```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

(* return: value, globals, entities, cards *)
exception ReturnException of Ast.expr * Ast.expr NameMap.t * Ast.expr NameMap.t * Ast.
expr NameMap.t
(* return: string or string list of winners, or [] for no winners *)
exception GameOverException of Ast.expr

(* seed random number generator with current time *)
let _ = Random.init (truncate (Unix.time()))
let entityData = []

(* Main entry point: run a program *)

let run (program) =
    let spec = fst(program)
    in
    let funcs = snd(program)
    in
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals entities cards =

  (* Evaluate an expression and return (value, updated environment) *)
  let rec eval env = function
      Null -> Null, env
    | Noexpr -> Noexpr, env

    | IntLiteral(i) -> IntLiteral(i), env
    | StringLiteral(i) -> StringLiteral(i), env
    | BoolLiteral(i) -> BoolLiteral(i), env
    | CardLiteral(i) -> CardLiteral(i), env

    (* Return (list of evaluated expressions), env *)
    (* Applicative order: evaluate each argument, updating env each time *)
    | ListLiteral(ls) ->
        (match ls with
          []       -> ListLiteral([]), env
        | hd :: tl ->
          let evalhd, env = eval env hd in
          let evaltl, env = eval env (ListLiteral(tl)) in
          (match evaltl with
            ListLiteral(lstl) -> ListLiteral(evalhd :: lstl), env
          | _ -> raise (Failure ("invalid ListLiteral construction"))))

    | Binop(e1, op, e2) ->
        let v1, env = eval env e1 in
        let v2, env = eval env e2 in
        let boolean i = if i then BoolLiteral(true) else BoolLiteral(false) in
        (match v1, op, v2 with
          IntLiteral(i1), Add, IntLiteral(i2)  -> IntLiteral(i1 + i2)
        | IntLiteral(i1), Sub, IntLiteral(i2)  -> IntLiteral(i1 - i2)
        | IntLiteral(i1), Mult, IntLiteral(i2) -> IntLiteral(i1 * i2)
        | IntLiteral(i1), Div, IntLiteral(i2)  -> IntLiteral(i1 / i2)
        | IntLiteral(i1),    Equal, IntLiteral(i2)    -> boolean (i1 = i2)
        | StringLiteral(i1), Equal, StringLiteral(i2) -> boolean (i1 = i2)
        | CardLiteral(i1),   Equal, CardLiteral(i2)   -> boolean (i1 = i2)
        | BoolLiteral(i1),   Equal, BoolLiteral(i2)   -> boolean (string_of_bool i1 =
string_of_bool i2)
```

```
        | IntLiteral(i1),    Neq, IntLiteral(i2)    -> boolean (i1 <> i2)
        | StringLiteral(i1), Neq, StringLiteral(i2) -> boolean (i1 <> i2)
        | CardLiteral(i1),   Neq, CardLiteral(i2)   -> boolean (i1 <> i2)
        | BoolLiteral(i1),   Neq, BoolLiteral(i2)   -> boolean (string_of_bool i1 <> s
tring_of_bool i2)
        | Null,    Equal, Null    -> boolean(true)
        | Null,    Neq, Null    -> boolean(false)
        | IntLiteral(i1),    Equal, Null    -> boolean(false)
        | StringLiteral(i1), Equal, Null -> boolean(false)
        | CardLiteral(i1),   Equal, Null   -> boolean(false)
        | Variable(VarExp(id, Entity)),   Equal, Null   -> boolean(false)
        | BoolLiteral(i1),   Equal, Null   -> boolean(false)
        | IntLiteral(i1),    Neq, Null    -> boolean(true)
        | StringLiteral(i1), Neq, Null -> boolean(true)
        | CardLiteral(i1),   Neq, Null   -> boolean(true)
        | Variable(VarExp(id, Entity)),   Neq, Null   -> boolean(true)
