ATL Template

```
module Mancala
( Player(P1,P2)
, Status(Turn,Finished)
, Spaces
, State(State, status, spaces, score1, score2), state_init
, Action(Action)
, update
) where

import Debug
```

1 Status

```
data Player = P1 | P2
    deriving (Show)

get_other_player :: Player -> Player
get_other_player player = case player of
    P1 -> P2
    P2 -> P1

data Status
    = Turn Player
    | Finished

instance Show Status where
    show (Turn player) = (show player) ++ "'s turn"
    show Finished = "game finished"
```

2 Spaces

```
type Spaces = [(Int, Int)] -- (index, pieces)

(+%) :: Int -> Int -> Int
x +% y = (x + y) `mod` 12

indexed :: [a] -> [(Int, a)]
indexed ls = let
    helper []     _ = []
    helper (x:xs) i = (i,x) : helper xs (i+1)
    in helper ls 0


spaces_init = indexed $ take 12 $ repeat 4

-- spaces_init = indexed $
--     [ 4, 4, 4, 4, 4, 4
```

```
    -- , 4, 4, 4, 4, 4, 4 ]

spaces_empty = indexed $ take 12 $ repeat 0
```

3 State

```
data State = State
    { status :: Status
    , spaces :: Spaces
    , score1 :: Int
    , score2 :: Int }

instance Show State where
    show state =
        let format s = case length s of { 1 -> s ++ " "; 2 -> s; _ -> error s }
            s i = format $ show $ get_space i state
        in foldl (++) ""
            [ "\n"
            , show $ status state
            , "\n\n"
            , "score2: " ++ (show $ score2 state) ++ "\n"
            , "\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 9,  " | ", s 8, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 10, " | ", s 7, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 11, " | ", s 6, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 0,  " | ", s 5, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 1,  " | ", s 4, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "    | ", s 2,  " | ", s 3, " |\n"
            , "    +----" ++   "+"  ++ "----+\n"
            , "\n"
            , "score1: " ++ (show $ score1 state) ++ "\n"
            , "\n" ]

state_init = State
    (Turn P1)
    spaces_init
    (0)
    (0)

--
-- setters
--

set_status :: Status -> State -> State
set_status status (State _ ss s1 s2) = State status ss s1 s2
```

```
set_spaces :: Spaces -> State -> State
set_spaces spaces (State st _ s1 s2) = State st spaces s1 s2

set_score1 :: Int -> State -> State
set_score1 score1 (State st ss _ s2) = State st ss score1 s2

set_score2 :: Int -> State -> State
set_score2 score2 (State st ss s1 _) = State st ss s1 score2

--
-- useful functions
--

add_score :: Player -> Int -> State -> State
add_score player x state = case player of
    P1 -> set_score1 (x + score1 state) state
    P2 -> set_score2 (x + score2 state) state

get_space :: Int -> State -> Int
get_space index state = snd $ (spaces state) !! index

set_space :: Int -> Int -> State -> State
set_space index x_new state = let
    spaces_new = map
        (\(i, x) -> if i == index then (i, x_new) else (i, x))
        (spaces state)
    in set_spaces spaces_new state

add_space :: Int -> Int -> State -> State
add_space index x_new state =
    set_space index (x_new + get_space index state) state
```

4 Action

```
data Action = Action Int

get_active_index :: Player -> Action -> Int
get_active_index player (Action index) =
    index + case player of
        P1 -> 0
        P2 -> 6
```

