Appendix

Translator Code Listing

```
(*action.ml by Ernesto *)
open Ast
open Checktype
open Check



module type ACTION = sig

   val kill_to_java : string -> string list
   val grab_to_java : string -> string -> string list
   val drop_to_java : string -> string -> string list
   val show_to_java : string -> string -> ('a VarMap.t VarMap.t) -> string list
   val hide_to_java : string -> string -> ('a VarMap.t VarMap.t) -> string list

end

module Action : ACTION = struct

let kill_to_java str =
  ["killFunction(\"" ^ str ^ "\");"]
let grab_to_java str1 str2 =
  [str1 ^ ".addItem (\"" ^ str2 ^"\", currentLocation, locations);"]
let drop_to_java str1 str2 =
  [str1 ^ ".removeItem (\"" ^ str2 ^"\", currentLocation, locations);"]

let show_to_java str1 str2 mapt =
  if (VarMap.mem (str2,Item) mapt) then
      [str1 ^ ".addItem (\"" ^ str2^ "\",items);"]
  else [str1 ^ ".showCharacter (\"" ^ str2^ "\", characters);"]

let hide_to_java str1 str2 mapt =
  if (VarMap.mem (str2,Item) mapt) then
      [str1 ^ ".removeItem (\"" ^ str2 ^"\");"]
  else [str1 ^ ".hideCharacter (\"" ^ str2 ^"\");"]

end
```

```
(* ast.mli by Xiao *)
type operator =
      Add
  | Sub
  | Mul
  | Div
  | Or
  | And
  | Eq
  | Lt
  | Gt
  | Neq
  | Leq
  | Geq



type pridec =
      Strdec of string
  | Intdec of string
  | Strdecinit of string * expr
  | Intdecinit of string * expr

and membervarlist = membervar list

and membervar =
      Primember of pridec
  | Varref of string

and expr =
    Binop of expr * operator * expr
  | Asn of id * expr
  | Lit of int
  | LitS of string
  | Exists of string * string
  | Neg of expr
  | Not of expr
  | Ident of id

and id =
        Var of string
      | Has of string * string

and probexpr =
      Unitprob of int * stmt

and probexprlist = probexpr list

and actiondec =
      Unitaction of string * string * string

and whenexpr =
      Unitwhen of string * stmt * string

and stmt =
      Ifelse of expr * stmt * stmt
  | Chwhen of actiondeclist * whenexprlist
  | Prob of probexprlist
  | Kill of string
  | Grab of string * string
  | Drop of string * string
  | Show of string * string
  | Hide of string * string
  | Atomstmt of expr
  | Cmpdstmt of block
  | Nostmt of int
  | Print of expr
and block = stmt list
and actiondeclist = actiondec list
and whenexprlist = whenexpr list

and globaldec =
    IntStrdec of pridec
  | Charadec of string * membervar list * membervar list
  | Itemdec of string * membervar list
  | Locdec of string * membervar list * membervar list * membervar list
  | Startend of string * expr * stmt

and globaldecs = globaldec list

and program = globaldecs
```

```
(* check.ml by Xiao *)
open Ast
open Checktype

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

module StringMap = Map.Make(String)

exception DupVar of string
exception NotFound of string
exception WrongType of string
exception ProbError of string
exception InvalidKey of string


let print_next_type = function
      Integer -> print_string "int"
  | String -> print_string "string"
  | Character -> print_string "character"
  | Item -> print_string "item"
  | Location -> print_string "location"
  | Action -> print_string "action"
  | Key -> print_string "key"

let print_symboltable symt =
  let print_entry (name,t) tt =
    print_string name;
    print_next_type t;
    print_endline "" in
  VarMap.iter print_entry symt


let check_id symt = function
    Var (name) ->
      if (not (VarMap.mem (name,String) symt)) &&
         (not (VarMap.mem (name,Integer) symt)) &&
         (not (VarMap.mem (name,Location) symt)) &&
         (not (VarMap.mem (name,Character) symt)) &&
         (not (VarMap.mem (name,Item) symt)) then
         raise ( NotFound("undefined variable " ^ name))
        else if (VarMap.mem (name,String) symt) then String
        else if (VarMap.mem (name,Integer) symt) then Integer
        else if (VarMap.mem (name,Location) symt) then Location
        else if (VarMap.mem (name,Character) symt) then Character
        else Item
  | Has (name, subname) ->
      if (not (VarMap.mem (name,Character) symt) ) &&
         (not (VarMap.mem (name,Item) symt) ) &&
         (not (VarMap.mem (name,Location) symt) ) then
            raise ( NotFound("undefined variable " ^ name))
        else
            if (VarMap.mem (name,Character) symt) then
              let subsymt = VarMap.find (name,Character) symt in
              if (not (VarMap.mem (subname, Integer) subsymt)) &&
                 (not (VarMap.mem (subname, String) subsymt)) then
                  raise ( NotFound("member not found " ^ name ^"." ^ subname))
              else
                  if (VarMap.mem (subname, Integer) subsymt) then Integer
                  else String
            else if (VarMap.mem (name,Location) symt) then
              let subsymt = VarMap.find (name,Location) symt in
              if (not (VarMap.mem (subname, Integer) subsymt)) &&
                 (not (VarMap.mem (subname, String) subsymt)) then
                raise ( NotFound("member not found " ^ name ^"." ^ subname))
              else
                  if (VarMap.mem (subname, Integer) subsymt) then Integer
                  else String
            else
              let subsymt = VarMap.find (name,Item) symt in
              if (not (VarMap.mem (subname, Integer) subsymt)) &&
                 (not (VarMap.mem (subname, String) subsymt)) then
                  raise ( NotFound("member not found " ^ name ^"." ^ subname))
              else
                  if (VarMap.mem (subname, Integer) subsymt) then Integer
                  else String


let rec check_expr symt = function
    Binop (expr1 , op, expr2) ->
      if ((op = Eq) || (op = Neq)) then
      if ((check_expr symt expr1)=String &&
         (check_expr symt expr2) = String) then
        Integer
      else if ((check_expr symt expr1)=Integer &&
              (check_expr symt expr2) = Integer) then
        Integer
      else
        raise (WrongType("Type does not match"))
      else
      if (check_expr symt expr1)=Integer &&
         (check_expr symt expr2) = Integer then
        Integer
      else raise (WrongType("Type does not match"))
  | Asn (id, expr) ->
      if ((check_id symt id) = Integer &&
         (check_expr symt expr) = Integer) then Integer
        else if ((check_id symt id) = String &&
        (check_expr symt expr) = String) then String
        else raise (WrongType("Type does not match"))
  | Lit (intvalue) -> Integer
  | LitS (strvalue) -> String
  | Exists (name, subname) ->
      if (VarMap.mem (name, Location) symt) then
            (*print_symboltable subsymt;*)
          if ( (not (VarMap.mem (subname, Item) symt)) &&
             (not (VarMap.mem (subname, Character) symt)) ) then
              raise ( NotFound("Exist error 1" ^ name ^"." ^ subname))
            else
              Integer
        else if (VarMap.mem (name, Character) symt) then
            (*let subsymt = VarMap.find (name,Character) symt in*)
            if ( not (VarMap.mem (subname, Item) symt) ) then
