Advanced Programming

Exam 2018

Exam Number: 95, Username: zlp432

November 8, 2018

Contents

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 Solver

4 Earls of Ravnica

The code for this task can be found in Appendix

4.1 Solution

4.2 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

4.3 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

4.4 All states

Messages which get accepted in all states.

4.4.1 get_description

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

4.5 Under configuration

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

4.5.1 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.

4.5.2 trigger

Under configuration also a trigger can be added to the server, here always the last one gets taken (overwritting whit the newest one). Trigger gets rung whenever a creature enters or leaves a district.

4.6 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.

4.6.1 activate

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

4.7 Active

In the active state, no more new connections can be added, also no triggers. So as soon as a district and it's neighbors is activated, it should only be possible to either run `get_description`, `enter` or `take_action` and of course shutting down.

4.8 Shutting down

When shutting down is called all neighbors of a district will be shut down as well and this can be propagated until all districts and it's nieghbors are shutdown.

4.8.1 shutdown

4.9 Territories with cycle

A Code Listing

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

```
1 module Utils where
2
```

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

import Defs

instance Ord Version where
    (<=) (V []) (V []) = True
    (<=) (V ((VN _ _):_)) (V []) = False
    (<=) (V []) (V ((VN _ _):_)) = True
    (<=) (V ((VN v1int v1str) : vnmbr1)) (V ((VN v2int v2str) : vnmbr2))
        | v1int < v2int = True
        | v1int > v2int = False
        | length(v1str) < length(v2str) = True
        | length(v1str) > length(v2str) = False
        | v1str < v2str = True
        | v1str > v2str = False
        | otherwise = (V vnmbr1) <= (V vnmbr2)

merge :: Constrs -> Constrs -> Maybe Constrs
merge [] [] = Just []
merge c1 [] = Just c1
merge [] c2 = Just c2
merge (const:c1) (c2) = case constInC2 const c2 [] of
                            Just x -> merge c1 (x)
                            Nothing -> Nothing

-- Check if Constraint from c1 is in the Constraint list C2
constInC2 :: (PName, PConstr) -> Constrs -> Constrs -> Maybe Constrs
constInC2 const [] x = Just (x ++ [const])
constInC2 const (c2const:c2tail) x =
                case fst const == fst c2const of
                    True -> case mergeConst (snd const) (snd c2const) of
                                Nothing -> Nothing
                                Just mconst -> Just (x ++ [(fst const,
                                ↪  mconst)] ++ c2tail)
                    False -> constInC2 const c2tail (x ++ [c2const])

-- Compare the 2 Constraints with
mergeConst :: PConstr -> PConstr -> Maybe PConstr
mergeConst (b1,c1v1,c1v2) (b2,c2v1,c2v2)
        | c1v2 <= c2v1 = Nothing
        | c2v2 <= c1v1 = Nothing
        | b1 == True && b2 == True = Just (b1, (largest c1v1 c2v1),
        ↪  (smallest c1v2 c2v2))
```

```haskell
        | b1 == False && b2 == False = Just (b1, (largest c1v1 c2v1),
        ↪  (smallest c1v2 c2v2))
        | b1 == True && b2 == False = Just (b1, (largest c1v1 c2v1),
        ↪  (smallest c1v2 c2v2))
        | b1 == False && b2 == True = Just (b2, (largest c1v1 c2v1),
        ↪  (smallest c1v2 c2v2))
mergeConst _ _ = Nothing

-- Return the smaller of 2 Versions
smallest :: Version -> Version -> Version
smallest v1 v2 =
    case v1 <= v2 of
        True -> v1
        False -> v2

