar Changes 1

1.3 Utility functions 2

1.4 The Database Parser 3

1.5 Constraint Solving 3

1.6 Testing 5

1.7 Conclusion 5

2 Earls of Ravnica **6**

2.1 The District Module 6

2.2 Assumptions and Pitfalls 8

2.3 Testing 8

2.4 Conclusion 8

Appendices **9**

A Program Code **9**

A.1 The appm Package Manager 9

A.1.1 `Utils.hs` 9

A.1.2 `ParserImpls.hs` 13

A.1.3 `SolverImpls.hs` 19

A.2 Earls of Ravnica 21

A.2.1 `district.erl` 21

B Test Code **35**

B.1 The appm Package Manager 35

B.1.1 Unit tests (BB tests): `Main.hs` 35

B.1.2 QuickCheck properties: `Properties.hs` 38

B.1.3 QuickCheck tests: `Main.hs` 40

B.2 Earls of Ravnica 43

B.2.1 Unit tests: `unittests.erl` 43

B.2.2 Unit tests: `triggertest.erl` 49

B.2.3 QuickCheck tests: `district_qc.erl` 52

References **55**

1 The appm Package Manager

How to run tests: run the command `stack test` from within the `appm` directory to run both black-box tests and property-based tests. The code for this assignment can be found under the same directory, or in appendix A.1.

1.1 Chosen Parser Library

I have used **Parsec** as the framework for parsing the database, mainly because I have used it before and believe that an event such as the Advanced Programming exam is not the right place to spent unnecessary time on figuring out a new parser library. Parsec also features little or no backtracking if the grammar tokens are sufficiently decidable from their first symbol; this is more efficient in cases where a significant amount of backtracking would otherwise occur, at the cost of potentially having to rewrite the grammar.

1.2 Grammar Changes

The given grammar lacks certain properties that we desire. Namely, we want to be able to decide what nonterminal we should parse from the first symbol alone. I have left factorized and removed left recursion in the grammar using the techniques described in [1], resulting in the following new grammar:

Database	::=	ϵ
	|	Package Database
Package	::=	’package’ ’{’ Clauses ’}’
Clauses	::=	Clause Clauses’
	|	ϵ Clauses’
Clauses’	::=	’;’ Clauses Clauses’
	|	ϵ
Clause	::=	’name’ PName
	|	’version’ Version
	|	’description’ String
	|	’requires’ PList
	|	’conflicts’ PList
PList	::=	PItem PList’
PList’	::=	’,’ PItem PList’
	|	ϵ
PItem	::=	PName PItem’
PItem’	::=	’>=’ Version
	|	’<’ Version
Version	::=	(see assignment text)
PName	::=	(see assignment text)
String	::=	(see assignment text)

Table 1: The modified grammar after left factorization and removed left recursion.

The grammar still poses some problems with respect to the restrictions on the database and packages. For once, it is still grammatically allowed to have zero or several clauses that denote the name of the package, which is not well formed with respect to the semantic constraint that any package must have exactly one name. Thus, any such conflicts with the semantics of a database must still be addressed. I do so when parsing, but an alternative could have been to enforce such restrictions on the grammar directly.

1.3 Utility functions

Version Ordering

The `Version` instance of `Ord` is pretty straight forward: inspect the numerical part of the first elements of each version. If the order is still ambiguous, inspect any suffix; go on to the ensuing element in the version sequence. I informally tested on different versions and it seemed to work. Had the ordering been more complex, rigorous testing would have been appropriate.

Merging Constraint Sets

The merge implementation is a little more tricky. It looks as follows:

```
merge c1 [] = Just c1
merge [] c2 = Just c2
merge c1 c2 = foldM mergeFolder c2 c1
```

This obviously hides the juicy parts of the implementation, but notice that if any of the two sets are empty, the other one is returned. Otherwise, we fold over the first input with the second input as starting accumulator.

Packages that have constraints in both sets are resolved, if possible, by taking the highest value of the lower bounds, and the lowest value of the higher bounds. If the interval of these two is empty and the package is required in either of the two sets, then the merge is inherently unsatisfiable. Otherwise we modify the package with the new bounds and sets it as a requirement if either of the two sets required the package.

The difficulties was to correctly discern the allowed merges between e.g. two conflicting constraints on a package, in which case empty intervals are satisfiable, from disallowed merges where empty intervals are not satisfiable. Tests of merges are included in the test suite.

The input to merge is assumed to be well formed. For this reason, some extra checks have to be done in the parsing of packages when checking for well-formedness.

1.4 The Database Parser

There are two main aspects of this task. One is parsing the grammar itself into an Abstract Syntax Tree (AST). The other is to check the AST for semantic well-formedness. I have chosen to check for well-formedness after parsing each package of the database. If the check goes well, we continue parsing the database; otherwise the package is not well-formed and an error is returned at this point.

Figuring out how to parse the grammar, one major task was to parse clauses. Since a clause can be either one of five different expressions, and since the set of clauses can be empty, this provided some edge cases of concern. Namely, it should be possible to parse a sequence of '`;`', since this is a sequence of empty clauses. For similar reasons, including and excluding a final '`;`' are both viable in the grammar, which the parser should handle correctly. A part of my solution is to denote a data type `Clause`, which entails all the different clauses; this way, we get one sequence of all the package clauses, indifferent to the clause type.

Checking each package for well-formedness was more straight forward, using the utility function merge. Each package is inspected, ensuring that there is exactly one name, and so on. The constraints are merged together one by one into an accumulated constraint set, which may or may not be satisfiable. Because each constraint clause can be comma-separated constraints, this process is repeated on the constraints internally before merging them together. The specifics are probably easier understood by looking at the code.

1.5 Constraint Solving

The solver uses the list monad to keep track of all possible alternative ways to increment the partial solution. The idea is to try adding all packages that satisfy the first requirement in the constraints, given that the package does not conflict with the other constraints, nor that a package by the same name is in the partial solution. If the addition of the package to the partial solution gives a full solution, then we add that possibility to the possible outcomes. If not, we recursively solve the remaining constraints merged with the constraints of the added package, and with the new partial solution as input. If the recursion bottoms out, or the merge of of constraints is unsatisfiable, the outcome is simply thrown away.

I believe that the solver satisfies the properties (a) to (i) because of the following reasons:

(a) Since we only look at packages from the database, any solution must consist of a subset of these. Potentially, the initial requirements can be for packages that are not in the database, but the solver is then simply unable to locale packages that match the constraints, and so the solution is empty.

(b) before any package is considered to be added to the partial solution, the solver, checks that a package by the same name is not already in the partial solution. Thus, no package with the same name will be added twice in any possible solution.

Also, the initial call is done with the empty partial solution, and every time a requirement is fulfilled, it is removed from the constraints. Thus, no requirement will demand the same package twice, securing that each package is present at most one in the solution.

(c) The initial call by the user will call solve with the single constraint `[(pkgname, (True, minV, maxV))]`, where `pkgname` is the requested package. Thus any solution will have to fulfill this requirement, and so the requested package must be in any possible solution list.

(d) Thus property is true, since for each package that is added to a potential solution, its dependencies are added to the current constraints; thus any recursive call will have to find alternatives that satisfy these dependencies as well. Similarly, when we check if the package added to the partial solution now constitutes a full solution, it is done with respect to the input constraints AND the dependencies of the package.

(e) Thus must be the case for any feasible solution, since the constraints and the dependencies of the considered package are merged together in each call. We only consider the package any further if this merge yields satisfiable constraints. This property must hold for any satisfiable solution, and so it must also hold for this particular solution.

(f) The solution only considers adding a package if it is required. Thus no packages are added that are not required, and so no package can be removed without violating a requirement. This is true since no two requirements require the same package.

(g) The installed solution is the first of the possible solutions produced by the solver. The solver goes through packages in order from left to right. The normalization of the database ensures that packages are sorted in decreasing order from left to right, thus for any two potential solution candidates in the final outcome, the leftmost will carry newer/higher versions by this sorting, since these are considered and added by the solver first. Since the first is chosen, it must by this property have the newest versions possible.

(h) The top level solver, `install`, applies the solver to generate potential solutions. The first of these are then returned. The solver either returns an empty list, if no combination of packages can satisfy the constraints, or a non-empty list of potential ways to satisfy the constraints. Thus, the top level solver can just return the first element in the returned list, if the list is non-empty, or it can return nothing if the list is empty.

(i) For irrelevant packages, the solver only spends an amount of time proportional to how long it takes to decide whether a package is required, conflicts with constraints or is already in the partial solution. I will argue that this is necessary and not a drastic consequence to the running time, since its proportional to the length of the constraints and the partial solution.

A drastic effect would be if the solver, e.g. naively tries all possible subsets of the database as a candidate solution, which heavily depends on the database size.

1.6 Testing

Each major implemented function `merge`, `parseDatabase` and `solve` has their own unit tests. I have tried to cover some of the weirder cases on which any of those functions might misbehave. In addition, I generate QuickCheck tests, using an arbitrary class for `Database`. In order to make the generated results suitable, i.e. well-formed and with non-trivial solutions, I have used the predicate suchThat to enforce certain of these properties. If I had more time, I would also have created a 'pretty printer'/unparser for the database in order to test the parser more thoroughly. The parser is probably the one that I am most uncertain about, for this reason.

In addition, I have restricted the database to 16 packages, which is clearly a restriction on the generated database samples. The reason for this decision was to increase the probability of well formed samples and non-trivial (non-empty) solutions.

All four properties (a), (b), (c) and (d) have been implemented and tested.

1.7 Conclusion

The parser, installer and utility functions seem to work; the tests I have been able to come up with all passes, which is of some comfort with respect to correctness. It also satisfied all the required properties as far as I have been able to test. This gives some assurance, but might also simply reflect on my lack of imagination when doing tests.

The QuickCheck tests passes, but this might only mean that I have failed to generate databases that are diverse enough to potentially capture all property errors. In any case, it gives comfort in the correctness of the solver.

The code is well commented, but some of the monadic structures could definitely be more elegant. Still, it should be possible to understand what is going on. The main parser is a combination of many smaller parsers, which might themselves be combinations yet again. Each parser roughly corresponds to a specification in the grammar, which should help understanding the structure. The main parser is located first in the code, but the ordering of the remaining parsers could be more structured. I tried to place parsers in their area of use, but this has not always been possible. The same can be said about the arbitrary generators in the QuickCheck code.

Code for removing whitespace and comments have been factored out in its own parts, so it does not clutter the essential parsing for a given clause.

2 Earls of Ravnica

How to run tests: In the `ravnica` folder, open a terminal an run `erl`. In any case, compile the district with `c(district).`.

