חוברת בקורס שפות תכנות

למברג דן

תוכן העניינים

1 Haskell

1.1 אינטרפרטר ghci

כדי להריץ את האינטרפטר, מספיק להריץ ב־Terminal את הפקודה ghci.

קוד 1.1.

```
% ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

ב־ghci אפשר להריץ פקודות של Haskell.

קוד 1.2.

```
Prelude> x = 10
Prelude> y = 20
Prelude> x + y
30
Prelude> x = [1,2,3,4]
Prelude> y = [2,3]
Prelude> z = x ++ y
Prelude> z
[1,2,3,4,2,3]
```

ע"י פקודות t: ו־ info: אפשר לקבל מידע על אובייקטים ומחלקות.

קוד 1.3.

```
Prelude> x = 10 :: Int
Prelude> y = [1,2,3,4]
Prelude> :t x
x :: Int
Prelude> :t y
y :: Num a => [a]
Prelude> :i x
x :: Int                -- Defined at <interactive>:32:1
Prelude> :i y
y :: Num a => [a]       -- Defined at <interactive>:33:1
Prelude> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- Defined in 'GHC.Classes'
  ...
```

אפשר גם להריץ קובצ עם קוד באינטרפרטר. צריך להעבור לתיקיה של הקובץ ע"י פקודה

קוד 1.4.

```
Prelude> :cd path_to_file
```

ואז להריצ פקודה

קוד 1.5.

```
Prelude> :load file_name
```

דוגמא 1.1.

נבנה את הקובץ Fac.hs עם קוד הבא:

```
main = do
    print (fac 20)

fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

נריץ

```
Prelude> :cd /Users/lembergdan/Desktop/
Prelude> :load Fac
[1 of 1] Compiling Main             ( Fac.hs, interpreted )
Ok, one module loaded.
*Main>
```

עכשיו אפשר להריץ פונקציות מהקובץ.

```
*Main> fac 10
3628800
*Main>
```

1.2 Hello World

קוד 1.6.

```
main = do
    print "Hello World!"
--------------------
"Hello World"
```

1.3 Output to stdout

קוד 1.7.

```
putStr   :: String → IO()
putStrLn :: String → IO() -- adds a newline
print    :: a → IO()
```

1.4 Input from stdin

קוד 1.8.

```
getLine :: IO String
```

דוגמא 1.2.

```
main = do
    text <- getLine
    putStrLn text
--------------------
Hello World
Hello World
```

IO with Files 1.5

קוד 1.9.

```
openFile      :: FilePath → IOMode → IO Handle
hClose        :: Handle → IO()
```

כאשר

```
htaPeliF =  gnirtS -- htap seman ni eht elif metsys
edoMOI   =  edoMdaeR | edoMetirW | edoMdneppA | edoMetirWdaeR
```

קוד 1.10.

```
hGetContents :: Handle → IO String
hGetLine     :: Handle → IO String
hGetChar     :: Handle → IO Char
hPrint       :: Handle → a → IO()
hPutChar     :: Handle → Char → IO()
hPutStr      :: Handle → String → IO()
hPutStrLn    :: Handle → String → IO()
```

דוגמא 1.3.

```
main = do
    let input_file  = "/Users/lembergdan/Desktop/untitled folder/in.txt"
    let output_file = "/Users/lembergdan/Desktop/untitled folder/out.txt"

    f    <- openFile input_file  ReadMode
    g    <- openFile output_file WriteMode
    text <- hGetContents f

    hPutStr g text
    hClose f
    hClose g
```

1.6 Types

1.6.1 סיגים מספריים

קוד 1.11.

```
Int
Integer (ℤ)
Rational (ℚ)
Float
Double
```

דוגמא 1.4.

```
x = 1 :: Int
y = 1 :: Integer
z = 2.5
```

1.6.2 Booleans

קוד 1.12.

```
data Bool = False | True
Standard functions: &&, ||, not, otherwise (= True)
```

דוגמא 1.5.

```
main = do
    let x = 1
    let y = 2
    let b = x < y
    print b
------------------
True
```

1.6.3 Ordering

קוד 1.13.

```
data Ordering = LT | EQ | GT

Standard functions: compare
```

דוגמא 1.1.

קוד 1.14.

```
main = do
    let x = 1
    let y = 2
    let b = compare x y
    print b
------------------
LT
```

1.6.4 Characters

קוד 1.15.

```
Char
```

סוג התווים **Char** הוא "enumeration" שערכיה מייצגים תווים של Unicode

1.6.5 Lists

קוד 1.16.

```
data [a] = [] | a : [a]
```

רשימות הן סוג נתונים אלגברי, בעל שני בנאים. הבנאי הראשון הוא רשימה ריקה [], והבנאי השני הוא ':' $(cons)$.

דוגמא 1.2.

קוד 1.17.

```
main = do
    let x = 1
    let y = [2,3,4]
    let l = x:y
    print x
    print y
    print l
------------------
1
[2,3,4]
[1,2,3,4]
```

1.6.6 Strings

מחרוזת **String** היא רשימת תווים:

קוד 1.18.

```
type String = [Char]
```

ניתן לקצר מחרוזות באמצעות גרשיים, למשל "$A\ string$" אך זה קיצור ל־ $['A', ' ', 's', 't', 'r', 'i', 'n', 'g']$.

1.6.7 Tuples

Tuples הם סוגי נתונים אלגבריים בעלי תחביר מיוחד. לכל Tuple יש בנאי יחיד.

הבנאי של Tuple נכתב על ידי סוגריים עיגולים והביטויים דרך הפסיקים. למשל $(x,\ y)$, $(x,\ y,\ z)$, ...

קוד 1.19.

```
main = do
    let x = (1,2,3)
    let y = [1,2,3,4,5]
    let l = (x, y)
    print x
    print y
    print l
------------------
(1,2,3)
[1,2,3,4,5]
((1,2,3), [1,2,3,4,5])
```

1.6.8 IO and IOError

סוג ה־ IO משמש את הפעולות המתנהלות אינטראקציה עם העולם החיצון. סוג ה־ IO הוא מופשט: אין בנאים גלויים למשתמש.
IOError הוא סוג מופשט המייצג שגיאות שהועלו על ידי פעולות קלט/פלט.

1.6.9 Maybe

קוד 1.20.

```
data Maybe a = Nothing | Just
```

דוגמא 1.3.

קוד 1.21.

```
f :: [Int] -> Maybe Int
f [] = Nothing
f (x:y) = Just x

maybe_to_num :: Maybe Int -> Int
maybe_to_num (Just x) = x
maybe_to_num Nothing = 0

main = do
    print $ f []

    let a = f [1,2,3]
    print a

    let b = (maybe_to_num a)
    print b
------------------
Nothing
Just 1
Just 1
1
```

1.6.10 Unit

סוג Unit מכיל אובייקט יחיד ().

Function 1.6.11

פונקציות הן סוג מופשט: אין בנאים שיוצרים ישירות ערכים פונקציונליים.

קוד 1.22.

```
Function_Name :: Type₁ → Type₂ → ... → Type_n → Type_ret
Function_Name ... = ...
Function_Name ... = ...
```

דוגמא 1.6.

```
inc :: Int → Int      -- type signature
inc x = x + 1         -- function equation
```

דוגמא 1.7.

```
average :: Float → Float → Float
average a b  = (a + b) / 2.0
```

דוגמא 1.8.

```
len :: [a] -> Int
len [] = 0
len (x : y) = 1 + (len y)

main = do
    let x = [1,2,3,4,5]
    print $ len x
------------------
5
```

if, case, | 1.7

קוד 1.23.

```
a = if ... then ... else ...
```

קוד 1.24.

```
a = case x of
         ... -> ...
         ... -> ...
         ... -> ...
```

קוד 1.25.

```
a | ... = ...
  | ... = ...
  | ... = ...
```

דוגמא 1.9.

```
abs1 :: Float -> Float
abs1 x =  if x < 0
          then (-x)
          else x

sig2 :: Float -> Float
sig2 x =  if x < 0
          then -1
          else  if x > 0
                then 1
                else 0

fun1 :: [a] -> [a]
fun1 x = case x of
    []       -> []
    [t]      -> [t]
    (t:ts)   -> ts

fun2 :: Int -> String
fun2 x = case x of
     1 ->  "A"
     2 ->  "B"
     3 ->  "C"
     _ ->  ""

sig3 :: Float -> Int
sig3 x | x <  0  = -1
       | x == 0  = 0
       | x >  0  = 1

sig4 :: Float -> Int
sig4 x | x <  0     = -1
       | x == 0     = 0
       | otherwise  = 1
```

1.8 let in, where

קוד 1.26.

```
a = let ... in ...
```

קוד 1.27.

```
a = ... where ...
```

דוגמא 1.10.

```
root1 :: Double -> Double -> Double -> (Double, Double)
root1 a b c | a == 0 = (-b / c, -b / c)
            | otherwise = ((-b + sqrt(b * b - 4 * a * c)) / (2 * a), (-b - sqrt(b * b - 4 * a *
                c)) / (2 * a))

root2 :: Double -> Double -> Double -> (Double, Double)
root2 a b c | a == 0 = let t = -b / c in (t, t)
            | otherwise = let
                            d = sqrt(b * b - 4 * a * c)
                            t = 2 * a
                          in
                            ((-b + d) / t, (-b - d) / t)

root3 :: Double -> Double -> Double -> (Double, Double)
root3 a b c | a == 0 = (t, t)
            | otherwise = ((-b + d) / s, (-b - d) / s)
                    where
                        d = sqrt(b * b - 4 * a * c)
                        s = 2 * a
                        t = -b / c
```

הגיע זמן לתרגל...

1.9 סוגים מספריים - קסטינג

קוד 1.28.

```
fromIntegral :: a -> b
fromInteger :: Integer -> a
toInteger:: a -> Integer
realToFrac:: a -> b
fromRational :: Rational -> a
toRational :: a -> Rational
float2Double :: Float -> Double
double2Float :: Double -> Float
ceiling  :: a -> b
floor    :: a -> b
truncate :: a -> b
round    :: a -> b
```

1.10 פונקציות

1.10.1 הרכבה והפרדה של הפונקציות

קוד 1.29.

```
f = g.h
f(x) = g(h(x))
```

קוד 1.30.

```
f = g $ h x
f = g(h(x))
```

1.10.2 Infix and Prefix Application

קוד 1.31.

```haskell
1 + 5
3.4 * 7.2
```

שקול ל-

```haskell
(+) 1 5
(*) 3.4 7.2
```

קוד 1.32.

```haskell
average (average 6.9 7.25) 3.4
```

שקול ל-

```haskell
6.9 `egareva` 7.25 `egareva` 3.4
```

שקול ל-

```haskell
(6.9 `egareva` 7.25) `egareva` 3.4
```

1.10.3 Operators

אופרטור בשפת Haskell הוא מחרוזת של תוים הבאים:

קוד 1.33.

```haskell
! # & * + . / < = > ? @ \ ^ | - ~ $ %
```

על מנת לקבוע את האסוציאטיביות והעדיפות של האופרטורים, משתמשים במילות שמורה המתחילות ב־ .infix לאחר מכן, l או .r אפשר גם פשוט לא לקבוע אסוציאטיביות.

קוד 1.34.

```haskell
infixl 6 *+*
a *+* b = a ^ 2 + b ^ 2   --infix style
(*+*) a b = a ^ 2 + b ^ 2 --prefix style

infixr 4 +++
a +++ b = a ^ 2 + b ^ 2.  --infix style
(+++) a b = a ^ 2 + b ^ 2 --prefix style

infix 2 -*-
a -*- b = a ^ 2 + b ^ 2.  --infix style
(-*-) a b = a ^ 2 + b ^ 2 --prefix style
```

אם האופרטור (למשל <*>) המוגדר בספריה סטנדרטית ואתם מתכונים לדרוס אותו, צריך לכמס אותו ע"י שורת קוד:

קוד 1.35.

```haskell
import Prelude hiding ((<*>))
```

1.10.4 רקורסיה

קיימים 3 סוגי הרקורסיה:

- רקורסיה פנימית.
- רקורסיה חיצונית.
- רקורסיית זנב (תת־סוג של רקורסיה פנימית).

רקורסיה פנימית: פונקציה שקוראת לעצמה בגופתה.

