Advanced Programming

Exam 2018

Silvan Robert Adrian

November 5, 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

```
1 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
- 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

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
```

```
25  --
    ↪  Just [(pname2, (bool1,miv1, mxv2))]
26  --                                                                  else
27  --
    ↪  Nothing
28  --                                                            else
29  --
    ↪  Nothing
30  merge (c1) (c2) = Just (c1 ++ c2)
```

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

```
1   -module(district).
2   -behaviour(gen_statem).
3   -export([create/1,
4            get_description/1,
5            connect/3,
6            activate/1,
7            options/1,
8            enter/2,
9            take_action/3,
10           shutdown/2,
11           trigger/2]).
12  %% Gen_statem callbacks
13  -export([terminate/3, code_change/4, init/1, callback_mode/0]).
14  %State Functions
15  -export([under_configuration/3, active/3]).
16  -type passage() :: pid().
17  -type creature_ref() :: reference().
18  -type creature_stats() :: map().
19  -type creature() :: {creature_ref(), creature_stats()}.
20  -type trigger() :: fun((entering | leaving, creature(), [creature()])
21                                                      -> {creature(),
                                                        ↪  [creature()]}).
22
23
24  -spec create(string()) -> {ok, passage()} | {error, any()}.
25  create(Desc) ->
26      gen_statem:start(?MODULE, Desc, []).
27
28  -spec get_description(passage()) -> {ok, string()} | {error, any()}.
29  get_description(District) ->
30      gen_statem:call(District, get_description).
```

```
31
32  -spec connect(passage(), atom(), passage()) -> ok | {error, any()}.
33  connect(From, Action, To) ->
34      gen_statem:call(From, {connect, Action, To}).
35
36  -spec activate(passage()) -> active | under_activation | impossible.
37  activate(District) ->
38      gen_statem:call(District, activate).
39
40  -spec options(passage()) -> {ok, [atom()]} | none.
41  options(District) ->
42      gen_statem:call(District, options).
43
44  -spec enter(passage(), creature()) -> ok | {error, any()}.
45  enter(District, Creature) ->
46      gen_statem:call(District, {enter, Creature}).
47
48  -spec take_action(passage(), creature_ref(), atom()) -> {ok, passage()} |
    ↪  {error, any()}.
49  take_action(From, CRef, Action) ->
50      gen_statem:call(From, {take_action, CRef, Action}).
51
52  -spec shutdown(passage(), pid()) -> ok.
53  shutdown(District, NextPlane) ->
54      gen_statem:call(District, {shutdown, NextPlane}).
55
56  -spec trigger(passage(), trigger()) -> ok | {error, any()} |
    ↪  not_supported.
57  trigger(_, _) ->
58      not_supported.
59
60
61  %% States
62  handle_event({call,From}, get_description, Data) ->
63    case maps:is_key(description, Data) of
64     true -> {keep_state, Data, {reply, From, {ok, maps:get(description,
       ↪  Data)}}};
65     false ->  {error, "No Description"}
66    end;
67
68  handle_event({call, From}, options, Data) ->
69    {keep_state, Data, {reply, From, {ok,
      ↪  maps:keys(maps:get(connections,Data))}}};
70
```

```
71  % ignore all other unhandled events
72  handle_event({call, From}, activate_instantion, Data) ->
73    {next_state, active, Data, {reply, From, ok}};
74
75  handle_event({call, From}, {run_action, CRef}, Data) ->
76    case maps:is_key(CRef, maps:get(creatures, Data)) of
77      true -> {keep_state, Data, {reply, From, {error, "Creature is already
       ↪  in this District"}}};
78      false ->  NewCreatures = maps:put(CRef, empty, maps:get(creatures,
       ↪  Data)),
79                NewData = maps:update(creatures,NewCreatures,Data),
80                {keep_state, NewData, {reply, From, ok}}
81    end;
82
83  % Handle Enter on other states
84  handle_event({call, From}, {enter, _}, Data) ->
85    {keep_state, Data, {reply, From, {error, "Can't enter in this state"}}};
86
87  % Shutdown can be called in any state