1. If you want to run unit tests, copile them with `c(unittests).` and run the tests with `unittests:test().`.
2. There is a bigger trigger test, which has a file on its own. To run, compile the test with `c(triggertest).`, and run the test with `triggertest:test().`.
3. If you want to run QuickCheck tests, compile the tests with `c(district_qc).` and run the tests with `eqc:module(district_qc).`.

2.1 The District Module

Each district is modelled using a generic state machine (`gen_statem` in Erlang), which then takes care of the gritty parts of communication, state loops, etc. The states noted in the assignment text are modelled using the states of the state machine. The implemented module provides all the functionality described in the assignment, meaning all of the API functions. There are certain critical assumptions, which I have described in detail in section 2.2. Most of the functions are synchronous, meaning that potential deadlocks and race conditions could apply in certain scenarios; I have tried to elaborate on these pitfall-conditions in section 2.2 as well.

I will not go into details with all of the API implementations; instead I have selected some points of interest to elaborate on and explain:

i. **Activation of districts:** In order to `activate` to work properly on graphs with cycles, I had to, from the point of the activated district, figure out how to activate neighbours while still be able to process requests from others, in case someone tries to activate me. I ended up spawning a new process/district whose neighbours are the same as my own, but whose sole purpose is to activate the neighbours and then report back to me on how it went. Meanwhile, the original district switches state to `under_activation` and is free to answer any requests while the subprocess waits on the neighbours. Thus, in case of a cycle, the original district is free to assure the requester that it is under activation. This way, no cycle of districts will wait forever on each other (or at least not because of this).

ii. **Shutting down districts:** When shutting down districts, similar problems may arise: A cycle of districts may shut down the original district, making the original call return an error. In some other scenario, districts may wait forever on each other to shut down. Finally, some district may try to contact a district which is already shut down, potentially yielding an error if one is not careful.

    So, in order to avoid problems like these, a district told to shut down spawns a process similar to when activating. In addition, this process carries a list of known districts that are already shutting down. The subprocess will then only try to shut down neighbours that are not on this list. That way, in a cycle `A -> B -> C -> A`, `A` will not be shut down by `C`, since it is already on the list. Each district adds itself to the list before spawning the subprocess. In addition, any error returned by a shutdown call is assumed to be because the process is already shut down. This is not always true, but it is better than crashing. Also, since the original district has put off the work of shutting down neighbours to a subprocess, it is itself free to answer requests, and so noone is waiting forever on each other. Actually, because the list of shutting down processes already provide a district with this information, the original process should not be requested to be shut down as a consequence of its own chain of shutdowns.

iii. **Communication:** The above implementation requires some extra communication between districts. Namely, there are special districts that function solely as subprocesses for doing some task, and then dies afterward. These have their own little API, which is assumed to be used only by its master district. To make this more secure, one could generate references to identify the right correspondent, but it seemed overkill.

iv. **Trigger events:** The implementation supports triggers. First of all, the district might not even have a trigger, in which case any action is taken 'normally'. If there is a trigger, the value of the trigger is evaluated in a spawned process, in case that the evaluation fails. If the district has to wait more than 2 seconds on the spawned process, the evaluation is discarded. However, if the evaluation returns in time, we sanity check that the result adheres to the requirements.

    This implementation assumes that the trigger does not call any functions from the `district` itself, since this could cause all kinds of problems, e.g. if the trigger shuts down the original district. It should however be possible to interact with other districts, given that these calls do not indirectly alter the state of the triggered district. This statement is not tested, however, and so I might be wrong.

    Triggers should never return a wrong result, as the result is sanity checked before used. Any errors occurring from the trigger is caught as well, after 2 seconds.

2.2 Assumptions and Pitfalls

This is a concurrent system of districts, and so there are various potential pitfalls of deadlocks or race conditions. One such deadlock is if a cycle of districts tries to synchronously prompt each other, e.g. by taking some action moving creatures. If we are unlucky, they will deadlock each other in the cycle. Maybe one could work with this by introducing time-outs on requests. This might however also have the side effect that districts are not correctly updated, or calls that should be adressed are simply ignored.

For this reason, an error-free implementation would have to assume that these scenarios do not occur. I am not sure that this is a realistic assumption. Had I had more time, I would try to address some of these issues.

2.3 Testing

I have written a suite of unit tests, each district module function having a couple of tests trying to cover some of the interesting cases. That could be cycles, self-loops, existing creatures and so on. I have also implemented the QuickCheck tests, which tests the two properties on `activate` and `take_action`. The properties I have tried to test are: if a district is active, so are its neighbours; if I take an action from an active district, the creature should end up at the destination of the action. The actual properties that I have implemented are slightly weaker forms of these properties.

Ideally, when testing `take_action`, one would want to check that the creature ends up at the receiving end, which could be done by shutting down the receiving district, and make it send its creatures to us as 'next plane'. However, when testing the districts, I do not want to shut down some of the districts, because I want to do something with them later. Maybe I could have used `?IMPLIES`, to work around this. But no time.

My generation of a territory has some restrictions. For one, it chooses between a finite set of actions, currently of size 10. Any district will have an expected number of 5 connections, based on the frequency weights used on generating connections. Both are restrictions that limit the diversity of generated samples, and so limits the range of what is tested. This choice was made to ensure in expectation that generated districts are connected with each other.

2.4 Conclusion

Based on the tests I have written and the satisfied QuickCheck properties, I believe that my solution is correct to a large extent. However, there are some cases where deadlock can occur, which should definitely be noted.

The `district` module is very large, and it would definitely help readability to split it up into smaller modules. The code itself is well commented, and so should hopefully be readily understood. Most functions are of fair size, which should help in their understanding. Most utility-functions have their own implementation and are not embedded in the code of other functions.

Appendices

A Program Code

A.1 The appm Package Manager

A.1.1 Utils.hs

```haskell
module Utils where

-- Any auxiliary code to be shared by Parser, Solver, or tests
-- should be placed here.

import Defs
import Data.List
import Control.Monad


instance Ord Version where
  -- if the first version is empty, then it is always less than or equal
  (<=) (V []) _ = True
  -- similarly, if the first is non-empty, but the second is empty,
  -- the first is largest
  (<=) (V (_:_)) (V []) = False
  -- otherwise, we inspect the elements of the version
  (<=) (V (VN i1 s1 : xs)) (V (VN i2 s2 : ys))
    -- if the numeric parts are distinct, then we can simply compare those
    | i1 /= i2 = i1 < i2
    -- otherwise, we have to compare suffixes. If they are equal as well,
    -- we have to look at the remainder of the version number.
    | s1 == s2 = V xs <= V ys
    -- if the suffixes are distinct, but same length, we can simply compare them
    | length s1 == length s2 = s1 <= s2
    -- otherwise, we know that the shortest is always smaller
    | otherwise = length s1 < length s2

merge :: Constrs -> Constrs -> Maybe Constrs
-- if either of the given constraints are empty, return the nonempty input.
merge c1 [] = Just c1
merge [] c2 = Just c2
merge c1 c2 = foldM mergeFolder c2 c1

mergeFolder :: Constrs -> (PName, PConstr) -> Maybe Constrs
mergeFolder cs (n1, c1) =
  -- if name is already in accumulator, then check it
  case matches of
    -- the package is not in the second list, and we can safely add it
    [] -> return $ (n1, c1) : cs
    -- the package is in both lists, so we have to check for feasibility
    [(_, c2)] ->
```

```haskell
      -- extract the bounds and required bool from both constraints
      let ((b1, lo1, hi1), (b2, lo2, hi2)) = (c1, c2)
          -- find lower and upper bounds
          lower = max lo1 lo2 :: Version
          upper = min hi1 hi2 :: Version
      in
        -- if the bounds denote an empty interval, then we have nothing
        -- that is, if at least one of them is required
        if lower >= upper && (b1 || b2) then Nothing
        -- otherwise, we return the new bounds for the package, plus the rest
        else return $ (n1, (b1 || b2, lower, upper)) : remainder
    _ -> Nothing -- Any package should be mentioned at most once in the list

  -- we split the second input into packages with same name, and the others
  where (matches, remainder) = partition (\(n2, _) -> n2 == n1) cs

------------------------------------
-- Utility-functions for the SOLVER
------------------------------------

-- partitions constraints into requirements and conflicts
reqsAndConfs :: Constrs -> (Constrs, Constrs)
reqsAndConfs = partition (\(_, (bool, _, _) ) -> bool)

-- finds all duplicates of a package in a list of packages
findDuplicatePackages :: Pkg -> [Pkg] -> [Pkg]
findDuplicatePackages p = filter f
  where f pkg = (name pkg == name p) && (ver pkg == ver p)

findConsistentPackage :: Pkg -> [Pkg] -> Maybe Pkg
findConsistentPackage pkg [] = Just pkg
findConsistentPackage pkg (p:ps) =
  if (desc pkg == desc p) && (Just depspkg == merge depspkg depsp)
  then findConsistentPackage pkg ps
  else Nothing
  where depspkg = deps pkg
        depsp   = deps p

isRequiredBy :: Pkg -> (PName, PConstr) -> Bool
isRequiredBy p (n, (True, lo, hi)) = name p == n && lo <= v && v < hi
                                     where v = ver p
isRequiredBy _ (_, (False, _, _)) = False

doesntConflictWith :: Pkg -> Constrs -> Bool
doesntConflictWith p c =
  not(any (\(n, (b, lo, hi)) -> pn == n && (lo > v || v >= hi) && not b) c)
  where pn = name p
        v = ver p

-- checks if input database is consistent
-- returns a consistent db without duplicate package versions if possible
-- or returns an error if db is not consistent
isConsistentDB :: Database -> Either String Database
-- the empty db is consistent
isConsistentDB (DB [])      = Right $ DB []
-- for a non-empty db, find any duplicates of the first package p in the db
isConsistentDB (DB (p:ps)) = let duplicates = findDuplicatePackages p ps in
  -- check if p is consistent with all its duplicates
  case findConsistentPackage p duplicates of
    -- if it is, then recursively check the remaining packages
    -- with found duplicates removed
    Just pkg -> case isConsistentDB (DB (ps \\ duplicates)) of
      -- TODO: monad implementation
      -- if the recursive call succeeds, then return the db with pkg added
      Right (DB rest) -> Right $ DB (pkg:rest)
      -- otherwise, we have an error, which is passed on
      Left e          -> Left e
    -- if the package is not consistent with its duplicates, then we have error
    Nothing  -> Left "Database not consistent; a package had conflicting copies"

-- Used to sort in packages in descending order.
pkgCompare :: Pkg -> Pkg -> Ordering
pkgCompare p1 p2 = compare (ver p2) (ver p1)

sorted :: Database -> Database
sorted (DB db) = DB $ sortBy pkgCompare db

----------------------------------------
-- Utility functions for the PROPERTIES
----------------------------------------

getNameVerList :: Database -> [(PName, Version)]
getNameVerList (DB db) = map (\p -> (name p, ver p)) db

ofSomeVersionIn :: PName -> Database -> Maybe Version
ofSomeVersionIn pname (DB db) =
  case find (\p -> name p == pname) db of
    Just pkg -> Just (ver pkg)
    Nothing -> Nothing

requirementsOf :: Pkg -> [(PName, Version, Version)]
requirementsOf pkg =
  let reqs = filter (\(_, (b, _, _)) -> b) (deps pkg) in
    map (\(pn, (_, lo, hi)) -> (pn, lo, hi)) reqs

-- Returns True iff all package names in the input list are unique
allDiff :: [(PName, Version)] -> Bool
allDiff [] = True
allDiff ((n, _):ss) = not (any (\(n', _) -> n == n') ss) && allDiff ss