              raise ( NotFound("Exist error 2" ^ name ^"." ^ subname))
            else
              Integer
        else
            raise ( NotFound("Exist error 3" ^ name ^"." ^ subname))



  | Neg (expr) ->
      if (check_expr symt expr) = Integer then Integer
      else raise (WrongType("Type does not match"))
  | Not (expr) ->
      if (check_expr symt expr) = Integer then Integer
      else raise (WrongType("Type does not match"))
  | Ident (id) -> check_id symt id


let check_action = fun actionmap actiondec ->
  match actiondec with
      Unitaction (vname, outstr, key) ->
        if (VarMap.mem (vname,Action) actionmap) then
          raise ( DupVar("duplicated action name " ^ vname))
        else if (VarMap.mem (key,Key) actionmap) then
          raise ( DupVar("Multiple binding for key " ^ key))
        else if not (String.length key = 1) then
          raise (InvalidKey ("Key should be one single character" ^ key))
        else if not ( (Char.code key.[0]) >= (Char.code 'a') &&
                      (Char.code key.[0]) <= (Char.code 'z') ||
                        (Char.code key.[0]) >= (Char.code 'A') &&
                        (Char.code key.[0]) <= (Char.code 'Z') ||
                              (Char.code key.[0]) >= (Char.code '0') &&
                              (Char.code key.[0]) <= (Char.code '9') ) then
            raise (InvalidKey ("Key should be either a digit or a letter" ^ key))
        else
            let actionmap = VarMap.add (vname, Action) Action actionmap in
            VarMap.add (key, Key) Key actionmap

let rec check_probexpr symt total probexpr =
  match probexpr with
      Unitprob (pvalue, probstmt) ->
        check_stmt symt probstmt;
        total+pvalue
and check_whenexpr symt actionmap whenexpr =
  match whenexpr with
      Unitwhen (actionname, whenstmt, locname) ->
       (if not (VarMap.mem (actionname, Action) actionmap) then
          raise ( NotFound("action not defined " ^ actionname))
        else check_stmt symt whenstmt);
        if not (VarMap.mem (locname, Location) symt) then
          raise ( NotFound("Location not found in next " ^ locname))
        else ()
and check_stmt symt =  function
      Ifelse (cond, truestmt, falsestmt) ->
        if not ((check_expr symt cond) = Integer) then
          raise (WrongType("Type does not match"))
        else
          check_stmt symt truestmt;
          check_stmt symt falsestmt
  | Chwhen (actiondeclist, whenexprlist) ->
      let actionmap = List.fold_left check_action VarMap.empty actiondeclist in
        List.iter (check_whenexpr symt actionmap) whenexprlist




  | Prob (probexprlist) ->
      let total = List.fold_left (check_probexpr symt) 0 probexprlist in
        if not (total = 100) then
          raise ( ProbError ("Total Probability is not 100 "))
        else ()
  | Kill (name) ->
      if (not (VarMap.mem (name,Character) symt) ) &&
         (not (VarMap.mem (name,Item) symt) ) then
        raise ( NotFound("Var not found, kill fail " ^ name))
        else ()
  | Grab (name, subname) ->
      if (not (VarMap.mem (name,Character) symt) ) then
        raise ( NotFound("Charactor not found, invalid grab " ^ name))
        else if (not (VarMap.mem (subname,Item) symt) ) then
          raise ( NotFound("Item not found, invalid grab " ^ subname))
        else ()
  | Drop (name, subname) ->
      if (not (VarMap.mem (name,Character) symt) ) then
        raise ( NotFound("Charactor not found, invalid drop " ^ name))
        else if (not (VarMap.mem (subname,Item) symt) ) then
          raise ( NotFound("Item not found, invalid drop " ^ subname))
        else ()
  | Show (name, subname) ->
      if (not (VarMap.mem (name,Location) symt) ) then
        raise ( NotFound("Location not found, invalid show " ^ name))
        else if ( (not (VarMap.mem (subname,Item) symt)) &&
                        (not (VarMap.mem (subname,Character) symt)) ) then
            raise (NotFound("Item or Character not found, invalid show " ^ subname))
        else ()
  | Hide (name, subname) ->
      if (not (VarMap.mem (name,Location) symt) ) then
        raise ( NotFound("Location not found, invalid hide " ^ name))
        else if ( (not (VarMap.mem (subname,Item) symt)) &&
                  (not (VarMap.mem (subname,Character) symt)) ) then
            raise (NotFound("Item or Character not found, invalid hide " ^ subname))
        else ()
  | Atomstmt (exp) -> ignore (check_expr symt exp)
  | Cmpdstmt (blk) -> List.iter (check_stmt symt) blk
  | Nostmt (nonsense) -> ()
  | Print  (exp) ->
      if (not ((check_expr symt exp) = Integer)) &&
         (not ((check_expr symt exp) = String)) then
        raise (WrongType("Type does not match"))
        else ()

let pridec_tostr = function
   Strdec (name) -> name
 | Intdec (name) -> name
 | Strdecinit (name,initexpr) -> name
 | Intdecinit (name,initexpr) -> name

let check_pridec symt = function
      Strdec (name) ->
        (*print_endline("symt----------");
            print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
             (* *)
          (print_endline("error");
            raise ( DupVar("duplicated identifier " ^ name)))
        else
            VarMap.add (name, String) VarMap.empty symt

  | Intdec (name) ->
      (*print_endline("symt----------");
            print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
          (print_endline("error");
          (* *)
            raise ( DupVar("duplicated identifier " ^ name)))
        else
            VarMap.add (name, Integer) VarMap.empty symt

  | Strdecinit (name,initexpr) ->
      ignore (check_expr symt initexpr);
        (*print_endline("symt----------");
        print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
          (* *)
            (print_endline("error");
            raise ( DupVar("duplicated identifier " ^ name)))
        else
            VarMap.add (name, String) VarMap.empty symt
  | Intdecinit (name,initexpr) ->
      ignore (check_expr symt initexpr);
        (*print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
          (* *)
            (print_endline("error");
            raise ( DupVar("duplicated identifier " ^ name)))
        else
            VarMap.add (name, Integer) VarMap.empty symt
```

```
let rec check_membervarlist_intstr subsymt = function
    [] -> subsymt
  | member::tl ->
      match member with
            Primember (pridec) ->
              check_membervarlist_intstr (check_pridec subsymt pridec) tl
        | Varref (varname) -> raise (WrongType("Type does not match "^varname))

let rec check_membervarlist_item symt subsymt = function
    [] -> subsymt
  | member::tl ->
      match member with
            Primember (pridec) ->
              raise (WrongType("Type does not match " ^(pridec_tostr pridec)))
        | Varref (varname) ->
            if not (VarMap.mem (varname, Item) symt) then
                  raise ( NotFound("undefined variable " ^ varname))
              else
                  if (VarMap.mem (varname,Item) subsymt) then
                    raise ( DupVar("duplicated identifier " ^ varname))