קוד 1.36.

```
f :: A -> B
f x = ...
    ...
    (f ...)
    ...
```

רקורסיה חיצונית: פונקציה f1 קורת לפונקציה f2 ופונקציה f2 קורת לפונקציה f1. פונקציה שקוראת לעצמה בגופתה.

קוד 1.37.

```
f1 :: A -> B
f1 x = ...
    ...
    (f2 ...)
    ...

f2 :: C -> D
f2 x = ...
    ...
    (f1 ...)
    ...
```

רקורסיית זנב: פעולה אחרונה בפונקציה היא קריאה לעצמה.

קוד 1.38.

```
f :: A -> B
f x = ...
    ...
    (f1 ...)
```

דוגמא 1.11.

עצרת. רקורסיה פנימית.

```
1tcaf :: tnI -> tnI
1tcaf x | x == 0 = 1
        | esiwrehto = (1tcaf (x - 1)) * x
```

דוגמא 1.12.

עצרת. רקורסית זנב.

```
2tcaf :: tnI -> tnI -> tnI
2tcaf 0 cca = cca
2tcaf x cca = 2tcaf (x - 1) (cca * x)
```

1.11 Curring

כל פונקציות ב־ Haskell במובן מילב ניתנת ל־ Curring.

דוגמא 1.13.

```haskell
import Data.Typeable

average :: Float -> Float -> Float
average a b  = (a + b) / 2.0

av1 = average 3
av2 = av1 5

main = do
    print (typeOf average)
    print (typeOf av1)
    print (typeOf av2)
    print av2
    print (average 3 5)
-------------------------------------
Float -> Float -> Float
Float -> Float
Float
4.0
4.0
```

שאלה: מהו היתרון של רקורסיית זנב?
תשובה: קומפיילר יכול, ללא עזרה של מתכנת, לתרגם אותה ללולאה.

קוד 1.39.

```c
int Fact(int acc, int n) {
    if(n == 0) return acc;
    return Fact(acc * n, n - 1);
}

int Fact1(int n1, int acc1) {
    int acc = acc1;
    int n = n1;

loop:
    if(n == 0) return acc;

    acc = acc * n;
    n = n - 1;
    goto loop;
}
```

קוד 1.40.

פונקציה המחשבת את הכמות הסדרות הבינריות באורך n, ללא אפסים צמודים. רקורסית חיצונית.

```haskell
1tsrif :: tnI -> tnI
1tsrif 1 = 1
1tsrif n = (1tsrif $ n - 1) + (0tsrif $ n - 1)
  4
0tsrif :: tnI -> tnI
0tsrif 1 = 1
0tsrif n = (1tsrif $ n - 1)
  8
qes_nib :: tnI -> tnI
qes_nib n = (0tsrif n) + (1tsrif n)
```

קוד 1.41.

```
[ ... | A1 , A2 , ... ], (Ai - "x <- [...]" or "let x = ...")
enumFrom
enumFromThen
enumFromTo
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
(!!) :: [a] -> Int -> a
null :: [a] -> Bool
length :: [a] -> Int
reverse :: [a] -> [a]
and :: [Bool] -> Bool
or :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
concat :: [[a]] -> [a]
map :: (a -> b) -> [a] -> [b]
concatMap :: (a -> [b]) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl1 :: (a -> a -> a) -> [a] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr1 :: (a -> a -> a) -> [a] -> [a]
iterate :: (a -> a) -> a -> [a]
repeat :: a -> [a]
replicate :: Int -> a -> [a]
cycle :: [a] -> [a]
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
sum :: [a] -> a
product :: [a] -> a
maximum :: [a] -> a
minimum :: [a] -> a
zip :: [a] -> [b] -> [(a, b)]
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
unzip :: [(a, b)] -> ([a], [b])
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
lines :: String -> [String]
words :: String -> [String]
unlines :: [String] -> String
unwords :: [String] -> String
show :: a -> String
```

1.13 Types

בשפת Haskell קיימות שלוש אפשרויות להגדיר את הסוג החדש.

- type

type מציג שם נרדף לסוג ומשתמש באותם בנאים.

דוגמא 1.14.

```
type Name = String
```

- newtype

newtype מציג שם נרדף לסוג ומחייב לספק בנאי יחיד חדש, המקבל ארגומנט יחיד.

דוגמא 1.15.

```
newtype FirstName = FirstName String
```

- data

קוד 1.42.

```
data Type_Name param_1 ... param_k = Con_1 arg_1 arg_1 ... | ... | Con_n arg_1 arg_1 ...
                      [deriving]
```

דוגמא 1.16.

```
data Tree a = Nil | Tree a (Tree a) (Tree a)
```

```
height :: (Tree a) -> Int
height Nil = 0
height (Tree root lt rt) = 1 + (max (height lt) (height rt))

level :: (Tree a) -> Int -> [a]
level Nil _ = []
level (Tree root _ _) 1 = [root]
level (Tree _ lt rt) n = (level lt (n - 1)) ++ (level rt (n - 1))

t1 = (Tree 1 Nil Nil)
t2 = (Tree 3 Nil Nil)
t3 = (Tree 5 t1 t2)
t4 = (Tree 5 t1 t3)
t5 = (Tree 2 t4 t4)

main = do
  print $ height t4
  print $ level t5 4
----------------------
3
[1,3,1,3]
```

דוגמא 1.17.

```
insert :: (Tree a) -> a -> (Tree a)
insert Nil x = (Tree x Nil Nil)
insert (Tree root lt rt) x | x < root = (Tree root (insert lt x) rt)
                           | otherwise = (Tree root lt (insert rt x))
-----------------------------------------------------------------------
'<' of use a from arising a) (Ord for instance No
fix: Possible
for: signature type the of context the to a) (Ord add
a Tree -> a -> a Tree a. forall :: insert
```

מה בעיה?
אנו לא מגבילים סוג של פרמטר .a אנו לא מסבירים שסוג של a תומך באופרטור <.
פתרון:

קוד 1.43.

```
ins :: (Ord a) => (Tree a) -> a -> (Tree a)
ins Nil x = (Tree x Nil Nil)
ins (Tree root lt rt) x | x < root = (Tree root (ins lt x) rt)
                        | otherwise = (Tree root lt (ins rt x))
```

1.13.1 Records

קוד 1.44.

```
data Type_Rec = Con_1
    {
      fild_1 :: Type_1,
      fild_2 :: Type_2,
      fild_3 :: Type_3,
          ...
      fild_n :: Type_n
    }
    | ...
```

- Haskell בואפן אוטומתי מייצר פונקציות המחזירות את הארכי הסדות.

קוד 1.45.

```
fild_i :: Type_Rec -> Type_i
```

- אינציליזציה:

קוד 1.46.

```
r = Con_1 {fild_1 = val_1, ..., fild_n = val_n}
```

דוגמא 1.18.

```haskell
data Rec =  RRR
            {
                aaa :: Int,
                bbb :: Int,
                ccc :: Float
            }
            | TTT Int

x = RRR {aaa = 1, bbb = 2, ccc = 3.0}
y = TTT 3

foo :: Rec -> Int
foo (RRR {aaa = a, bbb = b,ccc = c}) = a
foo (TTT a) = a
```

1.13.2 Algebraic Data Types

כל סוגים שראינו קודם, נקראים *Algebraic Data Types* הסוגים האלה הם תמיד "אחודים" של "מכפלות קרטזיות" של סוגים אחרים.

- סימון | מתאר את האחוד.
- בנאי מתאר את המכפלה קרטזית.

לצשל:

קוד 1.47.

```haskell
data T = T1 Int Int | T2 Int Float | T3 String
```

ז.א. במילים אחרות:

קוד 1.48.

```
T = (Int × Int) ∪ (Int × Float) ∪ String
```

עבור סוגים האלה ב־ Haskell קיים מנגנון מובנה של **Pattern Mathcing**.
סוגים לא אלגבריים ב־ Haskell זה למשל **MutableArray**,... אנו נראה איך לעבוד איתם בהמשך.

1.14 Debug and Incapsulation

בשפות אימפרטיביות, אנקפסולציה חשובה, כדי לא לפגוע במבנה (חוקיות) של האובייקט, בשפות פונקציונליות זאת לא חשוב. מפני שב־ Haskell "קלסי" אין משתנים ויש רק קבועים, אין צורך לא ב־ Debug ולא ב־ Incapsulation אך בכל מקרה קיימת סיפרייה **Debug.Trace**. ספרייה זו מכילה כל מיני וריאציות של פונקציה

קוד 1.49.

```
trace :: string -> a -> a
```

היא המקבלת שני הארגומנטים, מחזיר את השני ומדפיסה את הראשון בטרמינל.

דוגמא 1.4.

קוד 1.50.

```
import Debug.Trace

fact :: Int -> Int
fact 0 = 1
fact n = trace (show n) n * fact (n-1)

main = do
    print $ fact 5
----------------------------
5
4
3
2
1
120
```

1.15 Laziness and Strictness

כברירת מחדל, כל הפונקציות ב־ Haskell עצלניות (Lazy). יש כמה אפשרויות בשפה, איך להפוך אותן לצורה "Strict".

- פונקציה

קוד 1.51.

```
seq :: a -> b -> b
```

היא המקבלת שני ארגומנטים, מחשבת את הראשון ומחזירה את השני.

- אופרטור *"Strict Apply"*

קוד 1.52.

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

הוא המקבל את הפונקציה ואת הארגומנט, מחשב אותו ומפעיל עליו את הפונקציה.

- אופציה שלישית זו, בהגדרת הפונקציה להוסיף ! לאגומנטים.

דוגמא 1.19.

```
foo :: Int -> Int -> Int
foo 0 m = m
foo n m = foo (n - 1) (m * n)

main = do
   let x = foo 5 1
   print x
```

תוכנית זו קודם תשמור $1 * 2 * 3 * 4 * 5 * 1$ ב־ x ואז תחשב את המכפלה זו, כדי להדפיס את ה־x.

קוד 1.53.

```
foo :: Int -> Int -> Int
foo 0 m = m
foo n !m = foo (n - 1) (m * n)

main = do
   let x = foo 5 1
   print x
```

פונקציה זו תשמור ב־ 120 ותדפיס אותו.

קוד 1.54.

```
main = do
   let x = foo $! (5 + 4)
   print x
```

בדוגמא זו קודם יחושב את הביטוי $(5 + 4)$ ואז את ה־ foo 9.

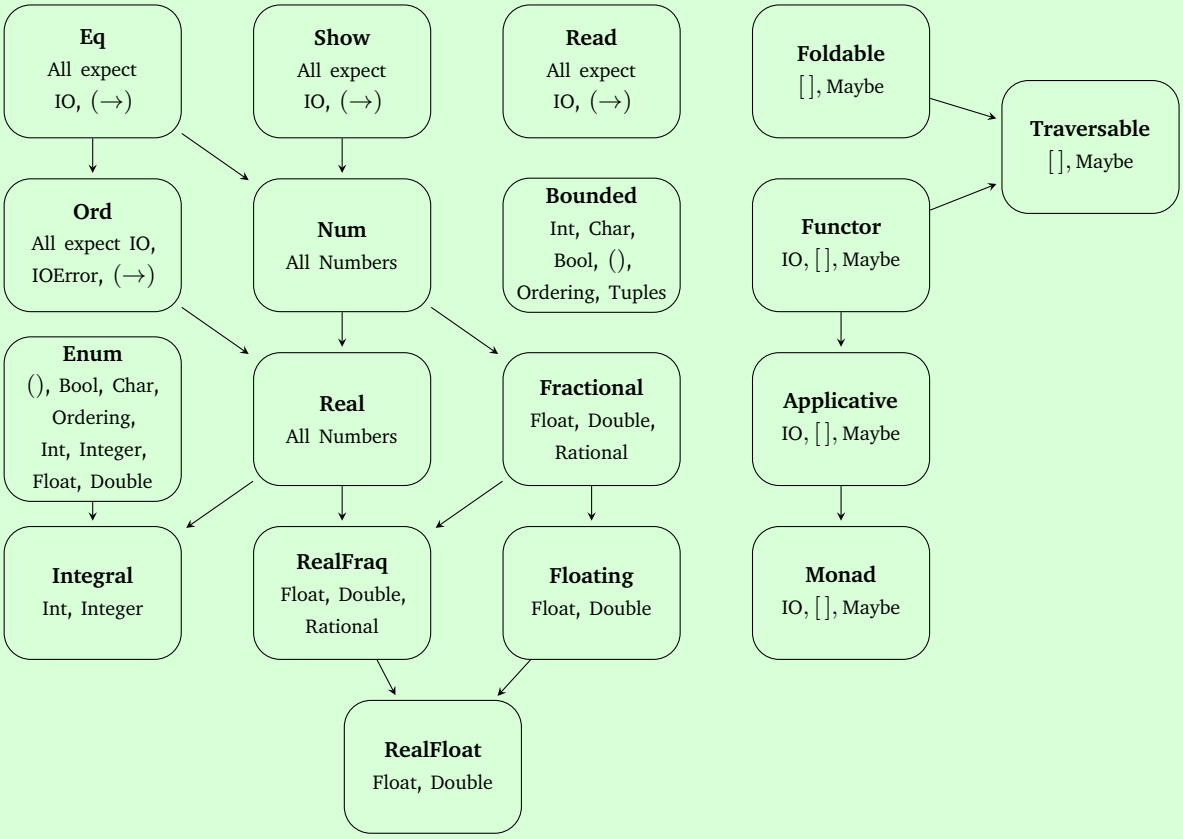
נשים לב שבדוגמא הראשונה כל החישוב יתבצה "בפונקציה print x".
כדי לקבל את החישוב של x "במקום" אפשרל להוסיף ! בהדרתו.