```

```haskell
-- checks if a solution satifies all constraints
satisfies :: Sol -> Constrs -> Bool
satisfies _ [] = True
satisfies sol cs = foldl folder True cs
  where folder acc (cn, (requires, lo, hi)) =
          if requires
          then acc && any (\(sn, sv) -> cn == sn && lo <= sv && sv < hi) sol
          else acc && all (\(sn, sv) -> cn /= sn || (lo <= sv && sv < hi)) sol

-- goes through all the constraints in the input and merges them
-- this basically checks that the conjoined constraints are satisfiable
sanityCheck :: Constrs -> Maybe Constrs
sanityCheck [] = Just []
sanityCheck (c:cs) = merge [c] cs

isWellFormed :: [Constrs] -> Maybe Constrs
isWellFormed constraints = do
  collected <- mapM sanityCheck constraints
  foldM merge [] collected
```

A.1.2 ParserImpls.hs

```haskell
-- hlint told me to use lambda cases, so that's why this is included
{-# LANGUAGE LambdaCase #-}

module ParserImpl where

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

import Defs
import Utils

import Data.Char
import Text.Parsec.Char
import Text.Parsec.Combinator
import Text.Parsec.Prim hiding (token)
import Text.Parsec.String
import Control.Monad


-- The structure of both parseVersion and parseDatabase is very close to what
-- we did in assignment A2 for the substript parser.
parseVersion :: String -> Either ErrMsg Version
parseVersion s = case parse versionParser "" s of
  Left  err     -> Left $ show err
  Right version -> Right version

parseDatabase :: String -> Either ErrMsg Database
parseDatabase s = case parse databaseParser "" s of
  Left  err -> Left $ show err
  Right db  -> Right db

databaseParser :: Parser Database
databaseParser = token $ do
  spacesAndCommentsParser
  packages <- many packageParser
  eof
  return $ DB packages

-- data used for parsing clauses
data Clause = NC PName
            | VC Version
            | DC String
            | CC Constrs
            | Epsilon

-- really ugly, sorry! but I had trouble making it work with (<|>)
clauseParser :: Parser Clause
clauseParser = token $ do
  -- maybe parse a name clause
  name <- optionMaybe (do keywordParser "name"
                          packageNameParser)
  case name of
    Just n -> return $ NC n
    Nothing -> do
      -- maybe parse a version clause
      ver <- optionMaybe (do keywordParser "version"
                             versionParser)
      case ver of
        Just v -> return $ VC v
        Nothing -> do
          -- maybe parse a requied constraint
          req <- optionMaybe requiredParser

          case req of
            Just r -> return $ CC r
            Nothing -> do
              -- maybe parse a conflict constraint
              con <- optionMaybe conflictsParser

              case con of
                Just c -> return $ CC c
                Nothing -> do
                  -- maybe parse a description
                  descr <- optionMaybe (do keywordParser "description"
                                           stringParser)
                  case descr of
                    Just d -> return $ DC d
                    -- if none of these work, then we have the empty clause
                    Nothing -> return Epsilon

-- used by package parser to check that clauses are well-formed
cleanUp :: [ Clause ] -> Maybe (PName, Version, String, Constrs)
cleanUp []      = Nothing
cleanUp clauses =
  -- filter out all the disticts clauses from the Clause data type
  let nameClauses = filter (\case NC _ -> True; _ -> False) clauses
      versClauses = filter (\case VC _ -> True; _ -> False) clauses
      descClauses = filter (\case DC _ -> True; _ -> False) clauses
      consClauses = filter (\case CC _ -> True; _ -> False) clauses
      -- we need to check the number of some of these clauses
      ln = length nameClauses
      lv = length versClauses
      ld = length descClauses

      -- we're not using a monad for this, so have to unpack
      unpackedConstraints =
        map (\case CC c -> c; _ -> error "") consClauses

      -- check that there is at most one specified version
      version = if lv == 0 then V [VN 1 ""]
                else case head versClauses of VC v -> v; _ -> error ""
```

```haskell

      -- check the description
      description = if ld == 0 then ""
                    else case head descClauses of DC s -> s; _ -> error ""

  -- ok, so we have a problem if there is not exactly one name,
  -- or if we have more than one version,
  -- or if we have more than one description
  in if ln /= 1 || lv > 1 || ld > 1 then Nothing
     else let name = case head nameClauses of NC n -> n; _ -> error ""
          -- check that constraints are well-formed
          in case isWellFormed unpackedConstraints of
              -- and return them in order if possible
              Just nice -> Just (name, version, description, nice)
              Nothing   -> Nothing

-- parses a package
packageParser :: Parser Pkg
packageParser = token $ do
  keywordParser "package"
  token $ char '{'
  clauses <- sepBy1 clauseParser (token (char ';'))
  token $ char '}'
  -- check that parsed clauses are well-formed
  case cleanUp clauses of
    Just (name, ver, descr, constr) -> return $ Pkg name ver descr constr
    Nothing -> fail "Clauses are not semantically well-formed."

-- used by required and conflicts parsers to parse version bounds
boundParser :: String -> Parser Version
boundParser op = token $ do
  token $ string op
  versionParser

requiredParser :: Parser [(PName, PConstr)]
requiredParser = token $ do
  keywordParser "requires"
  sepBy1 (singleConstParser True ">=" "<") (token (char ','))

conflictsParser :: Parser [(PName, PConstr)]
conflictsParser = token $ do
  keywordParser "conflicts"
  sepBy1 (singleConstParser False "<" ">=") (token (char ','))

singleConstParser :: Bool -> String -> String -> Parser (PName, PConstr)
singleConstParser bool op1 op2 = token $ do
  name <- packageNameParser
  low  <- option minV (boundParser op1)
  high <- option maxV (boundParser op2)
  if low <= high then return (name, (bool, low, high))
  else fail "version interval is empty"

```

```
-- Parses the version type
versionParser :: Parser Version
versionParser = token $ do
  -- parse initial whitespace and comments
  spacesAndCommentsParser
  -- parse at least one version number, each separated by '.'
  versionNumbers <- sepBy1 singleVNumParser (char '.')
  -- return as a version type
  return $ V versionNumbers

singleVNumParser :: Parser VNum
singleVNumParser = do
   -- parse at least one digit for the number of the version
  numbers <- many1 digit
  -- parse an optional suffix of at most 4 chars
  optionalSuffix <- many (satisfy isAsciiLower)
  -- check constraints on number range and suffic length before returning VNum
  if (read numbers < 1000000) || length optionalSuffix < 5
  then return $ VN (read numbers) optionalSuffix
  else error "Number is greater than 999 999 or suffix is too long."

isAlphaNumOrHyphen :: Char -> Bool
isAlphaNumOrHyphen c = isAlphaNum c || (c == '-')

-- I added hyphen to they keyword parser from A2, so that reserved keywords
-- are not followed by either alphanum or a hyphen (ver 1.0 of description)
-- I also added case insensitiveness
-- Otherwise it is pretty much the same function in case you wonder
keywordParser :: String -> Parser ()
keywordParser ""   = token $ notFollowedBy (satisfy isAlphaNumOrHyphen)
keywordParser (c:cs) = do
  oneOf [toUpper c, toLower c]
  keywordParser cs

asciiLetters = ['A' .. 'Z'] ++ ['a' .. 'z']

lettersAndDigitsParser :: Parser String
lettersAndDigitsParser = many1 (oneOf asciiLetters <|> digit)

hyphenParser :: Parser String
hyphenParser = do
  hyp <- char '-'
  lads <- lettersAndDigitsParser
  return $ hyp : lads

-- a parser for the simple package name
simplePackageNameParser :: Parser String
simplePackageNameParser = token $ do
  head  <- oneOf asciiLetters
  slack <- many (oneOf asciiLetters)
  rest  <- many hyphenParser
```

```haskell
  return $ head : slack ++ concat rest

stringParser :: Parser String
stringParser = token $ do
  char ’\"’
  stringContent <- stringContentParser
  char ’\"’
  return stringContent

-- a parser for the general package name
generalPackageNameParser :: Parser String
generalPackageNameParser = stringParser

isValidStringChar :: Char -> Bool
isValidStringChar c = isAscii c && (c /= ’"’)

-- Helper functions for string content parser:
-- asciiNotQuoteParser parses one or more of all ascii chars that are not ’\"’
asciiNotQuoteParser = many1 $ satisfy isValidStringChar
-- quoteParser parses one or more of the string "\"\""
quoteParser = do
  doublequotes <- many1 $ string [’\"’, ’\"’]
  -- we only want to return one quote for each pair of quotes we parsed
  return [’\"’ | _ <- [1 .. length doublequotes]]

stringContentParser :: Parser String
stringContentParser = do
  a <- many (asciiNotQuoteParser <|> try quoteParser)
  return $ concat a

-- This is the top package name parser
packageNameParser :: Parser PName
packageNameParser = token $ do
  -- either parse a simple or a general name and return it as a PName
  name <- simplePackageNameParser <|> generalPackageNameParser
  return $ P name

-- THE FOLLOWING CODE IS DIRECTLY TAKEN FROM OUR HANDIN OF ASSIGNMENT A2
-- It is used for parsing comments, whitespace, tokens, etc., and so is almost
-- completely similar. Comments are now starting with ’--’ instead of ’//’

-- One-line comments
commentsParser :: Parser ()
commentsParser = do
  string "--"
  many $ noneOf "\n"
  spaces
  return ()

-- Spaces and comments
```

```haskell
spacesAndCommentsParser :: Parser ()
spacesAndCommentsParser = do
  spaces
  many commentsParser
  return ()

-- Parse the parser followed by whitespace/comments
token :: Parser a -> Parser a
token p = do
  r <- p
  spacesAndCommentsParser
  return r

-- Parse the string followed by whitespace/comments
symbol :: String -> Parser String
symbol s = token $ string s
```