-- Returns the bigger of 2 Versions
largest :: Version -> Version -> Version
largest v1 v2 =
    case v1 >= v2 of
        True -> v1
        False -> v2
```

A.2 Question 2.1: handin/ravnica/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 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().
```

```
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(District, Trigger) ->
58    gen_statem:call(District, {trigger, Trigger}).
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, Data) ->
73    {next_state, active, Data, {reply, From, ok}};
74
75  handle_event({call, From}, {run_action, CRef, Stats}, 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, Stats, 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    {next_state, shutting_down, Data, {next_event, internal, {From,
      ↪  NextPlane}}};
91
92  handle_event({call, From}, {trigger, _Trigger}, Data) ->
93    {keep_state, Data, {reply, From, {error, "Can't set a trigger in this
      ↪  state"}}};
94
95  handle_event({call, From}, {connect, _Action, _To}, Data) ->
96    {keep_state, Data, {reply, From, {error, "Can't connect in this
      ↪  state"}}};
```

```
97
98  % ignore all other unhandled events
99  handle_event(_EventType, _EventContent, Data) ->
100   {keep_state, Data}.
101
102 under_configuration({call, From}, {connect, Action, To}, Data) ->
103   case is_process_alive(To) of
104     true -> case maps:is_key(Action, maps:get(connections, Data)) of
105               false -> Connections = maps:put(Action, To,
                  ↪  maps:get(connections, Data)),
106                 NewData = maps:update(connections, Connections, Data),
107                 {keep_state, NewData, {reply, From, ok}};
108               true -> {keep_state, Data, {reply, From, {error, "Action
                  ↪  already exists"}}}
109             end;
110     false -> {keep_state, Data, {reply, From, {error, "Process not alive
        ↪  anymore"}}}
111   end;
112
113 under_configuration({call, From}, activate, Data) ->
114   {next_state, under_activation, Data, {next_event, internal, From}};
115
116
117 under_configuration({call, From}, {trigger, Trigger}, Data) ->
118   NewData = maps:update(trigger, Trigger, Data),
119   {keep_state, NewData, {reply, From, ok}};
120
121 %% General Event Handling for state under_configuration
122 under_configuration(EventType, EventContent, Data) ->
123   handle_event(EventType, EventContent, Data).
124
125 under_activation(internal, From, Data) ->
126   Result = broadcast_connection(maps:to_list(maps:get(connections, Data)),
      ↪  From, active),
127   case Result of
128     impossible -> {next_state, under_configuration, Data, {reply, From,
        ↪  Result}};
129     active -> {next_state, active, Data, {reply, From, Result}}
130   end;
131
132 under_activation({call, From}, activate, Data) ->
133   {keep_state, Data, {reply, From, under_activation}};
134
135 %% General Event Handling for state under_activation
```

```
136 under_activation(EventType, EventContent, Data) ->
137   handle_event(EventType, EventContent, Data).
138
139 active({call, From}, {enter, {Ref, Stats}}, Data) ->
140   case maps:is_key(Ref, maps:get(creatures, Data)) of
141     true -> {keep_state, Data, {reply, From, {error, "Creture is already
      ↪  in this District"}}};
142     false -> Creatures = maps:get(creatures, Data),
143       case maps:get(trigger, Data) of
144         none -> Creature1 = none, Creatures1 = none;
145         Trigger -> case run_trigger(Trigger, entering, {Ref, Stats},
          ↪  Creatures) of
146                      {error, _} -> Creature1 = none, Creatures1 = none;
147                      {Creature1, Creatures1} -> {Creature1, Creatures1}
148                    end
149       end,
150       case {Creature1, Creatures1} of
151         {none, none} -> NewCreatures = maps:put(Ref, Stats,
          ↪  maps:get(creatures, Data)),
152           NewData = maps:update(creatures, NewCreatures, Data);
153         {{Ref1, Stats1}, NewCreatures1} -> NewCreatures = maps:put(Ref1,
          ↪  Stats1, maps:from_list(NewCreatures1)),
154           NewData = maps:update(creatures, NewCreatures, Data)
155       end,
156       {keep_state, NewData, {reply, From, ok}}
157   end;
158
159 active({call, From}, {take_action, CRef, Action}, Data) ->
160   case maps:is_key(Action, maps:get(connections, Data)) of
161     true ->
162       case maps:is_key(CRef, maps:get(creatures, Data)) of
163         false -> {keep_state, Data, {reply, From, {error, "Creature
          ↪  doesn't exist in this district"}}};
164         true -> case maps:get(trigger, Data) of
165                   none -> Creature1 = none, Creatures1 = none;
166                   Trigger ->
167                     RemoveCreature = maps:remove(CRef, maps:get(creatures,
                      ↪  Data)),
168                     RemovedData = maps:update(creatures, RemoveCreature,
                      ↪  Data),
169                     case run_trigger(Trigger, leaving, {CRef,
                      ↪  maps:get(CRef, maps:get(creatures, Data))},