        | BoolLiteral(i1),   Neq, Null   -> boolean(true)
        | Null,    Equal, IntLiteral(i2)    -> boolean(false)
        | Null, Equal, StringLiteral(i2) -> boolean(false)
        | Null,   Equal, CardLiteral(i2)   -> boolean(false)
        | Null, Equal, Variable(VarExp(id, Entity)) -> boolean(false)
        | Null,   Equal, BoolLiteral(i2)   -> boolean(false)
        | Null,    Neq, IntLiteral(i2)    -> boolean(true)
        | Null, Neq, StringLiteral(i2) -> boolean(true)
        | Null,   Neq, CardLiteral(i2)   -> boolean(true)
        | Null, Neq, Variable(VarExp(id, Entity)) -> boolean(true)
        | Null,   Neq, BoolLiteral(i2)   -> boolean(true)
        | IntLiteral(i1),    Less, IntLiteral(i2)    -> boolean (i1 < i2)
        | StringLiteral(i1), Less, StringLiteral(i2) -> boolean (i1 < i2)
        | CardLiteral(i1),   Less, CardLiteral(i2)   -> boolean (i1 < i2) (* cmp cards
 as string? *)
        | IntLiteral(i1),    Leq, IntLiteral(i2)    -> boolean (i1 <= i2)
        | StringLiteral(i1), Leq, StringLiteral(i2) -> boolean (i1 <= i2)
        | CardLiteral(i1),   Leq, CardLiteral(i2)   -> boolean (i1 <= i2) (* cmp cards
 as string? *)
        | IntLiteral(i1),    Greater, IntLiteral(i2)    -> boolean (i1 > i2)
        | StringLiteral(i1), Greater, StringLiteral(i2) -> boolean (i1 > i2)
        | CardLiteral(i1),   Greater, CardLiteral(i2)   -> boolean (i1 > i2) (* cmp ca
rds as string? *)
        | IntLiteral(i1),    Geq, IntLiteral(i2)    -> boolean (i1 >= i2)
        | StringLiteral(i1), Geq, StringLiteral(i2) -> boolean (i1 >= i2)
        | CardLiteral(i1),   Geq, CardLiteral(i2)   -> boolean (i1 >= i2) (* cmp cards
 as string? *)
        | BoolLiteral(i1), And, BoolLiteral(i2) -> boolean (i1 && i2)
        | BoolLiteral(i1), Or, BoolLiteral(i2)  -> boolean (i1 || i2)
        | StringLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(i1 ^ i2)  (* w
e want String concat, right? *)
        | StringLiteral(i1), Concat, CardLiteral(i2) -> StringLiteral(i1 ^ i2)  (* we
want String concat, right? *)
        | StringLiteral(i1), Concat, Variable(VarExp(id, Entity)) -> StringLiteral(i1
^ id)  (* we want String concat, right? *)
        | StringLiteral(i1), Concat, IntLiteral(i2) -> StringLiteral(i1 ^ string_of_in
t i2)  (* we want String concat, right? *)
        | StringLiteral(i1), Concat, BoolLiteral(i2) -> StringLiteral(i1 ^ string_of_b
ool i2)  (* we want String concat, right? *)
        | CardLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(i1 ^ i2)  (* we
want String concat, right? *)
        | Variable(VarExp(id, Entity)), Concat, StringLiteral(i2) -> StringLiteral(id
^ i2)  (* we want String concat, right? *)
        | IntLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(string_of_int i1
^  i2)  (* we want String concat, right? *)
        | BoolLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(string_of_bool i
1 ^  i2)  (* we want String concat, right? *)

        | _, _, _ ->
                    raise (Failure ("invalid binary operation - likely comparing two i
ncompatible types"))
        ), env
```

```
    | Rand(e) ->
        let v, env = eval env e in
        (match v with
          IntLiteral(i) -> IntLiteral(Random.int i), env
        | _ -> raise (Failure ("invalid argument for random operator ~. Must supply an
 int."))