A.1.3 SolverImpls.hs

```haskell
module SolverImpl where

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

import Defs
import Utils
import Data.List
import Parser(parseDatabase)

-- Helper functions used by the SOLVER
-- parseFile TAKEN DIRECTLY FROM ASSIGNMENT A2. Used to read test files
parseFile :: FilePath -> IO (Either ErrMsg Database)
parseFile path = parseDatabase <$> readFile path

-- normalize function
normalize :: Database -> Either String Database
normalize (DB []) = Right $ DB []
-- sorts the database and then checks the result for consistency
-- see Util for associated functions isConsistentDB and sorted
normalize db = isConsistentDB (sorted db)

-- the package resolver
solve :: Database -> Constrs -> Sol -> [Sol]
solve db c sol =
  -- divide constraints into required and conflicting packages
  let (reqs, conflicts) = reqsAndConfs c in
    -- then, use the solver’ to solve
    solve’ db reqs conflicts sol

-- the meaty part of the solver
solve’ :: Database -> Constrs -> Constrs -> Sol -> [Sol]
solve’ _ [] _ sol = [sol]
solve’ (DB db) (r:rs) cs sol = do
  -- take any package from the database
  p <- db
  let isNotIn p ss = not (any (\(n, v) -> name p == n) ss)
  -- if these checks, we want to consider the package
  if (p ‘isRequiredBy‘ r) && (p ‘doesntConflictWith‘ cs) && p ‘isNotIn‘ sol then
    -- s is the package name and version, to fit in the partial solution
    let s    = (name p, ver p)
        -- add the package to the partial solution
        sol’ = s:sol
        -- what are the dependencies we need to satisfy for the new package?
        pd   = deps p
        -- merge these with the current constraints
        merged = merge pd (rs ++ cs)
    in case merged of
```

```
      -- are the merged constraints satisfiable? And do we in fact satify
   them?
      -- if the solution is complete, we add it to the final result
      Just c' -> if sol' `satisfies` (r:c') then [sol']
                 -- if constraints are satisfiable, but the solution is
   partial
                 -- we recurse
                 else solve (DB db) c' sol'
      -- otherwise, discard this solution
      Nothing -> fail "Impossible."
  -- do not look any further at this package
  else fail "Dont need this package"

install :: Database -> PName -> Maybe Sol
install (DB db) pname =
  case find (\p -> name p == pname) db of
    Just _ -> let r = (pname, (True, minV, maxV)) in
      case solve (DB db) [r] [] of
        []  -> Nothing
        sol -> Just (head sol)
    Nothing -> Nothing
```

A.2 Earls of Ravnica

A.2.1 district.erl

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

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

% gen_statem callback module: init, callback_mode and terminate
-export([init/1, callback_mode/0, terminate/3]).

% gen_statem state functions
-export([ under_configuration/3
        , active/3
        , under_activation/3
        , shutting_down/3
        , activationsubroutine/3
        , shutdownsubroutine/3
        , kill_me/3
        ]).

%%%%%%%%%%%%%%%%%%%%%
% CALLBACK FUNCTIONS
%%%%%%%%%%%%%%%%%%%%%

% Callback mode:
% state_functions are used, where events are handled by one function per state
callback_mode() -> [state_functions, state_enter].

% Init:
% Takes a district description as input and returns
% ok, as a confirmation code that the operation was successful
% under_configuration atom, denoting that the district is under construction
% a state, containing
%   (1) a description of the disctrict,
%   (2) a map of connections,
%   (3) a triggerlist
init(Desc) ->
    Connections = maps:new(),
    Trigger = none,
    { ok
    , under_configuration
    , {Desc, Connections, Trigger}
    }.

% called when terminating a district
terminate(_Reason, _StateName, _StateData) -> void.

%%%%%%%%%%%%%%%%%%%%%
% Ravnica Client API
%%%%%%%%%%%%%%%%%%%%%

-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()]}).

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

% Call the specified district using the atom getDescription,
% denoting to the genstate district that it should send back a description
-spec get_description(passage()) -> {ok, string()} | {error, any()}.
get_description(District) -> gen_statem:call(District, getDescription).


% Call the specified district From, requesting a connection to To with Action
-spec connect(passage(), atom(), passage()) -> ok | {error, any()}.
connect(From, Action, To) -> gen_statem:call(From, {connect, Action, To}).

% Try to activate the input district
-spec activate(passage()) -> active | under_activation | impossible.
activate(District) -> gen_statem:call(District, activate).

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

-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(District, CRef, Action) ->
    gen_statem:call(District, {takeAction, CRef, Action}).

-spec shutdown(passage(), pid()) -> ok.
shutdown(District, NextPlane) -> internalshutdown(District, NextPlane, []).

