Programare declarativă

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță - seria 33
Ioana Leuștean - seria 34

Departamentul de Informatică, FMI, UB
traian.serbanuta@fmi.unibuc.ro
ioana@fmi.unibuc.ro

Transparență Referențială

Transparența Referențială

Transparența referențială este proprietatea expresiilor de a fi înlocuite cu valoarea rezultată, fără a schimba semantica programului.

Transparența Referențială

Transparența referențială este proprietatea expresiilor de a fi înlocuite cu valoarea rezultată, fără a schimba semantica programului.

```
zece :: Integer
zece = 10

plusZece :: Integer -> Integer
plusZece n = n + zece
--                  ^^ poate fi inlocuit cu 10
```

Transparența Referențială

Transparența referențială este proprietatea expresiilor de a fi înlocuite cu valoarea rezultată, fără a schimba semantica programului.

```
zece :: Integer
zece = 10

plusZece :: Integer -> Integer
plusZece n = n + zece
--                  ^^ poate fi inlocuit cu 10

imparePlusZece :: Integer -> [Integer]
imparePlusZece n = map plusZece (filter odd [1 .. n])
```

Transparența Referențială

Transparența referențială este proprietatea expresiilor de a fi înlocuite cu valoarea rezultată, fără a schimba semantica programului.

```
zece :: Integer
zece = 10

plusZece :: Integer -> Integer
plusZece n = n + zece
--                  ^^ poate fi inlocuit cu 10

imparePlusZece :: Integer -> [Integer]
imparePlusZece n = map plusZece (filter odd [1 .. n])
--                             ^^                ^^
--                pot fi inlocuite cu definitiile lor
```

Transparența Referențială

Transparența referențială este proprietatea expresiilor de a fi înlocuite cu valoarea rezultată, fără a schimba semantica programului.

```
zece :: Integer
zece = 10

plusZece :: Integer -> Integer
plusZece n = n + zece
--                  ^^ poate fi inlocuit cu 10

imparePlusZece :: Integer -> [Integer]
imparePlusZece n = map plusZece (filter odd [1 .. n])
--                             ^^                ^^
--                pot fi inlocuite cu definitiile lor
```

Fiecare din termenii de mai sus pot fi înlocuiți cu rezultatul lor, iar programele vor fi neschimbate.

Avantajele Transparenței Referențiale

În general, transparența referențială prezintă o serie de avantaje:

- simplificarea algoritmilor (Haskell, Idris)
- evaluare lenesă (Haskell)
- optimizări prin memoizare (Haskell)
- optimizări prin eliminarea subexpresiilor comune (Haskell)
- demonstrarea corectitudinii unui program (Coq, Agda, Idris)
- asistarea programatorlui în refactorizarea programelor (PureScript)

Efecte secundare

Cum putem implementa funcția `showPlusZece`, care primește un argument, îl afișează, apoi întoarce rezultatul adunării argumentului cu zece?

Efecte secundare

Cum putem implementa funcția `showPlusZece`, care primește un argument, îl afișează, apoi întoarce rezultatul adunării argumentului cu zece?

Să presupunem că ar exista o funcție:

```
showInteger :: Integer -> Integer
```

care afișează numărul primit ca parametru, apoi îl întoarce.

Efecte secundare

Cum putem implementa funcția `showPlusZece`, care primește un argument, îl afișează, apoi întoarce rezultatul adunării argumentului cu zece?

Să presupunem că ar exista o funcție:

```
showInteger :: Integer -> Integer
```

care afișează numărul primit ca parametru, apoi îl întoarce.
Atunci putem scrie funcția `showPlusZece`:

```
showPlusZece :: Integer -> Integer
showPlusZece n = showInteger n + 10
```

Efecte secundare

Cum putem implementa funcția `showPlusZece`, care primește un argument, îl afișează, apoi întoarce rezultatul adunării argumentului cu zece?