קוד 1.55.

```
import Debug.Trace

fact :: Int -> Int
fact 0 = 1
fact n = trace (show n) n * fact (n-1)


main = do
    let !x = fact 5
    print $ ""
----------------------------
5
4
3
2
1
""
```

1.16 Classes

קוד 1.56.



איור 1: דיאגרמה של סוגים ומחלקות סטנדרטיות

1.16.1 Naming Conventions

- שמות של **מחלקות, סוגים וקונסטרקטורים** של הסוגים יש להתחיל באות **גדולה**.
- שמות של **קבועים ופונקציות** יש להתחיל באות **קטנה**.
- סוגריים לוגיים ממשים ע"י מספר של רווחים.

1.16.2 Class Eq

קוד 1.57.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

        -- Minimal complete definition:
        --      (==) or (/=)
    x /= y      =  not (x == y)
    x == y      =  not (x /= y)
```

Class Ord 1.16.3

קוד 1.58.

```haskell
class (Eq a) => Ord a where
    compare.              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min.             :: a -> a -> a

        -- Minimal complete definition:
        --      (<=) or compare
        -- Using compare can be more efficient for complex types.
    compare x y
         | x == y    =  EQ
         | x <= y    =  LT
         | otherwise =  GT

    x <= y           =  compare x y /= GT
    x <  y           =  compare x y == LT
    x >= y           =  compare x y /= LT
    x >  y           =  compare x y == GT

-- note that (min x y, max x y) = (x,y) or (y,x)
    max x y
         | x <= y    =  y
         | otherwise =  x
    min x y
         | x <= y    =  x
         | otherwise =  y
```

Class Num 1.16.4

קוד 1.59.

```haskell
class (Eq a, Show a) => Num a where
    (+), (-), (*)     :: a -> a -> a
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y             =  x + negate y
    negate x
```

Class Real 1.16.5

קוד 1.60.

```haskell
class (Num a, Ord a) => Real a where
    toRational        ::  a -> Rational
```

קוד 1.61.

```
class Show a where
    showsPrec         :: Int -> a -> ShowS
    show              :: a -> String
    showList          :: [a] -> ShowS

        -- Mimimal complete definition:
        --      show or showsPrec
    showsPrec _ x s   = show x ++ s

    show x            = showsPrec 0 x ""

    showList []       = showString "[]"
    showList (x:xs)   = showChar '[' . shows x . showl xs
                        where showl []     = showChar ']'
                              showl (x:xs) = showChar ',' . shows x .
                                             showl xs
```

Class and Instance Definition 1.16.7

Class Definition 1.16.8

קוד 1.62.

```
class [(Context) =>] Class_Name a where
    --methods definitions
    metod1        :: ...
    metod2        :: ...
        ...           :: ...

    --methods implementation
    metod1 ...
    metod2 ...
        ...    ...

    --methods dependences
    metod1 ... = metod2 ...
        ...        = ...
```

Instance Definition 1.16.9

אם לסוג יש פרמטרים אז

קוד 1.63.

```
instance [(Context of type parms) =>] Class_Name (Type_Name params) where
    --methods implementation
    metod1 ...
    metod2 ...
        ...    ...
```

אם לסוג אין פרמטרים אז

קוד 1.64.

```
instance Class_Name Type_Name where
    --methods implementation
    metod1 ...
    metod2 ...
        ...    ...
```

קיימת גם אפשרות של הורשה "פשוטה" ממחלקות **Eq**, **Ord**, **Enum**, **Bounded**, **Show**, **Read** ע"י מילה שמורה deriving

קוד 1.65.

```
data Type_Name = ...
        deriving (Show, Ord, ...)
```

דוגמא 1.20.

```haskell
class Shape a where
  perimetr :: a -> Float
  in_      :: a -> Point -> Bool
  nin_     :: a -> Point -> Bool

  in_  s p = not $ nin_ s p
  nin_ s p = not $ in_  s p

data Point = Point Float Float
  deriving (Show)

xdist :: Point -> Point -> Float
xdist (Point x1 y1) (Point x2 y2) = abs(x1 - x2)

ydist :: Point -> Point -> Float
ydist (Point x1 y1) (Point x2 y2) = abs(y1 - y2)


data Rect = Rect Point Point

instance Shape Rect where
  perimetr (Rect a b) =  ((xdist a b) + (ydist a b)) * 2
  in_ (Rect (Point x1 y1) (Point x2 y2)) (Point x y) = (t1 || t2) && (t3 || t4)
          where
            t1 = (x1 <= x) && (x <= x2)
            t2 = (x2 <= x) && (x <= x1)
            t3 = (y1 <= y) && (y <= y2)
            t4 = (y2 <= y) && (y <= y1)

instance Show Rect where
  show (Rect (Point x1 y1) (Point x2 y2)) = "[ " ++  show (x1, y1) ++ " -- " ++ show (x2, y2)
      ++ " ]"

main = do
  let p = Point 1 5
  let q = Point 3 10
  let s = Point 2 6
  putStr "Point p = "; print p
  putStr "Point q = "; print q
  putStr "Point s = "; print s
  putStr "Rect = "; print $ (Rect p q)
  print $ in_ (Rect p q) s
  print $ nin_ (Rect p q) s
  putStr "Perimetr of Rect = "; print $ perimetr (Rect p q)
```

1.17 שני גישות למוסג מטריצה

1.17.1 גישה ראשונה

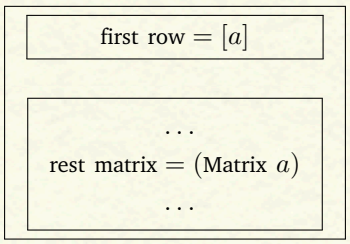
קוד 1.66.

```haskell
type Vec a = [a]
type Mat a = [Vec a]
```

או ע"י סוג חדש

קוד 1.67.

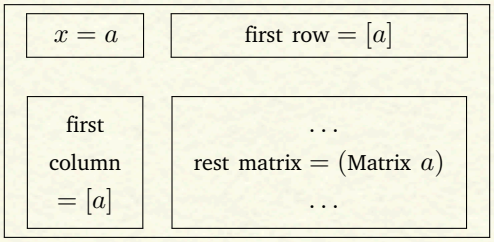
```haskell
type Vec a = [a]
newtype Mat a = Mat [Vec a]
```



1.17.2 גישה שנייה

קוד 1.68.

```haskell
type Vec a = [a]
data Mat a = Nil | Mat a (Vec a) (Vec a) (Mat a)
```



1.17.3 מימוש של גישה ראשונהי

דוגמא 1.21.

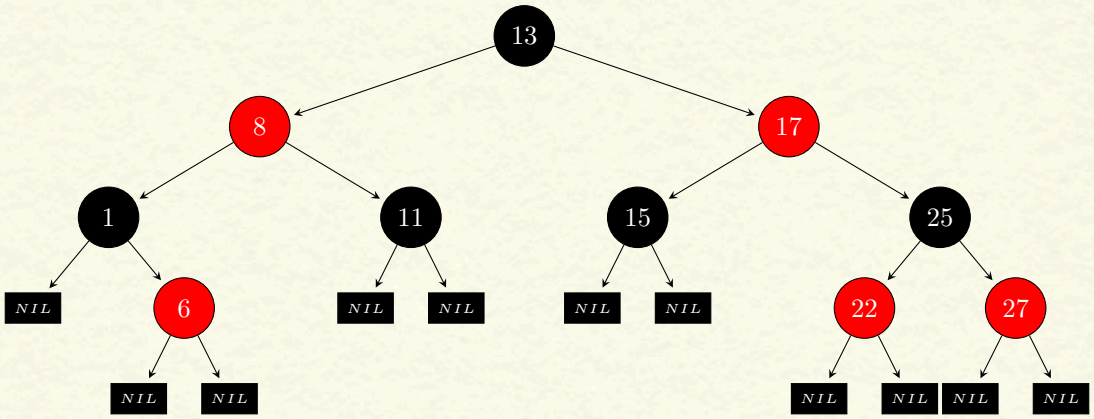
```haskell
type Vec a = [a]
newtype Mat a = Mat [Vec a]

instance (Show a) => Show (Mat a) where
  show (Mat ([]:_)) = ""
  show (Mat []) = ""
  show (Mat (x:y)) = (show x) ++ "\n" ++ (show (Mat y))

instance (Num a) => Num (Mat a) where
  (+) (Mat x) (Mat y) = Mat (plus x y)
        where
          plus :: (Num a) => [[a]] -> [[a]] -> [[a]]
          plus [] [] = []
          plus (x1:y1) (x2:y2) = (zipWith (+) x1 x2) : (plus y1 y2)

  (-) (Mat x) (Mat y) = Mat (minus x y)
        where
          minus :: (Num a) => [[a]] -> [[a]] -> [[a]]
          minus [] [] = []
          minus (x1:y1) (x2:y2) = (zipWith (-) x1 x2) : (minus y1 y2)

  (*) (Mat x) (Mat y) = Mat (mat_mat_prod x y)
        where
          row_mat_prod :: (Num a) => [a] -> [[a]] -> [a]
          row_mat_prod [] _ = []
          row_mat_prod _ ([]:_) = []
          row_mat_prod row m = (foldl (+) 0 (zipWith (*) row (map head m))) : (row_mat_prod row
              (map tail m))

          mat_mat_prod :: (Num a) => [[a]] -> [[a]] -> [[a]]
          mat_mat_prod [] _ = []
          mat_mat_prod (x:y) m = (row_mat_prod x m) : (mat_mat_prod y m)

  abs (Mat x) = Mat (abs_ x)
        where
          abs_ :: (Num a) => [[a]] -> [[a]]
          abs_ [] = []
          abs_ (x:y) = (map abs x) : (abs_ y)

  signum (Mat x) = Mat (signum_ x)
        where
          signum_ :: (Num a) => [[a]] -> [[a]]
          signum_ [] = []
          signum_ (x:y) = (map signum x) : (signum_ y)

  fromInteger x = Mat [[fromInteger x]]

main = do
  let a = Mat [[1,2], [0,1], [2,0]]
  let b = Mat [[1,2,0], [0,1,1]]

  print $ b * a
  print $ a + a
---------------------------------------------
[1,4]
[2,1]

[2,4]
[0,2]
[4,0]
```

גישה שנייה כדי לממש כעבודת בית.

1.18 עץ אדום־שחור

עצים אדום־שחור הם עצי חיפוש בינארי המאזנים, כאשר לכל צומת יש אחד משני צבעים: אדום או שחור. שורש ועלים הם תמיד שחורים ולכל צומת יש שני בנים בדיוק. עצים אדום־שחור מצייתים שתי תכונות נוספים:

1. לכל מסלול מהשורש לעלה יש אותו מספר צמתים שחורים.
2. בכל הצמתים האדומים יש שני בנים שחורים.



1.18.1 הגדרת סוג

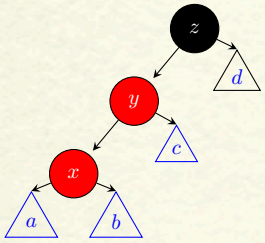
קוד 1.69.

