Curs 5

2020-2021

Fundamentele Limbajelor de Programare

Semantica limbajelor de programare

- Limbaj de programare: sintaxă și semantică
- Feluri de semantică
  - Limbaj natural — descriere textuală a efectelor
  - Operațională — asocierea unei demonstrații a execuției
  - Axiomatică — Descrierea folosind logică a efectelor unei instrucțiuni
  - Denotațională — prin asocierea unui obiect matematic (denotație)
  - Statică — Asocierea unui sistem de tipuri care exclude programe eronate

IMP: un limbaj IMPerativ foarte simplu

Ce conține

- Expresii
  - Aritmetice
  - Booleene
- Blocuri de instrucțiuni
  - De atribuire
  - Condiționale
  - De ciclare
  - De interacțiune I/O

```
var n = 0; read("n=", n);
var prime = true;
var i = 1;
while (prime && i * i < n) {
      i = i + 1;
      if (n % i == 0) prime = false
      else {}
};
if (prime) print("Is␣prime:␣", n)
else print("Is␣not␣prime:␣", n)
```

Sintaxă formală

$$
\begin{aligned}
E ::= {} & n \mid x \mid b \\
& \mid E \texttt{ + } E \mid E \texttt{ * } E \mid E \texttt{ / } E \\
& \mid E \texttt{ <= } E \mid E \texttt{ == } E \\
& \mid \texttt{! } E \mid E \texttt{ \&\& } E
\end{aligned}
$$

$$
\begin{aligned}
C ::= {} & \texttt{var } x \texttt{ = } E \mid x \texttt{ = } E \\
& \mid \texttt{if (} B \texttt{) } C \texttt{ else } C \\
& \mid \texttt{while (} B \texttt{) } C \\
& \mid \texttt{read (} str \texttt{ , } x \texttt{) } \mid \texttt{print (} str \texttt{ , } E \texttt{)} \\
& \mid \texttt{\{ } \{C \texttt{ ;}\}^* \texttt{ \}}
\end{aligned}
$$

$$
P ::= \{C \texttt{ ;}\}^*
$$

unde n reprezintă numere întregi, b constante de adevăr, x identificatori și str șiruri de caractere.

Semantică statică

Semantică Statică - Motivație

- Este sintaxa unui limbaj de programare prea expresivă?
- Sunt programe care n-aș vrea să le pot scrie, dar le pot?

- Putem detecta programe greșite înainte de rulare?
- Putem garanta că execuția programului nu se va bloca?

Semantică Statică - Motivație

- Este sintaxa unui limbaj de programare prea expresivă?
- Sunt programe care n-aș vrea să le pot scrie, dar le pot?
  - Pot aduna întregi cu Booleeni
  - 2 <= 3 <= 5

- Putem detecta programe greșite înainte de rulare?
- Putem garanta că execuția programului nu se va bloca?
  - folosirea variabilelor fără a le declara
  - tipuri incompatibile (variabile, dar și expresii)

Semantică Statică - Motivație

- Este sintaxa unui limbaj de programare prea expresivă?
- Sunt programe care n-aș vrea să le pot scrie, dar le pot?
  - Pot aduna întregi cu Booleeni
  - 2 <= 3 <= 5
- Putem detecta programe greșite înainte de rulare?
- Putem garanta că execuția programului nu se va bloca?
  - folosirea variabilelor fără a le declara
  - tipuri incompatibile (variabile, dar și expresii)
- Soluție: Sistemele de tipuri

Sisteme de tipuri

- Descriu programele „bine formate“
- Pot preveni anumite erori
  - folosirea variabilelor nedeclarate/neinițializate
  - detectarea unor bucați de cod inaccesibile
  - erori de securitate
- Ajută compilatorul
- Pot influența proiectarea limbajului

Scop (ideal)

Progamele „bine formate“, i. e., cărora li se poate asocia un tip nu eșuează

Reguli intuitive pentru IMP

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimba tipul în timpul execuției
- operanzii asociați unui operator într-o expresie trebuie să se evalueze la valori corespunzătoare tipurilor așteptate de operator
- Condițiile din if și while se evaluează la valori Booleene
- Se fac operații I/O doar cu valori de tip întreg