170                       maps:get(creatures, RemovedData)) of
171                       {error, _} -> Creature1 = none, Creatures1 = none;
```

```
172                         {Creature1, Creatures1} -> {Creature1, Creatures1}
173                       end
174                   end,
175             case {Creature1, Creatures1} of
176               {none, none} -> NewDataCreatures = Data;
177               {{Ref, Stats}, _} -> NewCreatures = maps:put(Ref, Stats,
                  ↪  maps:get(creatures, Data)),
178                 NewDataCreatures = maps:update(creatures, NewCreatures,
                    ↪  Data)
179             end,
180             {NewData, To} = creature_leave(CRef, Action, From,
                ↪  NewDataCreatures),
181             case NewData of
182               error -> {keep_state, Data, {reply, From, {error, To}}};
183               _ -> {keep_state, NewData, {reply, From, {ok, To}}}
184             end
185         end;
186       false -> {keep_state, Data, {reply, From, {error, "Action doesn't
          ↪  exist"}}}
187     end;
188
189 active({call, From}, activate, Data) ->
190   {keep_state, Data, {reply, From, active}};
191
192 %% Handle Calls to active
193 active(EventType, EventContent, Data) ->
194   handle_event(EventType, EventContent, Data).
195
196 shutting_down(internal, {From, NextPlane}, Data) ->
197   Result = broadcast_shutdown(maps:to_list(maps:get(connections, Data)),
      ↪  From, NextPlane),
198   {stop_and_reply, normal, {reply, From, Result}};
199
200 shutting_down({call, From}, activate, Data) ->
201   {keep_state, Data, {reply, From, impossible}};
202
203 shutting_down({call, From}, options, Data) ->
204   {keep_state, Data, {reply, From, none}};
205
206 shutting_down({call, From}, shutdown, Data) ->
207   {keep_state, Data, {reply, From, ok}};
208
209 %% Handle Calls to shutting_down
210 shutting_down(EventType, EventContent, Data) ->
```

```
211   handle_event(EventType, EventContent, Data).
212
213 %% Mandatory callback functions
214 terminate(_Reason, _State, _Data) ->
215   void.
216
217 code_change(_Vsn, State, Data, _Extra) ->
218   {ok, State, Data}.
219
220 % initial State under_configuration
221 init(Desc) ->
222   %% Set the initial state + data
223   State = under_configuration, Data = #{description => Desc, connections
      ↪  => #{}, creatures => #{}, trigger => none},
224   {ok, State, Data}.
225
226 callback_mode() -> state_functions.
227
228 %% Synchronous Call which should wait until each response
229 broadcast_shutdown([], _, _NextPlane) -> ok;
230 broadcast_shutdown([{_Action, To} | Actions], {Pid, Ref}, NextPlane) ->
231   case is_process_alive(To) of
232     true ->
233       case term_to_binary(To) == term_to_binary(Pid) of
234         true -> void;
235         false -> case term_to_binary(To) == term_to_binary(self()) of
236                    true -> void;
237                    false -> gen_statem:call(To, {shutdown, NextPlane})
238                  end
239       end;
240     false -> void
241   end,
242   broadcast_shutdown(Actions, {Pid, Ref}, NextPlane).
243
244 %% Synchronous Call which should wait until each response
245 broadcast_connection([], _, Result) -> Result;
246 broadcast_connection([{_Action, To} | Actions], {Pid, Ref}, _) ->
247   case is_process_alive(To) of
248     false -> Result1 = impossible;
249     true -> Result1 = active,
250       case term_to_binary(To) == term_to_binary(Pid) of
251         false -> case term_to_binary(To) == term_to_binary(self()) of
252                    true -> void;
253                    false -> gen_statem:call(To, activate)
```

```
254                     end;
255           true -> void
256         end
257     end,
258     broadcast_connection(Actions, {Pid, Ref}, Result1).
259
260 creature_leave(CRef, Action, {_Pid, _}, Data) ->
261     To = maps:get(Action, maps:get(connections, Data)),
262     Stats = maps:get(CRef, maps:get(creatures, Data)),
263     case is_process_alive(To) of
264       true -> case term_to_binary(self()) == term_to_binary(To) of
265                 true -> {Data, To};
266                 false -> case gen_statem:call(To, {run_action, CRef, Stats})
                    ↪  of
267                            ok -> NewCreatures = maps:remove(CRef,
                           ↪  maps:get(creatures, Data)),
268                              NewData = maps:update(creatures, NewCreatures,
                              ↪  Data),
269                              {NewData, To};
270                            {error, Reason} -> {error, Reason}
271                          end
272               end;
273       false -> {error, "District is shutdown"}
274     end.
275
276 run_trigger(Trigger, Event, Creature, Creatures) ->
277     Self = self(),
278     spawn(fun() -> Self ! {self(), Trigger(Event, Creature,
      ↪  maps:to_list(Creatures))} end),
279     receive
280       {_Pid, {Creature1, Creatures1}} -> {Creature1, Creatures1}
281     after
282       2000 -> {error, "didnt't run function"}
283     end.
```

