Advanced Programming

Exam 2018

Silvan Robert Adrian

November 6, 2018

Contents

1 Question 1.1: Utility functions

The Code for this task is attached in the appendix A.1.

1.1 Version

2 Question 1.2: Parsing appm databases

2.1 Choice of parser library

I implemented the Parser for appm in parsec, mostly out of this reason:

- Better Error handling compared to ReadP
- I do have more experience with Parsec then ReadP

2.2 Transform Grammar

The existing grammar has some ambiguities, like allowing many names, version etc. which now transformed to only allow once

```
Database ::= \epsilon
```

3 Earls of Ravnica

The code for this task can be found in Appendix

3.1 Solution

3.2 Implementation

The earls of Ravnica can be seen as a state machine for which I chose to use gen_statem. The following states exist:

- Under Configuration
- Under Activation
- Active
- Shutting down

3.3 Data Structure

The Data structure I used to implement Ravnica consists of a map with following entries:

- **description** Saves the description which gets saved when starting a server
- **connections** Map for Handling the connections from one District to an other
- **creatures** Map for handling all the entered/active creatures on a Server
- **trigger** Set a trigger for a district

3.4 All states

Messages which get accepted in all states.

3.4.1 get_description

Gets the description `Desc` which gets set on create of a District.

3.5 Under configuration

As soon as a Server started it is in the under_configuration state.

3.5.1 connect

Connects 2 District with a Action, by saving it in the `connections` map, connects can only be made while district is under configuration in other states an error gets returned.

3.5.2 trigger

Under configuration also a trigger can be added to the server, here always the last one gets taken (overwritting whit the newest one).

3.6 Under activation

When `activate` gets called the district and it's neighbors need to get activated, under_activation is a intermediate state until all neighbors and the district itself are activated. In case the neighbors can't be activated (for example when a neighbor got shutdown), then the server goes back to the state of under_configuration.

3.6.1 activate

Activate tries to activate all it's neighbors and changes the state of the server to `active` or back to `under_configuration`.

3.7 Active

In the active state, no more new connections can be added

3.8 Shutting down

3.9 Territories with cycle

A Code Listing

A.1 Question 1.1: handin/appm/src/Utils.hs

```haskell
module Utils where

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

import Defs

instance Ord Version where
  (<=) (V[]) _ = False
  (<=) (V(_:_)) (V []) = True
  (<=) (V[VN v1int v1str])  (V[VN v2int v2str]) =
    if checkVersion v1int v2int v1str v2str then True else False
  (<=) (V(VN _ _:xs))  (V(VN _ _:ys)) = V(xs) <= V(ys)

checkVersion :: Int -> Int -> String -> String -> Bool
checkVersion a b c d = a <= b && (c <= d || length(c) <= length(d))

merge :: Constrs -> Constrs -> Maybe Constrs
merge [] [] = Just []
merge c1 [] = Just c1
merge [] c2 = Just c2
-- merge ((pname1,(bool1, miv1, mxv1)):xs) ((pname2,(bool2,
↪  miv2,mxv2)):ys) =
--                                                           if miv1 <=
↪  (miv2 <= mxv1)  then
```

```
--                                                                      if miv1
↪  <= (mxv1 <= mxv2) then
--
↪  Just [(pname2, (bool1,miv1, mxv2))]
--                                                                      else
--
↪  Nothing
--                                                                 else
--
↪  Nothing
merge (c1) (c2) = Just (c1 ++ c2)
```

A.2 Question 2.1: handin/ravnica/district.erl

```
-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 callbacks
-export([terminate/3, code_change/4, init/1, callback_mode/0]).
%State Functions
-export([under_configuration/3, active/3, shutting_down/3,
↪  under_activation/3]).
-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(?MODULE, Desc, []).

```

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

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

-spec activate(passage()) -> active | under_activation | impossible.
activate(District) ->
    gen_statem:call(District, activate).

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

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

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

-spec shutdown(passage(), pid()) -> ok.
shutdown(District, NextPlane) ->
    gen_statem:call(District, {shutdown, NextPlane}).

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


%% States
handle_event({call,From}, get_description, Data) ->
  case maps:is_key(description, Data) of
   true -> {keep_state, Data, {reply, From, {ok, maps:get(description,
   ↪  Data)}}};
   false ->  {error, "No Description"}
  end;

handle_event({call, From}, options, Data) ->
```

```
  {keep_state, Data, {reply, From, {ok,
  ↪  maps:keys(maps:get(connections,Data))}}};

% ignore all other unhandled events
handle_event({call, From}, activate_instantion, Data) ->
  {next_state, active, Data, {reply, From, ok}};

handle_event({call, From}, {run_action, CRef}, Data) ->
  case maps:is_key(CRef, maps:get(creatures, Data)) of
    true -> {keep_state, Data, {reply, From, {error, "Creature is already
    ↪  in this District"}}};
    false ->  NewCreatures = maps:put(CRef, empty, maps:get(creatures,
    ↪  Data)),
              NewData = maps:update(creatures,NewCreatures,Data),
              {keep_state, NewData, {reply, From, ok}}
  end;