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

```
    .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Functions used internally by the districts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Used by districts to shut each other down
% only difference from this and shutdown is that this has a list Ds as
   argument
internalshutdown(District, NextPlane, Ds) ->
    ok = gen_statem:call(District, {shutdown, NextPlane, Ds}),
    gen_statem:stop(District),
    ok.

% Used for shutting down subroutines
endsubprocess(Process) -> gen_statem:stop(Process).

% Removes self loops. Used in the shutting down process
removeLoops(Cons, D) -> maps:filter (fun(_K, V) -> V /= D end, Cons).

% subroutine used for shutting down neighbours
shutdownSubroutine(Subprocess, Neighbours, Ds, NextPlane) ->
    gen_statem:call(Subprocess, {shutdownsubroutine, Neighbours, Ds,
   NextPlane}).

% Folds through a list of connections and activates.
actfolder({_Action, District}, R) ->
    if R == impossible -> impossible;
       true -> activate(District)
    end.

% Folds through a list of connections and shuts down.
shutfolder(Ds, {_Action, District}, {ok, NextPlane}) ->
    % First, try to shut down the neighbour
    try
        IsInDs = lists:member(District, Ds),
        if IsInDs -> {ok, NextPlane};
           true -> {internalshutdown(District, NextPlane, Ds), NextPlane}
        end
    catch
        % Well, maybe the neighbour is already shutdown, in which case this
        % results in an exit error. In that case, we know that the
   neighbour
        % is already shut down (or at least non-existing).
        exit:_Reason  -> {ok, NextPlane}
    end.

% Initiates the activate subroutine
activateSubroutine(Subprocess, Neighbours) ->
    gen_statem:call(Subprocess, {activationsubroutine, Neighbours}).

trigRun(From, Trigger, Event, Creature, Creatures) ->
```

```erlang
    Me = self(),
    try {_, _} = Result = Trigger(Event, Creature, Creatures),
        From ! {Me, Result}
    catch
        _ -> From ! {Me, error}
    end.

% This is used in trigger-handling, to check creatures are well-formed
creaturesCheck(Old, New) ->
    % same length; no new creatures have risen
    SameLength = length(Old) == length(New),
    % check that for all creatures in the old list,
    % there is a creature in the new list with the same ref
    % if the lists are same length, this mapping is one to one
    Same = lists:all (fun({R1, _}) ->
                lists:any(fun({R2, _}) -> R1 == R2 end, New)
            end, Old),
    % return the combined boolean
    Same and SameLength.

% sanity check that the result of the trigger is actually well formed
sanitycheck(Result, Creature, Creatures) ->
    % try to pattern match with creature tuple
    try {{Ref, _}, NewCreatures} = Result,
        % get creature ref
        {CRef, _} = Creature,
        % check refs for all remaining creatures
        Same = creaturesCheck(Creatures, NewCreatures),
        % these should all be the same
        Same and (CRef == Ref)
    catch _ -> false
    end.

% We use runTrigger to evaluate the trigger of a given district
runTrigger(Trigger, Event, Creature, Creatures) ->
    Me = self(),
    % We don't want to crash, so we spawn a function that does the work for us
    % It might potentially crash, but we wait at most 2 seconds, in which case
    % we return the original input, and thus the program survives
    Pid = spawn(fun() -> trigRun(Me, Trigger, Event, Creature, Creatures) end),
    receive
        % wait for a response from the worker and sanity check its answer
        {Pid, Result} -> SanityCheck = sanitycheck(Result, Creature, Creatures),
                         % if it passes the check, return the result
                         if SanityCheck -> Result;
                            % otherwise give back original creatures
                            true -> {Creature, Creatures}
                         end
    % do no wait for more than 2 seconds for a response, though
    after 2000 ->
        {Creature, Creatures}
    end.

%%%%%%%%%%%%%%%%%%%
% STATE FUNCTIONS
%%%%%%%%%%%%%%%%%%%

% ACTIVATION SUBROUTINE STATE
activationsubroutine(enter, _OldState, Data) ->
    {Master, _Desc, Connections, _Trigger} = Data,
    % Get all neighbours, and for each of them, call the activate subroutine
    Neighbours = maps:to_list(Connections),
    % Ask all neighbours in turn to activate
    Result = lists:foldl(fun(A, B) -> actfolder(A, B) end, active, Neighbours),
    % Reply back to the master process with the result
    {keep_state_and_data, [{reply, Master, Result}]};

% Any other interaction with subroutine is futile
% This is assumed not to happen
activationsubroutine(_, _, _) -> keep_state_and_data.

% SHUTDOWN SUBROUTINE STATE
shutdownsubroutine(enter, _OldState, Data) ->
    {Master, NextPlane, Ds, _Desc, Connections, _Trigger} = Data,
    % Get all neighbours, and for each of them, call the activate subroutine
    Neighbours = maps:to_list(Connections),
    % Ask all neighbours in turn to activate
    {ok, _} = lists:foldl(fun(A, B) -> shutfolder(Ds, A, B) end, {ok, NextPlane}, Neighbours),
    % Reply back to the master process with the result
    {keep_state_and_data, [{reply, Master, ok}]};

% Any other interaction with subroutine is futile
% This is assumed not to happen
shutdownsubroutine(_, _, _) -> keep_state_and_data.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% UNDER CONFIGURATION STATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% State enter call (unused, ignore)
under_configuration(enter, _OldState, _Data) -> keep_state_and_data;

% Special case used only for activation in subroutines
under_configuration({call, From}, {activationsubroutine, Connections}, Data) ->
    {Desc, _OldConnections, Trigger} = Data,
```

```erlang
    NewData = {From, Desc, Connections, Trigger},
    % We're gonna go into the subroutine state,
    % remembering who the caller/master process is
    {next_state, activationsubroutine, NewData, [{reply, From, ok}]};

% Special case used only for shutting down in subroutines
under_configuration({call, From}, {shutdownsubroutine, Cons, Ds, P}, Data) ->
    {Desc, _OldConnections, Trigger} = Data,
    NewData = {From, P, Ds, Desc, Cons, Trigger},
    % We're gonna go into the subroutine state,
    % remembering who the caller/master process is
    {next_state, shutdownsubroutine, NewData, [{reply, From, ok}]};

% handling description requests
under_configuration({call, From}, getDescription, Data) ->
    {Desc, _Connections, _Trigger} = Data,
    {keep_state_and_data, [{reply, From, {ok, Desc}}]};

% Handling connections
under_configuration({call, From}, {connect, Action, To}, Data) ->
    % First, declare data to access the connections
    {Desc, Connections, Trigger} = Data,
    % Check if the action is already used for a connection
    ActionIsAlreadyUsed = maps:is_key(Action, Connections),
    if ActionIsAlreadyUsed ->
        % If it is, then return error without changing anything
        Msg = "Action is already in use.",
        {keep_state_and_data, [{reply, From, {error, Msg}}]};
       true ->
        % Otherwise, add the link to the connections and return 'ok'
        % We anticipate activation by denoting a bool for activity
        NewConnections = maps:put(Action, To, Connections),
        NewData = {Desc, NewConnections, Trigger},
        {keep_state, NewData, [{reply, From, ok}]}
    end;

% Someone is trying to get action options from this state, which is ok
under_configuration({call, From}, getOptions, Data) ->
    {_Desc, Connections, _Trigger} = Data,
    Actions = maps:keys(Connections),
    {keep_state_and_data, [{reply, From, {ok, Actions}}]};

% Someone is erroneusly trying to put a creature in a shutting down district!
under_configuration({call, From}, {enter, _Creature}, _Data) ->
    Msg = "Under configuration. No creatures are allowed to enter.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% Someone is erroneusly trying to take action in a shutting down district!
under_configuration({call, From}, {takeAction, _CRef, _Action}, _Data) ->
    Msg = "Under configuration. No actions are allowed.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% Someone treats this district as the 'Next Plane'
under_configuration(cast, {shutting_down, _D, _Cs}, _) ->
    keep_state_and_data;

% Someone is adding a trigger to the district
under_configuration({call, From}, {trigger, Trigger}, Data) ->
    {Desc, Cons, _OldTrigger} = Data,
    % Overwrite whatever trigger we had stored, and return 'ok'
    NewData = {Desc, Cons, Trigger},
    {keep_state, NewData, {reply, From, ok}};

% Ok, we're being shut down. Commence.
under_configuration({call, From}, {shutdown, NextPlane, Ds}, Data) ->
    % Pattern match on data
    {Desc, Cons, Trigger} = Data,
    % Ok, since we already have a subprocess going on, we're just gonna kill it
    NewData = {From, Desc, Cons, [], Trigger, NextPlane, Ds, noSubProcess},
    {next_state, shutting_down, NewData};

% Someone is activating us! So we should go ahead and activate
under_configuration({call, From}, activate, Data) ->
    % Pattern match on data
    {Desc, Connections, Trigger} = Data,
    % Remember who called for activation.
    NewStateData = {From, Desc, Connections, Trigger, noSubProcess},
    % Change state, since we're now under activation
    {next_state, under_activation, NewStateData}.

%%%%%%%%%%%%%%%%%%%%%%%%%
% UNDER ACTIVATION STATE
%%%%%%%%%%%%%%%%%%%%%%%%%

% This is where we initially try to activate all the neighbours of the district
under_activation(enter, _OldState, Data) ->
    % Pattern match on data to get connections
    {_Caller, _Desc, Connections, _Trigger, _SubProcess} = Data,
    % Spawn a process, technically a district, that has
    % this district's neighbours, and which takes care of activating neighbours
    {ok, SubDistrict} = create("This is a subdistrict, used for activation."),
    ok = activateSubroutine(SubDistrict, Connections),
    % Then, go on, and wait for answers (while also answering queries)!
    NewData = {_Caller, _Desc, Connections, _Trigger, SubDistrict},
    {keep_state, NewData};

% This happens when someone is trying to figure out if we are active or not
under_activation({call, From}, activate, _Data) ->
```

```erlang
    % We just tell them that we're under activation and that's it
    {keep_state_and_data, [{reply, From, under_activation}]};

% handling initialization call
under_activation({call, From}, getDescription, Data) ->
    {_Caller, Desc, _Connections, _Trigger, _SubProcess} = Data,
    {keep_state_and_data, [{reply, From, {ok, Desc}}]};

% Handling connections
under_activation({call, From}, {connect, _Action, _To}, _Data) ->
    Msg = "District is under activation, and connections can not be formed.",
    {keep_state_and_data, [{reply, From, {ok, Msg}}]};

% Someone is trying to get action options from this state, which is not ok!
under_activation({call, From}, getOptions, _Data) ->
    {keep_state_and_data, [{reply, From, none}]};

% Someone is erroneusly trying to put a creature in a shutting down district!
under_activation({call, From}, {enter, _Creature}, _Data) ->
    Msg = "Under activation. No creatures are allowed to enter.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% Someone is erroneusly trying to take action in a district under activation!
under_activation({call, From}, {takeAction, _CRef, _Action}, _Data) ->
    Msg = "Under activation. No actions are allowed.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% Someone is sending us creatures??
under_activation(cast, {shutting_down, _D, _Cs}, _) -> keep_state_and_data;

% Someone is erroneusly adding a trigger to the district
under_activation({call, From}, {trigger, _Trigger}, _Data) ->
    Msg = "You cannot add a trigger to a district under activation.",
    {keep_state_and_data, {reply, From, {error, Msg}}};

% This matches the case where we get the result back from the subroutine
under_activation(info, {_Ref, Result}, Data) ->
    {Caller, Desc, Connections, Trigger, SubProcess} = Data,
    % OK, we need to be able to model creatues, which we store in a map
    Creatures = [],
    NewData = {Desc, Connections, Creatures, Trigger},
    % End the subprocess, since its work is now done
    endsubprocess(SubProcess),
    if Result == impossible ->
        % It was impossible to activate the district!
        % In that case, we revert back to being under configuration
        NewData = {Desc, Connections, Trigger},
        {next_state, under_configuration, NewData, [{reply, Caller, Result}]};
```

```
       true -> {next_state, active, NewData, [{reply, Caller, active}]}
    end;

% Ok, we're being shut down. Commence.
under_activation({call, From}, {shutdown, NextPlane, Ds}, Data) ->
    % Pattern match on data
    {Caller, Desc, Cons, Trigger, SubProcess} = Data,
    % Ok, since we already have a subprocess going on, we're just gonna kill it
    NewData = {From, Desc, Cons, [], Trigger, NextPlane, Ds, noSubProcess},
    try
        endsubprocess(SubProcess),
        % Change state, since we're now under shutdown
        % Also, notify caller that activation is impossible
        {next_state, shutting_down, NewData, [{reply, Caller, impossible}]}

    catch
        _ -> % Change state, since we're now under shutdown
             {next_state, shutting_down, NewData, [{reply, Caller, impossible}]}
    end.

%%%%%%%%%%%%%%%
% ACTIVE STATE
%%%%%%%%%%%%%%%

% A creature enters the dungeon!
active({call, From}, {enter, Creature}, Data) ->
    % First we match the data and the creature
    {Desc, Connections, Creatures, Trigger} = Data,
    {CRef, _Stats} = Creature,
    Event = entering,
    % check if the creature is already in the district
    % IsKey = maps:is_key(Ref, Creatures),
    IsKey = lists:keymember(CRef, 1, Creatures),
    % If so, return error
    if IsKey -> Msg = "A creature is already in the district.",
                {keep_state_and_data, [{reply, From, {error, Msg}}]};
       % otherwise, we can add the creature
       true ->
           % if there's no trigger, then we just go ahead
           if Trigger == none ->
                      NewCreatures = lists:append([Creature], Creatures),
                      NewData = {Desc, Connections, NewCreatures, Trigger},
                      {keep_state, NewData, [{reply, From, ok}]};
              % otherwise, we trigger the trigger and use the result
              true -> {C, Cs} = runTrigger(Trigger, Event, Creature, Creatures),
                    %   {CRef, CStats} = C,
                      NewCreatures = lists:append([C], Cs),
                      NewData = {Desc, Connections, NewCreatures, Trigger},
                      {keep_state, NewData, [{reply, From, ok}]}
```

```
      end
  end;

% State enter call (unused, ignore)
active(enter, _OldState, _Data) -> keep_state_and_data;

% handling description requests
% You can get description in all states
active({call, From}, getDescription, Data) ->
  {Desc, _Connections, _Creatures, _Trigger} = Data,
  {keep_state_and_data, [{reply, From, {ok, Desc}}]};

% Handling connections
active({call, From}, {connect, _Action, _To}, _Data) ->
  Msg = "District is active, and connections can not be formed.",
  {keep_state_and_data, [{reply, From, {ok, Msg}}]};

% Someone is trying to activate us
active({call, From}, activate, _Data) ->
  % We just tell them that we're activated, and that's it
  {keep_state_and_data, [{reply, From, active}]};

% Someone is taking an action in the district
active({call, From}, {takeAction, CRef, Action}, Data) ->
  % So, first we pattern match on the state data
  {Desc, Connections, Creatures, Trigger} = Data,
  % then we check if the specified creature is actually here
  IsCreature = lists:keymember(CRef, 1, Creatures),
  % we also check that the specified action is available
  IsAction = maps:is_key(Action, Connections),

  % If the creature is in the district AND the action is valid
  if IsCreature and IsAction ->
    % we find the destination
    To = maps:get(Action, Connections),
    % and the creature
    Creature = lists:keyfind(CRef, 1, Creatures),
    % Stats = maps:get(CRef, Creatures),
    Me = self(),
    % Now, we have to check if any trigger messes with the creatures
    % if there's no trigger, then we just go ahead
    if Trigger == none ->
     % First we try to skip the creature off to the next district
     % If its a self-loop, don't bother
     if Me == To ->
       {keep_state_and_data, [{reply, From, {ok, To}}]};
       true -> S = enter(To, Creature),
        if S == ok ->
         % we remove it from the current district and reply back
         NewCreatures = lists:keydelete(CRef, 1, Creatures),
         NewData = {Desc, Connections, NewCreatures, Trigger},
         {keep_state, NewData, [{reply, From, {ok, To}}]};
```

```
                 % on the other hand, if it went wrong, then we report error
                 % and keep the creature here
                 true -> Msg = "Creature could not enter new district.",
                         {keep_state_and_data, [{reply, From, {error, Msg}}]}
               end
          end;
          % if this went well,

           % otherwise, we trigger the trigger
           true ->
             % We run the trigger and try to send the creature off
             {C, Cs} = runTrigger(Trigger, leaving, Creature, Creatures),
              % If we have a self-loop, then just keep him here
              % After running trigger-enter
              if Me == To ->
                 {C2, Cs2} = runTrigger(Trigger, entering, C, Cs),
                 NewCreatures = [C2 | Cs2],
                 NewData = {Desc, Connections, NewCreatures, Trigger},
                 {keep_state, NewData, [{reply, From, {ok, To}}]};
               true ->
                S = enter(To, C),
                % if this went ok, we keep the new creatues and return {ok, To}
                if S == ok ->
                   NewCreatures = Cs,
                   NewData = {Desc, Connections, NewCreatures, Trigger},
                   {keep_state, NewData, [{reply, From, {ok, To}}]};
                   % on the other hand, if it went wrong, then we report error
                   % and keep the creature here, with NO changes to any creature
                 true -> Msg = "Creature could not enter new district.",
                         {keep_state_and_data, [{reply, From, {error, Msg}}]}
                end
              end
        end;
       true -> Msg = "Either action or creature does not exist here.",
               {keep_state_and_data, [{reply, From, {error, Msg}}]}
    end;

% Someone is erroneusly adding a trigger to the district
active({call, From}, {trigger, _Trigger}, _Data) ->
    Msg = "You cannot add a trigger to an active district.",
    {keep_state_and_data, {reply, From, {error, Msg}}};

% Someone is trying to get action options from this state, which is ok
active({call, From}, getOptions, Data) ->
    {_Desc, Connections, _Creatures, _Trigger} = Data,
    Actions = maps:keys(Connections),
    {keep_state_and_data, [{reply, From, {ok, Actions}}]};

% Someone is giving us creatures? Ignore!
active(cast, {shutting_down, _D, _Cs}, _) -> keep_state_and_data;

% Ok, we're being shut down. Commence.
active({call, From}, {shutdown, NP, Ds}, Data) ->
    % Pattern match on data
    {Desc, Cons, Creatures, Trigger} = Data,
    % Ok, since we already have a subprocess going on, we're just gonna kill it
    NewData = {From, Desc, Cons, Creatures, Trigger, NP, Ds, noSubProcess},
    {next_state, shutting_down, NewData}.

%%%%%%%%%%%%%%%%%%%%%%
% SHUTTING DOWN STATE
%%%%%%%%%%%%%%%%%%%%%%

% State enter call for shutting down. Things happen here!
shutting_down(enter, _OldState, Data) ->
    % Pattern match on data to get connections
    {Caller, Desc, Connections, Creatures, Trigger, NextPlane, Ds, _} = Data,
    % send creatures to NextPlane
    Me = self(),
    gen_statem:cast(NextPlane, {shutting_down, Me, Creatures}),
    % Spawn a process, technically a district, that has
    % this district's neighbours, and which takes care of activating neighbours
    {ok, SubDistrict} = create("This is a subdistrict, used for shutdown."),

    % remove selfloops from connections before handing them over
    NoLoopsBrother = removeLoops(Connections, Me),
    % We also tell the subroutine to NOT kill me, or anyone on the Ds list
    NewDs = lists:append([self()], Ds),
    ok = shutdownSubroutine(SubDistrict, NoLoopsBrother, NewDs, NextPlane),
    % Get all neighbours, and for each of them, call the activate subroutine
    % Neighbours = maps:to_list(Connections),
    % Each entry has format {Key, {Action, District, IsActive}}
    % lists:foreach(fun({_, D, _}) -> activateSubroutine(D) end, Neighbours),
    % Then, go on, and wait for answers!
    NewData = { Caller
              , Desc
              , Connections
              , Creatures
              , Trigger
              , NextPlane
              , Ds
              , SubDistrict},
    % keep state, but save caller and subproces
    {keep_state, NewData};

% This matches the case where we get the result back from the subroutine
shutting_down(info, {_Ref, ok}, Data) ->
    {Caller, _, _, _, _, _, _, SubProcess} = Data,
    gen_statem:stop(SubProcess),
    % goto one final state, where you just end yourself
    {next_state, kill_me, [], [{reply, Caller, ok}]};

% handling description requests
% You can get description in all states
shutting_down({call, From}, getDescription, Data) ->
    {_Caller, Desc, _Cons, _Creatures, _Trigger, _NextPlane, _Ds, _} = Data,
    {keep_state_and_data, [{reply, From, {ok, Desc}}]};

% Someone is trying to get action options from this state, which is not ok!
shutting_down({call, From}, getOptions, _Data) ->
    {keep_state_and_data, [{reply, From, none}]};

% Someone is erroneusly adding a trigger to the district
shutting_down({call, From}, {trigger, _Trigger}, _Data) ->
    Msg = "You cannot add a trigger to a district that is shutting down.",
    {keep_state_and_data, {reply, From, {error, Msg}}};

% Someone is erroneusly trying to put a creature in a shutting down district!
shutting_down({call, From}, {enter, _Creature}, _Data) ->
    Msg = "District is shutting down. No creatures are allowed to enter.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% Someone is erroneusly trying to take action in a shutting down district!
shutting_down({call, From}, {takeAction, _CRef, _Action}, _Data) ->
    Msg = "District is shutting down. No actions are allowed.",
    {keep_state_and_data, [{reply, From, {error, Msg}}]};

% This happens when someone is trying to shut us down
shutting_down({call, From}, {shutdown, _NextPlane, _Ds}, _Data) ->
    % We just tell them that we're already shutting down and that's it
    {keep_state_and_data, [{reply, From, ok}]};

% Someone is sending us creatures; ignore!
shutting_down(cast, {shutting_down, _D, _Creatures}, _) -> keep_state_and_data;

% Handling connections
shutting_down({call, From}, {connect, _Action, _To}, _Data) ->
    Msg = "District is shutting down, and connections can not be formed.",
    {keep_state_and_data, [{reply, From, {ok, Msg}}]}.

% Not really used for anything; the district enters this state right before
```

```
     the
% process calling the shutdown in the district terminates it
kill_me(enter, _OldState, _Data) -> keep_state_and_data.
```

