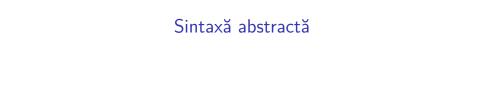


Descriere generală a problemei

Pornind de la limbajul SIMPLE prezentat in curs, să se implementeze un interpretor pentru acesta, integrat cu parser-ul și type-checker-ul implementate în laboratoarele precedente.



Expresii

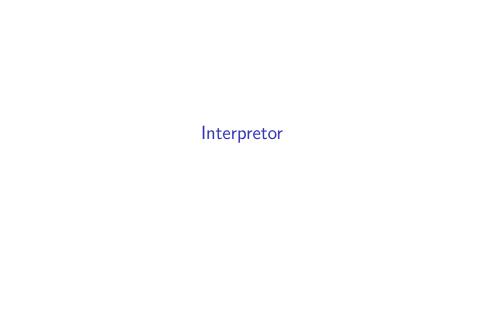
```
type Name = String
data BinAop = Add | Mul | Sub | Div | Mod
data BinCop = Lt | Lte | Gt | Gte
data BinEop = Eq | Neq
data BinLop = And | Or
data Exp
   = Td Name
    I Integer
    I B Bool
    UMin Exp
    BinA BinAop Exp Exp
    BinC BinCop Exp Exp
    | BinE BinEop Exp Exp
     BinL BinLop Exp Exp
     Not Exp
```

Instrucțiuni

data Stmt

- = Asgn Name Exp
- | If Exp Stmt Stmt
- | Read String Name
- | Print String Exp
- | While Exp Stmt
- | Block [Stmt]
- | Decl Name Exp

deriving (Show)



Valori

Avem valori de tip intreg și de tip Boolean:

```
data Value = IVal Integer | BVal Bool
  deriving (Show, Eq)
```

Stare (Envioronment și store)

Deoarece avem blocuri și variabile locale, este posibil ca un nume de variabilă să indice locații diferite de memorie în funcție de contextul (blocul) în care apare. De aceea, vom exprima starea programului folosind două map-uri: store, care reprezintă memoria efectivă, asociind valori locatiilor de memorie; și env, care asociază variabilelor vizibile în contextul curent locațiile lor în memorie. În plus, pentru a ști cu ușurință care este următoarea locație disponibilă când alocăm memorie, vom memora ultima locatie alocată în nextLoc.

```
data ImpState = ImpState
    { env :: Map String Int
    , store :: Map Int Value
    , nextLoc :: Int
    }
    deriving (Show)
emptyState :: ImpState
emptyState = ImpState Map.empty Map.empty 0
```

Monada interpretorului

Deoarce interpretorul nostru va avea ca efecte laterale atât menținerea acestei stări cât și interacțiunea I/O cu consola, vom folosi o monadă care combină aceste efecte.

```
runM :: M a -> IO (a, ImpState)
runM m = runStateT m emptyState
```

În această monadă, operațiile corespunzătoare monadei State pot fi realizate la fel ca în monada State, iar cele de tip I/O pot fi realizate folosind comanda liftIO :: IO a -> M a.

Citirea valorii curente a unui identificator

- obținem din env locația curentă 1 asociată lui x
- obținem din store valoarea stocată pentru locatia 1

```
lookupM :: String -> M Value
lookupM x = do
   Just l <- Map.lookup x <$> gets env
   Just v <- Map.lookup l <$> gets store
   return v
```

Scrierea valorii curente pentru un identificator

- obţinem din env locaţia curentă 1 asociată lui x
- actualizăm în store valoarea stocată pentru locatia 1 la valoarea v
- actualizăm starea pentru a face vizibilă modificarea ei în viitor.

```
updateM :: String -> Value -> M ()
updateM x v = do
   Just l <- Map.lookup x <$> gets env
   st <- gets store
   let st' = Map.insert l v st
   modify' (\s -> s {store = st'})
```

Exercițiu

În fișierul lab7.hs găsiți o implementare parțială a interpretorului pentru limbajul SIMPLE. Completați această implementare pentru a putea rula programele cu extensia .imp.

Următoarele slide-uri oferă explicații pentru părțile deja implementate.

evalExp

```
\mathtt{evalExp} \; :: \; \mathtt{Exp} \; \mathord{\hspace{1pt}\text{--}\hspace{1pt}} \mathsf{M} \; \, \mathtt{Value}
```

Identificatori

Pentru obținerea valorii unui identificator folosim lookupM.

```
evalExp (Id x) = lookupM x
```

evalExp — Operatori binari (de comparație)

Pentru evaluarea operatorilor binari - evaluam expresiile, așteptând rezultate de tipurile potrivite - fapt garantat de type checker - aplicam operația potrivită acelor valori.

```
evalExp (BinC op e1 e2) = do
    IVal i1 <- evalExp e1
    IVal i2 <- evalExp e2
    return (BVal $ cop op i1 i2)

cop :: BinCop -> Integer -> Integer -> Bool
cop Lt = (<)
cop _ = undefined</pre>
```

evalStmt

```
evalStmt :: Stmt -> M ()
```

Atribuire

Evaluarea atribuirii se face prin

- evaluarea expresiei ce se atribuie folosins evalExp
- actualizarea valorii pentru variabila căreia i se atribuie folosind updateM.

```
evalStmt (Asgn x e) = do
  v <- evalExp e
  updateM x v</pre>
```

Citire de la tastatură

- ► Folosim liftIO pentru a executa secvența de acțiuni I/O
 - putStr (pentru a afișa prompterul de citire),
 - ▶ hFlush stdout pentru a asigura că mesajul a fost scris,
 - readLn pentru a citi valoarea i.
- Executăm o operație de atribuire pentru a atribui valoarea nou citită variabilei în care trebuia ea citită.

```
evalStmt (Read s x) = do
   i <- liftIO (putStr s >> hFlush stdout >> readLn)
   evalStmt(Asgn x (I i))
```

Declararea unei variabile

- evaluam expresia de inițializare la o valoare v
- stocam valorii v la locatia nextLoc în store
- setam numele x către locația nextLoc în env
- creștem cu o unitate a locației nextLoc.

Observați folosirea modify' pentru a modifica conținutul stării.

```
evalStmt (Decl x e) = do
    v <- evalExp e
    modify' (declare v)
  where
    declare v st = ImpState env' store' nextLoc'
      where
        1 = nextLoc st
        nextLoc' = 1 + nextLoc st
        store' = Map.insert l v (store st)
        env' = Map.insert x l (env st)
```

Blocuri

Evaluarea unui bloc se face astfel:

- ▶ se salvează continutul existent al env în oldEnv
- ▶ se execută (în ordine) instrucțiunile din bloc
- se modifică starea pentru a restaura env la oldEnv

```
evalStmt (Block sts) = do
  oldEnv <- gets env
  mapM_ evalStmt sts
  modify' (\s -> s {env = oldEnv})
```

Exercitiul 2

Adăugați o expresie de forma ++i la limbaj (parser, type-checker, interpetor).