Să presupunem că ar exista o funcție:

```
showInteger :: Integer -> Integer
```

care afișează numărul primit ca parametru, apoi îl întoarce.
Atunci putem scrie funcția `showPlusZece`:

```
showPlusZece :: Integer -> Integer
showPlusZece n = showInteger n + 10
```

Se poate înlocui funcția `showPlusZece` cu rezultatul acesteia?

Efecte secundare

Cum putem implementa funcția `showPlusZece`, care primește un argument, îl afișează, apoi întoarce rezultatul adunării argumentului cu zece?

Să presupunem că ar exista o funcție:

```
showInteger :: Integer -> Integer
```

care afișează numărul primit ca parametru, apoi îl întoarce.
Atunci putem scrie funcția `showPlusZece`:

```
showPlusZece :: Integer -> Integer
showPlusZece n = showInteger n + 10
```

Se poate înlocui funcția `showPlusZece` cu rezultatul acesteia?

Răspunsul este NU. Am obține același rezultat, dar am pierde efectul secundar de afișare al numărului.

Modelarea efectelor secundare

Problema este că tipul de ieșire nu este suficient de descriptiv. **Integer** poate reprezenta toate numerele întregi, dar nu poate reprezenta existența efectelor secundare. Merită menționat că am vrea să putem afișa multe alte tipuri de date, nu doar numere întregi.

Modelarea efectelor secundare

Problema este că tipul de ieșire nu este suficient de descriptiv. **Integer** poate reprezenta toate numerele întregi, dar nu poate reprezenta existența efectelor secundare. Merită menționat că am vrea să putem afișa multe alte tipuri de date, nu doar numere întregi.

Este necesar un tip de date care să modeleze efectele secundare:

data **IO** a

Modelarea efectelor secundare

Problema este că tipul de ieșire nu este suficient de descriptiv. **Integer** poate reprezenta toate numerele întregi, dar nu poate reprezenta existența efectelor secundare. Merită menționat că am vrea să putem afișa multe alte tipuri de date, nu doar numere întregi.

Este necesar un tip de date care să modeleze efectele secundare:

```
data IO a
```

Avem nevoie și de o funcție care afișează un șir de caractere. Această funcție nu are o valoare pe care o poate întoarce, așa că ea va întoarce `()`, tipul *unitate*:

```
data () = ()

putStr :: String -> IO ()
```

IO

IO

Pentru că vrem să întoarcem o valoare de tip întreg, dar pentru că avem și un efect secundar, tipul pe care îl întoarce `showPlusZece` trebuie să fie **`IO Integer`**:

```
showPlusZece :: Integer -> IO Integer
showPlusZece n = putStr (show n) ?? n + 10
```

IO

Pentru că vrem să întoarcem o valoare de tip întreg, dar pentru că avem și un efect secundar, tipul pe care îl întoarce `showPlusZece` trebuie să fie **`IO Integer`**:

```
showPlusZece :: Integer -> IO Integer
showPlusZece n = putStr (show n) ?? n + 10
-- Daca stim ca
putStr (show n) :: IO ()
n + 10          :: Integer
-- Atunci ce tip are (??):
(??) ::
```

IO

Pentru că vrem să întoarcem o valoare de tip întreg, dar pentru că avem și un efect secundar, tipul pe care îl întoarce `showPlusZece` trebuie să fie **`IO Integer`**:

```
showPlusZece :: Integer -> IO Integer
showPlusZece n = putStr (show n) ?? n + 10
-- Daca stim ca
putStr (show n) :: IO ()
n + 10          :: Integer
-- Atunci ce tip are (??):
(??) :: IO () -> Integer -> IO Integer
```