% Handle Enter on other states
handle_event({call, From}, {enter, _}, Data) ->
  {keep_state, Data, {reply, From, {error, "Can't enter in this state"}}};

% Shutdown can be called in any state
handle_event({call, From}, {shutdown, NextPlane}, Data) ->
  NextPlane ! {shutting_down, From, maps:to_list(maps:get(creatures,
  ↪  Data))},
  {next_state, shutting_down, Data, {next_event, internal, {From,
  ↪  NextPlane}}};

handle_event({call, From}, {trigger, _Trigger}, Data) ->
  {keep_state, Data, {reply, From, {error, "Can't set a trigger in this
  ↪  state"}}};

handle_event({call, From}, {connect, _Action, _To}, Data) ->
  {keep_state, Data, {reply, From, {error, "Can't connect in this
  ↪  state"}}};

% ignore all other unhandled events
handle_event(_EventType, _EventContent, Data) ->
  {keep_state, Data}.

under_configuration({call, From}, {connect, Action, To}, Data) ->
  case maps:is_key(Action, maps:get(connections,Data)) of
    false -> Connections = maps:put(Action, To, maps:get(connections,
    ↪  Data)),
      NewData = maps:update(connections, Connections, Data),
      {keep_state, NewData, {reply, From, ok}};
  true -> {keep_state, Data, {reply, From, {error, "Action already
  ↪  exists"}}}
 end;

under_configuration({call, From}, activate, Data) ->
  {next_state, under_activation, Data, {next_event, internal, From}};


under_configuration({call, From}, {trigger, Trigger}, Data) ->
 NewData = maps:update(trigger, Trigger, Data),
 {keep_state, NewData, {reply, From, ok}};

%% General Event Handling for state under_configuration
under_configuration(EventType, EventContent, Data) ->
 handle_event(EventType, EventContent, Data).

under_activation(internal, From, Data) ->
 Result = broadcast_connection(maps:to_list(maps:get(connections, Data)),
 ↪  active),
 case Result of
  impossible ->   {next_state, under_configuration, Data, {reply, From,
  ↪  Result}};
  active ->   {next_state, active, Data, {reply, From, Result}}
 end;

under_activation({call, From}, activate, Data) ->
 {keep_state, Data, {reply, From, under_activation}};

under_activation(_EventType, _EventContent, Data) ->
 {keep_state, Data}.

active({call, From}, {enter, {Ref, Stats}}, Data) ->
 case maps:is_key(Ref, maps:get(creatures, Data)) of
  true -> {keep_state, Data, {reply, From, {error, "Creture is already
  ↪  in this District"}}};
  false ->  NewCreatures = maps:put(Ref, Stats, maps:get(creatures,
  ↪  Data)),
       NewData = maps:update(creatures, NewCreatures, Data),
       case maps:get(trigger, Data) of
           none -> void;
           Trigger -> Trigger(entering, {Ref, Stats},
           ↪  maps:to_list(NewCreatures))
       end,
       {keep_state, NewData, {reply, From, ok}}
 end;

active({call, From}, {take_action, CRef, Action}, Data) ->
 case maps:is_key(Action, maps:get(connections, Data)) of
  true ->
   case maps:is_key(CRef, maps:get(creatures, Data)) of
    false -> {keep_state, Data, {reply, From, {error, "Creature
    ↪  doesn't exist in this district"}}};
    true -> {NewData, To} = creature_leave(CRef, Action, Data),
        case NewData of
         error -> {keep_state, Data, {reply, From, {error, To}}};
         _ -> case maps:get(trigger, Data) of
               none -> void;
               Trigger -> Trigger(leaving, maps:get(CRef,
               ↪  maps:get(creatures, Data)),
               ↪  maps:to_list(maps:get(creatures, Data)))
              end,
           {keep_state, NewData, {reply, From, {ok, To}}}
        end
   end;
  false -> {keep_state, Data, {reply, From, {error, "Action doesn't
  ↪  exist"}}}
 end;

active({call, From}, activate, Data) ->
 {keep_state, Data, {reply, From, active}};

%% Handle Calls to active
active(EventType, EventContent, Data) ->
 handle_event(EventType, EventContent, Data).

shutting_down(internal, {From, NextPlane}, Data) ->
 case broadcast_shutdown(maps:to_list(maps:get(connections, Data)),
 ↪  NextPlane, ok) of
  ok -> {stop_and_reply, normal, {reply, From, ok}};
  nok -> {stop_and_reply, normal, {reply, From, impossible}}
 end;

shutting_down({call, From}, activate, Data) ->
 {keep_state, Data, {reply, From, impossible}};

shutting_down({call, From}, options, Data) ->
```

```erlang
  {keep_state, Data, {reply, From, none}}.

%% Mandatory callback functions
terminate(_Reason, _State, _Data) ->
  void.

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

% initial State under_configuration
init(Desc) ->
  %% Set the initial state + data
  State = under_configuration, Data = #{description => Desc, connections
  ↪  => #{}, creatures => #{}, trigger => none},
  {ok, State, Data}.

callback_mode() -> state_functions.