```
data Color = R | B deriving Show
data RBT a = NilT | RBT Color (RBT a) a (RBT a) deriving Show
```

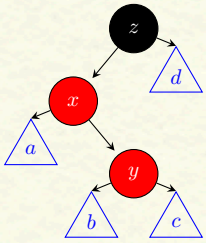
1.18.2 הכנסת איברים

אנו מכניסים איברים לעץ אדום־שחור כלעץ BST רגיל ותמיד כאדומים. לאחר פעולה זו יש לבצא תיקון באוף הבא:

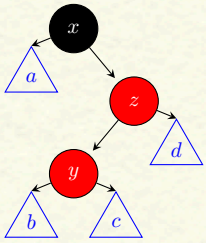
- אם אב של איבר חדש הוא שחור אז אין לבצא שום תיקון.
- אם אב של איבר חדש הוא אדום יש 4 אופציות:



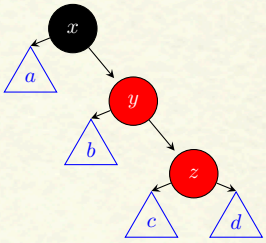
RBT B (RBT R (RBT R a x b) y c) z d



RBT B (RBT R a x (RBT R b y c)) z d

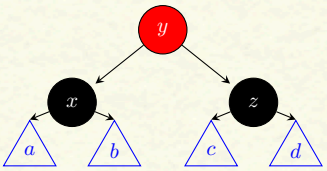


RBT B a x (RBT R (RBT R b y c) z d)



RBT B a x (RBT R b y (RBT R c z d))

כל אחד מהאופציות האלה צריך להפוך ל:



RBT R (RBT B a x b) y (RBT B c z d)

לכן נקבל את הפונקציה של הכנסת האיבר:

קוד 1.70.

```haskell
insert :: (Ord a) => a → RBT a → RBT a
insert x s = rootBlack $ ins s
  where ins NilT = RBT R NilT x NilT
        ins (RBT color a y b)
          | x <= y  = balance (RBT color (ins a) y b)
          | x >  y  = balance (RBT color a y (ins b))
        rootBlack (RBT _ a y b) = RBT B a y b

balance :: RBT a → RBT a
balance (RBT B (RBT R (RBT R a x b) y c) z d) = RBT R (RBT B a x b) y (RBT B c z d)
balance (RBT B (RBT R a x (RBT R b y c)) z d) = RBT R (RBT B a x b) y (RBT B c z d)
balance (RBT B a x (RBT R (RBT R b y c) z d)) = RBT R (RBT B a x b) y (RBT B c z d)
balance (RBT B a x (RBT R b y (RBT R c z d))) = RBT R (RBT B a x b) y (RBT B c z d)
balance (RBT color a x b) = RBT color a x b
```

1.19 Abstract Algebraic Types

בשפת Haskell אפשר להגדיר סוגים אלגבריים מופשטים. למשל נגדיר את הסוג של המספרים הטבעים $\mathbb{N}$.

קוד 1.71.

```haskell
{-# LANGUAGE UnicodeSyntax #-}
import Prelude hiding ((+), (-), (*), (%), (/), (<))    -- For Solve Name Conflict
data ℕ = 𝟘 | 𝕊 ℕ

(+) :: ℕ -> ℕ -> ℕ
(+) 𝟘 n = n
(+) (𝕊 m) n = 𝕊 (m + n)

(*) :: ℕ -> ℕ -> ℕ
(*) 𝟘 n = 𝟘
(*) (𝕊 m) n = n + (m * n)

(<) :: ℕ -> ℕ -> Bool
(<) 𝟘 𝟘 = False
(<) 𝟘 _ = True
(<) _ 𝟘 = False
(<) (𝕊 m) (𝕊 n) = m < n

(-) :: ℕ -> ℕ -> ℕ
(-) 𝟘 _ = 𝟘
(-) n 𝟘 = n
(-) (𝕊 m) (𝕊 n) = m - n

(%) :: ℕ -> ℕ -> ℕ
(%) m n | m < n = m
        | otherwise = (m - n) % n

(/) :: ℕ -> ℕ -> ℕ
(/) m n | m < n = 𝟘
        | otherwise = 𝕊 $ (m - n) / n

_0  = 𝟘 :: ℕ;      _1  = 𝕊 _0 :: ℕ
_2  = 𝕊 _1 :: ℕ;   _3  = 𝕊 _2 :: ℕ
_4  = 𝕊 _3 :: ℕ;   _5  = 𝕊 _4 :: ℕ
_6  = 𝕊 _5 :: ℕ;   _7  = 𝕊 _6 :: ℕ
_8  = 𝕊 _7 :: ℕ;   _9  = 𝕊 _8 :: ℕ
_10 = 𝕊 _9 :: ℕ
```

קוד 1.72.

```
instance Eq ℕ where
    (==) 𝕆 𝕆 = True
    (==) 𝕆 n = False
    (==) n 𝕆 = False
    (==) (𝕊 m) (𝕊 n) = m == n

instance Show ℕ where
    show n | n < _10 = dig_to_string n
           | otherwise = (show $ n / _10) ++ (dig_to_string $ n % _10)
          where
            dig_to_string :: ℕ -> String
            dig_to_string n | n == _0 = "0"
                            | n == _1 = "1"
                            | n == _2 = "2"
                            | n == _3 = "3"
                            | n == _4 = "4"
                            | n == _5 = "5"
                            | n == _6 = "6"
                            | n == _7 = "7"
                            | n == _8 = "8"
                            | n == _9 = "9"

𝕀 :: String -> ℕ
𝕀 s  = help $ reverse s
    where
        help :: String -> ℕ
        help "0" = _0
        help "1" = _1
        help "2" = _2
        help "3" = _3
        help "4" = _4
        help "5" = _5
        help "6" = _6
        help "7" = _7
        help "8" = _8
        help "9" = _9
        help (x:y) = (help [x]) + _10 * (help y)

main = do
    let x = (𝕊 (𝕊 𝕆))
    let y = 𝕀 "6"

    putStr $ "x = " ++ (show x) ++ ",\t\t\t\t"
    putStr "y = ";            print y
    putStr "x + y = ";        putStr $ (show $ x + y) ++ ",\t\t\t"
    putStr "x - y = ";        print  $ x - y
    putStr "x * y = ";        putStr $ (show $ x * y) ++ ",\t\t\t"
    putStr "x / y = ";        print  $ x / y
    putStr "y % x = ";        putStr $ (show $ y % x) ++ ",\t\t\t"
    putStr "x % y = ";        print  $ x % y
    putStr "y * y * y = ";    print  $ y * y * y
-----------------------------------------------
x = 2,                          y = 6
x + y = 8,                      x - y = 0
x * y = 12,                     x / y = 0
y % x = 0,                      x % y = 2
y * y * y = 216
```

2 Standard Library

2.1 Unicode

Installation

קוד 2.1.

```
cabal install base-unicode-symbols
cabal install --lib base-unicode-symbols
```

Preamble

קוד 2.2.

```
import Prelude.Unicode
 {-# LANGUAGE UnicodeSyntax #-}
```

2.2 List

Preamble

קוד 2.3.

```
import Data.List
```

2.3 Set

Preamble

קוד 2.4.

```
import Data.Set
```

2.4 Map

Preamble

קוד 2.5.

```
import Data.Map
```

2.5 Sequence

Preamble

קוד 2.6.

```
import Data.Sequence
```

סוג Seq מוגדר באופן דומה לסוג List, אך יש בו בנאי נוסף.

קוד 2.7.

```
data Seq a = Empty | a :<| (Seq a) | (Seq a) :|> a
```

- בנאי :<| : מוסיף איבר לתחילת הסדרה. O(1).
- בנאי :|> : מוסיף איבר לסוף הסדרה. O(1).

Constructors 2.5.1

קוד 2.8.

```
empty          :: Seq a                        -- O(1).
singleton      :: a -> Seq a                   -- O(1).
(<|)           :: a -> Seq a -> Seq a          -- O(1).
(|>)           :: Seq a -> a -> Seq a          -- O(1).
(><)           :: Seq a -> Seq a -> Seq a      -- O(log(min(n1, n2))).
-- Concatenate two sequences.
fromList       :: [a] -> Seq a                 -- O(n).
fromFunction   :: Int -> (Int -> a) -> Seq a -- O(n).
-- Convert a given sequence length and a function representing that sequence into a sequences.
replicate      :: Int -> a -> Seq a            -- O(log n).
-- replicate n x is a sequence consisting of n copies of x.
```

Folding 2.5.2

קוד 2.9.

```
null   :: Seq a -> Bool                          -- O(1).
-- Is this the empty sequence?
length :: Seq a -> Int                           -- O(1).
-- The number of elements in the sequences.
scanl  :: (a -> b -> a) -> a -> Seq b -> Seq a  -- O(n).
-- scanl is similar to foldl, but returns a sequence of reduced values from the left.
scanl1 :: (a -> a -> a) -> Seq a -> Seq a.      -- O(n).
-- scanl1 is a variant of scanl that has no starting value argument.
scanr  :: (a -> b -> b) -> b -> Seq a -> Seq b. -- O(n).
-- scanr is the right-to-left dual of scanl.
scanr1 :: (a -> a -> a) -> Seq a -> Seq a.      -- O(n).
-- scanr1 is a variant of scanr that has no starting value argument.
```

Filter and Sort 2.5.3

קוד 2.10.

```
partition :: (a -> Bool) -> Seq a -> (Seq a, Seq a)    -- O(n).
-- The partition function takes a predicate p and a sequence xs and returns sequences of those
    elements which do and do not satisfy the predicate.
filter    :: (a -> Bool) -> Seq a -> Seq a             -- O(n).
-- The filter function takes a predicate p and a sequence xs and returns a sequence of those
    elements which satisfy the predicate.
sort      :: Ord a => Seq a -> Seq a                   -- O(n log n).
-- sort sorts the specified Seq by the natural ordering of its elements. The sort is stable. If
    stability is not required, unstableSort can be slightly faster.
sortBy    :: (a -> a -> Ordering) -> Seq a -> Seq a.   -- O(n log n).
-- sortBy sorts the specified Seq according to the specified comparator.
sortOn    :: Ord b => (a -> b) -> Seq a -> Seq a.      -- O(n log n).
-- sortOn sorts the specified Seq by comparing the results of a key function applied to each
    element.
```

Indexing 2.5.4

קוד 2.11.