IO

Pentru că vrem să întoarcem o valoare de tip întreg, dar pentru că avem și un efect secundar, tipul pe care îl întoarce `showPlusZece` trebuie să fie **`IO Integer`**:

```
showPlusZece :: Integer -> IO Integer
showPlusZece n = putStr (show n) ?? n + 10
-- Daca stim ca
putStr (show n) :: IO ()
n + 10          :: Integer
-- Atunci ce tip are (??):
(??) :: IO () -> Integer -> IO Integer
-- (??) se numeste ($>) din modulul Data.Functor
($>) :: Functor f => f a -> b -> f b
-- In cazul nostru f => IO, a => () iar b => Integer:
($>) :: IO () -> Integer -> IO Integer
```

IO

Pentru că vrem să întoarcem o valoare de tip întreg, dar pentru că avem și un efect secundar, tipul pe care îl întoarce `showPlusZece` trebuie să fie **`IO Integer`**:

```
showPlusZece :: Integer -> IO Integer
showPlusZece n = putStr (show n) ?? n + 10
-- Daca stim ca
putStr (show n) :: IO ()
n + 10          :: Integer
-- Atunci ce tip are (??):
(??) :: IO () -> Integer -> IO Integer
-- (??) se numeste ($>) din modulul Data.Functor
($>) :: Functor f => f a -> b -> f b
-- In cazul nostru f => IO, a => () iar b => Integer:
($>) :: IO () -> Integer -> IO Integer
-- Acum putem completa:
showPlusZece n = putStr (show n) $> n + 10
```

Folosirea funcțiilor cu IO

```
Prelude> showPlusZece 3
313
```

Observăm afișarea parametrului `3`, urmată de afișarea rezultatului expresiei, `13`.

Folosirea funcțiilor cu IO

```
Prelude> showPlusZece 3
313
```

Observăm afișarea parametrului `3`, urmată de afișarea rezultatului expresiei, `13`.

```
Prelude> rezultat = showPlusZece 3
```

Nu se afișează nimic pentru expresia de mai sus.

Folosirea funcțiilor cu IO

```
Prelude> showPlusZece 3
313
```

Observăm afișarea parametrului `3`, urmată de afișarea rezultatului expresiei, `13`.

```
Prelude> rezultat = showPlusZece 3
```

Nu se afișează nimic pentru expresia de mai sus. În schimb, dacă încercăm să evaluăm expresia:

```
Prelude> rezultat
313
```

atunci se va evalua atât afișarea cât și rezultatul expresiei. Ce tip are această expresie?

Folosirea funcțiilor cu IO

```
Prelude> showPlusZece 3
313
```

Observăm afișarea parametrului `3`, urmată de afișarea rezultatului expresiei, `13`.

```
Prelude> rezultat = showPlusZece 3
```

Nu se afișează nimic pentru expresia de mai sus. În schimb, dacă încercăm să evaluăm expresia:

```
Prelude> rezultat
313
```

atunci se va evalua atât afișarea cât și rezultatul expresiei. Ce tip are această expresie?

```
Prelude> :t rezultat
rezultat :: IO Integer
```

Lucrul cu IO

Cum putem aplica de două ori același algoritm:

- afișăm numărul primit
- adăugăm 10
- afișăm rezultatul
- adăugăm 10 și întoarcem valoarea obținută

Lucrul cu IO

Cum putem aplica de două ori același algoritm:

- afișăm numărul primit
- adăugăm 10
- afișăm rezultatul
- adăugăm 10 și întoarcem valoarea obținută

Trebuie să putem trimite ca argument un **IO Integer** unei funcții care primește un **Integer** :

```
showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n =
    (showPlusZece n) ?? showPlusZece
```

Lucrul cu IO

Cum putem aplica de două ori același algoritm:

- afișăm numărul primit
- adăugăm 10
- afișăm rezultatul
- adăugăm 10 și întoarcem valoarea obținută

Trebuie să putem trimite ca argument un **IO Integer** unei funcții care primește un **Integer**:

```
showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n =
    (showPlusZece n) ?? showPlusZece
--   ^^ IO Integer      ^^ Integer -> IO Integer

(??) ::
```

Lucrul cu IO

Cum putem aplica de două ori același algoritm:

- afișăm numărul primit
- adăugăm 10
- afișăm rezultatul
- adăugăm 10 și întoarcem valoarea obținută

Trebuie să putem trimite ca argument un **IO Integer** unei funcții care primește un **Integer**:

```
showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n =
    (showPlusZece n) ?? showPlusZece
--   ^^ IO Integer       ^^ Integer -> IO Integer

(??) :: IO Integer -> (Integer -> IO Integer) -> IO Integer
```

Bind

Operatorul care permite compunerea de efecte se numește bind:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
-- In cazul nostru avem m => IO, a => Integer, b => Integer
(>>=) :: IO Integer -> (Integer -> IO Integer) -> IO Integer
```

Bind

Operatorul care permite compunerea de efecte se numește bind:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
-- In cazul nostru avem m => IO, a => Integer, b => Integer
(>>=) :: IO Integer -> (Integer -> IO Integer) -> IO Integer

showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n = showPlusZece n >>= showPlusZece
```

Bind

Operatorul care permite compunerea de efecte se numește bind:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
-- In cazul nostru avem m => IO, a => Integer, b => Integer
(>>=) :: IO Integer -> (Integer -> IO Integer) -> IO Integer

showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n = showPlusZece n >>= showPlusZece

Prelude> showPlusZece 3
313
Prelude> showPlusZeceX2 3
```

Bind

Operatorul care permite compunerea de efecte se numește bind:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
-- In cazul nostru avem m => IO, a => Integer, b => Integer
(>>=) :: IO Integer -> (Integer -> IO Integer) -> IO Integer

showPlusZeceX2 :: Integer -> IO Integer
showPlusZeceX2 n = showPlusZece n >>= showPlusZece

Prelude> showPlusZece 3
313
Prelude> showPlusZeceX2 3
31323
```

Interpretarea valorii afișate este:

- 3 este valoarea de intrare pentru primul showPlusZece
- 13 este valoarea de intrare pentru al doilea showPlusZece
- 23 este rezultatul afișat de către GHCi (rezultatul evaluării expresiei)

Monad

Tipul **IO** face parte din clasa **Monad**. Aceasta va fi studiată în detaliu în cursurile următoare.

Momentan, așa cum am arătat că putem folosi orice funcție care necesită clasa Foldable cu termeni de tipul listă, putem folosi orice funcție care necesită clasa **Monad** cu termeni de tipul **IO**.

Citire și notația do

Citire de la tastatură

Am văzut că un termen de tipul `IO a` reprezintă atât o valoare de tipul `a` cât și un efect (afișarea unei valori). Similar, citirea de la tastatură se face folosind un termen de tipul `IO String`:

```
getLine :: IO String
```

```
Prelude> getLine
```

Citire de la tastatură

Am văzut că un termen de tipul `IO a` reprezintă atât o valoare de tipul `a` cât și un efect (afișarea unei valori). Similar, citirea de la tastatură se face folosind un termen de tipul `IO String`:

```
getLine :: IO String
```

```
Prelude> getLine
Haskell
```

Citire de la tastatură

Am văzut că un termen de tipul `IO a` reprezintă atât o valoare de tipul `a` cât și un efect (afișarea unei valori). Similar, citirea de la tastatură se face folosind un termen de tipul `IO String`:

```
getLine :: IO String
```

```
Prelude> getLine
Haskell
"Haskell"
```

După ce apelăm funcția `getLine` putem scrie un șir de caractere urmat de `enter`. În cazul acesta, am scris *Haskell*. `GHCi` întoarce acest șir de caractere între ghilimele.