                  else
                    check_membervarlist_item symt
                    (VarMap.add (varname,Item) VarMap.empty subsymt) tl

let rec check_membervarlist_chara symt subsymt = function
    [] -> subsymt
  | member::tl ->
      match member with
            Primember (pridec) ->
              raise (WrongType("Type does not match " ^(pridec_tostr pridec)))
        | Varref (varname) ->
            if not (VarMap.mem (varname, Character) symt) then
                raise ( NotFound("undefined variable " ^ varname))
              else
                  if (VarMap.mem (varname,Character) subsymt) then
                    raise ( DupVar("duplicated identifier " ^ varname))
                  else
                    check_membervarlist_chara symt
                    (VarMap.add (varname,Character) VarMap.empty subsymt) tl


let check_globaldec symt locmap = function
    IntStrdec (pridec) -> check_pridec symt pridec, locmap
  | Charadec (name, memberlist1, memberlist2) ->
      (*print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
          raise ( DupVar("duplicated identifier in character dec " ^ name))
        else
            let subsymt = VarMap.empty in
            let subsymt = check_membervarlist_intstr subsymt memberlist1 in
            let subsymt = check_membervarlist_item symt subsymt memberlist2 in
            (VarMap.add (name, Character) subsymt symt) , locmap
  | Itemdec (name, memberlist1) ->
      (*print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
            raise ( DupVar("duplicated identifier in item dec " ^ name))
        else
            let subsymt = VarMap.empty in
            let subsymt = check_membervarlist_intstr subsymt memberlist1 in
            (VarMap.add (name, Item) subsymt symt) , locmap
  | Locdec (name, memberlist1,memberlist2, memberlist3)->
      (*print_symboltable symt;*)
        if (VarMap.mem (name, Location) symt) ||
             (VarMap.mem (name, Character) symt) ||
             (VarMap.mem (name, Item) symt) ||
             (VarMap.mem (name, String) symt) ||
             (VarMap.mem (name, Integer) symt) then
          raise ( DupVar("duplicated identifier in location dec " ^ name))
        else
            let subsymt = VarMap.empty in
            let subsymt = check_membervarlist_intstr subsymt memberlist1 in
            let subsymt = check_membervarlist_item symt subsymt memberlist2 in
            let subsymt = check_membervarlist_chara symt subsymt memberlist3 in
            (VarMap.add (name, Location) subsymt symt), locmap
  | Startend (name, cond, logicstmt) ->
      (*print_symboltable symt;*)
        if not (VarMap.mem (name, Location) symt)  then
            raise ( NotFound("undefined variable in start stmt " ^ name))
        else
            ignore (check_expr symt cond);
            check_stmt symt logicstmt;
            symt, (StringMap.add name 1 locmap)



let rec check_program (symt,locmap) = function
    [] -> let match_loc tuple dc =
            let name = fst tuple in
            let t = snd tuple in
            if (t = Location) && (not (StringMap.mem name locmap)) then
              raise ( NotFound("No body definition for location " ^ name))
            else
              ()
        in
        VarMap.iter match_loc symt;
        symt
  | dec::tl -> check_program  (check_globaldec symt locmap dec) tl
```

```
(* checktype.mli by Xiao *)
type t =
      Integer
  | String
  | Character
  | Item
  | Location
  | Action (*hack for action name in choose stmt *)
  | Key    (*hack for key binded for an action *)
```

```
(* compile.ml by Everybody *)
open Ast
open Expression
open Declaration
open Action
open Selection
open Start
open Statement
open Check

module type COMPILE =
  sig
    exception CompileError of string
    val javacode :
      globaldec list -> ('a VarMap.t VarMap.t) -> string list * string list
    val stmt_to_java :
      ('a VarMap.t VarMap.t) -> string list * string list -> stmt
      -> string list * string list
    val global_dec_to_java :
      string list * string list -> globaldec
      -> ('a VarMap.t VarMap.t) -> string list * string list
  end

module Compile : COMPILE = struct

exception CompileError of string

let rec startend_stmt_check (expression:string) (statement:string list) =
  match statement with
    []->[]
  |hd::tl ->
    if (String.contains hd ';') then
      [hd] @ ["if (" ^expression ^")"; "endGame();"] @
        (startend_stmt_check expression tl)
      else [hd] @ (startend_stmt_check expression tl)

let rec actiondeclist_to_java list num = match list with
    [] -> []
  | hd::tail ->
      match hd with
        Unitaction(action, actionname, key) ->
          ["keysToActionName" ^ string_of_int num ^
           ".put(\"" ^ key ^ "\", \"" ^ action ^ "\"" ^ ");";
           "actionNameToOutput" ^ string_of_int num ^
           ".put(\"" ^ action ^ "\", \"" ^ actionname ^ "\");";
           "System.out.println(\"Type " ^ key ^
           " for " ^ actionname ^ "\");"] @ actiondeclist_to_java tail num

let rec prob_sum list = match list with
    [] -> 0
  | hd::tail ->
      match hd with Unitprob(i, stmt) -> i + (prob_sum tail)

let rec stmt_to_java tmap (playcode, startfns) stmt =
  match stmt with
    Ifelse (expr, stmt1, stmt2) ->
      let (expr_precode, expr_exp) =
          Expression.expr_to_java_boolean expr tmap in
      let (stmt1_playcode, stmt1_startfns) =
          stmt_to_java tmap ([], []) stmt1  in
      let (stmt2_playcode, stmt2_startfns) =
          stmt_to_java tmap ([], []) stmt2  in
      (playcode @ expr_precode @ ["if(" ^ expr_exp ^ ") {"] @
      stmt1_playcode @ ["}"; "else {"] @ stmt2_playcode @ ["}"],
      startfns @ stmt1_startfns @ stmt2_startfns)
  | Chwhen (actiondeclist, whenexprlist) ->
      let num = List.length(playcode) in
      let mapDecl = ["Map<String,String> keysToActionName" ^
                    string_of_int num ^ " = new HashMap<String, String>();";
                    "Map<String, String> actionNameToOutput" ^
                    string_of_int num ^
                    " = new HashMap<String, String>();"] in
      let actiondecs = ["System.out.println(\"CHOOSE AN ACTION:\");"]
                       @ actiondeclist_to_java actiondeclist num in
      let getinput = ["Scanner in" ^ string_of_int num ^
                      " = new Scanner(System.in);";
                     "String input" ^ string_of_int num ^
                     " = in" ^ string_of_int num ^ ".nextLine();";
                     "while(!keysToActionName" ^ string_of_int num ^
                     ".containsKey(input" ^ string_of_int num ^ ")) {";
                     "System.out.println(\"Invalid input, try again\");";
                     "input" ^ string_of_int num ^ " = in" ^
                     string_of_int num ^ ".nextLine();";
                     "}";
                     "System.out.println(\"You typed \" + input" ^
                     string_of_int num ^ ");";
                     "String action" ^ string_of_int num ^
                     " = keysToActionName" ^ string_of_int num ^
