Interprete del MiniCaml

Tipi di dato

Definizione di tipi per la sintassi astratta del linguaggio:

```
(* Identificatori *)
type ide = string;;

(* I tipi *)
type tname =
  | TInt
  | TBool
  | TString
  | TClosure
  | TRecClosure
  | TUnBound

(* Abstract Expressions = espressioni nella sintassi astratta,
compongono l'Albero di Sintassi Astratta *)
type exp =
  | EInt of int
  | CstTrue
  | CstFalse
  | EString of string
  | Den of ide
  (* Operatori binari da interi a interi *)
  | Sum of exp * exp
  | Diff of exp * exp
  | Prod of exp * exp
  | Div of exp * exp
  (* Operatori da interi a booleani *)
  | IsZero of exp
  | Eq of exp * exp
  | LessThan of exp * exp
  | GreaterThan of exp * exp
  (* Operatori su booleani *)
  | And of exp * exp
  | Or of exp * exp
  | Not of exp
  (* Controllo del flusso, funzioni *)
  | IfThenElse of exp * exp * exp
  | Let of ide * exp * exp
  | Letrec of ide * ide * exp * exp
  | Fun of ide * exp
  | Apply of exp * exp
```

Ambiente e valori esprimibili

L'implementazione più semplice è tramite coppia `(identificatore, valore)` e una funzione `lookup` che fornisce il valore associato all'identificatore

```
(* Ambiente polimorfo *)
type 't env = ide -> 't
```

```
type 't env = ide -> 't
```

Tipi esprimibili

```
(* Evaluation types = tipi esprimibili *)
type evT =
  | Int of int
  | Bool of bool
  | String of string
  | Closure of ide * exp * evT env
  | RecClosure of ide * ide * exp * evT env
  | UnBound
```

```
type evT =
  | Int of int
  | Bool of bool
  | String of string
  | Closure of ide * exp * evT env
  | RecClosure of ide * ide * exp * evT env
  | UnBound
```

Ambiente vuoto

```
(* Ambiente vuoto *)
let emptyenv = function x -> UnBound
```

```
val emptyenv : 'a -> evT = <fun>
```

Binding

```
(* Binding fra una stringa x e un valore primitivo evT *)
let bind (s: evT env) (x: ide) (v: evT) =
function (i: ide) -> if (i = x) then v else (s i)
```

```
val bind : evT env -> ide -> evT -> ide -> evT = <fun>
```

Typechecking

```
(* Funzione da evT a tname che associa a ogni valore il suo descrittore di
let getType (x: evT) : tname =
  match x with
  | Int(n) -> TInt
  | Bool(b) -> TBool
  | String(s) -> TString
  | Closure(i,e,en) -> TClosure
  | RecClosure(i,j,e,en) -> TRecClosure
  | UnBound -> TUnBound
```

```
val getType : evT -> tname = <fun>
```

```
(* Type-checking *)
let typecheck ((x, y) : (tname * evT)) =
  match x with
  | TInt -> (match y with
    | Int(u) -> true
    | _ -> false
    )
  | TBool -> (match y with
    | Bool(u) -> true
    | _ -> false
    )
  | TString -> (match y with
    | String(u) -> true
    | _ -> false
    )
  | TClosure -> (match y with
    | Closure(i, e, n) -> true
    | _ -> false
    )
  | TRecClosure -> (match y with
    | RecClosure(i, j, e, n) -> true
    | _ -> false
    )
  | TUnBound -> (match y with
    | UnBound -> true
    | _ -> false
    )
```

```
val typecheck : tname * evT -> bool = <fun>
```

Eccezioni

```
(* Errori a runtime *)
exception RuntimeError of string
```

```
exception RuntimeError of string
```