88  handle_event({call, From}, {shutdown, NextPlane}, Data) ->
89    NextPlane ! {shutting_down, From, maps:to_list(maps:get(creatures,
       ↪  Data))},
90    broadcast_shutdown(maps:to_list(maps:get(connections, Data)),
       ↪  NextPlane),
91    {stop_and_reply, normal, {reply, From, ok}};
92
93  % ignore all other unhandled events
94  handle_event(_EventType, _EventContent, Data) ->
95    {keep_state, Data}.
96
97  under_configuration({call, From}, {connect, Action, To}, Data) ->
98    case maps:is_key(Action, maps:get(connections,Data)) of
99      false -> Connections = maps:put(Action, To, maps:get(connections,
       ↪  Data)),
100             NewData = maps:update(connections, Connections, Data),
101             {keep_state, NewData, {reply, From, ok}};
102     true -> {keep_state, Data, {reply, From, {error, "Action already
       ↪  exists"}}}
103   end;
104
105 under_configuration({call, From}, activate, Data) ->
106     case broadcast_connection(maps:to_list(maps:get(connections, Data)))
       ↪  of
107       active -> {next_state, active, Data, {reply, From, active}};
```

```
108         _ -> {next_state, active, Data, {reply, From, impossible}}
109       end;
110
111   %% General Event Handling for state under_configuration
112   under_configuration(EventType, EventContent, Data) ->
113     handle_event(EventType, EventContent, Data).
114
115   active({call, From}, {enter, {Ref, Stats}}, Data) ->
116     case maps:is_key(Ref, maps:get(creatures, Data)) of
117       true -> {keep_state, Data, {reply, From, {error, "Creture is already
          ↪  in this District"}}};
118       false ->  NewCreatures = maps:put(Ref, Stats, maps:get(creatures,
          ↪  Data)),
119                 NewData = maps:update(creatures, NewCreatures, Data),
120                 {keep_state, NewData, {reply, From, ok}}
121     end;
122
123   active({call, From}, {take_action, CRef, Action}, Data) ->
124     case maps:is_key(Action, maps:get(connections, Data)) of
125       true ->
126         case maps:is_key(CRef, maps:get(creatures, Data)) of
127           false -> {keep_state, Data, {reply, From, {error, "Creature
              ↪  doesn't exist in this district"}}};
128           true -> {NewData, To} = creature_leave(CRef, Action, Data),
129                   case NewData of
130                     error -> {keep_state, Data, {reply, From, {error, To}}};
131                     _ ->      {keep_state, NewData, {reply, From, {ok, To}}}
132                   end
133         end;
134       false -> {keep_state, Data, {reply, From, {error, "Action doesn't
          ↪  exist"}}}
135     end;
136
137   %% Handle Calls to active
138   active(EventType, EventContent, Data) ->
139     handle_event(EventType, EventContent, Data).
140
141   %% Mandatory callback functions
142   terminate(_Reason, _State, _Data) ->
143     void.
144
145   code_change(_Vsn, State, Data, _Extra) ->
146     {ok, State, Data}.
147
```

```
148 % initial State under_configuration
149 init(Desc) ->
150   %% Set the initial state + data
151   State = under_configuration, Data = #{description => Desc, connections
      ↪  => #{}, creatures => #{}},
152   {ok, State, Data}.
153
154 callback_mode() -> state_functions.
155
156 %% Synchronous Call which should wait until each response
157 broadcast_shutdown([], _NextPlane) -> ok;
158 broadcast_shutdown([{_Action, To} | Actions ], NextPlane) ->
159   gen_statem:call(To, {shutdown, NextPlane}),
160   broadcast_shutdown(Actions, NextPlane).
161
162 %% Synchronous Call which should wait until each response
163 broadcast_connection([]) -> active;
164 broadcast_connection([{_Action, To} | Actions ]) ->
165   gen_statem:call(To, activate_instantion),
166   broadcast_connection(Actions).
167
168 creature_leave(CRef, Action, Data) ->
169   To = maps:get(Action, maps:get(connections, Data)),
170   case gen_statem:call(To, {run_action, CRef}) of
171     ok -> NewCreatures = maps:remove(CRef, maps:get(creatures, Data)),
172           NewData = maps:update(creatures, NewCreatures, Data),
173           Return = {NewData, To};
174     {error, Reason} -> Return = {error, Reason}
175   end,
176   Return.
```