```
(!?)   :: Seq a -> Int -> Maybe a.          -- O(log(min(i, n-i))).
-- The element at the specified position.
index  :: Seq a -> Int -> a                 -- O(log(min(i, n-i))).
-- The element at the specified position.
update :: Int -> a -> Seq a -> Seq a        -- O(log(min(i, n-i))).
-- Replace the element at the specified position. If the position is out of range, the original
    sequence is returned.
adjust :: (a -> a) -> Int -> Seq a -> Seq a -- O(log(min(i, n-i))).
-- Update the element at the specified position
take :: Int -> Seq a -> Seq a               -- O(log(min(i, n-i))).
-- The first i elements of a sequence. If i is negative, take i s yields the empty sequence. If
    the sequence contains fewer than i elements, the whole sequence is returned.
drop :: Int -> Seq a -> Seq a               -- O(log(min(i, n-i))).
-- Elements of a sequence after the first i. If i is negative, drop i s yields the whole
    sequence. If the sequence contains fewer than i elements, the empty sequence is returned.
```

Map and Zip 2.5.5

קוד 2.12.

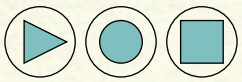
```
mapWithIndex :: (Int -> a -> b) -> Seq a -> Seq b                                  -- O(n).
reverse :: Seq a -> Seq a                                                          -- O(n).
zip       :: Seq a -> Seq b -> Seq (a, b)                                          -- O(n).
zip3      :: Seq a -> Seq b -> Seq c -> Seq (a, b, c)                              -- O(n).
zip4      :: Seq a -> Seq b -> Seq c -> Seq d -> Seq (a, b, c, d)                  -- O(n).
zipWith   :: (a -> b -> c) -> Seq a -> Seq b -> Seq c                              -- O(n).
zipWith3  :: (a -> b -> c -> d) -> Seq a -> Seq b -> Seq c -> Seq d                -- O(n).
zipWith4  :: (a -> b -> c -> d -> e) -> Seq a -> Seq b -> Seq c -> Seq d -> Seq e -- O(n).
unzip     :: Seq (a, b) -> (Seq a, Seq b)                                          -- O(n).
unzipWith :: (a -> (b, c)) -> Seq a -> (Seq b, Seq c)                              -- O(n).
```

3 Functor, Applicative, Monad

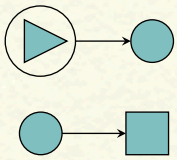
3.1 Container Data Type

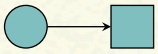
בפרק זה אנו נסמן **Data Types** כצורות גיאומטריות, למשל

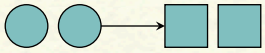
Container Data Type כצורוצ גיומטריות בתוך העיגול, למשל



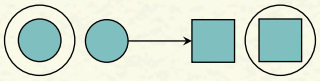
ופונקציות נסמן כצורוצ גיומטריות מחוברות ע"י חץ, למשל



העפלה של פונקציה מסוג  לאובייקט מסוג אפשר לתאר באופן הבא:



דיאגרמה זו, היא מספיק טבעית, במובן שסוג של הפונקציה מתאים לסוגים של הארגומנט והערך המוחזר.
מה לעשות במקרה שיש איהאתמה בין הסוגים? למשל איך לטפל במצב

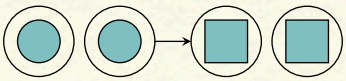


נתבונן בדוגמא הבאה:

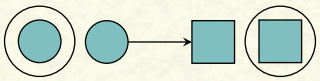
קוד 3.1.

```
foo :: Int -> Float
foo x = fromIntegral x / 2

maybe_foo :: Maybe Int -> Maybe Float
maybe_foo (Just x) = Just (foo x)
maybe_foo Nothing = Nothing
```

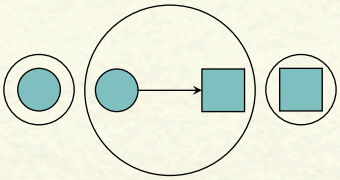
אנו רואים שאי אפשר להפעיל את הפונקציה foo ישירות על Container Type "Maybe Int" כדי להפעיל את הפונקציה זו יש בכתוב "מתאם" מסויים. בוא נחשוב אלו "מתאמים" אנו נצתרך, כדי לעבוד עם Container Types.



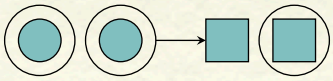
איור 2: אין צורך במתאם.



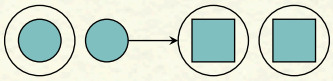
איור 3: מתאם - Functor.



איור 4: מתאם - Applicative.



איור 5: מתאם - Monad (return).



איור 6: מתאם - Monad (bind).

עכשיו נראה איך זה ממומש ב־Haskell.

קוד 3.2.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

דרישות:

קוד 3.3.

```
fmap id = id                                -- Identity
fmap (g . h) = (fmap g) . (fmap h)          -- Homomorphism
```

קוד 3.4.

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

דרישות:

קוד 3.5.

```
pure id <*> v = v                             -- Identity
pure f <*> pure x = pure (f x)                -- Homomorphism
u <*> pure y = pure ($ y) <*> u               -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

קוד 3.6.

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> k = m >>= \_ -> k
  return :: a -> m a
  return = pure
```

דרישות:

קוד 3.7.

```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

דוגמא 3.1.

למשל עלינו לכתוב פונקציה

```
mfact :: Int -> Maybe Int
```

היא המקבלת את המספר שלם n ואם הוא חיובי היא מחזיה את ה- $n!$, אחרת היא מחזירה את Nothing.
ניסוי ראשון:

```
mfact :: Int -> Maybe Int
mfact n | n < 0  = Nothing
        | n == 0 = Just 1
        | otherwise = n * mfact (n - 1)
```

בניסוי זה יש בעייה לא מוגדר כפל בין Int ו־ Maybe Int. אפשר לכוב אחרת:

קוד 3.8.

```
mfact :: Int -> Maybe Int
mfact n | n < 0  = Nothing
        | n == 0 = Just 1
        | otherwise = mfact (n - 1) >>= (\x -> Just (x * n))
```

פונקציה זו כבר עובדת, אך קוד לא "ידידותי". קוד קריא יותר אפשר לקבל עיי' Do Notation.

קוד 3.9.

```
mfact :: Int -> Maybe Int
mfact n | n < 0  = Nothing
        | n == 0 = Just 1
        | otherwise = do
            x <- mfact (n - 1)
            return $ n * x
```

נשים לב שעם Do Notation אנו עובדים תמיד בפונקציה main.
באופן כללי Do Notation מאפשרת לכתוב קוד ב־ Haskell בסיגנון אימפרטיבי. באופן פורמאלי:

- Sintax of Do Notation

קוד 3.10.

```
lexp    →      do { stmts }.                (do expression)
stmts   →      stmt1 … stmtn exp [;].       (n ⩾ 0)
stmt    →      exp        ;
            |  pat <- exp ;
            |  let decls  ;
            |             ;                 (empty statement)
```

- Translation of Do Notation

קוד 3.11.

```
do {e}.                  =       e
do {e; stmts}.           =       e >> do {stmts}
do {p <- e; stmts}       =   let ok p = do {stmts}
                                 ok _ = fail "..."
                                 in e >>= ok
do {let decls; stmts}    =   let decls in do {stmts}
```

שימו לב שאם בבלוק Do יש לפחות שני לא let ביטויים אם דורש Monad.

3.2 ST, STRef

סוג ST דו קיצור מ־ Transformer State. סוג זה מוגדר בספייה Control.Monad.ST באופן הבא

קוד 3.12.

```
data ST s a
```

אצל סוג זה אין בנאים (זו לא סוג אלגבריי), פרמטר s מייצג מצב מסויים ופרמטר a מייצג סוג כלשהו.
פונקציות בסיסיות של סוג זה הן:

קוד 3.13.

```
runST :: (forall s. ST s a) -> a
```

קוד 3.14.

```
-- Coercions between IO and ST

stToIO         :: ST RealWorld a -> IO a
ioToST         :: IO a -> ST RealWorld a
unsafeIOToST   :: IO a -> ST s a
unsafeSTToIO   :: ST s a -> IO a
```

בעזרת סוג זה אפשר להתמש ב־ Mutable Vareables ע"י סוג

קוד 3.15.

```
data STRef s a
```

המוגדר בספרייה Data.STRef.
סוג זה גם לא מכיל בנאים, פונקציות בסיסיות של סוג זה, הן:

קוד 3.16.

```
newSTRef     :: a -> ST s (STRef s a)
readSTRef    :: STRef s a -> ST s a
writeSTRef   :: STRef s a -> a -> ST s ()
```

דוגמא 3.2.

נתרגם מספת C, C++ קוד הבא ל- Haskell.

```
#include <stdio.h>

int Sum (int n) {
    int t = 0;
    for (int i = 0; i < n; i++) {
        t = t + i;
    }
}

int main() {
    printf("%d", Sum(5));
    return 0;
}
```

קוד 3.17.

```haskell
import Control.Monad.ST
import Data.STRef

stSum :: forall s. Int -> ST s Int
stSum n = do
    t <- newSTRef 0
    i <- newSTRef 0

    let loop :: Int -> ST s ()
        loop k | k == n     = return ()
               | otherwise = do
                   t_val <- readSTRef t
                   i_val <- readSTRef i

                   writeSTRef t (t_val + i_val)
                   writeSTRef i (i_val + 1)

                   loop (k + 1)
    loop 0
    readSTRef t

main = do
    print $ runST (stSum 5)
```

הערה 3.1.

שימו לב על תוספת `s. forall` היא מגדירה את ה- Scope של משתנה s. היא מסבירה לקומפיילר שבהגדרות

```haskell
stSum :: forall s. Int -> ST s Int
let loop :: Int -> ST s ()
```

יש אותו s. לגבי פקודה `forall` יש שני כללים:

- `forall` צריך להיות רק בפונקציה "חיצונית".
- אם `forall` מופיע בהגדרת הפונקציה, אז היא חייבת להופיע לכל מישתנים סוגיים.

בספרייה `Control.Monad` קיימת פונקציה `forM` בעזרתה אפשר לקצר את הקוד:

קוד 3.18.

```haskell
import Control.Monad.ST
import Data.STRef
import Control.Monad

stSum :: forall s. Int -> ST s Int
stSum n = do
    t <- newSTRef 0
    i <- newSTRef 0

    forM [0 .. (n - 1)] $ \i -> do
        t_val <- readSTRef t
        writeSTRef t (t_val + i)

    readSTRef t

main = do
    print $ runST (stSum 5)
```

באותה ספרייה יש גם הפונקציה when המתאימה למימוש של לולאה `while`. אתם מוזמנים לקרוא עליה, באופן עצמי.

3.3 Array, UArray, STArray, STUArray

שפת Haskell תומכת בכמה סוגים של מערכים.

- **Array** - Boxed Immutable Array
- **UArray** - Unboxed Immutable Array
- **STArray** - Boxed Mutable Array
- **STUArray** - Unboxed Mutable Array

Type של כל Immutable Array הוא שלישיה (a i e) כאשר

- a - סוג של מערך.
- i - סוג של אינדקסים.
- e - סוג של איברים.

כל מערכים האלה הם Instances של מחלקה:

קוד 3.19.

```haskell
class IArray a e
```

בחלקה זו מוגדרות פונקציות הבאות:

קוד 3.20.

```haskell
-- Constructs an immutable array from a pair of bounds and a list of
initial associations.

array   :: (IArray a e, Ix i)
        => (i, i)       -- bounds of the array: (lowest, highest)
        -> [(i, e)]     -- list of pairs (index, element)
        -> a i e
```

קוד 3.21.

```haskell
-- Constructs an immutable array from a list of initial elements.

listArray :: (IArray a e, Ix i) => (i, i) -> [e] -> a i e
```

קוד 3.22.

```haskell
-- Constructs an immutable array using a generator function.

genArray :: (IArray a e, Ix i) => (i,i) -> (i -> e) -> a i e
```

קוד 3.23.

```haskell
-- Returns the element of an immutable array at the specified index.

(!) :: (IArray a e, Ix i) => a i e -> i -> e
```

קוד 3.24.

```haskell
-- Returns 'Just' the element of an immutable array at the specified index,
-- or 'Nothing' if the index is out of bounds.

(!?) :: (IArray a e, Ix i) => a i e -> i -> Maybe e
```

קוד 3.25.

```
-- Returns a list of all the valid indices in an array.

indices :: (IArray a e, Ix i) => a i e -> [i]
```

קוד 3.26.

```
-- Returns a list of all the elements of an array, in the same order
-- as their indices.

elems :: (IArray a e, Ix i) => a i e -> [e]
```

קוד 3.27.

```
-- Returns the contents of an array as a list of pairs.

assocs :: (IArray a e, Ix i) => a i e -> [(i, e)]
```

קוד 3.28.

```
-- Takes an array and a list of pairs and returns an array identical to
-- the left argument except that it has been updated by the associations
-- in the right argument.

(//) :: (IArray a e, Ix i) => a i e -> [(i, e)] -> a i e
```

קוד 3.29.

```
-- Returns a new array derived from the original array by applying a
-- function to each of the elements.

amap :: (IArray a e', IArray a e, Ix i) => (e' -> e) -> a i e' -> a i e
```

קוד 3.30.

```
-- Returns a new array derived from the original array by applying a
-- function to each of the indices.

ixmap :: (IArray a e, Ix i, Ix j) => (i,i) -> (i -> j) -> a j e -> a i e
```

3.3.1 UArray

סוג זה מוגדר בספרייה Data.Array.Unboxed באופן הבא:

קוד 3.31.

```
data UArray i e
```

כאשר i הוא Index Type ו־ e הוא Elements Type.
מערכים האלה מוגדרים רק לסוגים הבאים:

קוד 3.32.

```
Int16, Int32, Int64, Int8, Int,
Word16, Word32, Word64, Word8, Word,
Bool,
Char,
Double, Float,
```

קוד 3.32. (המשך)

```
(FunPtr a), (Ptr a), (StablePtr a)
```

דוגמא 3.3.

```haskell
import Data.Array.Unboxed

data Mat a = IArray UArray a => Mat {
                                      arr  :: UArray Int a,
                                      rows :: Int,
                                      cols :: Int
                                    }

(<!>) :: IArray UArray a => Mat a -> (Int, Int) -> a
m <!> (i, j) = arr m ! (i * cols m + j)

fromLists :: IArray UArray a => [[a]] -> Mat a
fromLists l = Mat {arr = listArray (0, m * n - 1) (concat l), rows = m, cols = n}
     where
       m = length l
       n = length $ head l

instance Show a => Show (Mat a) where
    show :: forall a. Show a => Mat a -> String
    show Mat {arr = arr_, rows = m, cols = n} = toStr (elems arr_) m
       where
        toStr :: [a] -> Int -> String
        toStr l m | m == 1    = show l
                  | otherwise = show (take n l) ++ "\n" ++ toStr (drop n l) (m - 1)

trans :: IArray UArray a => Mat a -> Mat a
trans mat = Mat {arr = t_arr, rows = n, cols = m}
     where
       m     = rows mat
       n     = cols mat
       t_arr = ixmap (0, m * n - 1) indF (arr mat)

       indF :: Int -> Int
       indF i = c * n + r
          where
            r = i `div` m
            c = i `mod` m

(<+>) :: (IArray UArray a, Num a) => Mat a -> Mat a -> Mat a
mat1 <+> mat2 = Mat {arr = ar, rows = m, cols = n}
    where
      m = rows mat1
      n = cols mat1
      arr1 = arr mat1
      arr2 = arr mat2
      ar = listArray (0, m * n - 1) (map (\i -> (arr1 ! i) + (arr2 ! i)) [0 .. m * n - 1])

main :: IO ()
main = do
    let l   = [[1,2,3,4], [2,3,4,5], [3,4,5,6]] :: [[Float]]
    let mat = fromLists l
    print $ mat
    print $ trans mat
    print $ mat <+> mat
```

הערה 3.2.

נשים לב, שכדי לממש את אופרטור <+> היינו צריכים להשתמש ברשימות, לכן אנו לא קיבלנו performans טובה יותר מבמימוש את המטריצה כרשימה.

3.3.2 STUArray

בואו נסכם, מה היה חסר לנו Immutable Arrays.

- אין אפשרות לשנות (במקום) ערכים של איברי המערך.
- אין אפשלות להפתר מעבודה עם רשימות.
- אין קילים איטרטיביים (כמו במונדות).

פתרון הוא לשלב (ST s) עם UArray.

Type של כל Mmutable Array הוא שלישיה (a i e) כאשר

- a - סוג של מערך.
- i - סוג של אינדקסים.
- e - סוג של איברים.

כל מערכים האלה הם Instances של מחלקה:

קוד 3.33.

```
class Monad m => MArray a e m
```

מערכים האלה מוגדרים בספרייות Data.Array.ST, Data.Array.MArray.

סוגים של מערכים האלה:

קוד 3.34.

```
data STArray s i e
data STUArray s i e
```

בחלקה זו מוגדרות פונקציות הבאות:

קוד 3.35.

```
-- Builds a new array, with every element initialised to the supplied
-- value.

newArray    :: Ix i => (i, i) -> e -> m (a i e)
```

קוד 3.36.

```
-- Constructs a mutable array from a list of initial elements.
-- The list gives the elements of the array in ascending order
-- beginning with the lowest index.

newListArray :: (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
```

קוד 3.37.

```
-- Constructs a mutable array using a generator function.
-- It invokes the generator function in ascending order of the indices.

newGenArray :: (MArray a e m, Ix i) => (i, i) -> (i -> m e) -> m (a i e)
```

קוד 3.38.

```
-- Read an element from a mutable array

readArray :: (MArray a e m, Ix i) => a i e -> i -> m e
```

קוד 3.39.

```
-- Write an element in a mutable array

writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
```

קוד 3.39. (המשך)

קוד 3.40.

```
-- Modify an element in a mutable array

modifyArray :: (MArray a e m, Ix i) => a i e -> i -> (e -> e) -> m ()
```

קוד 3.41.

```
-- Return a list of all the elements of a mutable array

getElems :: (MArray a e m, Ix i) => a i e -> m [e]
```

קוד 3.42.

```
-- Return a list of all the associations of a mutable array, in index order.

getAssocs :: (MArray a e m, Ix i) => a i e -> m [(i, e)]
```

קוד 3.43.

```
-- Constructs a new array derived from the original array by applying a
-- function to each of the elements.

mapArray :: (MArray a e' m, MArray a e m, Ix i) => (e' -> e) -> a i e' -> m (a i e)
```

קוד 3.44.

```
-- Constructs a new array derived from the original array by applying a
-- function to each of the indices.

mapIndices :: (MArray a e m, Ix i, Ix j) => (i,i) -> (i -> j) -> a j e -> m (a i e)
```

קוד 3.45.

```
-- Converts a mutable array (any instance of 'MArray') to an
-- immutable array (any instance of 'IArray') by taking a complete copy of it.
freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
freezeSTUArray :: STUArray s i e -> ST s (UArray i e)
```

קוד 3.46.

```
-- A safe way to create and work with a mutable array before returning an immutable array
-- for later perusal.  This function avoids copying the array before returning it - it
-- uses 'unsafeFreeze' internally, but this wrapper is a safe interface to that function.

runSTArray :: (forall s . ST s (STArray s i e)) -> Array i e
```

קוד 3.47.

```
-- A safe way to create and work with an unboxed mutable array before returning an immutable
-- array for later perusal.  This function avoids copying the array before returning it - it
-- uses 'unsafeFreeze' internally, but this wrapper is a safe interface to that function.

runSTUArray :: (forall s . ST s (STUArray s i e)) -> UArray i e
```

קוד 3.47. (המשך)

דוגמא 3.4.

```haskell
import Data.Array.ST
import Control.Monad
import Control.Monad.ST

data Mat s a = MArray (STUArray s) a (ST s) => Mat {
                                                    arr  :: STUArray s Int a,
                                                    rows :: Int,
                                                    cols :: Int
                                                  }
{-# INLINE get #-}
get :: MArray (STUArray s) a (ST s) => Mat s a -> Int -> Int -> ST s a
get mat i j = readArray (arr mat) (i * cols mat + j)
{-# INLINE set #-}
set :: MArray (STUArray s) a (ST s) => Mat s a -> Int -> Int -> a -> ST s ()
set mat i j = writeArray (arr mat) (i * cols mat + j)

fromLists :: MArray (STUArray s) a (ST s) => [[a]] -> ST s (Mat s a)
fromLists l = do
    let m = length l
    let n = length $ head l
    ar <- newListArray (0, m * n - 1) (concat l)
    return Mat { arr = ar, rows = m, cols = n }

toStr :: forall s a. (MArray (STUArray s) a (ST s), Show a) => Mat s a -> ST s String
toStr Mat {arr = arr_, rows = m, cols = n} = do
    let toStr_ :: [a] -> Int -> String
        toStr_ l m | m == 1    = show l
                   | otherwise = show (take n l) ++ "\n" ++ toStr_ (drop n l) (m - 1)
    l <- getElems arr_
    return $ toStr_ l m

trans_mat :: (MArray (STUArray s) a (ST s), Num a) => Mat s a -> ST s (Mat s a)
trans_mat mat = do
    let m  = rows mat
    let n  = cols mat

    arr_t <- newArray (0, m * n - 1) 0
    let mat_t = Mat { arr = arr_t, rows = n, cols = m }

    forM [0 .. (m - 1)] $ \i -> do
        forM [0 .. (n - 1)] $ \j -> do
            x <- get mat i j
            set mat_t j i x
    return mat_t

(<+>) :: (MArray (STUArray s) a (ST s), Num a) => Mat s a -> Mat s a -> ST s (Mat s a)
mat1 <+> mat2 = do
    let m  = rows mat1
    let n  = cols mat1

    arr_ <- newArray (0, m * n - 1) 0
    let mat_ = Mat { arr = arr_, rows = m, cols = n }

    forM [0 .. (m - 1)] $ \i -> do
        forM [0 .. (n - 1)] $ \j -> do
            x <- get mat1 i j
            y <- get mat2 i j
            set mat_ i j (x + y)
    return mat_

```

דוגמא 3.4. (המשך)

```
main = do
    let l = [[1,2,3,4], [2,3,4,5], [3,4,5,6]] :: [[Float]]
    mat       <- stToIO $ fromLists l
    s_mat     <- stToIO $ toStr mat

    mat_t     <- stToIO $ trans_mat mat
    s_mat_t   <- stToIO $ toStr mat_t

    mat_sum   <- stToIO $ mat <+> mat
    s_mat_sum <- stToIO $ toStr mat_sum

    putStrLn s_mat;      putStrLn ""
    putStrLn s_mat_t;    putStrLn ""
    putStrLn s_mat_sum;  putStrLn ""
----------------------------------------------
[1.0,2.0,3.0,4.0]
[2.0,3.0,4.0,5.0]
[3.0,4.0,5.0,6.0]

[1.0,2.0,3.0]
[2.0,3.0,4.0]
[3.0,4.0,5.0]
[4.0,5.0,6.0]

[2.0,4.0, 6.0, 8.0]
[4.0,6.0, 8.0,10.0]
[6.0,8.0,10.0,12.0]
```

בואו נסכם, מה היה מסובך בקוד הזה.

- יש יותר מדי משחקים עם בסוגים.
- מפני שערך המוחזר של כל הפונקציות הוא מונדה, יש יותר מדי פעולות נוספות.

פתרון:

- להשתמש בפונקציות וסוגים לא פולימורפיים, זאת מפשט את הקוד מעלה את הביצעוים.
- לקבל ולהחזיר `UArray` ולהשתמש ב־ `STUArray` רק בתוך הפונקציות. זה מפשט את האינטרפייס של העבודה עם האובייקטים.

```haskell
import Data.Array.ST
import Control.Monad
import Data.Array.Unboxed

data Mat = Mat { arr  :: UArray Int Float, rows :: Int, cols :: Int }

instance Show Mat where
    show :: Mat -> String
    show Mat {arr = arr_, rows = m, cols = n} = toStr (elems arr_) m
        where
            toStr :: [Float] -> Int -> String
            toStr l m | m == 1    = show l
                      | otherwise = show (take n l) ++ "\n" ++ toStr (drop n l) (m - 1)

fromLists :: [[Float]] -> Mat
fromLists l = Mat {arr = listArray (0, m * n - 1) (concat l), rows = m, cols = n}
        where
            m = length l
            n = length $ head l

binOp :: (Float -> Float -> Float) -> Mat -> Mat -> Mat
binOp op mat1 mat2 = Mat { arr = arr_, rows = m, cols = n }
    where
        m  = rows mat1
        n  = cols mat1
        arr1 = arr mat1
        arr2 = arr mat2
        arr_ = binOp_ arr1 arr2

        binOp_ arr1 arr2 = runSTUArray $ do
            arr_ <- newArray (0, m * n - 1) 0

            forM [0 .. (m - 1)] $ \i -> do
                forM [0 .. (n - 1)] $ \j -> do
                    let x = ar1 ! (i * n + j)
                    let y = ar2 ! (i * n + j)
                    writeArray arr_ (i * n + j) (op x y)
            return arr_

uniOp :: (Float -> Float) -> Mat -> Mat
uniOp op mat = Mat { arr = arr_, rows = m, cols = n }
    where
        m    = rows mat
        n    = cols mat
        arr1 = arr mat
        arr_ = uniOp_ arr1

        uniOp_ arr1 = runSTUArray $ do
            arr_ <- newArray (0, m * n - 1) 0

            forM [0 .. (m * n - 1)] $ \i -> do
                    let x = arr1 ! i
                    writeArray arr_ i (op x)
            return arr_

instance Num Mat where
    mat1 + mat2 = binOp (+) mat1 mat2
    mat1 - mat2 = binOp (-) mat1 mat2
    mat1 * mat2 = binOp (*) mat1 mat2
    signum mat  = uniOp signum mat
    abs mat     = uniOp abs mat
    fromInteger mat = Mat { arr = listArray (0,0) [1.0], rows = 1, cols = 1 }