        )

    | GetType(e) ->
        let v, env = eval env e in
        (match v with
          Null -> StringLiteral("null")
        | IntLiteral(_) -> StringLiteral("int")
        | StringLiteral(_) -> StringLiteral("string")
        | BoolLiteral(_) -> StringLiteral("bool")
        | CardLiteral(_) -> StringLiteral("Card")
        | ListLiteral(_) -> StringLiteral("list")
        | Variable(VarExp(_, Entity)) -> StringLiteral("CardEntity")
        | _ -> raise (Failure ("internal error: unrecognized type in GetType"))
        ), env

    | Variable(var) ->
        let locals, globals, entities, cards = env in
        (match var with
          VarExp(id, scope) ->
            (match scope with
              Local ->
                (* NameMap maps var name to (literalvalue) *)
                if NameMap.mem id locals then
                  NameMap.find id locals, env
                else raise (Failure ("undeclared local variable " ^ id))
            | Global ->
                if NameMap.mem id globals then
                  NameMap.find id globals, env
                else raise (Failure ("undeclared global variable " ^ id))
            | Entity ->
                if NameMap.mem id entities then
                  (* return the entity variable *)
                  Variable(var), env
                else raise (Failure ("undeclared CardEntity " ^ id))
            )
        | GetIndex(id, scope, index) ->
            let evalidx, env = eval env index in
            (match scope, evalidx with
              Local, IntLiteral(i) ->
                if NameMap.mem id locals then
                  (match NameMap.find id locals with
                    ListLiteral(ls) -> List.nth ls i
                  | Variable(VarExp(origid, Entity)) ->
                      if NameMap.mem origid entities then
                        (match NameMap.find origid entities with
                          ListLiteral(ls) -> List.nth ls i
                        | _ -> raise (Failure ("internal error: CardEntity "^origid^"
not storing ListLiteral")))
                      else raise (Failure ("internal error: "^id^" holding invalid ref
erence to CardEntity "^origid))
                  | _ -> raise (Failure ("You can only dereference a list or CardEntit
y"))
                  ), env
                else raise (Failure ("undeclared local variable " ^ id))
            | Global, IntLiteral(i) ->
                if NameMap.mem id globals then
                  (match NameMap.find id globals with
                    ListLiteral(ls) -> List.nth ls i
                  | Variable(VarExp(origid, Entity)) ->
                      if NameMap.mem origid entities then
                        (match NameMap.find origid entities with
                          ListLiteral(ls) -> List.nth ls i
                        | _ -> raise (Failure ("internal error: CardEntity "^origid^"
```

```
not storing ListLiteral")))
                    else raise (Failure ("internal error: "^id^" holding invalid ref
erence to CardEntity "^origid))
                | _ -> raise (Failure ("You can only dereference a list or CardEntit
y"))
                ), env
              else raise (Failure ("undeclared global variable " ^ id))
          | Entity, IntLiteral(i) ->
              if NameMap.mem id entities then
                (match NameMap.find id entities with
                  ListLiteral(ls) -> List.nth ls i
                | _ -> raise (Failure ("internal error: CardEntity "^id^" not storin
g ListLiteral"))
                ), env
              else raise (Failure ("undeclared CardEntity " ^ id))
          | _, _ ->
              raise (Failure ("invalid list dereference, probably using non-integer
index"))
          ))

    | Assign(var, e) ->
        let v, (locals, globals, entities, cards) = eval env e in
        (match var with
          VarExp(id, scope) ->
            (match scope with
              Local ->
                if NameMap.mem id locals then
                  v, (NameMap.add id v locals, globals, entities, cards)
                else raise (Failure ("undeclared local variable " ^ id))
            | Global ->
                if NameMap.mem id globals then
                  v, (locals, NameMap.add id v globals, entities, cards)
                else raise (Failure ("undeclared global variable " ^ id))
            | Entity ->
                raise (Failure ("You cannot assign to a cardentity"))