B Test Code

B.1 The appm Package Manager

B.1.1 Unit tests (BB tests): Main.hs

```
module Main where

-- Put your black-box tests in this file

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

import Test.Tasty
import Test.Tasty.HUnit
import Data.Either

-- parseFile taken directly from A2. Used to read test files
parseFile :: FilePath -> IO (Either ErrMsg Database)
parseFile path = parseDatabase <$> readFile path

-- directory of the test files
path = "tests/BB/testfiles/"

-- Similarly, many of these structural procedures are taken directly
-- from earlier test files, and in some cases generalized
runTest pAct pExp = do
  act <- parseFile $ path ++ pAct
  exp <- fmap read $ readFile $ path ++ pExp
  act @?= Right exp

runNegTest pAct = do
  act <- parseFile $ path ++ pAct
  case act of
    Right _ -> assertFailure "Should not have been parsed!"
    ow      -> assertBool ".." $ isLeft ow

tests = testGroup "Unit tests"
  [ testGroup "Merge tests"
      [ testCase "examExample1"  $ c1  ‘merge‘ c2  @?= Just c3
      , testCase "forumExample1" $ c9  ‘merge‘ c10 @?= Just c11
      , testCase "req and conf"  $ c14 ‘merge‘ c15 @?= Just c16
      , testCase "examExample1"  $ c6  ‘merge‘ c7  @?= Just c8
      , testCase "examExample2"  $ c4  ‘merge‘ c5  @?= Nothing
      , testCase "forumExample2" $ c12 ‘merge‘ c13 @?= Nothing
      ]

  , testGroup "Parser tests"
      [ testCase "tiny" $ parseDatabase "package {name foo}" @?= Right db
      , testCase "intro"                          $ runTest "test1" "test1e"
      , testCase "intro2"                      $ runTest "test2" "test2e"
      , testCase "case insensitive keywords"   $ runTest "test3" "test3e"
      , testCase "jumbled clause order"        $ runTest "test4" "test4e"
      , testCase "no description"              $ runTest "test5" "test5e"
      , testCase "no version"                  $ runTest "test6" "test6e"
      , testCase "no ending semi colon"        $ runTest "test7" "test7e"
      , testCase "ending semi colon"           $ runTest "test8" "test8e"
      , testCase "simple name"                 $ runTest "test9" "test9e"
      , testCase "general name"                $ runTest "test10" "test10e"
      , testCase "version suffix"              $ runTest "test11" "test11e"
      , testCase "negtest: double names"       $ runNegTest "test15"
      , testCase "negtest: double description" $ runNegTest "test16"
      , testCase "negtest: double version"     $ runNegTest "test17"
      , testCase "negtest: bad name"           $ runNegTest "test18"
      , testCase "negtest: empty package"      $ runNegTest "test19"
      , testCase "negtest: missing semicolon"  $ runNegTest "test20"
      , testCase "negtest: bad version"        $ runNegTest "test21"
      ]

  , testGroup "Solver tests"
      [ testCase "tiny"   $ install db pname @?= Just [(pname, ver)]
      , testCase "intro"  $ do
          db1 <- fmap read $ readFile $ path ++ "test1e"
          case normalize db1 of
            Right ndb -> install ndb (P "foo") @?= e1
            _ -> fail ".."

      , testCase "intro2" $ do
          db2 <- fmap read $ readFile $ path ++ "test2e"
          case normalize db2 of
            Right ndb -> install ndb (P "foo") @?= e2
            _ -> fail ".."
      , testCase "large case1" $ do
          db3 <- fmap read $ readFile $ path ++ "test12e"
          case normalize db3 of
            Right ndb -> install ndb (P "chrome") @?= e3
            _ -> fail ".."
      , testCase "large case2" $ do
          eitherdb4 <- parseFile $ path ++ "test13"
          case eitherdb4 of
            Right db4 -> case normalize db4 of
                           Right ndb -> install ndb (P "a") @?= e4
                           _ -> fail ""
            _ -> fail ""
      , testCase "Small special case" $ do
          eitherdb4 <- parseFile $ path ++ "test14"
          case eitherdb4 of
            Right db4 -> case normalize db4 of
                           Right ndb -> install ndb (P "a") @?= Nothing
                           _ -> fail ""
            _ -> fail ""
      ]
```

```haskell
  ]
  where
    pname = P "foo"
    ver = V [VN 1 ""]
    db = DB [Pkg pname ver "" []]
    e1 = Just [ (P "bar",V [VN 2 "",VN 1 ""])
              , (P "foo",V [VN 2 "",VN 3 ""]) ]
    e2 = Just [ (P "baz",V [VN 6 "",VN 1 "",VN 2 ""])
              , (P "bar",V [VN 1 "",VN 0 ""])
              , (P "foo",V [VN 2 "",VN 3 ""]) ]
    e3 = Just [ (P "foo",V [VN 2 "",VN 3 ""])
              , (P "baz",V [VN 6 "",VN 1 "",VN 2 ""])
              , (P "bar",V [VN 5 "ff",VN 32 ""])
              , (P "chrome",V [VN 3 "",VN 0 "aa"])]
    e4 = Just [ (P "b",V [VN 5 "",VN 0 ""])
              , (P "c",V [VN 4 "",VN 0 ""])
              , (P "d",V [VN 5 "",VN 0 ""])
              , (P "a",V [VN 4 "",VN 0 ""])]
    c1 = [ (P "bar", (True,  V [VN 1 ""], V [VN 1000000 ""]))
         , (P "foo", (False, V [VN 2 "", VN 3 ""], V [VN 4 ""]))
         , (P "baz", (True,  V [VN 1 "", VN 3 ""], V [VN 5 ""]))
         ]
    c2 = [ (P "bar", (True,  V [VN 1 "a"], V [VN 2 ""]))
         , (P "foo", (True, V [VN 2 "", VN 4 ""], V [VN 2 "", VN 7 ""]))
         , (P "baz", (False,  V [VN 2 "", VN 0 ""], V [VN 3 "", VN 5 ""]))
         ]
    c3 = [ (P "baz", (True,  V [VN 2 "", VN 0 ""], V [VN 3 "", VN 5 ""]))
         , (P "foo", (True,  V [VN 2 "", VN 4 ""], V [VN 2 "", VN 7 ""]))
         , (P "bar", (True,  V [VN 1 "a"], V [VN 2 ""]))
         ]
    c4 = [ (P "foo", (True,  V [VN 5 "", VN 0 ""], maxV)) ]
    c5 = [ (P "foo", (True,  minV, V [VN 2 "", VN 0 ""])) ]
    c6 = [ (P "foo", (False,  V [VN 3 "", VN 4 "", VN 4 ""], V [VN 3 "", VN 4 "", VN 2 ""])) ]
    c7 = [ (P "bar", (True,  minV, V [VN 2 "", VN 0 ""])) ]
    c8 = [(P "foo",(False,V [VN 3 "",VN 4 "",VN 4 ""],V [VN 3 "",VN 4 "",VN 2 ""])),(P "bar",(True,V [VN 0 ""],V [VN 2 "",VN 0 ""]))]
    c9 = [(P "foo", (True, V [VN 2 ""], V [VN 8 ""]))]
    c10 = [(P "foo", (False, V [VN 4 ""], V [VN 6 ""]))]
    c11 = [(P "foo", (True, V [VN 4 ""], V [VN 6 ""]))]
    c12 = [(P "foo", (True, V [VN 6 ""], V [VN 8 ""]))]
    c13 = [(P "foo", (True, V [VN 4 ""], V [VN 6 ""]))]
    c14 = [(P "e", (False, minV, maxV))]
    c15 = [(P "c", (True, minV, maxV))]
    c16 = [(P "e", (False, minV, maxV)), (P "c", (True, minV, maxV))]

main = defaultMain tests
```