```

דוגמא 3.5. (המשך)

```
main = do
    let l1 = [[1,2,3,4], [2,3,4,5], [3,4,5,6]] :: [[Float]]
        l2 = [[1,2,5,4], [2,6,4,7], [1,1,5,1]] :: [[Float]]

        a = fromLists l1
        b = fromLists l2

        c = a + b
        d = a * b

    print a; putStrLn ""
    print b; putStrLn ""
    print c; putStrLn ""
    print d; putStrLn ""
-------------------------------------
[1.0,2.0,3.0,4.0]
[2.0,3.0,4.0,5.0]
[3.0,4.0,5.0,6.0]

[1.0,2.0,5.0,4.0]
[2.0,6.0,4.0,7.0]
[1.0,1.0,5.0,1.0]

[2.0,4.0,8.0, 8.0]
[4.0,9.0,8.0,12.0]
[4.0,5.0,10.0,7.0]

[1.0, 4.0,15.0,16.0]
[4.0,18.0,16.0,35.0]
[3.0, 4.0,25.0, 6.0]
```

HMatrix 3.4

Installation

קוד 3.48.

```
1.  Run msys2_shell.cmd from C:\ghcup\msys64
2.  pacman -Sy
3.  pacman -S make perl gcc-fortran
4.  pacman -S mingw-w64-x86_64-openblas
5.  pacman -S mingw-w64-x86_64-gsl
6.  pacman -S mingw-w64-x86_64-glpk
7.  Run cmd
8.  cabal update
9.  cabal install hmatrix --flag=openblas --extra-lib-dir=C:\ghcup\msys64\mingw64\lib
    --extra-include-dirs=C:\ghcup\msys64\mingw64\include
10. cabal install --lib hmatrix --flag=openblas --extra-lib-dir=C:\ghcup\msys64\mingw64\lib
    --extra-include-dirs=C:\ghcup\msys64\mingw64\include
11. Add C:\ghcup\msys64\mingw64\bin to PATH
```

Preamble

קוד 3.49.

```haskell
import Numeric.LinearAlgebra
import Numeric.LinearAlgebra.Data
import Numeric.LinearAlgebra.Devel
import Numeric.LinearAlgebra.HMatrix
import Numeric.LinearAlgebra.Static
```

PrimitiveTypes 3.4.1

קוד 3.50.

```haskell
type I = CInt
type Z = Int64
type R = Double
         Float
type C = Complex Double
         Complex Float
```

Constructors 3.4.2

קוד 3.51.

```haskell
vector :: [Double] -> Vector Double
(|>) :: Int -> [a] -> Vector a

vector [1 .. 5]
5 |> [1..]
------------------------------------
[1.0,2.0,3.0,4.0,5.0]

[1.0,2.0,3.0,4.0,5.0]
```

קוד 3.52.

```
range :: Int -> Vector I

range 5
------------------------------------
[0,1,2,3,4]
```

קוד 3.53.

```
idxs :: [Int] -> Vector I

l = [2,3,1,4] :: [Int]
l
:t l
v = idxs l
v
:t v
------------------------------------
[2,3,1,4]
l :: [Int]
[2,3,1,4]
v :: Vector I
```

Create a vector of indexes, useful for matrix extraction using (??)

קוד 3.54.

```
matrix :: Int -> [Double] -> Matrix Double -- (Int = number of rows)
(><) :: Int -> Int -> [a] -> Matrix a

matrix 5 [1 .. 15]
(2><3) [2, 4, 7 + 2 * iC, -3, 11, 0]
(2><3)[1..]

------------------------------------
(3><5)
 [  1.0,  2.0,  3.0,  4.0,  5.0
 ,  6.0,  7.0,  8.0,  9.0, 10.0
 , 11.0, 12.0, 13.0, 14.0, 15.0 ]

(2><3)
 [       2.0 :+ 0.0,  4.0 :+ 0.0, 7.0 :+ 2.0
 , (-3.0) :+ (-0.0), 11.0 :+ 0.0, 0.0 :+ 0.0 ]

(2><3)
 [ 1.0, 2.0, 3.0
 , 4.0, 5.0, 6.0 ]

[1.0,2.0,3.0,4.0,5.0]
```

קוד 3.55.

```
fromList :: [a] -> Vector a
toList :: Vector a -> [a]

fromLists :: [[a]] -> Matrix a
toLists :: Matrix a -> [[a]]

row :: [Double] -> Matrix Double
col :: [Double] -> Matrix Double
```

קוד 3.56.

```
flatten :: Matrix t -> Vector t
reshape :: Int -> Vector t -> Matrix t  -- Int = number of columns

asRow :: Vector a -> Matrix a.          --creates a 1-row matrix from a vector
asColumn :: Vector a -> Matrix a.       --creates a 1-col matrix from a vector

fromRows :: [Vector t] -> Matrix t
toRows :: Matrix t -> [Vector t]

fromColumns :: [Vector t] -> Matrix t
toColumns :: Matrix t -> [Vector t]

m = (3 >< 4) [1,2..]
m
flatten m

reshape 4 (fromList [1..12])
------------------------------------
(3><4)
 [ 1.0,  2.0,  3.0,  4.0
 , 5.0,  6.0,  7.0,  8.0
 , 9.0, 10.0, 11.0, 12.0 ]

[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0]

(3><4)
 [ 1.0,  2.0,  3.0,  4.0
 , 5.0,  6.0,  7.0,  8.0
 , 9.0, 10.0, 11.0, 12.0 ]
```

קוד 3.57.

```
konst :: a -> Int -> Vector a
konst :: a -> (Int, Int) -> Matrix a

konst 7 3 :: Vector Float
konst 2 (3,4) :: Matrix Float
-----------------------------------
[7.0,7.0,7.0]

(3><4)
 [ 2.0, 2.0, 2.0, 2.0
 , 2.0, 2.0, 2.0, 2.0
 , 2.0, 2.0, 2.0, 2.0 ]
```

קוד 3.58.

```
build :: Int -> (Int -> a) -> Vector a
build :: (Int, Int) -> (Int -> Int -> a) -> Matrix a

build 5 (**2) :: Vector Double
build 5 (\t -> 1 / (t + 1)) :: Vector Float

build (3, 4) (\i j -> i + j) :: Matrix Double
-----------------------------------
[0.0,1.0,4.0,9.0,16.0]

[1.0,0.5,0.33333334,0.25,0.2]

(3><4)
 [ 0.0, 1.0, 2.0, 3.0
 , 1.0, 2.0, 3.0, 4.0
 , 2.0, 3.0, 4.0, 5.0 ]
```

קוד 3.59.

```
assoc :: Int -> a -> [(Int, a)] -> Vector a
-- assoc length default_val [(ind, val) ... (ind, val)] :: Vector a
assoc :: (Int, Int) -> a -> [((Int Int), a)] -> Matrix a
-- assoc (rows, cols) default_val [((row, col), val) ... ((row, col), val)] :: Matrix a

assoc 5 0 [(3,7),(1,4)] :: Vector Double
assoc (2,3) 0 [((0,2), 7), ((1,0), 2 * iC - 3)] :: Matrix (Complex Double)

-----------------------------------
[0.0, 4.0, 0.0, 7.0, 0.0]

(2><3)
 [    0.0 :+ 0.0, 0.0 :+ 0.0, 7.0 :+ 0.0
 , (-3.0) :+ 2.0, 0.0 :+ 0.0, 0.0 :+ 0.0 ]
```

קוד 3.60.

```
accum :: Matrix a -> (a -> a -> a) -> [((Int Int), a)] -> Matrix a
-- accum matrix func [((row, col), val) ... ((row, col), val)] :: Matrix a
accum :: Vector a -> (a -> a -> a) -> [(Int, a)] -> Vector a
-- accum :: matrix func [(ind, val) ... (ind, val)] :: Vector a

accum (ident 5) (+) [((1,1),5),((0,3),3)] :: Matrix Double
accum (5 |> [1,2..]) (\x y -> x + 2 * y) [(1,5.0),(0,3.0)] :: Vector Double
----------------------------------
(5><5)
 [ 1.0, 0.0, 0.0, 3.0, 0.0
 , 0.0, 6.0, 0.0, 0.0, 0.0
 , 0.0, 0.0, 1.0, 0.0, 0.0
 , 0.0, 0.0, 0.0, 1.0, 0.0
 , 0.0, 0.0, 0.0, 0.0, 1.0 ]

[5.0,9.0,3.0,4.0,5.0]
```

קוד 3.61.

```
linspace Int -> (a, a) -> Vector a

linspace 5 (-3, 7) :: Vector Double
linspace 5 (8,2 + iC) :: Vector (Complex Double)
----------------------------------
[-3.0,-0.5,2.0,4.5,7.0]

[8.0 :+ 0.0,6.5 :+ 0.25,5.0 :+ 0.5,3.5 :+ 0.75,2.0 :+ 1.0]
```

קוד 3.62.

```
ident :: Int -> Matrix a                            -- Creates the identity matrix of
                                                    -- given dimension.
diag :: Vector a -> Matrix a                        -- Creates a square matrix with a
                                                    -- given diagonal.
diagl :: [Double] -> Matrix Double                  -- Create a real diagonal matrix
                                                    -- from a list.
diagRect :: a -> Vector a -> Int -> Int -> Matrix a -- Creates a rectangular diagonal
                                                    -- matrix.
takeDiag :: Matrix a -> Vector a                    -- Extracts the diagonal from a
                                                    -- rectangular matrix.

ident 3 :: Matrix Double
diag (fromList [1,2,3]) :: Matrix Float
diagl ([1,2,3])
diagRect 7 (fromList [10,20,30]) 4 5 :: Matrix Double
takeDiag (ident 3)
-----------------------------------
(3><3)
 [ 1.0, 0.0, 0.0
 , 0.0, 1.0, 0.0
 , 0.0, 0.0, 1.0 ]

 (3><3)
 [ 1.0, 0.0, 0.0
 , 0.0, 2.0, 0.0
 , 0.0, 0.0, 3.0 ]

 (3><3)
 [ 1.0, 0.0, 0.0
 , 0.0, 2.0, 0.0
 , 0.0, 0.0, 3.0 ]

 (4><5)
 [ 10.0,  7.0,  7.0, 7.0, 7.0
 ,  7.0, 20.0,  7.0, 7.0, 7.0
 ,  7.0,  7.0, 30.0, 7.0, 7.0
 ,  7.0,  7.0,  7.0, 7.0, 7.0 ]

[1.0,1.0,1.0]
```

Indexing and Sizes 3.4.5

קוד 3.63.

```
size :: Matrix a -> (Int, Int)
size :: Vector a -> Int
rows :: Matrix a -> Int
cols :: Matrix a -> Int

size $ (2><5)[1..10]
size $ vector [1,2,3]
rows $ (2><5)[1..10]
cols $ (2><5)[1..10]
------------------------------------
(2,5)
3
2
5
```

קוד 3.64.

```
(!) :: Vector a -> Int -> a
(!) :: Matrix a -> Int -> Vector a

v = vector [1,2,3] :: Matrix Float
m = (2><5)[1..10] :: Matrix Double
v ! 1
m ! 0
m ! 0 ! 2
------------------------------------
2.0
[1.0,2.0,3.0,4.0,5.0]
3.0
```

קוד 3.65.

```
atIndex :: Vector a -> Int -> a
atIndex :: Matrix a -> (Int, Int) -> a

v = vector [1,2,3] :: Matrix Float
m = (2><5)[1..10] :: Matrix Double
v `atIndex` 1
m `atIndex` 0
m `atIndex` (0, 2)
------------------------------------
2.0
[1.0,2.0,3.0,4.0,5.0]
3.0
```

הערה 3.3.

שימו לב שעבוק גישה לאיברי המטריצה יש להגיר את סוג שלה במפורט.