                     ".get(input" ^ string_of_int num ^ ");" ] in
      let whenexprs_playcode, whenexprs_startfns =
        whenexprs_to_java whenexprlist num tmap in
      (playcode @ mapDecl @ actiondecs @ getinput @
      whenexprs_playcode, startfns @ whenexprs_startfns)

  | Prob (probexprlist) ->
      let sum = prob_sum probexprlist in
      if sum != 100 then
        raise (CompileError "Probabilities did not sum to 100")
      else
        let randomcode = ["int num = r.nextInt(100);"] in
        let (probexprs_code, probexprs_startfns) =
          probexprs_to_java 0 probexprlist tmap in
      (playcode @ randomcode @ probexprs_code, startfns @ probexprs_startfns)

  | Kill (str) -> (playcode @ (Action.kill_to_java str), startfns)
  | Grab (str1, str2) ->
      (playcode @ (Action.grab_to_java str1 str2), startfns)
  | Drop (str1, str2) ->
      (playcode @ (Action.drop_to_java str1 str2), startfns)
  | Show (str1, str2) ->
      (playcode @ (Action.show_to_java str1 str2 tmap), startfns)
  | Hide (str1, str2) ->
      (playcode @ (Action.hide_to_java str1 str2 tmap), startfns)
  | Atomstmt (expr) ->
      (playcode @ ["dummy = (" ^(Expression.expr_to_java expr tmap)^");"],
      startfns)
  | Cmpdstmt (codeblock) ->
      let (blockcode, startfns) =
        List.fold_left (stmt_to_java tmap) ([], []) codeblock
      in
      (playcode @ ["{"] @ (blockcode) @ ["}"], startfns)


  | Nostmt (i) -> (playcode @ ["//Empty stmt"], startfns)
  | Print (str) ->
    (playcode @
    ["System.out.println(\"\"+ (" ^Expression.expr_to_java str tmap^"));"],
    startfns)


and whenexprs_to_java list num tmap = match list with
    [] -> ([], [])
  | hd :: tail ->
      let (hd_playcode, hd_startfns) = whenexpr_to_java hd num tmap in
      let (tail_playcode, tail_startfns) = whenexprs_to_java tail num tmap in
      (hd_playcode @ tail_playcode, hd_startfns @ tail_startfns)

and whenexpr_to_java whenexpr num tmap =
  match whenexpr with
    Unitwhen(action, stmt, loc) ->
      let (when_playcode, when_startfns) = stmt_to_java tmap ([], []) stmt in
      let nextcode = [loc ^ "();"] in
      (["if(action" ^ string_of_int num ^ ".equals(\"" ^ action ^   "\")) {"]
      @ when_playcode @ nextcode @ ["}"], when_startfns)
```

```
and probexprs_to_java start_num list tmap = match list with
    [] -> ([], [])
  | hd::tail ->
    let (hd_playcode, hd_startfns, curr_num) =
      probexpr_to_java hd start_num tmap in
    let (tail_playcode, tail_startfns) =
      probexprs_to_java curr_num tail tmap in
    (hd_playcode @ tail_playcode, hd_startfns @ tail_startfns)

and probexpr_to_java probexpr start_num tmap =
  match probexpr with
    Unitprob(i, stmt) ->
      let (prob_playcode, prob_startfns) = stmt_to_java tmap ([], []) stmt in
      (["if(num >= " ^ string_of_int start_num ^ " && num < "
      ^ string_of_int (start_num + i) ^ ") {"] @ prob_playcode @ ["}"],
      prob_startfns, start_num + i)

exception InvalidCode of string

(* TODO: FIX THIS *)
let global_dec_to_java (playcode, startfns) global_dec tmap =
  match global_dec with
    IntStrdec (pridec) ->
        let l = Declaration.intstrdec_to_java pridec tmap in
        (match l with
        []-> (playcode, startfns)
      | hd::hd2::tl -> (playcode @ tl, startfns @ [hd; hd2])
      | _::tl -> raise (InvalidCode("Invalid Code")))
  | Charadec (name, membervar1, membervar2) ->
      let l = Declaration.charadec_to_java name membervar1 membervar2 tmap in
        (match l with
        []-> (playcode, startfns)
      | hd::hd2::tl -> (playcode @ tl, startfns @ [hd; hd2])
      | _::tl -> raise (InvalidCode("Invalid Code")))
  | Itemdec (name, membervar) ->
      let l = Declaration.itemdec_to_java name membervar tmap in
      (match l with
        []-> (playcode, startfns)
      | hd::hd2::tl -> (playcode @ tl, startfns @ [hd; hd2])
      | _::tl -> raise (InvalidCode("Invalid Code")))
  | Locdec (name, membervar1, membervar2, membervar3) ->
      let l =
        Declaration.locdec_to_java name membervar1 membervar2 membervar3 tmap in
        (match l with
          []-> (playcode, startfns)
        | hd::hd2::tl -> (playcode @ tl, startfns @ [hd; hd2])
        | _::tl -> raise (InvalidCode("Invalid Code")))
  | Startend (name, expr, stmt) ->
      playcode @ ["//Location function call"; name ^ "();"],
      startfns @ ["//start funtion"; "public void " ^ name ^ "() {"] @
        (fst (Expression.expr_to_java_boolean expr tmap)) @
        ["currentLocation = \"" ^ name ^ "\";"]@
        ["if (" ^(snd (Expression.expr_to_java_boolean expr tmap)) ^")";
        "endGame();"] @
        ["while (!(" ^ (snd (Expression.expr_to_java_boolean expr tmap)) ^")){"] @
        (startend_stmt_check (snd (Expression.expr_to_java_boolean expr tmap))
        (fst (stmt_to_java tmap ([], []) stmt)) )
        @ (fst (Expression.expr_to_java_boolean expr tmap))@ ["}"  ; "}"]

let print_globaldec global_dec = match global_dec with
    IntStrdec (pridec) ->
      print_endline "pridec"
  | Charadec (name, membervar1, membervar2) ->
      print_endline ("char " ^ name)
  | Itemdec (name, membervar) ->
      print_endline ("item " ^ name)
  | Locdec (name, membervar1, membervar2, membervar3) ->
      print_endline ("loc " ^ name)
  | Startend (name, expr, stmt) -> print_endline ("Start "^name)

let rec javacode program symt = match program with
```

```
    [] -> ([], [])
  | hd::tl ->
    let tuple = javacode tl symt in
    (fst (global_dec_to_java ([], []) hd symt))@ (fst tuple),
    (snd (global_dec_to_java ([], []) hd symt))@ (snd tuple)
end
```

```
(* declaration.ml by Pri: Danny, Sec: Ernesto, Morgan *)
open Ast
open Expression
open Check

module type DECLARATION = sig
   val intstrdec_to_java : pridec -> ('a VarMap.t VarMap.t) -> string list
   val itemdec_to_java : string -> membervarlist ->
                         ('a VarMap.t VarMap.t) -> string list
   val charadec_to_java : string -> membervarlist -> membervarlist ->
                          ('a VarMap.t VarMap.t) -> string list
   val locdec_to_java : string -> membervarlist -> membervarlist ->
                        membervarlist -> ('a VarMap.t VarMap.t)  -> string list
end


module Declaration : DECLARATION = struct

let attr_to_java tmap (var:string) (attr:membervar) : string list =
  match attr with
    Primember(pridec) ->
      (match pridec with
        Strdec (str) ->
          [var ^ ".addStrAttr(\"" ^ str ^ "\", \"\");";
          "types.put(\"" ^ str ^ "\", Type.STRING);"]
      | Intdec (str) ->
          [var ^ ".addIntAttr(\"" ^ str ^ "\", 0);";
          "types.put(\"" ^ str ^ "\", Type.INT);"]
      | Strdecinit (str, expr) ->
          [var ^ ".addStrAttr(\"" ^ str ^ "\"," ^
          (Expression.expr_to_java expr tmap) ^ ");";
          "types.put(\"" ^ str ^ "\", Type.STRING);"]
      | Intdecinit (str, expr) ->
          [ var ^ ".addIntAttr(\"" ^ str ^ "\", " ^
          (Expression.expr_to_java expr tmap) ^ ");";
          "types.put(\"" ^ str ^ "\", Type.INT);"])
  | Varref(str) -> ["//OOPS THIS WAS BAD!"]