B.1.2 QuickCheck properties: Properties.hs

```haskell
module Properties where

import Defs
-- import Solver
import Data.List
import Utils

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

-- for reference; may discard after implementing full install_c
install_c' :: InstallProp
install_c' _db _p Nothing = True
install_c' _db _p (Just []) = False
install_c' _db _p (Just _) = True

-- All packages (with the indicated versions) are actually available in the db
install_a :: InstallProp
install_a _ _ Nothing = True
install_a (DB db) _ (Just sol) = let db' = getNameVerList (DB db) in
  all (`elem` db') sol

-- Any package name may only occur once in the list; in particular, it is not
-- possible to install two different versions of the same package simultaneously
install_b :: InstallProp
install_b _ _ Nothing  = True
install_b _ _ (Just s) = allDiff s

-- The package requested by the user is in the list
install_c :: InstallProp
install_c db p _ =
    case p `ofSomeVersionIn` db of
        Just _  -> True
        Nothing -> False

-- For any package in the list,
-- all the packages it requires are also in the list
install_d :: InstallProp
install_d _ _ Nothing   = True
install_d _ _ (Just []) = True
install_d (DB db) _ (Just ((n, v):ss)) =
  -- Find the package dependencies in the database
  case find (\p -> name p == n && ver p == v) db of
    -- Once found, extract the requirements for that package
    Just pkg -> let depsList = requirementsOf pkg in
      -- check that for all required packages
      all (\(n, lo, hi) ->
            -- there is one package in the solution that satisfies it
            any (\(pn, v) -> (pn == n) && (lo <= v) && (v < hi)) ss)
        depsList
    -- If no package is found in the db, then something is definitely wrong
    Nothing  -> False
```

B.1.3 QuickCheck tests: `Main.hs`

```haskell
module Main where

import Defs
import Properties
import Solver (install, normalize)
import Utils (isWellFormed)

import Test.Tasty
import Test.Tasty.QuickCheck
import Data.List

instance Arbitrary PName where
  arbitrary = nameGenerator

names = [ return $ P "Package01"
        , return $ P "Package02"
        , return $ P "Package03"
        , return $ P "Package04"
        , return $ P "Package05"
        , return $ P "Package06"
        , return $ P "Package07"
        , return $ P "Package08"
        , return $ P "Package09"
        , return $ P "Package10"
        , return $ P "Package11"
        , return $ P "Package12"
        , return $ P "Package13"
        , return $ P "Package14"
        , return $ P "Package15"
        , return $ P "Package16"
        ]

-- nameGenerator = oneof [simpleNameGenerator, generalNameGenerator]
nameGenerator = oneof names

simpleNameGenerator = simpleName
generalNameGenerator = generalName

-- Simple name generator
asciiLetter    = elements $ ['a'..'z'] ++ ['A'..'Z']
alphaNumHyphen = elements $ ['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9'] ++ ['-']
digits         = elements ['0' .. '9']

simpleName = do
  h <- asciiLetter
  n <- choose (0, 20)
  s <- vectorOf n alphaNumHyphen
  return $ P $ h : s

fieldsGenerator = do
```

```haskell
  n <- choose (1,5)
  vectorOf n fieldGenerator

fieldGenerator = do
  n <- choose (1,3)
  numeral <- vectorOf n digits
  suffix  <- oneof [return "", vectorOf n asciiLetter]
  return $ VN (read numeral) suffix

versionGenerator = V <$> fieldsGenerator

descriptionGenerator = do
  n <- choose (0, 10)
  vectorOf n alphaNumHyphen

constraintsGenerator = do
  n <- choose (0, 2)
  vectorOf n constraintGen

genConstraintWithName n = do
  name <- n
  bool <- elements [True, False]
  v1 <- oneof [versionGenerator, return minV]
  v2 <- oneof [versionGenerator, return maxV]
  if v1 <= v2 then return (name, (bool, v1, v2))
  else return (name, (bool, v2, v1))

constraintGen = do
  name <- nameGenerator
  bool <- elements [True, False]
  v1 <- oneof [versionGenerator, return minV]
  v2 <- oneof [versionGenerator, return maxV]
  if v1 <= v2 then return (name, (bool, v1, v2))
  else return (name, (bool, v2, v1))

instance Arbitrary Database where
  arbitrary = databaseGenerator
  -- the shrink simply takes all combinations of the db with one package removed
  shrink (DB db) = do
    p <- db
    return $ DB $ delete p db

databaseGenerator = do
  n <- choose (0,16)
  packages <- vectorOf n packageGenerator
  return $ DB packages

satisfiableConstraints p =
  case isWellFormed [deps p] of Just _ -> True; _ -> False

pName p = let nms = filter (\(n, _) -> name p == n) (deps p) in null nms
```

```

isWellFormedBool p = satisfiableConstraints p && pName p

packageGenerator = unsafePackageGen ‘suchThat‘ isWellFormedBool

unsafePackageGen = do
  name    <- nameGenerator
  version <- versionGenerator
  desc    <- descriptionGenerator
  Pkg name version desc <$> constraintsGenerator

genPackWithName :: PName -> Gen Pkg
genPackWithName pn = do
  ver <- versionGenerator
  des <- descriptionGenerator
  Pkg pn ver des <$> constraintsGenerator

-----------------------------------
-- Actual properties for testing
-----------------------------------

prop_install_a db p =
  case normalize db of
    Right d -> install_a d p (install db p)
    Left _ -> True

prop_install_b db p = install_b db p (install db p)

prop_install_c (DB db) p = do
  -- for whatever package name p that was generated, make a package for it
  pkg <- genPackWithName p
  -- and put it in the database before installing it
  let db’ = pkg:db in case normalize (DB db’) of
    Right d -> return $ install_c d p (install d p)
    Left  _ -> return True

prop_install_d db p = install_d db p (install db p)

tests = testGroup "QC tests" [ testProperty "Prop (a)" prop_install_a
                             , testProperty "Prop (b)" prop_install_b
                             , testProperty "Prop (c)" prop_install_c
                             , testProperty "Prop (d)" prop_install_d
                             ]

main = defaultMain tests

generalName = do
  n <- choose (0, 20)
  s <- vectorOf n (oneof [vectorOf 2 alphaNumHyphen, return "\"\""])
  return $ P $ [’\"’] ++ concat s ++ [’\"’]
```