Read

Dacă am vrea să citim un întreg de la tastatură pentru a îl trimite funcției `showPlusZece`, ar trebui întâi să putem transforma un șir de caractere într-un număr întreg. Pentru asta folosim funcția **`read`**:

```
read :: Read a => String -> a
```

Read

Dacă am vrea să citim un întreg de la tastatură pentru a îl trimite funcției `showPlusZece`, ar trebui întâi să putem transforma un șir de caractere într-un număr întreg. Pentru asta folosim funcția **`read`**:

```
read :: Read a => String -> a
```

```
Prelude> read "10" + 10
20
Prelude> read "False" || True
True
Prelude> read "[1,2,3]" ++ [4]
[1,2,3,4]
Prelude> read "1" || True
*** Exception: Prelude.read: no parse
```

Read

Dacă am vrea să citim un întreg de la tastatură pentru a îl trimite funcției `showPlusZece`, ar trebui întâi să putem transforma un șir de caractere într-un număr întreg. Pentru asta folosim funcția **`read`**:

```
read :: Read a => String -> a
```

```
Prelude> read "10" + 10
20
Prelude> read "False" || True
True
Prelude> read "[1,2,3]" ++ [4]
[1,2,3,4]
Prelude> read "1" || True
*** Exception: Prelude.read: no parse
```

Compilatorul va încerca să deducă tipul pe care trebuie să îl întoarcă funcția **`read`** din contextul în care este folosit. `Atentie:` **`read`** va arunca o excepție dacă șirul primit nu poate fi convertit la tipul dedus din context.

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece ::
```

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
```

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
-- Cu ce functie seamana ??
??   :: (String -> Integer) -> IO String -> IO Integer
```

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
-- Cu ce functie seamana ??
??   :: (String -> Integer) -> IO String -> IO Integer
map  :: (a      -> b      ) -> [a]        -> [b]
```

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
-- Cu ce functie seamana ??
??    :: (String -> Integer) -> IO String -> IO Integer
map   :: (a      -> b      ) -> [a]        -> [b]
fmap  :: (a      -> b      ) -> f a        -> f b
```

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția `showPlusZece`:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
-- Cu ce functie seamana ??
??   :: (String -> Integer) -> IO String -> IO Integer
map  :: (a      -> b      ) -> [a]        -> [b]
fmap :: (a      -> b      ) -> f a        -> f b


citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = fmap read getLine >>= showPlusZece
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> ?? (x + y)
        )
    )
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> ?? (x + y)
        )
    )


??      ::
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> ?? (x + y)
        )
    )


??     ::                Integer -> IO Integer
return :: Monad m => a           -> m  a
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> return (x + y)
        )
    )
```

```
??     ::                Integer -> IO Integer
return :: Monad m => a           -> m  a
```

Putem folosi `return` pentru a introduce valori pure în `IO`. Deși putem scrie funcții folosind `>>=`, codul devine greu de citit.

Notația **do**

Pentru a simplifica funcțiile care folosesc `>>=` si **return**, in Haskell există notația **do**. Funcția `suma` poate fi scrisă echivalent:

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> return (x + y)
        )
    )
```

Notația **do**

Pentru a simplifica funcțiile care folosesc `>>=` si **return**, in Haskell există notația **do**. Funcția `suma` poate fi scrisă echivalent:

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> return (x + y)
        )
    )

suma :: IO Integer
suma = do
    x <- fmap read getLine
    y <- fmap read getLine
    return (x + y)
```

Citire și scriere

Notația **do** ne permite de asemenea să combinăm scrierea și citirea:

```
citesteSiScrie :: IO Integer
citesteSiScrie = do
    putStrLn "Introduceti numele: "
    nume <- getLine
    putStrLn ("Buna ziua, " ++ nume)
    putStrLn "Introduceti doua numere:"
    x <- fmap read getLine
    y <- fmap read getLine
    let suma = x + y
    putStrLn ("Suma este: " ++ show suma)
    return suma
```

putStrLn este identic cu **putStr**, cu excepția că adaugă o linie la sfârșit.