Sisteme de tipuri

- Vom defini o relație $\Gamma \vdash \mathit{frag} : T$
- Citim *frag* are tipul T dacă Γ, unde
- Γ — tipuri asociate variabilelor din e

Exemple

$$\begin{array}{lcll} & \vdash & \texttt{if}\ \mathit{true}\ \texttt{then}\ \{\}\ \texttt{else}\ \{\} : \texttt{stmt} & \\ x : \texttt{int} & \vdash & x + 13 : \texttt{int} & \\ x : \texttt{int} & \nvdash & x = y + 1 : T & \text{pentru orice } T \end{array}$$

Tipuri în limbajul IMP

Tipurile expresiilor = tipurile gramaticale

$$T ::= \texttt{int} \mid \texttt{bool} \mid \texttt{stmt}$$

Γ — Mediul de tipuri

- Asociază tipuri variabilelor
- Notație: o listă de perechi locație-tip $x_1 : t_1, \ldots, x_n : t_n$

Observații pentru limbajul IMP

- variabile din Γ au tipul fie **`int`** fie **`bool`**
- Apariția unei variabile în Γ înseamnă că variabila a fost declarată

IMP: Reguli formale pentru tipuri

Tipul constantelor

$$(\textsc{int}) \quad \Gamma \vdash n : \mathbf{int} \quad \text{dacă } n \in \mathbb{Z}$$

$$(\textsc{bool}) \quad \Gamma \vdash b : \mathbf{bool} \quad \text{dacă } b \in \{\mathit{true}, \mathit{false}\}$$

IMP: Reguli formale pentru tipuri

Tipul constantelor

$$(\textsc{int}) \quad \Gamma \vdash n : \mathbf{int} \quad \text{dacă } n \in \mathbb{Z}$$

$$(\textsc{bool}) \quad \Gamma \vdash b : \mathbf{bool} \quad \text{dacă } b \in \{\mathit{true}, \mathit{false}\}$$

Tipul operatorilor

Operanzii asociați unui operator într-o expresie trebuie să se evalueze la valori corespunzătoare tipurilor așteptate de operator

$$(\textsc{op+}) \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

IMP: Reguli formale pentru tipuri

Tipul constantelor

$$(\textsc{int}) \quad \Gamma \vdash n : \mathbf{int} \quad \text{dacă } n \in \mathbb{Z}$$

$$(\textsc{bool}) \quad \Gamma \vdash b : \mathbf{bool} \quad \text{dacă } b \in \{true, false\}$$

Tipul operatorilor

Operanzii asociați unui operator într-o expresie trebuie să se evalueze la valori corespunzătoare tipurilor așteptate de operator

$$(\textsc{op}+) \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

$$(\textsc{op}\leq) \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \texttt{<=} e_2 : \mathbf{bool}}$$

$$(\textsc{op}!) \quad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}}$$

IMP: Reguli formale pentru tipuri

Tipul variabilelor și al declarațiilor

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimba tipul în timpul execuției

IMP: Reguli formale pentru tipuri

Tipul variabilelor și al declarațiilor

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimba tipul în timpul execuției

(LOC) $\Gamma \vdash x : t \quad \text{dacă } \Gamma(x) = t$

(DECL) $$\frac{\Gamma \vdash e : t \quad \Gamma, x \mapsto t \vdash sts : stmt}{\Gamma \vdash \mathbf{var}\ x = e; sts : stmt}$$

IMP: Reguli formale pentru tipuri

Tipul variabilelor și al declarațiilor

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimba tipul în timpul execuției

(LOC) $\Gamma \vdash x : t \quad \text{dacă } \Gamma(x) = t$

(DECL) $$\frac{\Gamma \vdash e : t \quad \Gamma, x \mapsto t \vdash sts : stmt}{\Gamma \vdash \mathbf{var}\ x = e; sts : stmt}$$

Instrucțiuni: Atribuire și secvențiere

- Variabilele nu își schimba tipul în timpul execuției

(ATRIB) $$\frac{\Gamma \vdash e : t}{\Gamma \vdash x = e : \mathbf{stmt}} \quad \text{dacă } \Gamma(x) = t$$

IMP: Reguli formale pentru tipuri

Tipul variabilelor și al declarațiilor

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimba tipul în timpul execuției