            )
        | GetIndex(id, scope, index) ->
            let evalidx, env = eval env index in
            (match scope, evalidx with
              Local, IntLiteral(i) ->
                if NameMap.mem id locals then
                  let rec inserthelper ls targetindex value curr =
                    if curr = targetindex then
                      (match ls with
                        []      -> [value]
                      | _ :: tl -> value :: tl)
                    else
                      (match ls with
                        []       -> raise (Failure ("index out of bounds"))
                      | hd :: tl -> hd :: (inserthelper tl targetindex value (curr+1))
)
                  in
                  (match NameMap.find id locals with
                    ListLiteral(ls) ->
                      v, (NameMap.add id (ListLiteral(inserthelper ls i v 0)) locals,
globals, entities, cards)
                  | Variable(vexp) ->
                      let ret, env = eval env (Assign(vexp, v)) in ret, env
                  | _ -> raise (Failure ("You can only dereference a list or CardEntit
y")))
                else raise (Failure ("undeclared local variable " ^ id))
            | Global, IntLiteral(i) ->
                if NameMap.mem id globals then
                  let rec inserthelper ls targetindex value curr =
                    if curr = targetindex then
                      (match ls with
                        []      -> [value]
                      | _ :: tl -> value :: tl)
                    else
```

```
                    (match ls with
                    | []       -> raise (Failure ("index out of bounds"))
                    | hd :: tl -> hd :: (inserthelper tl targetindex value (curr+1))
)
                in
                (match NameMap.find id globals with
                  ListLiteral(ls) ->
                    v, (locals, NameMap.add id (ListLiteral(inserthelper ls i v 0))
globals, entities, cards)
                | Variable(vexp) ->
                    let ret, env = eval env (Assign(vexp, v)) in ret, env
                | _ -> raise (Failure ("You can only dereference a list or CardEntit
y")))
              else raise (Failure ("undeclared global variable " ^ id))
            | Entity, IntLiteral(i) ->
                raise (Failure ("You must use the transfer operator (<-) to modify Car
dEntity"))
            | _, _ ->
                raise (Failure ("invalid list dereference, probably using non-integer
index"))
            ))

    | ListLength(vlist) ->
        let evlist, (locals, globals, entities, cards) = eval env vlist in
        (match evlist with
          ListLiteral(ls) -> IntLiteral(List.length ls), env
        | Variable(VarExp(id, Entity)) ->
            if NameMap.mem id entities then
              (match NameMap.find id entities with
                ListLiteral(ls) -> IntLiteral(List.length ls)
              | _ -> raise (Failure ("internal error: CardEntity "^id^" not storing Li
stLiteral"))), env
            else raise (Failure ("undeclared CardEntity " ^ id))
        | _ -> raise (Failure ("argument to list length operator must be a list or Car
d Entity")))

    | Append(vlist, e) ->
        let v, env = eval env e in
        let evlist, env = eval env vlist in
        (match evlist with
          ListLiteral(ls) -> ListLiteral(ls @ [v]), env
        | _ -> raise (Failure ("trying to append an element to a non-list")))

    | Transfer(cevar, card) ->
        let evalc, env = eval env card in
        (match cevar, evalc with
          VarExp(id, Entity), CardLiteral(c) ->
            if NameMap.mem c cards then
              let locals, globals, entities, cards = env in
              (* delete Card from original CardEntity's list *)
              let rec deletehelper ls value =
                (match ls with
                  []       -> []
                | hd :: tl -> if hd = value then tl else hd :: (deletehelper tl value)
)
              in
              let oldownerlit = NameMap.find c cards in
              (match oldownerlit with
                StringLiteral(oldowner) ->
                  let entities =
                  (if NameMap.mem oldowner entities then
                    let oldownercards = NameMap.find oldowner entities in
                      (match oldownercards with
                        ListLiteral(c1) -> NameMap.add oldowner (ListLiteral(deletehel