let rec attrlist_to_java var attrlist tmap =
  match attrlist with
    [] -> []
  | hd::tl -> (attr_to_java tmap var hd) @ (attrlist_to_java var tl tmap)

let itemdec_to_java str attrlist tmap =
  ["//itemdec"; "Item " ^ str ^ " = new Item();";
  "items.put(\"" ^ str ^ "\"," ^ str ^ ");";
  "types.put(\"" ^ str ^ "\", Type.ITEM);"] @
  (attrlist_to_java str attrlist tmap)

let item_to_java (var:string) (item:membervar) : string list =
  match item with
    Varref(str) -> [var ^".addItem(\"" ^str^ "\");"]
  | Primember(pridec) ->
      ["//This was bad. Not a reference to a complex variable"]

let rec itemlist_to_java var itemlist =
  match itemlist with
    [] -> []
  | hd::tl -> (item_to_java var hd) @ (itemlist_to_java var tl)

let charadec_to_java str attrlist itemlist tmap =
  ["//charadec"; "Character " ^ str ^ " = new Character();";
  "characters.put(\"" ^ str ^ "\"," ^ str ^ ");";
  "types.put(\"" ^ str ^ "\", Type.CHARACTER);"]@
  (attrlist_to_java str attrlist tmap) @ (itemlist_to_java str itemlist)

let character_to_java (var:string) (character:membervar) : string list =
  match character with
    Varref(str) -> [var ^".showCharacter(\"" ^str^ "\");"]
  | Primember(pridec) ->
      ["//This was bad. Not a reference to a complex variable"]
```

```
let rec characterlist_to_java var characterlist =
  match characterlist with
    [] -> []
  | hd::tl -> (character_to_java var hd) @ (characterlist_to_java var tl)

let locdec_to_java str attrlist itemlist charlist tmap =
  ["//locdec"; "Location " ^ str ^ " = new Location();";
  "locations.put(\"" ^ str ^ "\"," ^ str ^ ");";
  "types.put(\"" ^ str ^ "\", Type.LOCATION);"]@
  (attrlist_to_java str attrlist tmap) @
  (itemlist_to_java str itemlist) @
  (characterlist_to_java str charlist)

let intstrdec_to_java pridec tmap = match pridec with
    Strdec(str) -> ["//strdec"; "String " ^ str ^ " = \"\";"]
  | Intdec(str) -> ["//intdec"; "int " ^ str ^ ";"]
  | Strdecinit(str, expr) ->
      ["//strdecinit"; "String " ^ str ^ ";";
      str ^ " = " ^ (Expression.expr_to_java expr tmap) ^ ";"]
  | Intdecinit(str, expr) ->
      ["//intdecinit"; "int " ^ str ^ ";";
      str ^ " = " ^ (Expression.expr_to_java expr tmap) ^ ";"]
end
```

```
(* expression.ml by Pri: Danny, Sec: Morgan, Ernesto *)
open Ast
open Checktype
open Check


module type EXPRESSION =
   sig
    val expr_to_java : expr -> ('a VarMap.t VarMap.t) -> string
    val expr_to_java_boolean :
      expr -> ('a VarMap.t VarMap.t) -> string list * string
      (* string list contains statements that need
      to happen before the string condition is checked *)
    val next_type_to_string : Checktype.t -> string
    val check_type_to_string : string -> ('a VarMap.t VarMap.t) -> string
   end

module Expression : EXPRESSION = struct

exception InvalidComparison of string

let next_type_to_string = function
      Integer -> "int"
  | String -> "string"
  | Character -> "character"
  | Item -> "item"
  | Location -> "location"
  | Action -> "action"
  | Key -> "key"

let check_type_to_string name  = function
  symt ->
    if (not (VarMap.mem (name,String) symt)) &&
       (not (VarMap.mem (name,Integer) symt)) &&
       (not (VarMap.mem (name,Location) symt)) &&
       (not (VarMap.mem (name,Character) symt)) &&
       (not (VarMap.mem (name,Item) symt)) then
      raise ( NotFound("undefined variable " ^ name))
    else if (VarMap.mem (name,String) symt) then "String"
      else if (VarMap.mem (name,Integer) symt) then "Int"
      else if (VarMap.mem (name,Location) symt) then "Location"
      else if(VarMap.mem (name,Character) symt) then "Character"
    else "Item"

let rec expr_to_java exp tmap =
  match exp with
    Binop (exp1, op, exp2) ->
      if op == Add then
        "(" ^ (expr_to_java exp1 tmap) ^ " + " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Sub then
        "(" ^ (expr_to_java exp1 tmap) ^ " - " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Mul then
        "(" ^ (expr_to_java exp1 tmap) ^ " * " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Div then
        "(" ^ (expr_to_java exp1 tmap) ^ " / " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Or then
        "boolToInt(isTrue(" ^ (expr_to_java exp1 tmap) ^
        ") || isTrue(" ^ (expr_to_java exp2 tmap) ^ "))"
      else if op == And then
        "boolToInt(isTrue(" ^ (expr_to_java exp1 tmap) ^
        ") && isTrue(" ^ (expr_to_java exp2 tmap) ^ "))"
      else if op == Eq then
        let t = check_expr tmap exp1 in
        (match t with
          String ->
            "boolToInt(" ^ (expr_to_java exp1 tmap) ^
            ".equals(" ^ (expr_to_java exp2 tmap) ^ "))"
        | Integer ->
            "boolToInt(" ^ (expr_to_java exp1 tmap) ^
            " == " ^ (expr_to_java exp2 tmap) ^ ")"
        | _ -> raise (InvalidComparison("Invalid Comparison")))
      else if op == Lt then
```

```
        "boolToInt (" ^ (expr_to_java exp1 tmap) ^
        " < " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Gt then
        "boolToInt (" ^ (expr_to_java exp1 tmap) ^
        " > " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Leq then
        "boolToInt (" ^ (expr_to_java exp1 tmap) ^
        " <= " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Geq then
        "boolToInt (" ^ (expr_to_java exp1 tmap) ^
        " >= " ^ (expr_to_java exp2 tmap) ^ ")"
      else if op == Neq then
        "boolToInt (" ^ (expr_to_java exp1 tmap) ^
        " != " ^ (expr_to_java exp2 tmap) ^ ")"
      else raise (InvalidComparison("Invalid Comparison"))
  | Asn (id, exp) ->
      let t = check_id tmap id in
      (match id with
        Var(str) -> str ^ " = " ^ (expr_to_java exp tmap)
      | Has(name, subname) ->
          "entitySet" ^ (String.capitalize (next_type_to_string t)) ^
          "(\"" ^ name ^ "\", Type." ^
          (String.uppercase (check_type_to_string name tmap)) ^
          ", \"" ^ subname ^ "\", " ^ (expr_to_java exp tmap) ^ ")")
  | Lit (i) -> "(" ^ (string_of_int i) ^ ")"
  | LitS (str) -> "\"" ^ str ^ "\""
  | Exists (str1, str2) ->
      let t1 = check_id tmap (Var(str1)) in
      let t2 = check_id tmap (Var(str2)) in
      (match t2 with
        Item ->
          "entityExistsItem(\"" ^ str1 ^ "\", Type." ^
          (String.uppercase (next_type_to_string t1)) ^
          ", \"" ^ str2 ^ "\")"
      | Character ->
          "entityExistsCharacter(\"" ^ str1 ^ "\", Type." ^
          (String.uppercase (next_type_to_string t1))
          ^ ", \"" ^ str2 ^ "\")"
      | _ -> raise (InvalidComparison("Invalid Comparison")))
  | Ident (id) ->
      let t = check_id tmap id in
      (match id with
        Var(name) -> name
      | Has(name, subname) ->
          "entityHas" ^ (String.capitalize (next_type_to_string t)) ^
          "(\"" ^ name ^ "\", Type." ^
          (String.uppercase (check_type_to_string name tmap))
          ^ ", \"" ^ subname ^ "\")")
  | Neg (exp) -> "(-" ^ (expr_to_java exp tmap) ^ ")"