(LOC) $\Gamma \vdash x : t \quad \text{dacă } \Gamma(x) = t$

(DECL) $$\frac{\Gamma \vdash e : t \quad \Gamma, x \mapsto t \vdash sts : stmt}{\Gamma \vdash \mathbf{var}\ x = e; sts : stmt}$$

Instrucțiuni: Atribuire și secvențiere

- Variabilele nu își schimba tipul în timpul execuției

(ATRIB) $$\frac{\Gamma \vdash e : t}{\Gamma \vdash x = e : \mathbf{stmt}} \quad \text{dacă } \Gamma(x) = t$$

(SEQ) $$\frac{\Gamma \vdash st : \mathbf{stmt} \quad \Gamma \vdash sts : stmt}{\Gamma \vdash st; sts : \mathbf{stmt}} \quad \text{dacă } st \text{ nu e declaratie}$$

IMP: Reguli formale pentru tipuri

Tipuri pentru instrucțiuni

- Condițiile din if și while se evaluează la valori Booleene

$$(\textsc{if})\quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash t : \mathbf{stmt} \quad \Gamma \vdash e : \mathbf{stmt}}{\Gamma \vdash \mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ e : \mathbf{stmt}}$$

IMP: Reguli formale pentru tipuri

Tipuri pentru instrucțiuni

- Condițiile din if și while se evaluează la valori Booleene

$$(\textsc{if})\quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash t : \mathbf{stmt} \quad \Gamma \vdash e : \mathbf{stmt}}{\Gamma \vdash \mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ e : \mathbf{stmt}}$$

$$(\textsc{while})\quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash b : \mathbf{stmt}}{\Gamma \vdash \mathbf{while}\ (c)\ b : \mathbf{stmt}}$$

IMP: Reguli formale pentru tipuri

Tipuri pentru instrucțiuni

- Condițiile din if și while se evaluează la valori Booleene

$$(\textsc{if})\quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash t : \mathbf{stmt} \quad \Gamma \vdash e : \mathbf{stmt}}{\Gamma \vdash \mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ e : \mathbf{stmt}}$$

$$(\textsc{while})\quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash b : \mathbf{stmt}}{\Gamma \vdash \mathbf{while}\ (c)\ b : \mathbf{stmt}}$$

- Se fac operații I/O doar cu valori de tip întreg

$$(\textsc{read})\quad \Gamma \vdash \mathbf{read}\ (s, x) : \mathbf{stmt} \quad \textit{dacă}\ \Gamma(x) = \mathbf{int}$$

$$(\textsc{print})\quad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \mathbf{print}\ (s, e) : \mathbf{stmt}}$$

Haskell: limbajul SIMPLE

Limbajul SIMPLE

SIMPLE - exemplu

```
pFact= Block [
       Asgn "n" (I 5),
       Asgn "fact " (Id "n"),
       Asgn "i" (I 1),
       While (BinE Neq  (Id "n") (Id "i"))
             (Block [ Asgn "fact" (BinA Mul (Id "
                 fact") (Id "i")),
                    Asgn "i" (BinA Add (Id "i") (I
                        1))
                    ])
             ]

*SIMPLE> pFact
Block [Asgn "n" 5,Asgn "fact " n,Asgn "i" 1,While n!=
    i (Block [Asgn "fact" (fact*i),Asgn "i" (i+1)])]
```

Limbajul SIMPLE - operatori

```
type Name = String

data BinAop = Add | Mul | Sub | Div | Mod

data BinCop = Lt | Lte | Gt | Gte

data BinEop = Eq | Neq

data BinLop = And | Or
```

Limbajul SIMPLE - expresii

```
data Exp
     = Id Name
     | I Integer
     | B Bool
     | UMin Exp
     | BinA BinAop Exp Exp
     | BinC BinCop Exp Exp
     | BinE BinEop Exp Exp
     | BinL BinLop Exp Exp
     | Not Exp
```

Limbajul SIMPLE - instrucțiuni

```
data Stmt
     = Asgn Name Exp

     | If Exp Stmt Stmt

     | While Exp Stmt

     | Block [Stmt]

     | Decl Name Exp
```

SIMPLE: operatori binari

- operatori aritmetici

```
data BinAop = Add | Mul

instance Show BinAop where
    show Add = "+"
    show Mul = "*"
```