per c1 evalc)) entities
                      | _ -> raise (Failure ("internal error: CardEntity "^id^" not st
oring ListLiteral")))
                  else raise (Failure ("internal error: Card "^c^" invalid owner "^old
owner))
```

```
                  ) in
                  (* add mapping from Card name to StringLiteral containing CardEntity
's name *)
                  let cards = NameMap.add c (StringLiteral(id)) cards in
                  let rec insertunique ls value =
                    (match ls with
                      []           -> [value]
                    | hd :: tl     -> if hd = value then ls else hd :: (insertunique t
l value))
                  in
                  (* add updated ListLiteral to new entity's list *)
                  if NameMap.mem id entities then
                    let entitycards = NameMap.find id entities in
                    (match entitycards with
                      ListLiteral(c2) ->
                        StringLiteral(id), (locals, globals, NameMap.add id (ListLiter
al(insertunique c2 evalc)) entities, cards)
                    | _ -> raise (Failure ("internal error: CardEntity "^id^" not stor
ing ListLiteral")))
                  else raise (Failure ("Invalid CardEntity: " ^ id))
                | _ -> raise (Failure ("internal error: Card "^id^" not mapped to a Stri
ngLiteral")))

              else raise (Failure ("Invalid card name: " ^ c))
          | VarExp(id, _), CardLiteral(c) ->
              let ceref, env = eval env (Variable(cevar)) in
              (match ceref with
                Variable(VarExp(id2, Entity)) -> eval env (Transfer(VarExp(id2, Entity),
 evalc))
              | _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))
          | GetIndex(id, _, _), CardLiteral(c) ->
              let ceref, env = eval env (Variable(cevar)) in
              (match ceref with
                Variable(VarExp(id2, Entity)) -> eval env (Transfer(VarExp(id2, Entity),
 evalc))
              | _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))
          | _, _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))

      | Call(f, actuals) ->
          let fdecl =
            try NameMap.find f func_decls
            with Not_found ->
            raise (Failure ("undefined function " ^ f))
          in
          let actuals, env = List.fold_left
              (fun (actuals, values) actual ->
                let v, env = eval env actual in
                List.append actuals [v], values) ([], env) actuals
          in
          let (locals, globals, entities, cards) = env in
          try
            let globals, entities, cards = call fdecl actuals globals entities cards
            in  BoolLiteral(false), (locals, globals, entities, cards)
          with ReturnException(v, globals, entities, cards) -> v, (locals, globals, enti
ties, cards)
    in
    (* Execute a statement and return an updated environment *)
    let rec exec env = function
        Nostmt -> env
      | Expr(e) -> let _, env = eval env e in env
      | If(e, s1, s2) ->
          let v, env = eval env e in
          let b = (match v with
            BoolLiteral(b) -> b
            | _ -> raise (Failure ("Invalid conditional expression.")))
          in
          if b then
            List.fold_left exec env (List.rev s1)
          else
```

```
          List.fold_left exec env (List.rev s2)
    | While (e, s) ->
        let rec loop env =
          let v, env = eval env e in
          let b = (match v with
          BoolLiteral(b) -> b
          | _ -> raise (Failure ("Invalid conditional expression.")))
          in
          if b then
            loop (List.fold_left exec env (List.rev s))
          else env
        in loop env
    | Break ->
        env
    | Read(var) ->
        let input = read_line() in
        let v = (match input with
            a -> StringLiteral(a)
            |_ -> raise(Failure("Invalid input")))
        in
        let ret, env = eval env (Assign(var, v)) in env
    | Print(e) ->
        let v, env = eval env e in
        begin
            let str = (match v with
          BoolLiteral(b) -> string_of_bool b
          |IntLiteral(i) -> string_of_int i
          |CardLiteral(c) -> "[Card: " ^ c ^ "]"
          |StringLiteral(s) -> s
          | Variable(VarExp(id, Entity)) -> "[Card Entity: " ^ id ^ "]"
          | _ -> raise (Failure ("Invalid print expression.")))