Vector Extractors 3.4.6

קוד 3.66.

```
subVector :: Int -> Int -> Vector a -> Vector a
takesV :: [Int] -> Vector a -> [Vector a]  --Extract consecutive subvectors of the given sizes.
vjoin :: [Vector t] -> Vector t            --Concatenate a list of vectors

subVector 2 3 (fromList [1..10])
lst = takesV [5,3,2,3] (100 |> [1,2..])
vjoin lst
------------------------------------
[3.0,4.0,5.0]

[[1.0,2.0,3.0,4.0,5.0], [6.0,7.0,8.0], [9.0,10.0], [11.0,12.0,13.0]]

[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0]
```

Matrix Extractors 3.4.7

קוד 3.67.

```
(??) :: Matrix a -> (Extractor, Extractor) -> Matrix a

--Extractor is a:
     All
     Range Int Int Int
     Pos (Vector I)
     Take Int
     TakeLast Int
     Drop Int
     DropLast Int

m = (4><5)
 [  0,  1,  2,  3,  4
 ,  5,  6,  7,  8,  9
 , 10, 11, 12, 13, 14
 , 15, 16, 17, 18, 19 ]

 m ?? (Take 3, DropLast 2)
m ?? (Pos (idxs[2,1]), All)
m ?? (Pos (idxs [1, 2]), Range 4 (-2) 0)
------------------------------------
(3><3)
 [  0,  1,  2
 ,  5,  6,  7
 , 10, 11, 12 ]

 (2><5)
 [ 10, 11, 12, 13, 14
 ,  5,  6,  7,  8,  9 ]

 (2><3)
 [  9.0,  7.0,  5.0
 , 14.0, 12.0, 10.0 ]
```

קוד 3.68.

```
(?) :: Matrix a -> [Int] -> Matrix a    -- extract rows
(¿) :: Matrix a -> [Int] -> Matrix a    -- extract columns,    ¿ = Alt + Shift + ?
subMatrix :: (Int, Int) -> (Int, Int) -> Matrix a -> Matrix a

(20><4) [1..] ? [2,1,1]
(3><4) [1..] ¿ [3,0]

 m = (4><5)
 [  0,  1,  2,  3,  4
 ,  5,  6,  7,  8,  9
 , 10, 11, 12, 13, 14
 , 15, 16, 17, 18, 19 ]

 subMatrix (2, 3) (2, 2) m
------------------------------------
(3><4)
 [ 9.0, 10.0, 11.0, 12.0
 , 5.0,  6.0,  7.0,  8.0
 , 5.0,  6.0,  7.0,  8.0 ]

(3><2)
 [  4.0, 1.0
 ,  8.0, 5.0
 , 12.0, 9.0 ]

 (2><2)
 [ 13.0, 14.0
 , 18.0, 19.0 ]
```

קוד 3.69.

```
takeRows    :: Int -> Matrix a -> Matrix a
dropRows    :: Int -> Matrix a -> Matrix a
takeColumns :: Int -> Matrix a -> Matrix a
dropColumns :: Int -> Matrix a -> Matrix a
remap :: Matrix I -> Matrix I -> Matrix a -> Matrix a
-- (matrix of rows) -> (matrix of cols) -> Matrix a -> Matrix a

r =
(3><3)
 [ 1, 1, 1
 , 1, 2, 2
 , 1, 2, 3 ]

c =
(3><3)
 [ 0, 1, 5
 , 2, 2, 1
 , 4, 4, 1 ]
m =
(4><6)
 [  0,  1,  2,  3,  4,  5
 ,  6,  7,  8,  9, 10, 11
 , 12, 13, 14, 15, 16, 17
 , 18, 19, 20, 21, 22, 23 ]


remap r c m

-- The indexes are autoconformable.

c1
(3><1)
 [ 1
 , 2
 , 4 ]

 remap r c' m
------------------------------------
(3><3)
 [  6,  7, 11
 ,  8, 14, 13
 , 10, 16, 19 ]

(3><3)
 [  7,  7,  7
 ,  8, 14, 14
 , 10, 16, 22 ]
```

Block matrix 3.4.8

קוד 3.70.

```
fromBlocks     :: [[Matrix a]] -> Matrix a
(|||)          :: Matrix t -> Matrix t -> Matrix t -- Horizontal concatenation
(===)          :: Matrix t -> Matrix t -> Matrix t -- Vertical concatenation
diagBlock      :: [Matrix a] -> Matrix a
repmat         :: Matrix a -> Int -> Int -> Matrix a
toBlocks       :: [Int] -> [Int] -> Matrix a -> [[Matrix a]]
-- Partition a matrix into blocks with the given numbers of rows and columns. The remaining rows
    and columns are discarded.
toBlocksEvery :: Int -> Int -> Matrix a -> [[Matrix a]]
-- Fully partition a matrix into blocks of the same size. If the dimensions are not a multiple
    of the given size the last blocks will be smaller.

fromBlocks [[ident 5, 7, row[10,20]], [3, diagl [1,2,3], 0]]
repmat (ident 2) 2 3
------------------------------------
(8><10)
 [ 1.0, 0.0, 0.0, 0.0, 0.0, 7.0, 7.0, 7.0, 10.0, 20.0
 , 0.0, 1.0, 0.0, 0.0, 0.0, 7.0, 7.0, 7.0, 10.0, 20.0
 , 0.0, 0.0, 1.0, 0.0, 0.0, 7.0, 7.0, 7.0, 10.0, 20.0
 , 0.0, 0.0, 0.0, 1.0, 0.0, 7.0, 7.0, 7.0, 10.0, 20.0
 , 0.0, 0.0, 0.0, 0.0, 1.0, 7.0, 7.0, 7.0, 10.0, 20.0
 , 3.0, 3.0, 3.0, 3.0, 3.0, 1.0, 0.0, 0.0,  0.0,  0.0
 , 3.0, 3.0, 3.0, 3.0, 3.0, 0.0, 2.0, 0.0,  0.0,  0.0
 , 3.0, 3.0, 3.0, 3.0, 3.0, 0.0, 0.0, 3.0,  0.0,  0.0 ]

 (4><6)
 [ 1.0, 0.0, 1.0, 0.0, 1.0, 0.0
 , 0.0, 1.0, 0.0, 1.0, 0.0, 1.0
 , 1.0, 0.0, 1.0, 0.0, 1.0, 0.0
 , 0.0, 1.0, 0.0, 1.0, 0.0, 1.0 ]
```

Zip and Map 3.4.9

קוד 3.71.

```
conj :: Vector a -> Vector a
conj :: Matrix a -> Matrix a

cmap :: (a -> b) -> Vector a -> Vector b
cmap :: (a -> b) -> Matrix a -> Matrix b

zipVectorWith :: (a -> b -> c) -> Vector a -> Vector b -> Vector c

mapVectorWithIndex :: (Int -> a -> b) -> Vector a -> Vector b
mapMatrixWithIndex :: ((Int, Int) -> a -> b) -> Matrix a -> Matrix b
```

Sort and Find 3.4.10

קוד 3.72.

```
find :: (a -> Bool) -> Vector a -> [Int]
maxIndex   :: Vector a -> Int
minIndex   :: Vector a -> Int
maxElement :: Vector a -> a
minElement :: Vector a -> a
sortVector :: Vector a -> Vector a
sortIndex  :: Vector a -> Vector Int

find       :: (a -> Bool) -> Matrix a -> [(Int, Int)]
maxIndex   :: Matrix a -> (Int, Int)
minIndex   :: Matrix a -> (Int, Int)
maxElement :: Matrix a -> a
minElement :: Matrix a -> a
```

IO 3.4.11

קוד 3.73.

```
disp :: Int -> Matrix Double -> IO ()
-- Print a real matrix with given number of digits after the decimal point
loadMatrix :: String -> IO (Matrix Double) -- String = Path to file
-- load a matrix from an ASCII file formatted as a 2D table.
saveMatrix :: String -> String -> Matrix Double -> IO ()
-- First String = Path to file
-- Secon String = "printf" format (e.g. "%.2f", "%g", etc.)
-- Save a matrix as a 2D ASCII table

disp 5 $ ident 2 / 3
------------------------------------
2x2
0.33333  0.00000
0.00000  0.33333
```

קוד 3.74.

```

+, -, *, /, **   :: Vector a -> Vector a -> Vector a
+, -, *, /, **   :: Vector a -> a -> Vector a
+, -, *, /, **   :: a -> Vector a -> Vector a

+, -, *, /, **   :: Matrix a -> Matrix a -> Matrix a
+, -, *, /, **   :: Matrix a -> a -> Matrix a
+, -, *, /, **   :: a -> Matrix a -> Matrix a

-- One or both matrices can consist of one row or one column.

log, exp, sin, cos,...   :: Vector a -> Vector a -> Vector a
log, exp, sin, cos,...   :: Vector a -> a -> Vector a
log, exp, sin, cos,...   :: a -> Vector a -> Vector a

log, exp, sin, cos,...   :: Matrix a -> Matrix a -> Matrix a
log, exp, sin, cos,...   :: Matrix a -> a -> Matrix a
log, exp, sin, cos,...   :: a -> Matrix a -> Matrix a

-- One or both matrices can consist of one row or one column.
```

קוד 3.75.

```

dot   :: Vector a -> Vector a -> a -- Scalar product of vectors.
(<.>) :: Vector a -> Vector a -> a -- Scalar product of vectors.

vector [1,2,3,4] <.> vector [-2,0,1,1]
------------------------------------
5.0
```

קוד 3.76.

```haskell

(#>) :: Matrix a -> Vector a -> Vector a -- Matrix-Vector product.
(<#) :: Vector a -> Matrix a -> Vector a -- Vector-Matrix product.
(<>) :: Matrix a -> Matrix a -> Matrix a -- Matrix-Matrix product.

m =
(2><3)
 [ 1.0, 2.0, 3.0
 , 4.0, 5.0, 6.0 ]

v = vector [10,20,30]

a =
(3><5)
 [  1.0,  2.0,  3.0,  4.0,  5.0
 ,  6.0,  7.0,  8.0,  9.0, 10.0
 , 11.0, 12.0, 13.0, 14.0, 15.0 ]

b
(5><2)
 [  1.0, 3.0
 ,  0.0, 2.0
 , -1.0, 5.0
 ,  7.0, 7.0
 ,  6.0, 0.0 ]

m #> v
a <> b
------------------------------------
[140.0,320.0]

(3><2)
 [  56.0,  50.0
 , 121.0, 135.0
 , 186.0, 220.0 ]
```

קוד 3.77.

```haskell
(<\>) :: Matrix a -> Vector a -> Vector a -- Solution of system Ax = b.
(<\>) :: Matrix a -> Matrix a -> Matrix a -- Solution of system AX = B.
```

קוד 3.78.

```haskell
inv       :: Matrix t -> Matrix t -- Inverse of a square matrix
pinv      :: Matrix t -> Matrix t -- Pseudoinverse of a matrix
rank      :: Matrix t -> Int.     -- Number of linearly independent rows or columns.
det       :: Matrix t -> t.       -- Determinant of a square matrix.
orth      :: Matrix t -> Matrix t -- Orthonormal basis of the range space of a matrix.
nullspace :: Matrix t -> Matrix t -- Orthonormal basis of the null space of a matrix.
svd       :: Matrix t -> (Matrix t, Vector Double, Matrix t) -- SVD factorization.
eig       :: Matrix t -> (Vector (Complex Double), Matrix (Complex Double))
-- Eigenvalues (not ordered) and eigenvectors (as columns) of a general square matrix.
qr        :: Matrix t -> (Matrix t, Matrix t) -- QR factorization.
lu        :: Matrix t -> (Matrix t, Matrix t, Matrix t, t)
-- Explicit LU factorization of a general matrix.
schur     :: Matrix t -> (Matrix t, Matrix t) -- Schur factorization.
expm      :: Matrix t -> Matrix t -- Matrix exponential.
sqrtm     :: Matrix t -> Matrix t -- Matrix square root.
```

קוד 3.79.

```
rand  :: Int -> Int -> IO (Matrix Double) -- Pseudorandom matrix with uniform elements between 0
    and 1.
randn :: Int -> Int -> IO (Matrix Double) -- Pseudorandom matrix with normal elements.
```

Data.Time.Clock 3.5

קוד 3.80.

```
getCurrentTime -- Current time in miliseconds.
diffUTCTime    -- Difference of times.

import Data.Time.Clock
main = do
     tt_start <- getCurrentTime
     tt_end <- getCurrentTime
     print (diffUTCTime tt_end tt_start)
```