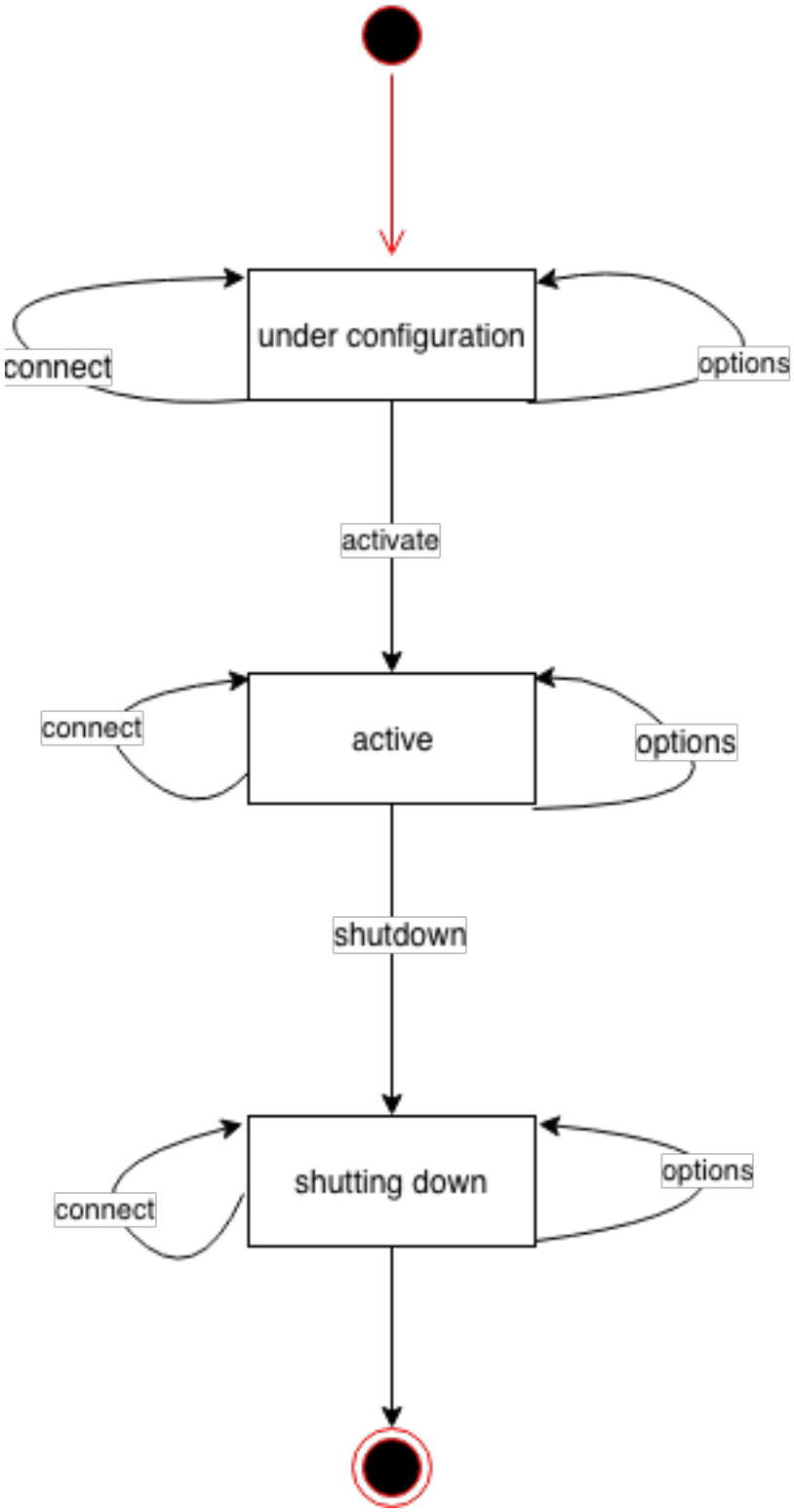

Figure 1: Simple State machine diagramm