Operazioni primitive

```
(* PRIMITIVE del linguaggio *)

(* Controlla se un numero è zero *)
let is_zero(x) =
  match (typecheck(TInt,x), x) with
  | (true, Int(v)) -> Bool(v = 0)
  | (_, _) -> raise (RuntimeError "Wrong type")

(* Uguaglianza fra interi *)
let int_eq(x, y) =
  match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v = w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

(* Somma fra interi *)
let int_plus(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

(* Differenza fra interi *)
let int_sub(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v - w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

(* Prodotto fra interi *)
let int_times(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v * w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

(* Divisione fra interi *)
let int_div(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) ->
    if w <> 0 then Int(v / w)
    else raise (RuntimeError "Division by zero")
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")
```

```
(* Operazioni di confronto *)
let less_than(x, y) =
  match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v < w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

let greater_than(x, y) =
  match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v > w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

(* Operazioni logiche *)
let bool_and(x, y) =
  match (typecheck(TBool,x), typecheck(TBool,y), x, y) with
  | (true, true, Bool(v), Bool(w)) -> Bool(v && w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

let bool_or(x, y) =
  match (typecheck(TBool,x), typecheck(TBool,y), x, y) with
  | (true, true, Bool(v), Bool(w)) -> Bool(v || w)
  | (_, _, _, _) -> raise (RuntimeError "Wrong type")

let bool_not(x) =
  match (typecheck(TBool,x), x) with
  | (true, Bool(v)) -> Bool(not v)
  | (_, _) -> raise (RuntimeError "Wrong type")
```

```
val is_zero : evT -> evT = <fun>
val int_eq : evT * evT -> evT = <fun>
val int_plus : evT * evT -> evT = <fun>
val int_sub : evT * evT -> evT = <fun>
val int_times : evT * evT -> evT = <fun>
val int_div : evT * evT -> evT = <fun>
val less_than : evT * evT -> evT = <fun>
val greater_than : evT * evT -> evT = <fun>
val bool_and : evT * evT -> evT = <fun>
val bool_or : evT * evT -> evT = <fun>
val bool_not : evT -> evT = <fun>
```

Interprete

```
(* Interprete *)
let rec eval (e:exp) (s:evT env) : evT =
  match e with
  | EInt(n) -> Int(n)
  | CstTrue -> Bool(true)
  | CstFalse -> Bool(false)
  | EString(s) -> String(s)
  | Den(i) -> (s i)
  | Prod(e1,e2) -> int_times((eval e1 s), (eval e2 s))
  | Sum(e1, e2) -> int_plus((eval e1 s), (eval e2 s))
  | Diff(e1, e2) -> int_sub((eval e1 s), (eval e2 s))
  | Div(e1, e2) -> int_div((eval e1 s), (eval e2 s))
  | IsZero(e1) -> is_zero (eval e1 s)
  | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
  | LessThan(e1, e2) -> less_than((eval e1 s),(eval e2 s))
  | GreaterThan(e1, e2) -> greater_than((eval e1 s),(eval e2 s))
  | And(e1, e2) -> bool_and((eval e1 s),(eval e2 s))
  | Or(e1, e2) -> bool_or((eval e1 s),(eval e2 s))
  | Not(e1) -> bool_not(eval e1 s)
  | IfThenElse(e1,e2,e3) ->
    let g = eval e1 s in
    (match (typecheck(TBool,g),g) with
      |(true, Bool(true)) -> eval e2 s
      |(true, Bool(false)) -> eval e3 s
      |(_,_) -> raise (RuntimeError "Wrong type")
    )
  | Let(i, e, ebody) -> eval ebody (bind s i (eval e s))
  | Fun(arg, ebody) -> Closure(arg,ebody,s)
  | Letrec(f, arg, fBody, leBody) ->
    let benv = bind (s) (f) (RecClosure(f, arg, fBody,s)) in
    eval leBody benv
  | Apply(eF, eArg) ->
    let fclosure = eval eF s in
    (match fclosure with
      | Closure(arg, fbody, fDecEnv) ->
        let aVal = eval eArg s in
        let aenv = bind fDecEnv arg aVal in
        eval fbody aenv
      | RecClosure(f, arg, fbody, fDecEnv) ->
```

```
        let aVal = eval eArg s in
        let rEnv = bind fDecEnv f fclosure in
        let aenv = bind rEnv arg aVal in
        eval fbody aenv
      | _ -> raise (RuntimeError "Wrong type")
    )
```

```
val eval : exp -> evT env -> evT = <fun>
```