SIMPLE: operatori binari

- operatori aritmetici

```
data BinAop = Add | Mul

instance Show BinAop where
    show Add = "+"
    show Mul = "*"
```

- operatori logici

```
data BinLop = And | Or

instance Show BinLop where
    show And = "&&"
    show Or = "||"
```

SIMPLE: relații

- relații de ordine

```
data BinCop = Lte | Gte

instance Show BinCop where
    show Lte = "<="
    show Gte = ">="
```

SIMPLE: relații

- relații de ordine

```
data BinCop = Lte | Gte

instance Show BinCop where
    show Lte = "<="
    show Gte = ">="
```

- relația de egalitate

```
data BinEop = Eq | Neq

instance Show BinEop where
    show Eq = "=="
    show Neq = "!="
```

SIMPLE: expresii simple

```
data Exp
     = Id Name
     | I Integer
     | B Bool
```

SIMPLE: expresii simple

```
data Exp
     = Id Name
     | I Integer
     | B Bool

instance Show Exp where
     show (Id x) = x
     show (I i) = show i
     show (B True) = "true"
     show (B False) = "false"
```

SIMPLE: expresii (complet)

```
data Exp
     = Id Name
     | I Integer
     | B Bool
     | UMin Exp
     | BinA BinAop Exp Exp
     | BinC BinCop Exp Exp
     | BinE BinEop Exp Exp
     | BinL BinLop Exp Exp
     | Not Exp
```

SIMPLE: expresii (complet)

```
instance Show Exp where
    show (Id x) = x
    show (I i) = show i
    show (B True) = "true"
    show (B False) = "false"
    show (UMin e) = "-" ++ show e
    show (BinA op e1 e2) = addParens $ show e1 ++
        show op ++ show e2
    show (BinL op e1 e2) = addParens $ show e1 ++
        show op ++ show e2
    show (BinC op e1 e2) = show e1 ++ show op ++ show
        e2
    show (BinE op e1 e2) = show e1 ++ show op ++ show
        e2
    show (Not e) = "!" ++ show e

addParens :: String -> String
addParens e = "(" ++ e ++ ")"
```

Exemplu: SIMPLE expresii

```
> :t BinC Lte  (Id "prime") (BinA Add (I 2) (Id "x"))
BinC Lte  (Id "prime") (BinA Add (I 2) (Id "x")) ::
    Exp

> :t BinL Lte  (Id "prime") (BinA Add (I 2) (Id "x"))
error: ...

> BinC Lte  (Id "prime") (BinA Add (I 2) (Id "x"))
prime<=(2+x)
```

Exemplu: SIMPLE expresii

Observați că:

```
> :t BinC Lte (B True) (BinA Add (I 2) (Id "x"))
BinC Lte (B True) (BinA Add (I 2) (Id "x")) :: Exp

> BinC Lte (B True) (BinA Add (I 2) (Id "x"))
true<=(2+x)
```

Exemplu: SIMPLE expresii

Observați că:

```
> :t BinC Lte (B True) (BinA Add (I 2) (Id "x"))
BinC Lte (B True) (BinA Add (I 2) (Id "x")) :: Exp

> BinC Lte (B True) (BinA Add (I 2) (Id "x"))
true<=(2+x)
```

Expresia de mai sus este **coerectă sintactic** dar greșită din punctul de vedere al verificării tipurilor.

SIMPLE: instrucțiuni

```
data Stmt
        =   Asgn Name Exp
          | If Exp Stmt Stmt
          | While Exp Stmt
          | Block [Stmt]
          | Decl Name Exp
  deriving (Show)
```

Exemplu: SIMPLE program

```
pFact= Block [
       Asgn "n" (I 5),
       Asgn "fact " (Id "n"),
       Asgn "i" (I 1),
       While (BinE Neq  (Id "n") (Id "i"))
             (Block [ Asgn "fact" (BinA Mul (Id "
                 fact") (Id "i")),
                      Asgn "i" (BinA Add (Id "i") (I
                           1))
                      ])
             ]
```

Exemplu: SIMPLE program

```
pFact= Block [
       Asgn "n" (I 5),
       Asgn "fact " (Id "n"),
       Asgn "i" (I 1),
       While (BinE Neq  (Id "n") (Id "i"))
             (Block [ Asgn "fact" (BinA Mul (Id "
                 fact") (Id "i")),
                      Asgn "i" (BinA Add (Id "i") (I
                          1))
                      ])
             ]

*SIMPLE> :t pFact
pFact :: Stmt
```