B.2 Earls of Ravnica

B.2.1 Unit tests: unittests.erl

```
-module(unittests).
-include_lib("eunit/include/eunit.hrl").

-on_load(setup/0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% How to run tests from current folder:       %%
%%  (1) in erl, type ’c(unittests).’ and enter %%
%%  (2) type ’unittests:test().’               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup() ->
  compile:file(district),
  ok.

create_test() -> {A, _} = district:create("A"), [?assertEqual(ok, A)].

% Get description from district under configuration
get_description1_test() ->
  Expected = "test",
  {ok, A} = district:create(Expected),
  {ok, Actual} = district:get_description(A),
  [?assertEqual(Expected, Actual)].

% Get description from active district
get_description2_test() ->
  Expected = "test",
  {ok, A} = district:create(Expected),
  active = district:activate(A),
  {ok, Actual} = district:get_description(A),
  [?assertEqual(Expected, Actual)].

connect_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  Return = district:connect(A, t, B),
  Return = district:connect(B, t, C),
  [?assertEqual(ok, Return)].

active_simple_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  Return = district:activate(A),
  [?assertEqual(active, Return)].
```

```erlang

active_selfloop_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(A, t2, A),
  Return = district:activate(A),
  [?assertEqual(active, Return)].

active_cycle_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(C, t, A),
  Return = district:activate(A),
  [?assertEqual(active, Return)].

active_cycleandloop_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(C, t, A),
  ok = district:connect(A, t1, A),
  Return = district:activate(A),
  [?assertEqual(active, Return)].

active_double_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(C, t, A),
  ok = district:connect(A, t1, A),
  Return = district:activate(A),
  [?assertEqual(active, Return)].

actions1_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(A, t1, A),
  Return = district:options(A),
  [?assertEqual({ok, [t, t1]}, Return)].

actions2_test() ->
```

```erlang
  {ok, A} = district:create("Test"),
  Return = district:options(A),
  [?assertEqual({ok, []}, Return)].

actions3_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  ok = district:connect(A, aa, A),
  active = district:activate(A),
  Return1 = district:options(A),
  Return2 = district:options(B),
  [?assertEqual({ok, [a, aa]}, Return1), ?assertEqual({ok, [b]}, Return2)].

enter1_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  ok = district:connect(A, aa, A),
  Creature = {make_ref(), #{}},
  Return = district:enter(A, Creature),
  Msg = "Under configuration. No creatures are allowed to enter.",
  [?assertEqual({error, Msg}, Return)].

enter2_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  ok = district:connect(A, aa, A),
  active = district:activate(A),
  Creature = {make_ref(), #{}},
  Return = district:enter(A, Creature),
  [?assertEqual(ok, Return)].

enter3_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  ok = district:connect(A, aa, A),
  active = district:activate(A),
  Creature = {make_ref(), #{}},
  Return1 = district:enter(A, Creature),
  Return2 = district:enter(A, Creature),
  Msg = "A creature is already in the district.",
  [ ?assertEqual(ok, Return1)
  , ?assertEqual({error, Msg}, Return2)].

take_action1_test() ->
```

```
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  active = district:activate(A),
  {CRef, _} = Creature = {make_ref(), #{}},
  ok = district:enter(A, Creature),
  ok = district:enter(B, Creature),
  Return = district:take_action(A, CRef, a),
  Msg = "Creature could not enter new district.",
  [ ?assertEqual({error, Msg}, Return)].

take_action2_test() ->
  {ok, A} = district:create("A"),
  {ok, B} = district:create("B"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  active = district:activate(A),
  {CRef, _} = Creature = {make_ref(), #{}},
  ok = district:enter(A, Creature),
  Return = district:take_action(A, CRef, a),
  Me = self(),
  ok = district:shutdown(B, Me),
  receive
    {_, {shutting_down, From, Cs}} ->
      ?assertEqual(From, B),
      ?assertEqual(Cs, [Creature])
  end,

  [ ?assertEqual({ok, B}, Return)].

take_action3_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  ok = district:connect(A, a, B),
  ok = district:connect(B, b, A),
  active = district:activate(A),
  {CRef, _} = Creature = {make_ref(), #{}},
  ok = district:enter(A, Creature),
  ok = district:enter(B, Creature),
  Return = district:take_action(A, CRef, c),
  Msg = "Either action or creature does not exist here.",
  [ ?assertEqual({error, Msg}, Return)].

take_action4_test() ->
  {ok, A} = district:create("Test"),
  ok = district:connect(A, a, A),
  active = district:activate(A),
  {CRef, _} = Creature = {make_ref(), #{}},
  ok = district:enter(A, Creature),
  Return = district:take_action(A, CRef, a),
  [ ?assertEqual({ok, A}, Return)].
```

```

shutdown_simple_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  {ok, NextPlane} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  Return1 = district:activate(A),
  Return2 = district:shutdown(A, NextPlane),
  try district:get_description(A),
      error
  catch exit:_ ->
    [ ?assertEqual(active, Return1)
    , ?assertEqual(ok, Return2)
    ]
  end.

shutdown_cycle_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  {ok, NextPlane} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(C, t, A),
  ok = district:connect(A, t1, A),
  Return1 = district:activate(A),
  Return2 = district:shutdown(A, NextPlane),
  try district:get_description(A),
      error
  catch exit:_ ->
    [ ?assertEqual(active, Return1)
    , ?assertEqual(ok, Return2)
    ]
  end.

shutdown_selfloop_test() ->
  {ok, A} = district:create("Test"),
  {ok, B} = district:create("Test"),
  {ok, C} = district:create("Test"),
  {ok, NextPlane} = district:create("Test"),
  ok = district:connect(A, t, B),
  ok = district:connect(B, t, C),
  ok = district:connect(C, t, A),
  ok = district:connect(A, t1, A),
  Return1 = district:activate(A),
  Return2 = district:shutdown(A, NextPlane),
  try district:get_description(A),
      error
  catch exit:_ ->
    [ ?assertEqual(active, Return1)
    , ?assertEqual(ok, Return2)
    ]
  end.
```

B.2.2 Unit tests: `triggertest.erl`

```erlang
% Example contributed by Joachim and Mathias
-module(triggertest).
-export([test/0]).

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

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

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

rest_a_bit(entering, Creature, Creatures) ->
    io:format("Sob..~n"),
    {make_sober(Creature), Creatures};
rest_a_bit(leaving, Creature, Creatures) ->
    {Creature, Creatures}.

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

lille_up1(entering, {CreatureRef, Stats}, Creatures, KenRef, AndrzejRef) ->
    CreatureRefs = lists:map(fun({Ref, _Stats}) -> Ref end, Creatures),
    KenPresent = lists:member(KenRef, CreatureRefs),
    AndrzejPresent = lists:member(AndrzejRef, CreatureRefs),
    if KenPresent and AndrzejPresent ->
        io:format("Surprise! Ken and Andrzej are here!~n"),
        {{CreatureRef, Stats#{stunned => true}}, Creatures};
       true ->
        {{CreatureRef, Stats}, Creatures}
    end;
lille_up1(leaving, _Creature, _Creatures, _KenRef, _AndrzejRef) ->
    io:format("Someone is leaving LilleUP1! This trigger should fail.~n"),
    % This is misbehaving, thus the trigger has no effect
    ok.

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

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

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

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

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

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

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

test() ->
    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},

    % io:fwrite("run_world(): We now enter, triggering triggers.~n", []),
    % 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),


    % io:fwrite("run_world(): We now take action, triggering triggers.~n",
    []),
    % =====| Following two lines changed in ver. 1.0.1 | =====
    {ok, _} = district:take_action(KensOffice, KenRef, restore_health),
    {ok, _} = district:take_action(AndrzejsOffice, AndrzejRef, sneak),
    {ok, _} = district:take_action(Cafeen, PrebenRef, need_to_pee),

    % 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
    ),
    Student = hd(StudentRefs),
    {ok, _} = district:take_action(Canteen, Student, have_courage),
    {ok, _} = district:take_action(LilleUP1, Student, rejuvenate),
    {KensOffice, AndrzejsOffice, Canteen}.
```

B.2.3 QuickCheck tests: district_qc.erl

```erlang
-module(district_qc).

-export([ex/0, setup_territory/1, territory/0]).

-export([prop_activate/0, prop_take_action/0]).

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

% Choose between SIZE different districts
-define(SIZE, 10).

% Used for connections between districts. atom() is basically an action.
atom() -> elements([a, b, c, d, e, f, g, h, i, j, k, l]).

% Generates an integer between 1 and SIZE
integer() -> choose(1, ?SIZE).

% Makes a list [{atom(), integer()}] of variable length; expected length about 5
connections() ->
    ?LAZY((frequency([{1, []},
          {5,
           ?LETSHRINK([L], [connections()],
          {call, lists, keystore,
           ?LETSHRINK([K], [atom()],
                [K, 1, L, {K, integer()}])})}]))).

% uses a map to create a territory
territory() ->
    ?LAZY((frequency([{1, {call, maps, new, []}},
          {4,
           ?LETSHRINK([M], [territory()],
          {call, maps, put,
           [integer(), connections(), M]})}]))).

% helper function used in setup_territory to connect districts
connectToNeighbours(From, A, L) ->
    lists:foldl(fun ({Action, IntTo}, A_To) ->
      % Is the neighbour already created?
      IsToCreated = maps:is_key(IntTo, A_To),
      % if so, we just connect and go on
      if IsToCreated ->
             To = maps:get(IntTo, A_To),
             district:connect(From, Action, To),
             A_To;
         % otherwise, we have to create the neighbour before connect
         true ->
             {ok, To} = district:create("District " ++ IntTo),
             % return the map, where we add the neighbour as well
             maps:put(IntTo, To, A_To)
      end
    end,
    A, L).

setup_territory(InputMap) ->
    % The accumulator is a map, pairing each integer with a district
    Map = maps:fold(fun (IntFrom, L, A_From) ->
          % Check if we have already created the district D
          IsFromCreated = maps:is_key(IntFrom, A_From),
          % if so, we find its pid and go fold through its neighbours
          if IsFromCreated ->
           % The district id associated with the district integer
           From = maps:get(IntFrom, A_From),
           connectToNeighbours(From, A_From, L);
             % This is the case if we have not already created From
             true ->
           % Create the district
           {ok, From} = district:create("District " ++
                  IntFrom),
           % Add the district to the map
           APlus = maps:put(IntFrom, From, A_From),
           % and connect to neighbours
           connectToNeighbours(From, APlus, L)
          end
        end,
        #{}, InputMap),
    maps:values(Map).

prop_activate() ->
    % We want to check that for all active districts,
    % any neighbour for such a given district is active as well.
    % However, since this is done by taking action -> see prop_take_action
    ()
    % This property just tests that all districts can be activated
    ?FORALL(Xs, (territory()),
      begin
        Eval = eval(Xs),
        % First, setup the world
        World = setup_territory(Eval),
        lists:all(fun (District) ->
        Check = district:activate(District),
        % check that the district is then actually active or at least
        % under activation
        if (Check == active) or (Check == under_activation) -> true;
           true -> false
        end
      end,
      World)
      end).

prop_take_action() ->
    ?FORALL(Xs, (territory()),
      begin
        Eval = eval(Xs),
        % First, setup the world
        World = setup_territory(Eval),
        lists:all(fun (District) ->
        Check = district:activate(District),
        if (Check == active) or (Check == under_activation) ->
            % get possible actions on the district
              {ok, Actions} = district:options(District),
            % for each action
              lists:foreach(fun (Action) ->
              % make a creature and make it enter the district
              Creature = {CRef, _} = {make_ref(), #{}},
                ok = district:enter(District, Creature),
              % use the action with that creature
                {ok, _} = district:take_action(District, CRef, Action)
              % ideally, one would want to check that the creature
              % ends up at the receiving end, which could be done
              % by shutting down the receiving district, and make
              % it send its creatures to us as 'next plane'
              % However, when testing the districts, I do not want
              % to shut down some of the districts, just to want to
              % do something with them later. Maybe I could have
              % used ?IMPLIES, to work around this. But no time.
            end, Actions),
              true;
           true -> false
        end
      end,
      World)
      end).
```

References

[1] Peter Sestoft and Ken Friis Larsen. *Grammars and parsing with Haskell Using Parser Combinators.*