  | Not (exp) -> "boolToInt(!" ^ "isTrue(" ^ (expr_to_java exp tmap) ^ ") )"


let rec expr_to_java_boolean exp tmap =
  match exp with
      Binop (exp1, op, exp2) ->
        if op == Add then
          ([], "(" ^ (expr_to_java exp1 tmap) ^ " + "
          ^ (expr_to_java exp2 tmap) ^ ") != 0" )
        else if op == Sub then
          ([], "(" ^ (expr_to_java exp1 tmap) ^ " - " ^
          (expr_to_java exp2 tmap) ^ ") != 0")
        else if op == Mul then
          ([], "(" ^ (expr_to_java exp1 tmap) ^ " * "
          ^ (expr_to_java exp2 tmap) ^ ") != 0")
        else if op == Div then
          ([], "(" ^ (expr_to_java exp1 tmap) ^ " / "
          ^ (expr_to_java exp2 tmap) ^ ") != 0")
        else if op == Eq then
          let t = check_expr tmap exp1 in
        (match t with
          String ->
            ([], (expr_to_java exp1 tmap) ^
            ".equals(" ^ (expr_to_java exp2 tmap) ^ ")")
        | Integer ->
            ([], (expr_to_java exp1 tmap) ^ " == " ^ (expr_to_java exp2 tmap))
        | _ -> raise (InvalidComparison("Invalid Comparison")))
        else if op == Lt then
          ([], "(" ^ (expr_to_java exp1 tmap) ^
          " < " ^ (expr_to_java exp2 tmap) ^ ")")
      else if op == Gt then
        ([], "(" ^ (expr_to_java exp1 tmap) ^
        " > " ^ (expr_to_java exp2 tmap) ^ ")")
        else if op == Neq then
          ([], "(" ^ (expr_to_java exp1 tmap) ^
          " != " ^ (expr_to_java exp2 tmap) ^ ")")
        else if op == Leq then
          ([], "(" ^ (expr_to_java exp1 tmap) ^
          " <= " ^ (expr_to_java exp2 tmap) ^ ")")
      else if op == Geq then
        ([], "(" ^ (expr_to_java exp1 tmap) ^
        " >= " ^ (expr_to_java exp2 tmap) ^ ")")
        else if op == Or then
          (fst (expr_to_java_boolean exp1 tmap) @
          fst (expr_to_java_boolean exp2 tmap),
          (snd (expr_to_java_boolean exp1 tmap)) ^
          " || " ^ (snd (expr_to_java_boolean exp2 tmap)))
      else if op == And then
        (fst (expr_to_java_boolean exp1 tmap) @
        fst (expr_to_java_boolean exp2 tmap),
        (snd (expr_to_java_boolean exp1 tmap)) ^
        " && " ^ (snd (expr_to_java_boolean exp2 tmap)))
        else ([], "false")
  | Asn (id, exp) ->
      let t = check_id tmap id in
      (match id with
        Var(str) ->
          ([str ^ " = " ^ (expr_to_java exp tmap)],
          "(" ^ (expr_to_java exp tmap) ^ ") != 0 ")
      | Has(name, subname) ->
          (["entitySet" ^ (String.capitalize (next_type_to_string t)) ^
          "(\"" ^ name ^ "\", Type." ^
          (String.uppercase (check_type_to_string name tmap)) ^
          ", \"" ^ subname ^ "\", " ^ (expr_to_java exp tmap) ^ ")"],
          "(" ^ (expr_to_java exp tmap) ^ ") != 0 "))
  | Lit (i) -> ([], "isTrue(" ^ (string_of_int i) ^ ")")
  | LitS (str) -> ([], "isTrue(" ^ str ^ ")")
  | Exists (str1, str2) ->
      let t1 = check_id tmap (Var(str1)) in
      let t2 = check_id tmap (Var(str2)) in
      (match t2 with
        Item ->
          ([], "isTrue(entityExistsItem(\"" ^ str1 ^
          "\", Type." ^ (String.uppercase (next_type_to_string t1)) ^
          ", \"" ^ str2 ^ "\"))")
      | Character ->
        ([], "isTrue(entityExistsCharacter(\"" ^ str1 ^
        "\", Type." ^ (String.uppercase (next_type_to_string t1)) ^
        ", \"" ^ str2 ^ "\"))")
      | _ -> raise (InvalidComparison("Invalid Comparison")))
  | Ident (id) -> ([], "isTrue(" ^ (expr_to_java exp tmap) ^ ")")
  | Neg (exp) -> ([], (expr_to_java exp tmap) ^ " != 0 ")
  | Not (exp) -> ([], "!isTrue(" ^ (expr_to_java exp tmap) ^ ")")

end
```

```
#Makefile
TARFILES = Makefile scanner.mll parser.mly ast.mli check.ml checktype.mli expression.ml
declaration.ml

action.ml selection.ml start.ml statement.ml compile.ml next.ml

OBJS = parser.cmo scanner.cmo check.cmo expression.cmo declaration.cmo action.cmo
selection.cmo

start.cmo statement.cmo compile.cmo next.cmo
LIBPATH = -I +sdl

next : $(OBJS)
      ocamlc -o next $(OBJS)

scanner.ml : scanner.mll
      ocamllex scanner.mll

parser.ml parser.mli : parser.mly
      ocamlyacc -v parser.mly

%.cmo : %.ml
      ocamlc -c $(LIBPATH) $<

%.cmi : %.mli
      ocamlc -c $(LIBPATH) $<

next.tar.gz : $(TARFILES)
      cd .. && tar zcf next/next.tar.gz $(TARFILES:%=next/%)

.PHONY : clean
clean :
      rm -f next parser.ml parser.mli scanner.ml *.cmo *.cmi *.class

# Generated by ocamldep *.ml *.mli
action.cmo:
action.cmx:
check.cmo: checktype.cmi ast.cmi
check.cmx: checktype.cmi ast.cmi
compile.cmo: statement.cmo start.cmo selection.cmo expression.cmo \
    declaration.cmo ast.cmi action.cmo
compile.cmx: statement.cmx start.cmx selection.cmx expression.cmx \
    declaration.cmx ast.cmi action.cmx
declaration.cmo: expression.cmo ast.cmi
declaration.cmx: expression.cmx ast.cmi
expression.cmo: ast.cmi
expression.cmx: ast.cmi
next.cmo: scanner.cmo parser.cmi compile.cmo check.cmo ast.cmi
next.cmx: scanner.cmx parser.cmx compile.cmx check.cmx ast.cmi
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
selection.cmo: ast.cmi
selection.cmx: ast.cmi
start.cmo: ast.cmi
start.cmx: ast.cmi
statement.cmo: ast.cmi
statement.cmx: ast.cmi
ast.cmi:
checktype.cmi:
parser.cmi: ast.cmi
```

```
(* next.ml by Pri: Morgan, Sec: Everybody *)
open Ast
open Compile
open Check

let java_of_prog program symt =
let (playcode, startfns) = Compile.javacode program symt in
"
import java.util.*;

public class Next {
   enum Type {INT, STRING, CHARACTER, ITEM, LOCATION}

   static Random r = new Random();
   Object dummy;
   String currentLocation;
   Map<String, Location> locations = new HashMap<String, Location>();
   Map<String, Character> characters = new HashMap<String, Character>();
   Map<String, Item> items = new HashMap<String, Item>();
   Map<String, Type> types = new HashMap<String, Type>();

   public static void main(String[] args) {
    (new Next()).play();
   }

   public int boolToInt(boolean value) {
        if(value) {
            return 1;
        }
        else {
            return 0;
        }
   }

   public String entitySetString(String key1, Type type1, String key2, String value) {
    boolean valueSet = false;
      if(type1 == Type.LOCATION) {
            Location loc = locations.get(key1);
            if(loc != null) {
                  loc.strAttrs.put(key2, value);
                  valueSet = true;
            }
      }
      else if(type1 == Type.CHARACTER) {
            Character character = characters.get(key1);
            if(character != null) {
                  character.strAttrs.put(key2, value);
                  valueSet = true;
            }
      }
      else if(type1 == Type.ITEM) {
            Item item = items.get(key1);
            if(item != null) {
                  item.strAttrs.put(key2, value);
                  valueSet = true;
            }
      }

      if(!valueSet) {
            throw new RuntimeException();
      }

      return value;
   }