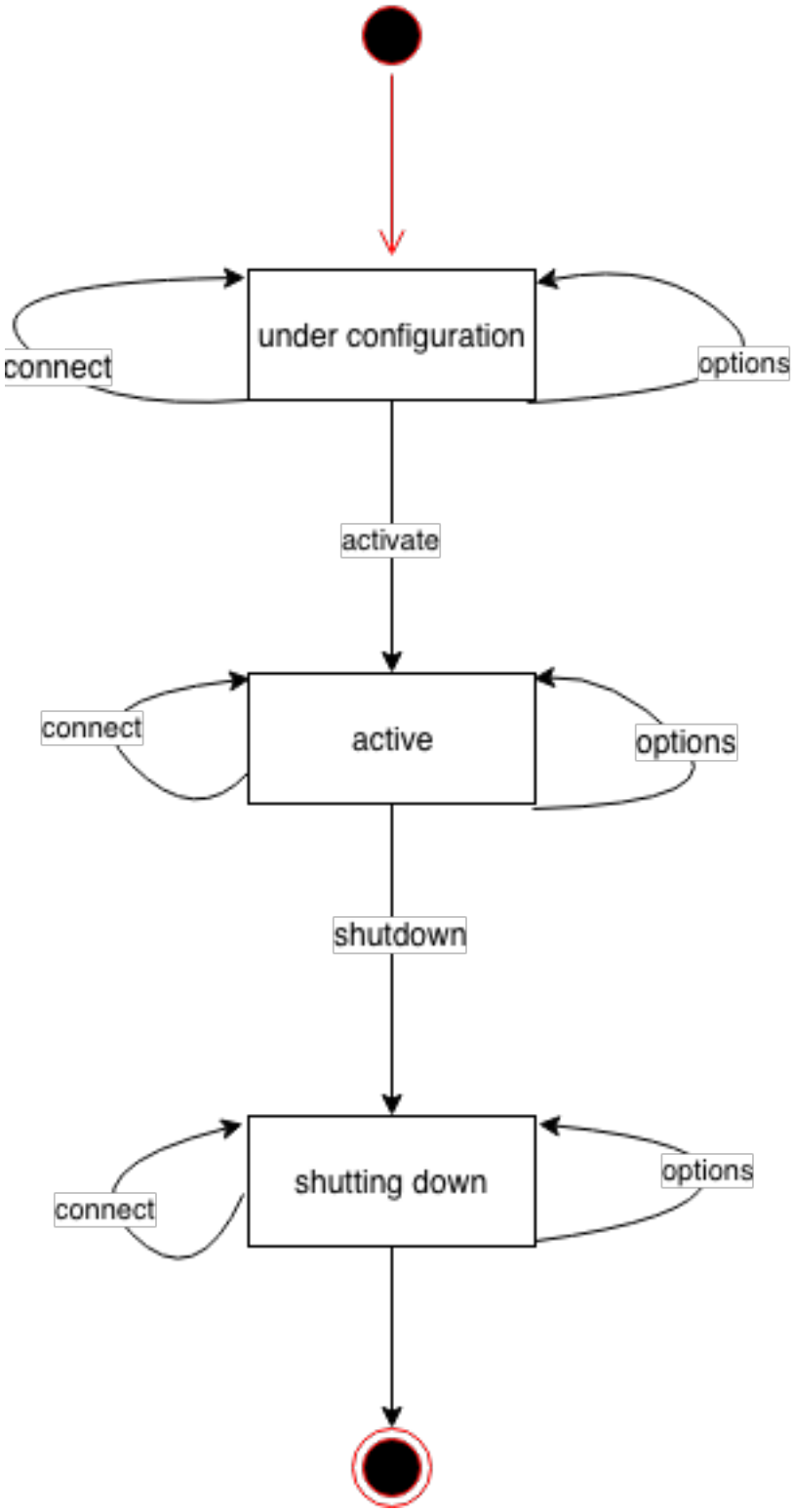

Figure 1: Simple State machine diagramm