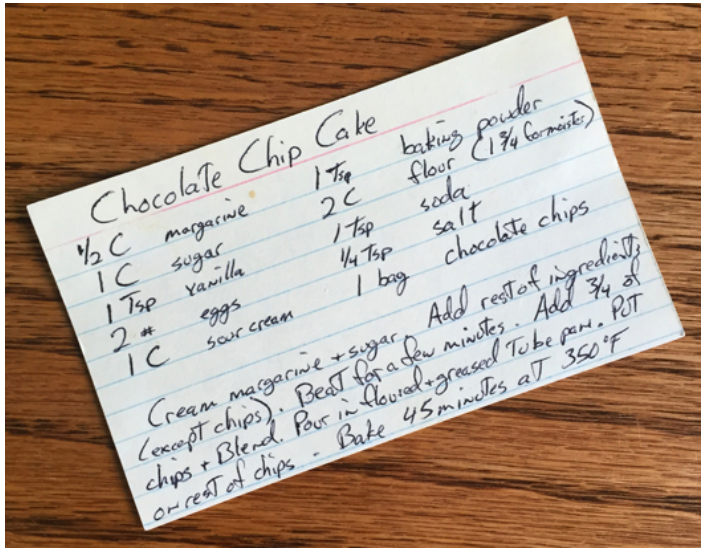
IO și transparența referențială

Este IO impur?

Haskell reușește să păstreze transparența referențială pentru că **IO Integer** nu reprezintă o valoare de tip **Integer**, ci o metodă pentru a obține o valoare de tip **Integer**.

Rețetă vs Prăjitură

tort :: Tort

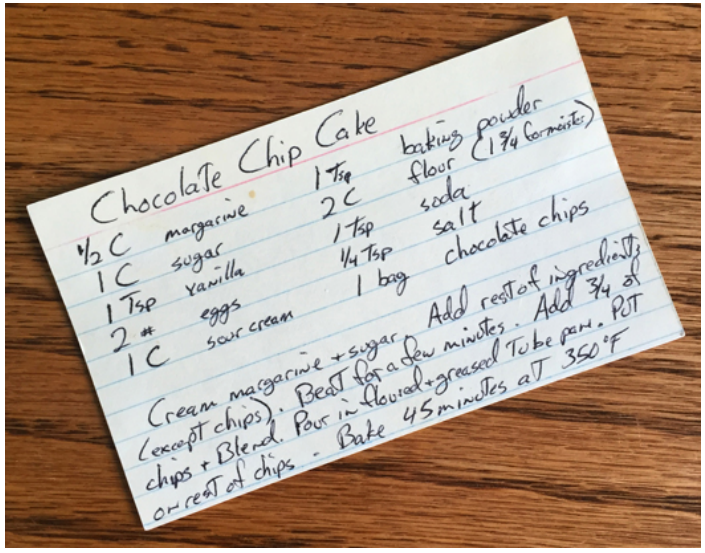


make tort :: **IO** Tort

- un termen de tipul **IO** a reprezintă o rețetă sau un set de instrucțiuni care generează o valoare de tipul a .

Rețetă vs Prăjitură

`tort :: Tort`

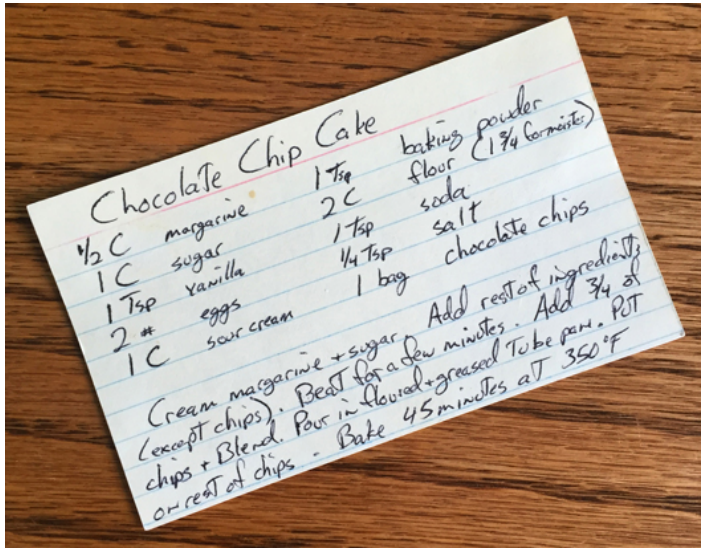


`make tort :: IO Tort`

- un termen de tipul `IO a` reprezintă o rețetă sau un set de instrucțiuni care generează o valoare de tipul `a`.
- nu putem să scoatem valoarea dintr-un `IO`, ci doar să operăm pe ea folosind funcții precum `fmap` sau `>>=`.

