Module Plan

1. Scripting.
The *world* contains everything that is needed in the game: *actors*, which are objects or characters that act independently and *areas* which are material and often graphical building blocks of the world.

Areas are in a tree-like hierarchy where areas can have subareas. Actors can have areas associated with them, for example the graphics of an object or character. Areas are also used for collision detection. Actors without associated area are *daemons*.

In the world, there area *actions* that can modify the world. There is a small set of *atomic actions* that are predefined. In scripting the atomic actions are combined to user-defined *composite actions*.

There will be three kinds of scripting in the game:

- World scripting that affects the world directly.
- Area scripting for area triggers that can affect the world directly or change the current area.
- Actor scripting for objects and characters.

World and area scripting depend on the world and the game much. All actions from world scripting should be usable in other kinds of scripting.

In actor scripting, the actors have *tasks* that execute the actions. You can imagine tasks as threads where actions are atomic commands that take no time to execute. One actor has only one big task that determines the behaviour of the actor. Tasks also handle events that are sent to actors. Some tasks need much interaction with other actors. For this purpose there are *interactive tasks*.

There are three different levels in the actor scripting: First level is the normal language level where normal language expressions are evaluated. These expressions cannot modify or access the world or the actor. The second level is the level of actions. Actions can modify and access the world. Third level is the level of tasks. Actually the expressions in this level are just actions that return objects that implement tasks (plans). They are actions because planning tasks happens in the world. Actions describe what happens in the world at a given time and tasks describe what actors do in the world. Tasks can cause actions to happen at different times.

2. Basic tasks.
When scripting, *pass v* is an action that has value *v*.

location obj returns the location of an object.

```
let location obj = pass obj.loc
```

atom action returns a task that executes script *action* when the task is activated.

move_obj obj loc returns a task that moves an object. Note that ^ is the string concatenation operator.

```
let move_obj obj loc = atom (print (obj.name ^ " moved"))
```

The expression *action1* >>= fun *a* → *action2* is an action where first *action1* is executed. *a* is the value of that action. Then action2 is executed. *action1* >> *action2* is the same but the value of *action1* is ignored.

name returns the name of the current actor. *drop obj* makes an actor drop an object and *pickup obj* makes an actor pick up an object. *goto_loc loc* makes an actor move to location *loc*.

```
let drop obj = atom (name >>= fun n → print (n ^ " dropped " ^ obj.name))
let pickup obj = atom (name >>= fun n → print (n ^ " picked " ^ obj.name))
let goto_loc loc = atom (name >>= fun n → print (n ^ " moved "))
```

3. Combined tasks.
The expression *task1* >>> *task2* where *task1* and *task2* return tasks, returns a new task where *task1* is made before *task2*.

install_stick loc makes an actor first move to location *loc*, then put a stick on the ground. *add_tent loc* is the same but a tent is put to the location. Note that *goto_loc loc* >> *atom* (...) won't work because *goto_loc loc* returns a task, and » would just ignore that task.

```
let install_stick loc =
  goto_loc loc >>>
  atom (name >>= fun n → print (n ^ " installed stick"))

let add_tent loc =
  goto_loc loc >>>
  atom (name >>= fun n → print (n ^ " added tent"))
```

task1 >>>= fun *a* → *task2*, where *task1* and *task2* return tasks, makes a task that first executes *task1*. This tasks returns a value *a*. Then the task *task2* is executed. The variable *a* can be used in the second task.

goto_obj' obj makes an actor go to the location of an object when the task is added. *goto_obj obj* makes an actor go to the location of an object when the task is executed.

```
let goto_obj' obj = location obj >>= fun loc → goto_loc loc
let goto_obj obj = atom (location obj) >>>= fun loc → goto_loc loc
```

The game can have several levels of detail for the scripts. *levels ~game* : *task1 ~day* : *task2* returns a task that executes the task *task1* if the game is running on normal level of detail and if the game is running on day-to-day level, another task (*task2*) is executed. Defining levels of detail is helpful if the world is very big. Then some areas can run in a lower level of detail. The problem is that the results of tasks might become different on different detail levels if they are not programmed carefully.