   public int entitySetInt(String key1, Type type1, String key2, int value) {
      boolean foundReturnValue = false;

      if(type1 == Type.LOCATION) {
            Location loc = locations.get(key1);
            if(loc != null) {
            loc.intAttrs.put(key2, value);
```

```
               foundReturnValue = true;
         }
   }
   else if(type1 == Type.CHARACTER) {
         Character character = characters.get(key1);
         if(character != null) {
               character.intAttrs.put(key2, value);
               foundReturnValue = true;
         }
   }
   else if(type1 == Type.ITEM) {
         Item item = items.get(key1);
         if(item != null) {
               item.intAttrs.put(key2, value);
               foundReturnValue = true;
         }
   }

   if(foundReturnValue == false) {
         throw new RuntimeException();
   }

   return value;
}

   public boolean isTrue(Object object) {
       if(object instanceof String) {
          if(((String)object).isEmpty()) {
              return false;
          }
       }
       else if(object instanceof Integer) {
          if((Integer)object == 0) {
              return false;
          }
       }
       else {
           if(object == null) {
               return false;
           }
       }

       return true;
   }

 public void killFunction(String varName){
   if (characters.containsKey(varName)){
         characters.remove(varName);
         for (String key: locations.keySet()){
                     locations.get(key).hideCharacter(varName);
         }
   }
   else if (items.containsKey(varName)){
         items.remove(varName);
         for (String key:locations.keySet()){
               locations.get(key).removeItem(varName);
               }
         for (String key : characters.keySet()){
               characters.get(key).removeItem(varName);
         }
   }
 }

 public int entityHasInt(String key1, Type type1, String key2) {
   int returnValue = 0;
   boolean foundReturnValue = false;

   if(type1 == Type.LOCATION) {
         Location loc = locations.get(key1);
         if(loc != null) {
               returnValue = loc.intAttrs.get(key2);
               foundReturnValue = true;
```

```
        }
  }
  else if(type1 == Type.CHARACTER) {
        Character character = characters.get(key1);
        if(character != null) {
              returnValue = character.intAttrs.get(key2);
              foundReturnValue = true;
        }
  }
  else if(type1 == Type.ITEM) {
        Item item = items.get(key1);
        if(item != null) {
              returnValue = item.intAttrs.get(key2);
              foundReturnValue = true;
        }
  }

  if(foundReturnValue == false) {
        throw new RuntimeException();
  }

  return returnValue;
}

public String entityHasString(String key1, Type type1, String key2) {
  String returnValue = null;
  if(type1 == Type.LOCATION) {
        Location loc = locations.get(key1);
        if(loc != null) {
              returnValue = loc.strAttrs.get(key2);
        }
  }
  else if(type1 == Type.CHARACTER) {
        Character character = characters.get(key1);
        if(character != null) {
              returnValue = character.strAttrs.get(key2);
        }
  }
  else if(type1 == Type.ITEM) {
        Item item = items.get(key1);
        if(item != null) {
              returnValue = item.strAttrs.get(key2);
        }
  }

  if(returnValue == null) {
        throw new RuntimeException();
  }

  return returnValue;
}

public Item entityHasItem(String key1, Type type1, String key2) {
  Item returnValue = null;
  if(type1 == Type.LOCATION) {
        Location loc = locations.get(key1);
        if(loc != null) {
              if(loc.items.contains(key2)) {
                    returnValue = items.get(key2);
              }
        }
  }
  else if(type1 == Type.CHARACTER) {
        Character character = characters.get(key1);
        if(character != null) {
              if(character.items.contains(key2)) {
                    returnValue = items.get(key2);
              }
        }
  }

  if(returnValue == null) {
         throw new RuntimeException();
   }

   return returnValue;
 }

 public Character entityHasCharacter(String key1, Type type1, String key2) {
   Character returnValue = null;
   if(type1 == Type.LOCATION) {
         Location loc = locations.get(key1);
         if(loc != null) {
               if(loc.characters.contains(key2)) {
                     returnValue = characters.get(key2);
               }
         }
   }

   if(returnValue == null) {
         throw new RuntimeException();
   }

   return returnValue;
 }

 public int entityExistsItem(String key1, Type type1, String key2) {
   Object returnValue = null;
   if(type1 == Type.LOCATION) {
         Location loc = locations.get(key1);
         if(loc != null) {
               if(loc.items.contains(key2)) {
                     returnValue = items.get(key2);
               }
         }
   }
   else if(type1 == Type.CHARACTER) {
         Character character = characters.get(key1);
         if(character != null) {
               if(character.items.contains(key2)) {
                     returnValue = items.get(key2);
               }
         }
   }

   if(returnValue == null) {
         return 0;
   }

   return 1;
 }

 public int entityExistsCharacter(String key1, Type type1, String key2) {
   Object returnValue = null;
   if(type1 == Type.LOCATION) {
         Location loc = locations.get(key1);
         if(loc != null) {
               if(loc.characters.contains(key2)) {
                     returnValue = characters.get(key2);
               }
         }
   }

   if(returnValue == null) {
         return 0;
   }

   return 1;
 }

public void endGame() {
   System.exit(0);
}
```

```
   public void play() {
"
 ^ (String.concat "\n" playcode) ^ "
   endGame();
   } \n"

 ^ (String.concat "\n" startfns) ^ "
}

abstract class Entity {

   Map<String, Integer> intAttrs = new HashMap<String, Integer>();
   Map<String, String> strAttrs = new HashMap<String, String>();

   public void addIntAttr(String name, int value) {
      intAttrs.put(name, value);
   }

   public void addStrAttr(String name, String value) {
      strAttrs.put(name, value);
   }
}

class Location extends Entity {
   Set<String> characters = new HashSet<String>();
   Set<String> items = new HashSet<String>();

   public void addItem(String name, Map<String, Item> itemses) {
      if(itemses.containsKey(name))
                  items.add(name);
      else
            System.out.println(\"Error: The item you attempted to add no longer
exists\");
   }

   public void addItem(String name){
      items.add(name);
      }

   public void removeItem(String name) {
      items.remove(name);
   }

   public void showCharacter(String name, Map<String, Character> characterses) {
      if(characterses.containsKey(name))
      characters.add(name);
      else
            System.out.println(\"Error: The character you attempted to use no longer