Exemplu: SIMPLE program

```
pFact= Block [
       Decl "n" (I 5),
       Decl "fact " (Id "n"),
       Decl "i" (I 1),
       While (BinE Neq  (Id "n") (Id "i"))
                (Block [ Asgn "fact" (BinA Mul (Id "
                   fact") (Id "i")),
                       Asgn "i" (BinA Add (Id "i") (I
                            1))
                       ])
              ]
```

Exemplu: SIMPLE program

```
pFact= Block [
       Decl "n" (I 5),
       Decl "fact " (Id "n"),
       Decl "i" (I 1),
       While (BinE Neq  (Id "n") (Id "i"))
             (Block [ Asgn "fact" (BinA Mul (Id "
                fact") (Id "i")),
                    Asgn "i" (BinA Add (Id "i") (I
                         1))
                    ])
             ]

*SIMPLE> pFact
Block [Asgn "n" 5,Asgn "fact " n,Asgn "i" 1,
While n!=i (Block [Asgn "fact" (fact*i),
Asgn "i" (i+1)])]
```

SIMPLE: verificarea sistemului de tipuri

SIMPLE type checker

- vom folosi o "stare" in care fiecare variabila are ascociat un tip; "stările" sunt definite folosind Data.Map

```
import Data.Map.Strict as Map
type CheckerState = Map Name Type

emptyCSt :: CheckerState -- "starea" vida
emptyCSt = Map.empty
```

SIMPLE type checker

- vom folosi o "stare" in care fiecare variabila are ascociat un tip; "stările" sunt definite folosind Data.Map

```
import Data.Map.Strict as Map
type CheckerState = Map Name Type

emptyCSt :: CheckerState -- "starea" vida
emptyCSt = Map.empty
```

- funcția de verificare va asocia unei construcții sintactice o valoare M Type, unde M este o monadă iar Type este un tip:

```
data Type = TInt | TBool

checkExp :: Exp -> M Type
```

SIMPLE type checker

- vom folosi o "stare" in care fiecare variabila are ascociat un tip; "stările" sunt definite folosind Data.Map

```
import Data.Map.Strict as Map
type CheckerState = Map Name Type

emptyCSt :: CheckerState -- "starea" vida
emptyCSt = Map.empty
```

- funcția de verificare va asocia unei construcții sintactice o valoare M Type, unde M este o monadă iar Type este un tip:

```
data Type = TInt | TBool

checkExp :: Exp -> M Type
```

Tipul "unit" () va fi tipul instrucțiunilor:

SIMPLE type checker

- vom folosi o "stare" in care fiecare variabila are ascociat un tip; "stările" sunt definite folosind Data.Map

```
import Data.Map.Strict as Map
type CheckerState = Map Name Type

emptyCSt :: CheckerState -- "starea" vida
emptyCSt = Map.empty
```

- funcția de verificare va asocia unei construcții sintactice o valoare M Type, unde M este o monadă iar Type este un tip:

```
data Type = TInt | TBool

checkExp :: Exp -> M Type
```

Tipul "unit" () va fi tipul instrucțiunilor: unde

```
checkStmt :: Stmt -> M ()
checkBlock :: [Stmt] -> M ()
```

SIMPLE type checker: monada

- Monada M este o combnație între monada Reader și monada **Either**

```
newtype EReader a =
    EReader {runEReader :: CheckerState ->(Either
        String a)}

instance Monad EReader where
     return a = EReader (\env ->  Right a)
     act >>= k = EReader  f
                 where
                  f env  = case (runEReader act env)
                      of
                            Left s -> Left s
                            Right va -> runEReader (k
                                va) env


type M = EReader
```

SIMPLE type checker: monada

- Monada EReader este o combnație între monada Reader și monada **Either**

```
newtype EReader a =
    EReader {runEReader :: CheckerState ->(Either
        String a)}

askEReader :: EReader CheckerState
askEReader =EReader (\env -> Right env)

localEReader ::(CheckerState ->CheckerState) ->
    EReader a -> EReader a
localEReader f ma = EReader (\env -> (runEReader ma)
    (f env))
```