Rețetă vs Prăjitură

`tort :: Tort`



`make tort :: IO Tort`

- un termen de tipul `IO a` reprezintă o rețetă sau un set de instrucțiuni care generează o valoare de tipul `a`.
- nu putem să scoatem valoarea dintr-un `IO`, ci doar să operăm pe ea folosind funcții precum `fmap` sau `>>=`.
- executarea propriu-zisă a instrucțiunilor din `IO` se face în momentul în care programul se execută (sau evaluează în `GHCI`).

Haskell Runtime System

Motorul care citește și execută instrucțiunile **IO** se numește *Haskell Runtime System* (RTS). Acest sistem reprezintă legătura dintre programul scris și mediul în care va fi executat, împreună cu toate efectele și particularitățile acestuia.

Haskell Runtime System

Motorul care citește și execută instrucțiunile **IO** se numește *Haskell Runtime System* (RTS). Acest sistem reprezintă legătura dintre programul scris și mediul în care va fi executat, împreună cu toate efectele și particularitățile acestuia.

Un termen de tipul **IO Integer** va produce același întreg de fiecare dată, cu condiția să primească aceleași date de intrare. În cazul acesta, datele de intrare nu sunt numai cele explicite, ci și cele furnizate sub formă de efecte secundare (intrări de la consolă, conținut al fișierelor, data și ora curentă, etc.).

Haskell Runtime System

Motorul care citește și execută instrucțiunile **IO** se numește *Haskell Runtime System* (RTS). Acest sistem reprezintă legătura dintre programul scris și mediul în care va fi executat, împreună cu toate efectele și particularitățile acestuia.

Un termen de tipul **IO Integer** va produce același întreg de fiecare dată, cu condiția să primească aceleași date de intrare. În cazul acesta, datele de intrare nu sunt numai cele explicite, ci și cele furnizate sub formă de efecte secundare (intrări de la consolă, conținut al fișierelor, data și ora curentă, etc.).

Este important de reținut că un termen de tipul **IO Integer** reprezintă o metodă pentru generarea unui întreg.

Lucrul cu fișierele

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile ::
```

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile  :: FilePath -> IO String
writeFile ::
```

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile ::
```

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Spre exemplu, putem scrie o funcție care citește un fișier și îl printează în consolă:

```
cat :: FilePath -> IO ()
cat path = do
  continut <- readFile path
  putStr continut
```

Exemplu

Exemplu: citirea unui fișier de intrare și convertirea tuturor caracterelor in majuscule:

```
toUpperFile :: IO ()
toUpperFile = do
  putStr "Fisier intrare: "
  inPath <- getLine

  continut <- readFile inPath
  let caps = map toUpper continut

  putStr "Fisier iesire: "
  outPath <- getLine

  writeFile outPath caps
```

Suma numerelor dintr-un fișier

```
lines :: String -> [String]
words :: String -> [String]
concat :: [[String]] -> [String]

sumaFisier :: FilePath -> IO Integer
sumaFisier path = do
    continut <- readFile path
    let linii = lines continut      -- linii   :: [String]
    let cuvinte = map words linii   -- cuvinte :: [[String]]
    let strNum = concat cuvinte     -- strNum  :: [String]
    let numere = map read strNum    -- numere  :: [Integer]
    return (sum numere)
```

Programul va calcula suma numerelor din fișierul de intrare. Acestea pot fi separate de spații sau linii.

Pe săptămâna viitoare!