exists\");
   }
   public void showCharacter(String name){

      characters.add(name);
      }

   public void hideCharacter(String name) {
      characters.remove(name);
   }
}

class Character extends Entity {
   Set<String> items = new HashSet<String>();

   public void addItem(String name, String locationNow, Map<String, Location> locations){

      if(locations.get(locationNow).items.contains(name)){
            locations.get(locationNow).removeItem(name);
            items.add(name);
      }
      else
            System.out.println(\"Error: The item you attempted to grab is not in this
location\");
}

public void removeItem(String name, String locationNow, Map<String, Location> locations)
{
            if (items.contains(name)){
            items.remove(name);
            locations.get(locationNow).addItem(name);
   }
      else
            System.out.println(\"Error: The character does not have the item you
attempted to

drop\");
}

   public void addItem(String name) {
      items.add(name);
   }

   public void removeItem(String name) {
      items.remove(name);
   }
}

class Item extends Entity {
}"

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let symt = check_program (VarMap.empty, StringMap.empty) program in
  let java = java_of_prog program symt in
  print_endline java
```

```
/* Parser.mly by Xiao */
%{ open Ast %}


%token PLUS MINUS TIMES DIVIDE LESSTHAN GREATERTHAN EQUAL NEQUAL LOGICAND LOGICOR ASSIGN
LOGICNOT

LEQTHAN GEQTHAN
%token COMMA SEMICOLON DOT
%token LBRACKET RBRACKET LPAREN RPAREN RPROBBLOCK LPROBBLOCK
%token IF THEN ELSE START END PROB WHEN NEXT CHOOSE KILL GRAB HIDE EXISTS DROP SHOW
%token CHARACTER LOCATION ACTION OUTPUT ITEM INT STRING
%token EOF
%token <int> LITERAL
%token <string> VARIABLE
%token <string> STRINGLIT


%nonassoc IF THEN ELSE START END PROB
%left SEMICOLON
%nonassoc OUTPUT
%left COMMA
%right ASSIGN
%left LOGICOR
%left LOGICAND
%left LOGICNOT
%left EQUAL NEQUAL
%left LESSTHAN GREATERTHAN LEQTHAN GEQTHAN
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
%nonassoc EXISTS
%left DOT
%nonassoc LPAREN RPAREN


%start program
%type < Ast.globaldecs> program
%type < Ast.globaldecs> file
%type < Ast.block> block
%type < Ast.expr> expr
%type < Ast.stmt> stmt
%type < Ast.pridec> pridec
%type < Ast.membervar> membervar
%type < Ast.probexpr> probexpr
%type < Ast.globaldec> globaldec

%%
program:
  file EOF {$1}
;

file:
            {[]}
| globaldecs {List.rev $1}
;

stmt:
  IF expr THEN stmt ELSE stmt {Ifelse($2,$4,$6)}
| KILL VARIABLE SEMICOLON                       {Kill($2)}
| GRAB VARIABLE DOT VARIABLE SEMICOLON                      {Grab($2,$4)}
| DROP VARIABLE DOT VARIABLE SEMICOLON                      {Drop($2,$4)}
| HIDE VARIABLE DOT VARIABLE SEMICOLON                      {Hide($2,$4)}
| SHOW VARIABLE DOT VARIABLE SEMICOLON                      {Show($2,$4)}
| RPROBBLOCK probexprlist LPROBBLOCK {Prob(List.rev $2)}
| expr SEMICOLON {Atomstmt ($1) }
| LBRACKET block RBRACKET {Cmpdstmt (List.rev $2) }
| LBRACKET RBRACKET { Nostmt (0) }
| SEMICOLON { Nostmt (0) }
| CHOOSE actiondeclist LBRACKET whenexprlist RBRACKET {Chwhen (List.rev $2, List.rev $4)}
| OUTPUT expr SEMICOLON { Print($2)}
;
```

```
globaldec:
  pridec SEMICOLON {IntStrdec($1)}
| CHARACTER VARIABLE LBRACKET LPAREN membervarlist RPAREN COMMA LPAREN membervarlist
RPAREN RBRACKET

{Charadec($2, List.rev $5, List.rev $9)}
| LOCATION VARIABLE LBRACKET LPAREN membervarlist RPAREN COMMA LPAREN membervarlist
RPAREN COMMA

LPAREN membervarlist RPAREN RBRACKET {Locdec($2,List.rev $5,List.rev $9, List.rev $13)}
| ITEM VARIABLE LBRACKET LPAREN membervarlist RPAREN RBRACKET {Itemdec($2, List.rev $5)}
| START VARIABLE END LPAREN expr RPAREN stmt {Startend ($2, $5, $7)}
;

globaldecs:
 globaldec {[$1]}
| globaldecs globaldec {$2::$1}
;

block:
  stmt { [$1] }
| block stmt { $2::$1 }
;

expr:
  expr PLUS expr              { Binop($1, Add, $3) }
| LOGICNOT expr                     { Not ($2)}
| expr MINUS  expr                  { Binop($1, Sub, $3) }
| expr TIMES  expr                  { Binop($1, Mul, $3) }
| expr DIVIDE expr                  { Binop($1, Div, $3) }
| expr LESSTHAN expr          { Binop($1, Lt, $3) }
| expr GREATERTHAN expr       { Binop($1, Gt ,$3) }
| expr LEQTHAN expr                 { Binop($1, Leq, $3)}
| expr GEQTHAN expr                 { Binop($1, Geq, $3)}
| expr EQUAL expr                   { Binop($1, Eq ,$3) }
| expr NEQUAL expr                  { Binop($1, Neq ,$3) }
| expr LOGICAND expr          { Binop($1, And ,$3) }
| expr LOGICOR expr           { Binop($1, Or ,$3) }
| LPAREN expr RPAREN        { $2 }
| id ASSIGN expr              { Asn($1, $3) }
| LITERAL                           { Lit($1) }
| STRINGLIT                         { LitS($1) }
| EXISTS VARIABLE DOT VARIABLE      { Exists($2,$4)}
| MINUS expr %prec NEG        { Neg($2)}
| id                                { Ident($1)}
;

id:
  VARIABLE DOT VARIABLE       { Has($1,$3)}
| VARIABLE                          { Var($1)}

membervarlist:
  {[]}
|  membervar {[$1]}
| membervarlist COMMA membervar {$3::$1}
;

membervar:
  pridec {Primember($1)}
| VARIABLE {Varref($1)}
;

pridec:
  STRING VARIABLE ASSIGN expr {Strdecinit($2,$4)}
| INT VARIABLE ASSIGN expr {Intdecinit($2,$4)}
| STRING VARIABLE {Strdec($2)}
| INT VARIABLE {Intdec($2)}
;


probexprlist:
  probexpr {[$1]}
| probexprlist probexpr {$2::$1}
;

probexpr:
  PROB LITERAL stmt { Unitprob ($2,$3)}
;

actiondeclist:
  actiondec {[$1]}
| actiondeclist actiondec {$2::$1}
;

actiondec:
  LPAREN VARIABLE COMMA STRINGLIT COMMA STRINGLIT RPAREN { Unitaction ($2,$4,$6)}
;

whenexprlist:
  whenexpr  {[$1]}
| whenexprlist whenexpr {$2::$1}
;

whenexpr:
  WHEN VARIABLE stmt NEXT VARIABLE { Unitwhen($2,$3,$5)}
;
```

```
(* scanner.mll by Ernesto *)
{ open Parser }


rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*" {comment lexbuf}
| '"' [^ '"']* '"' as s { STRINGLIT( String.sub s 1 (String.length s - 2)) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '<' { LESSTHAN }
| "<=" {LEQTHAN}
| ">=" {GEQTHAN}
| '>' { GREATERTHAN }
| "==" { EQUAL }
| "!=" { NEQUAL }
| "and" {LOGICAND}
| "or" {LOGICOR}
| ';' {SEMICOLON}
| '=' {ASSIGN}
| "if" {IF}
| "then" {THEN}
| "else" {ELSE}
| '{' {LBRACKET}
| '}' {RBRACKET}
| '(' {LPAREN}
| ')' {RPAREN}
| ',' {COMMA}
| "output" {OUTPUT}
| "not" {LOGICNOT}
|'.' {DOT}
|"start" {START}
|"end" {END}
|"prob" {PROB}
|"[?" {RPROBBLOCK}
|"?]" {LPROBBLOCK}
|"when" {WHEN}
|"next" {NEXT}
|"choose" {CHOOSE}
|"kill" {KILL}
|"grab" {GRAB}
|"drop" {DROP}
|"show" {SHOW}
|"hide" {HIDE}
|"exists" {EXISTS}
|"character" {CHARACTER}
|"location" {LOCATION}
|"item" {ITEM}
|"int" {INT}
|"string" {STRING}
| ['0'-'9']+ as lit { LITERAL(int_of_string lit) }
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']* as var { VARIABLE ("_" ^ var) }
| eof { EOF }

and comment = parse
| "*/" {token lexbuf }
| _ {comment lexbuf}
```