move_object_out area obj moves an object out of an area. If the game is running on normal level then the actor moves to the object, picks it up, goes to the new location and drops the object. If the game runs on day-to-day levels then the object is move automatically.

```
let move_object_out area obj =
  let x, y = corner area and w, h = size area in
```

```
let nloc = random_loc x y w h (Random.float 10.0) in
location obj >>= fun loc →
levels
    ~game : ( goto_obj obj >>> pickup obj >>> goto_loc nloc >>> drop obj )
    ~day : ( move_obj obj nloc )
```

get_objects area returns objects that are currently on an area.

```
let get_objects area = pass [{name ="foo"; loc = (0.0, 0.0)}]
```

do_all tasks returns a task that executes all tasks in the list *tasks*. The tasks might be executed in any order, depending on their priority. There is another version of *do_all* that can share the tasks with helpers or friends to finish the combined task faster.

clear_ground area moves all objects away from an area. It doesn't matter in which order the objects are moved. *List.map* is a normal expression that is here used to generate a task for moving out the objects for each object in a list.

```
let clear_ground area =
  get_objects area >>= fun objs →
  do_all (List.map (fun obj → move_object_out area obj) objs)
```

during x t modifies the task t so that it takes x extra seconds. Atomic tasks take no time. Also combining tasks doesn't make them take any additional time.

make_tent loc returns a task that makes a tent to location. If the game is running on day-to-day level then the tent is added after 5 minutes.

```
let make_tent loc =
  let sq = square_around loc 10.0 in
  clear_ground sq >>>
  levels ~game : (do_all [
                        install_stick (add loc (−5.0, −5.0));
                        install_stick (add loc (−5.0, 5.0));
                        install_stick (add loc (5.0, −5.0));
                        install_stick (add loc (5.0, 5.0)) ] >>>
                  add_tent loc)
         ~day : (during (minutes 5) (add_tent loc))
```

4. Recursive tasks.

An advantage in that the tasks system is embedded in a general purpose language is that features like selection and recursion do not have to be implemented. *looping act* returns a task that executes action *act* forever. Here *atom act >>> looping act* doesn't work because then another *looping act* would always be planned when *looping act* is called. Now *looping act* is planned another time when the previous action is executed.

```
let rec looping act = atom act >>>= fun () → looping act
```

count x returns a task that counts down from x to 0. If running at day-to-day level step 10.

```
let rec count x =
  if x ≤ 0 then atom (print "END") else
```

```
  ( atom (print (string_of_int x)) >>>= fun () →
    levels ~game : (count (x − 1)) ~day : (count (x − 10)) )
```

inf_count x counts from x to ∞.

```
let rec inf_count x =
  atom (print (string_of_int x)) >>>= fun () → inf_count (x + 1)
```

5. Adding tasks.
Make a world and some actors.

```
let world = World.empty
let aku = Actor.make "Aku" ["Hupu";"Lupu";"Tupu"]
let tupu = Actor.make "Tupu" []
let hupu = Actor.make "Hupu" []
let lupu = Actor.make "Lupu" []
```

Add actors to the world.

```
let world = add_actors world [hupu; lupu; tupu; aku]
```

A task to control Aku.

```
let aku_task =
  do_all [ make_tent (10.0, 10.0); count 10; atom (print "test") ]
```

Add some tasks for Aku.

```
let world = run world (modify "Aku" (task aku_task))
```

Result:

```
test
5
4
3
2
1
END
Aku moved
Aku picked foo
Aku moved
Aku dropped foo
Aku moved
Aku installed stick
Hupu moved
Lupu moved
Tupu moved
Aku moved
Tupu installed stick
Lupu installed stick
```

```
Hupu installed stick
Aku added tent
```

6. Priority and delays.
priority action t returns the task *t* modified so that it's priority is calculated by script *action*. *action* shouldn't modify the world. Default priority is 0.0. The priority of *do_all* task is the priority of the most important component task. The priority of »>= task is the priority of the first element.

at time task executes the task *task* when the clock is *time*.

schedule_tuesday schedules some events to happen at tuesday. The function *weekday* needs the current time to determine which tuesday is next.

```
let schedule_tuesday =
  clock >>= fun t →
  do_all [
    at (weekday t Tuesday "8:00") (priority (pass 2.0) exam);
    at (weekday t Tuesday "12:00") (priority (pass 1.0) meeting);
    at (weekday t Tuesday "11:00") eating ]
```

at could be implemented using priorities and conditional tasks, but it is built-in for better scheduling.

```
let wait_time x act =
  priority (clock >>= fun now → if now > x then pass 0.0 else pass (−10.0))
           (cond ~post : (clock >>= fun now → pass (now > x)) wait >>> act)
```

late time can_be_late minp maxp calculates priority for a task that can be late for at most *can_be_late* seconds. If the task is late then its priority raises from *minp* to *maxp* linearly.

```
let late time can_be_late minp maxp =
  clock >>= fun now →
  (* If the time is not yet, don't worry. *)
  if time ≥ now then pass minp
  (* If the time is over, the task has failed so don't care about it. *)
  else if time ≤ now +. can_be_late then pass 0.0 else
  (* Calculate how much is late already. *)
  let late = now −. time in
  pass (minp +. (maxp −. minp) *. time /. can_be_late)
```