%% Synchronous Call which should wait until each response
broadcast_shutdown([], _NextPlane, Result) -> Result;
broadcast_shutdown([{_Action, To} | Actions ], NextPlane, Result) ->
  case gen_statem:call(To, {shutdown, NextPlane}) of
    ok -> Result = ok;
    _ -> Result = nok
  end,
  broadcast_shutdown(Actions, NextPlane, Result).

%% Synchronous Call which should wait until each response
broadcast_connection([], Result) -> Result;
broadcast_connection([{_Action, To} | Actions ], _) ->
  case is_process_alive(To) of
    false -> Result1 = impossible;
    true ->  Result1 = active, gen_statem:call(To, activate_instantion)
  end,
  broadcast_connection(Actions, Result1).

creature_leave(CRef, Action, Data) ->
  To = maps:get(Action, maps:get(connections, Data)),
  case gen_statem:call(To, {run_action, CRef}) of
    ok -> NewCreatures = maps:remove(CRef, maps:get(creatures, Data)),
          NewData = maps:update(creatures, NewCreatures, Data),
          Return = {NewData, To};
    {error, Reason} -> Return = {error, Reason}
  end,
Return.
```

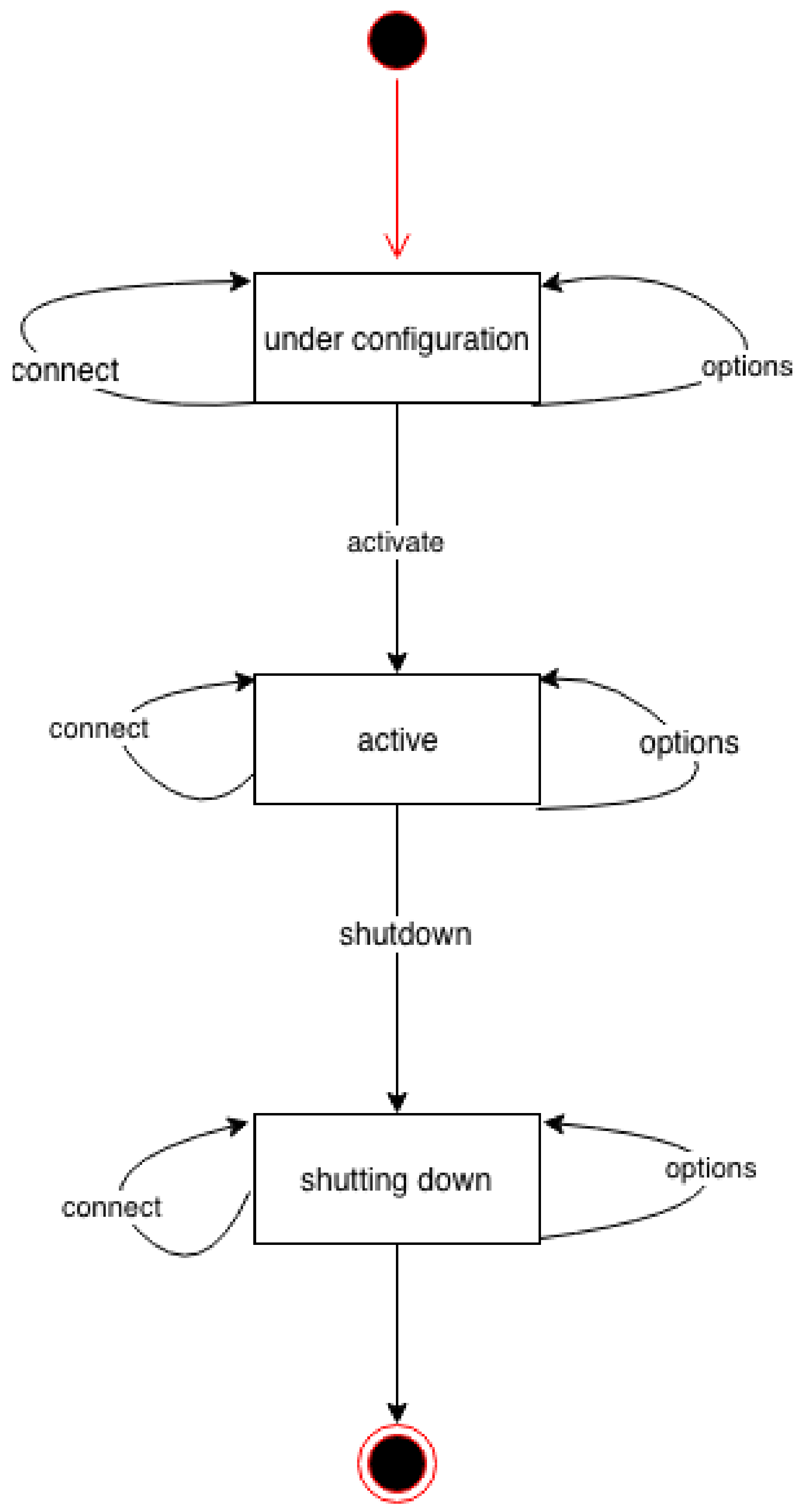

Figure 1: Simple State machine diagramm