5 Update

```
update :: Action -> State -> IO State
update action state_orig = let
    -- ‘active‘ player and ‘other‘ player
    (active, other) = case status state_orig of
        Turn P1 -> (P1, P2)
        Turn P2 -> (P2, P1)
```

```
-- (player 1) distribute selected pieces to the appropriate places
update_action :: State -> IO State
update_action state = let
    action_index = get_active_index active action
    action_pieces = get_space action_index state
    emptied_action_index = set_space action_index 0 state
    -- distribute action pieces
    helper :: Int -> Int -> State -> IO State
    helper index pieces state = let
        target = get_space (index +% 1) state
        after = get_space (index +% 2) state
        in case (active, index, pieces, target) of
            -- P1 drop last piece in score1
            (P1, 2, 1, _) -> do
                debug "P1 drop last piece in score1"
                return
                  $ add_score P1 1              -- P1 scores 1
                  $ set_status (Turn P1)        -- P1 takes extra turn
                        state
            -- P2 drop last piece in score2
            (P2, 8, 1, _) -> do
                debug "P2 drop last piece in score2"
                return
                  $ add_score P2 1              -- P2 scores 1
                  $ set_status (Turn P2)        -- P2 takes extra turn
                        state
            -- P1 drop last piece immediately after score1
            (P1, 2, 2, _) -> do
                debug "drop last piece immediately after score1"
                return
                    $ add_score P1 1            -- P1 scores 1
                    $ add_space (index +% 1) 1  -- drop 1 in `target`
                    $ set_status (Turn P2)      -- alternate turn to P2
                        state
            -- P2 drop last piece immediately after score1
            (P2, 8, 2, _) -> do
                debug "P2 drop last piece immediately after score1"
                return
                    $ add_score P2 1            -- P2 scores 1
                    $ add_space (index +% 1) 1  -- drop 1 in `target`
                    $ set_status (Turn P1)      -- alternate turn to P1
                        state
            -- P1 `target` is empty
            (_, _, 1, 0) -> do
                debug "target is empty"
                return
                  $ add_space (index +% 1) 1    -- drop 1 in `target`
                  $ add_score active after      -- active scores `after`
                  $ set_space (index +% 2) 0    -- empty `after`
                  $ set_status (Turn other)     -- alternate turn to other
                        state
            -- target is non-empty
            (_, _, 1, _) -> do
                debug "target is non-empty"
```

```
                    return
                      $ add_space (index +% 1) 1    -- drop 1 in `target`
                      $ set_status (Turn other)     -- alternate turn to P2
                            state
                -- P1 pass score1
                (P1, 2, _, _) -> do
                    debug "P1 pass score1"
                    helper (index +% 1) (pieces - 2)
                      $ add_space (index +% 1) 1    -- drop 1 in `target`
                      $ add_score P1 1              -- P1 scores 1
                            state
                -- P2 pass score2
                (P2, 8, _, _) -> do
                    debug "P2 pass score2"
                    helper (index +% 1) (pieces - 2)
                      $ add_space (index +% 1) 1    -- drop 1 in `target`
                      $ add_score P2 1              -- P2 scores 1
                            state
                -- normal
                (_, _, _, _) -> do
                    debug "normal"
                    helper (index +% 1) (pieces - 1)
                      $ add_space (index +% 1) 1    -- drop 1 in `target`
                            state
        in helper action_index action_pieces
            $ set_space action_index 0 state

    -- apply post-action rules
    update_cleanup :: State -> IO State
    update_cleanup state = let
        sum1 = sum $ take 6 $ map snd $ spaces state
        sum2 = sum $ drop 6 $ map snd $ spaces state
        in case (sum1, sum2) of
            -- P1's side is empty
            (0,_) -> return
                $ add_score P1 sum2                 -- P1 scores P2's side
                $ set_status Finished               -- game is finished
                    state
            -- P2's side is empty
            (_,0) -> return
                $ add_score P2 sum1                 -- P2 scores P1's side
                $ set_status Finished               -- game is finished
                    state
            -- neither side is empty
            (_,_) -> return state

    in case status state_orig of
        Finished -> return state_orig
        Turn _  -> foldl (>>=) (return state_orig)
                    [ update_action
                    , update_cleanup ]


p1 x = update (Action x) (set_status (Turn P1) state_init)
```

```
p2 x = update (Action x) (set_status (Turn P2) state_init)
```