          in
            print_endline str;
            env
        end
    | Return(e) ->
        let v, (locals, globals, entities, cards) = eval env e in
        raise (ReturnException(v, globals, entities, cards))
  in
  (* end of statement execution *)

  (* call: enter the function: bind actual values to formal args *)
  let locals =
    try List.fold_left2
      (fun locals formal actual -> NameMap.add formal actual locals)
      NameMap.empty fdecl.formals actuals
    with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments to " ^ fdecl.fname))
  in
  let locals = List.fold_left   (* Set local variables to Null (undefined) *)
    (fun locals local -> NameMap.add local Null locals)
    locals fdecl.locals
  in   (* Execute each statement; return updated global symbol table *)
  (match (List.fold_left exec (locals, globals, entities, cards) fdecl.body) with
    _, globals, entities, cards -> globals, entities, cards)

(* run: set global variables to Null; find and run "start" *)
in
(* initialize globals by reading from the globals block *)
let globals = List.fold_left
  (fun globals vdecl -> NameMap.add vdecl Null globals)
  NameMap.empty spec.glob.globals
in
(* initialize entities by reading from CardEntities block *)
let entities = List.fold_left
  (fun entities vdecl -> NameMap.add vdecl (ListLiteral([])) entities)
  NameMap.empty spec.cent.entities
in
(* initialize the cards symbol table to point to the first CardEntity *)
```

```
let firstentity =
  (match spec.cent.entities with
    hd :: _ -> hd
  | []      -> raise (Failure ("You must declare at least one CardEntity.")))
in
let deckstrings = ["C2";"C3";"C4";"C5";"C6";"C7";"C8";"C9";"C10";"CJ";"CQ";"CK";"CA";
                   "D2";"D3";"D4";"D5";"D6";"D7";"D8";"D9";"D10";"DJ";"DQ";"DK";"DA";
                   "H2";"H3";"H4";"H5";"H6";"H7";"H8";"H9";"H10";"HJ";"HQ";"HK";"HA";
                   "S2";"S3";"S4";"S5";"S6";"S7";"S8";"S9";"S10";"SJ";"SQ";"SK";"SA"]
in
let cards = List.fold_left
  (fun cards vdecl -> NameMap.add vdecl (StringLiteral(firstentity)) cards)
  NameMap.empty deckstrings
in
(* Add the cards to the first CardEntity too. they map to each other. *)
let deckcards =
  ListLiteral(List.fold_left
    (fun acc cardstring -> CardLiteral(cardstring) :: acc) [] (List.rev deckstrings))
in
let entities = NameMap.add firstentity deckcards entities in
try
  let startDecl = { fname = "Start";
                    formals = [];
                    locals = spec.strt.slocals;
                    body=spec.strt.sbody }
  in
  let func_decls = NameMap.add "Start" startDecl func_decls
  in
  let startDecl = { fname = "Play";
                    formals = [];
                    locals = spec.play.plocals;
                    body=spec.play.pbody }
  in
  let func_decls = NameMap.add "Play" startDecl func_decls
  in
  let startDecl = { fname = "WinningCondition";
                    formals = [];
                    locals = spec.wcon.wlocals;
                    body=spec.wcon.wbody }
  in
  let func_decls = NameMap.add "WinningCondition" startDecl func_decls
  in
  let (globals, entities, cards) =
    call (NameMap.find "Start" func_decls) [] globals entities cards
    in
      let rec loop a (globals, entities, cards) =
        let (globals, entities, cards) =
            call (NameMap.find "Play" func_decls) [] globals entities cards
        in
        try
            let (globals, entities, cards) =
                call (NameMap.find "WinningCondition" func_decls) [] globals entities
cards
            in (globals, entities, cards)
        with ReturnException(v, globals, entities, cards) ->
          (match v with
            Null -> loop a (globals, entities, cards)
          | _ -> raise (GameOverException (v)))

      in loop "blah" (globals, entities, cards)

  with
    Not_found -> raise (Failure ("did not find the start() function"))
  | GameOverException(winners) ->
      print_endline "Game over!"; exit 0
```