7. Failures.
Sometimes a task will fail. For this, exception handling is needed.

```
exception NoSuchObject
```

fail ex makes a task that fails with exception *ex*.

wield obj_type tries to wield an object with kind *obj_type*. If there is no such object, failure is returned.

```
let wield obj_type =
```

```
  atom possessions >>>= fun objs →
  ( try
      let obj = List.find (fun obj → obj#has_kind obj_type) objs in
      atom (carry_state obj Wield)
    with Not_found → fail NoSuchObject )
```

try_task t func tries first the task *t*. If that task fails, then *func* is called with the exception. This function returns a new action which returns a new task that is next tried instead.

ready_battle tries to wield a sword, then an axe and last just some weapon.

```
let ready_battle =
  try_task (wield "sword")
    ~handle : (fun _ → try_task (wield "axe")
        ~handle : (fun _ → try_task (wield "weapon")
            ~handle(fun _ → atom (pass ()))))
```

alternatives lst tries other alternatives if the first alternative fails.

```
let rec alternatives = function
 | [] → atom (pass ()) (* The case that is ran when the list is empty. *)
 (* The case where action is the head of the list and tl is the tail. *)
 | action :: tl → try_task action ~handle : (fun _ → alternatives tl)
```

ready_battle rewritten using *alternatives*.

```
let ready_battle = alternatives [wield "sword"; wield "axe"; wield "weapon"]
```

Atomic tasks can fail using action *throw exn* inside the task. The world is restored to the state it had before the atomic task started. *fail* could be implemented like this:

```
let fail exn = atom (throw exn)
```

8. Post-conditions.

A task might be interrupted or it might be pointless to continue doing a task for some reason. When this happens the actor needs to analyze what to do. For this there are post-conditions. They tell when the task can be stopped safely.

empty returns a task that does nothing and takes no time.

```
let empty = atom (pass ())
```

cond action ~post : cond returns the task returned by *action* modified so that if action *cond* returns true, the *task* ends. *cond* fails with *PostCondition* if the task ends when its postcondition is false.

clear_ground rewritten using *cond*.

```
let rec clear_ground area =
  try_task
    (cond ~post : (get_objects area >>= fun lst → pass (lst = []))
       begin
         get_objects area >>= fun objs →
         do_all (List.map (move_object_out area) objs)
```

```
    end) ~handle : (fun _ → clear_ground area)
```

A recovery strategy *retry ~post action* retries the task until it succeeds.

```
let rec retry ~post act =
  try_task (cond ~post act)
      ~handle : (function PostCondition → retry ~post act
                        | a → fail a)
```

clear_ground rewritten using *retry*.

```
let clear_ground area =
  retry ~post : (get_objects area >>= fun lst → pass (lst = []))
      begin
        get_objects area >>= fun objs →
        do_all (List.map (move_object_out area) objs)
      end
```

wander_around – move around without goals forever.

```
let wander_around =
  looping (move_relative (Random.float 10.0, Random.float 10.0))
```

search x moves around randomly until sees x. *List.mem x objs* tests if x is in the list *objs*.

```
let search x =
  cond ~post : (look >>= fun objs → pass (¬ (List.mem x objs)))
       wander_around
```

9. Events and reactions.
Actors can send and receive events. Received events are handled by the *running* tasks. Task is running if it is the main task, or it is an active part of a running composite task. The parts of *do_all* task are all active. In a >>> b, only a is active.

An event handler receives the event and the continuation of the task that has the handler. It returns the new task that will replace the old one.

beware is an event handler that tries to move away from dangerous phenomena.

```
let beware cont = function
  | See x when dangerous x → do_all [priority 10.0 (move_away x); cont]
  | _ → cont
```

reaction h task returns a task that is same as *task*, but adds new event handler h to the task.

```
let world = run world (modify "Aku" (task (reaction beware aku_task)))
```

Action *send ev obj* sends event *ev* to object *obj*.