SIMPLE type checker: funcții auxiliare

```
throwError :: String -> EReader a
throwError e = EReader (\_ -> (Left e))


expect ::(Show t,Eq t,Show e)=> t -> t -> e -> M ()
expect tExpect tActual e =
    if (tExpect /= tActual)
    then      (throwError
        $ "Type mismatch. Expected " ++ show tExpect
            ++ " but got " ++ show tActual ++ " for "
            ++ show e)
    else (return ())
```

SIMPLE type checker: **lookup**

```
data Type = TInt | TBool
  deriving (Eq)

type CheckerState = Map Name Type
type M = EReader

lookupM :: Name -> M Type
lookupM x = do
    env <- askEReader
    case (Map.lookup x env) of
        Nothing -> throwError $ "Variable " <> x <> "
            not declared"
        Just t -> return t
```

SIMPLE type checker: structura generală

```
checkExp :: Exp -> M Type

checkStmt :: Stmt -> M ()

checkBlock :: [Stmt] -> M ()

checkPgm :: [Stmt] -> Bool
checkPgm pgm =
 case (runEReader (checkBlock pgm)) emptyCSt of
        Left err -> error err
        Right _ -> True
```

SIMPLE type checker: structura generală

```
checkExp :: Exp -> M Type

checkStmt :: Stmt -> M ()

checkBlock :: [Stmt] -> M ()

checkPgm :: [Stmt] -> Bool
checkPgm pgm =
 case (runEReader (checkBlock pgm)) emptyCSt of
        Left err -> error err
        Right _ -> True

*Checker> checkPgm [pFact]
True
```

SIMPLE type checker: structura generală

```
checkExp :: Exp -> M Type

checkStmt :: Stmt -> M ()

checkBlock :: [Stmt] -> M ()

checkPgm :: [Stmt] -> Bool
checkPgm pgm =
 case (runEReader (checkBlock pgm)) emptyCSt of
        Left err -> error err
        Right _ -> True
```

SIMPLE type checker: structura generală

```
checkExp :: Exp -> M Type

checkStmt :: Stmt -> M ()

checkBlock :: [Stmt] -> M ()

checkPgm :: [Stmt] -> Bool
checkPgm pgm =
 case  (runEReader (checkBlock pgm)) emptyCSt of
        Left err -> error err
        Right _ -> True

*Checker> runEReader (checkExp (BinC Lte  (B True)
 (BinA Add (I 2) (Id "x")))) emptyCSt
Left "Type mismatch. Expected int but got bool for
    true"
```

SIMPLE type checker: verficarea blocurilor

```
checkBlock :: [Stmt] -> M ()

checkBlock [] = return ()

checkBlock (Decl x e : ss) = do
    t <- checkExp e
    localEReader (Map.insert x t) (checkBlock ss)

checkBlock (s : ss) = checkStmt s >> checkBlock ss
```

SIMPLE type checker: verificarea instrucțiunilor

```
checkStmt :: Stmt -> M ()
checkStmt (Decl _ _) = return ()
checkStmt (Block ss) = checkBlock ss
```

SIMPLE type checker: verificarea instrucțiunilor

```
checkStmt :: Stmt -> M ()
checkStmt (Decl _ _) = return ()
checkStmt (Block ss) = checkBlock ss

checkStmt (Asgn x e) = do
    tx <- lookupM x
    te <- checkExp e
    expect tx te e

checkStmt (If e s1 s2) = do
    te <- checkExp e
    expect TBool  te e
    checkStmt s1
    checkStmt s2

checkStmt (While e s) = ...
```

SIMPLE type checker: verificarea expresiilor

```
checkExp :: Exp -> M Type
checkExp (I _) = return TInt
checkExp (B _) = return TBool
checkExp (Id x) = lookupM x
```

SIMPLE type checker: verificarea expresiilor

```
checkExp :: Exp -> M Type
checkExp (I _) = return TInt
checkExp (B _) = return TBool
checkExp (Id x) = lookupM x

checkExp (BinA _ e1 e2) = do
    t1 <- checkExp e1
    expect TInt t1 e1
    t2 <- checkExp e2
    expect TInt t2 e2
    return TInt

checkExp (Not e) = do
    t <- checkExp e
    expect TBool t e
    return TBool
...
```

Pe săptămâna viitoare!