The event handler *wet_at_rain* sends the *Rain* event to all clothes of the actor when the actor receives *Rain* event.

```
let wet_at_rain cont = function
  | Rain →
```

```
    possessions >>= fun lst →
    seq (List.map (send Rain) (List.filter worn lst)) >>
    cont
  | _ → cont
```

wait does nothing forever.

```
let wait = looping (pass ())
```

wait_event pred is a task that waits until the actor receives an event *ev* such that *pred ev* is true.

```
let wait_event pred =
  reaction (fun cont ev → if pred ev then empty else cont) wait
```

get_event is a task that returns the next event that the actor receives.

```
let get_event = reaction (fun _ ev → pass ev) wait
```

10. Interactive tasks.
Often the task code contains a lot of exceptions and choices. An *interaction handler* is a sequence of events and choices. *Interaction players* are entities that send messages to each other. The players include two kinds of handlers: skills and resources. Skills can always be used. Resources can only be used once. Interactive tasks integrate this approach to the task system. They wait for event, and then reply with another event. There are different tasks associated to the events. These tasks are ran before the replies.

move a b action makes a handler that anticipates event *a* and replies *b*. *action* is the side effect.

a ||| *b* makes a handler that works like handler *a* or like handler *b*.

a &&& *b* makes a handler that first works like handler *a* and then like handler *b*.

hello_saying is interaction handler for saying hello. If another replies, prints `"Happy"`, otherwise `"Sad"`.

```
let hello_saying =
  ( move Notice (Say "Hello") (atom (print "Hello")) &&&
    ( move (Reply "Hello") End (atom (print "Happy")) |||
      move Silent End (atom (print "Sad")) ) ) |||
  move Notice Silent (atom (print "*silent*")) |||
  move (Say "Hello") (Reply "Hello") (atom (print "Hello")) |||
  move (Say "Hello") Silent (atom (print "*silent*"))
```

Function *make_player s r* returns an interaction player that has *s* as skills and *r* as resources.

make_hello x makes a hello saying handler for talking with *x*.

```
let make_hello x =
  make_player (map_handler (fun a → x, a) (make hello_saying)) (make none)
```

start_game selector start sender ev player makes an interactive task that uses *selector* to select the replies. *player* is the player for the current actor. *ev* is the event that is used to start the game and *sender* is the actor that has sent that event. *start* is a value that is used by other players to select suitable handlers and selectors for the interaction.

seeing_friend is an event handler that makes actor say hello to friends and people that say hello to it.

```
let seeing_friend cont = function
  | See x →
    friend x >>= function
      | true →
        get_id >>= fun name →
        start_game Interaction.random_answer (ChatWith name)
                   x Notice (make_hello x) >>> cont
      | false → cont
  | Hear (x, Say "Hello") →
    get_id >>= fun name →
    start_game Interaction.random_answer (ChatWith name)
               x (Say "Hello") (make_hello x) >>> cont
  | _ → cont
```

listener f waits for new interactions to start. When new interaction is started, the listener uses *f* to determine if it should connect to the interaction or not. *f* will also return the selector and player if it wants to connect to the interaction. If several tasks want to connect to interaction, the task that has best priority is selected.

wait_friend determines if to connect to an interaction. It will only connect to chatting interactions.

seeing_friend is a task that makes an actor to be prepared to say hello to friends.

```
let wait_friend = function
  | ChatWith name →
    pass (Some (make_hello x, Interaction.random_answer))
  | _ → pass None

let seeing_friend = do_all [reaction see_friend wait; listener wait_friend]
```

Index

add_tent, **3**, 3
aku, **5**, 5
aku_task, **5**, 5, 9
alternatives, **7**, 7
beware, **9**, 9
clear_ground, **3**, **8**, 3, 8
count, **4**, 4, 5
drop, **2**, 3
empty, **8**, 5, 9
fail, **7**, 7, 8
get_event, **9**
get_objects, **3**, 3, 8
goto_loc, **2**, 3
goto_obj, **3**, 3
goto_obj', **3**
hello_saying, **10**, 10
hupu, **5**, 5
inf_count, **4**, 4
install_stick, **3**, 3
late, **6**
location, **2**, 3
looping, **4**, 4, 8, 9
lupu, **5**, 5
make_hello, **10**, 10
make_tent, **3**, 5
move_obj, **2**, 3
move_object_out, **3**, 3, 8
NoSuchObject (exn), **7**
pickup, **2**, 3
post (label), **8**, 6, 8
ready_battle, **7**
retry, **8**, 8
schedule_tuesday, **6**
search, **8**
seeing_friend, **10**
tupu, **5**, 5
wait, **9**, 6, 9, 10
wait_event, **9**
wait_friend, **10**, 10
wait_time, **6**
wander_around, **8**, 8
wet_at_rain, **9**
wield, **7**, 7
world, **5**, **9**, 5, 9