Cursul 5

- Noi tipuri: Enumerări
- Sinonime pentru tipuri
- Introducere noi tipuri
- Construcția sintactică **do**
- Acțiuni IO
- Grafică

Definirea de noi tipuri de date

- Enumerări:
  - Enumerarea explicită a valorilor tipului
  - Tipul Bool este introdus prin enumerare
- Exemple:

```
data Triunghi = Esec|Isoscel|Echilateral|Scalen
analiza::(Int, Int, Int) -> Triunghi
analiza(x,y,z) | x+y<=z          = Esec
               | x==z            = Echilateral
               | (x==y)||(x==z)  = Isoscel
               | otherwise       = Scalen
```

  - Funcția **analiza** este corectă pentru x<=y<=z.

Enumerări

- Exemple:

```
data Zi = Lu|Ma|Mi|Jo|Vi|Sa|Du
```

  - Tipul Zi are 8 valori, cele enumerate plus "bottom"
  - Cele 7 constante se numesc constructorii tipului Zi
  - Constructorii unui tip trebuie să înceapă cu literă mare la fel ca şi numele unui nou tip

Enumerări

- Elementele unui tip enumerare pot fi comparate dacă se declară tipul ca fiind instanță a claselor Eq şi Ord
- Exemplu:

```
data Bool = False|True
instance Eq Bool where
(x==y) = (x&&y)||(`not` x && `not` y)
(x/=y) = `not`(x==y)
```

- Pentru enumerările cu număr mare de valori (ca Zi) nu este convenabilă declararea ca instanță în acest mod (pentru definirea operatorilor == şi /=)

Enumerări

- Soluția: o clasă Enum care are ca metodă transformarea valorilor în întregi şi compararea întregilor:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

Enumerări

- Declarăm tipul Zi ca fiind instanță a clasei Enum:

```
instance Enum Zi where
    fromEnum Du = 0
    fromEnum Lu = 1
    fromEnum Ma = 2
    fromEnum Mi = 3
    fromEnum Jo = 4
    fromEnum Vi = 5
    fromEnum Sa = 6
```

- Declarăm Zi instanță a claselor Eq şi Ord:

```
instance Eq Zi where
    (x==y)=(fromEnum x == fromEnum y)
instance Ord Zi where
    (x<y)=(fromEnum x < fromEnum y)
```

Enumerări - optimizarea definiției

- Declararea automata ca instanță a unor clase (**deriving**); sistemul generează metodele clasei pentru această instanță:

```
data Zi = Du|Lu|Ma|Mi|Jo|Vi|Sa
     deriving (Eq, Ord, Enum, Show)

zilucr::Zi -> Bool
zilucr d = (Lu <= d) && (d <= Vi)

maine:: Zi -> Zi
maine d = toEnum((fromEnum d + 1) `mod` 7)
```

Enumerări

```
Main> maine Lu
Ma
(40 reductions, 113 cells)
Main> zilucr Mi
True
(47 reductions, 91 cells)
Main> zilucr Du
False
(36 reductions, 61 cells)
Main> maine Du
Lu
(36 reductions, 51 cells)
```

Tipuri sinonime

- Sintaxa: `type Nume_nou = tip`
- Determinarea rădăcinilor unei ecuații de gradul 2:

```
radacini :: (Float, Float, Float) -> (Float, Float)
```

- Alternativă:

```
type Coef = (Float, Float, Float)
type Radacini = (Float, Float)
radacini :: Coef -> Radacini

type PozitieInPlan = (Float, Float)
type Unghi = Float
type Distanta = Float
type Pereche = (a, a)
type Automorfism = a -> a
```

Tipuri noi

- Tipurile sinonime nu sunt tipuri noi: metodele pentru acest tip sunt cele de la tipul pe care-l numeşte
- Uneori este necesar a schimba înțelesul unor metode: două unghiuri sunt egale dacă ele sunt egale modulo 2nπ.
- Soluția: tip nou şi nu sinonim:

```
data Unghi = MkUnghi Float
instance Eq Unghi where
        MkUnghi x == MkUnghi y = norm x == norm y

norm :: Float -> Float
norm x =          | x < 0          = norm(x + per)
                  | x >= per       = norm(x - per)
                  | otherwise = x
                  where per = 2*pi
```

"Side Effects" în Haskell

- Programele de până acum nu au "side-effects": programele sunt executate pentru a afla nişte valori care se afişează: programele modelează niște funcții care au niște argumente și produc niște rezultate

- Cum construim programe interactive? (citirea unor valori din fișier (în particular de la tastatură), scrierea unor valori în fişiere (sau la ecran), desenarea unei figuri etc.)

- La prima vedere acest lucru este imposibil în stilul funcțional pentru că ar trebui să avem de-a face cu "side-effects"

- Soluția: tipul input/output

"Side Effects" în Haskell

- Un program interactiv este conceput ca o funcție care are ca argument "state of the world" (o valoare de tip World) și produce o valoare de tip World care este o altă stare ce reflectă efectul colateral produs de program:

```
type IO = World -> World
```

- În general, un program interactiv poate returna și o anume valoare pe lângă efectul ce-l are asupra stării (de exemplu, se citește un caracter de la tastatură (acțiune) și se returnează acel caracter. Așadar, mai corect:

```
type IO a = World -> (a, World)
```

Exemplu: `IO Char, IO Int; IO ()`

- O expresie de tip `IO a` are asociate execuției sale *acțiuni,* ultima dintre ele fiind returnarea unei valori de tip `a`.

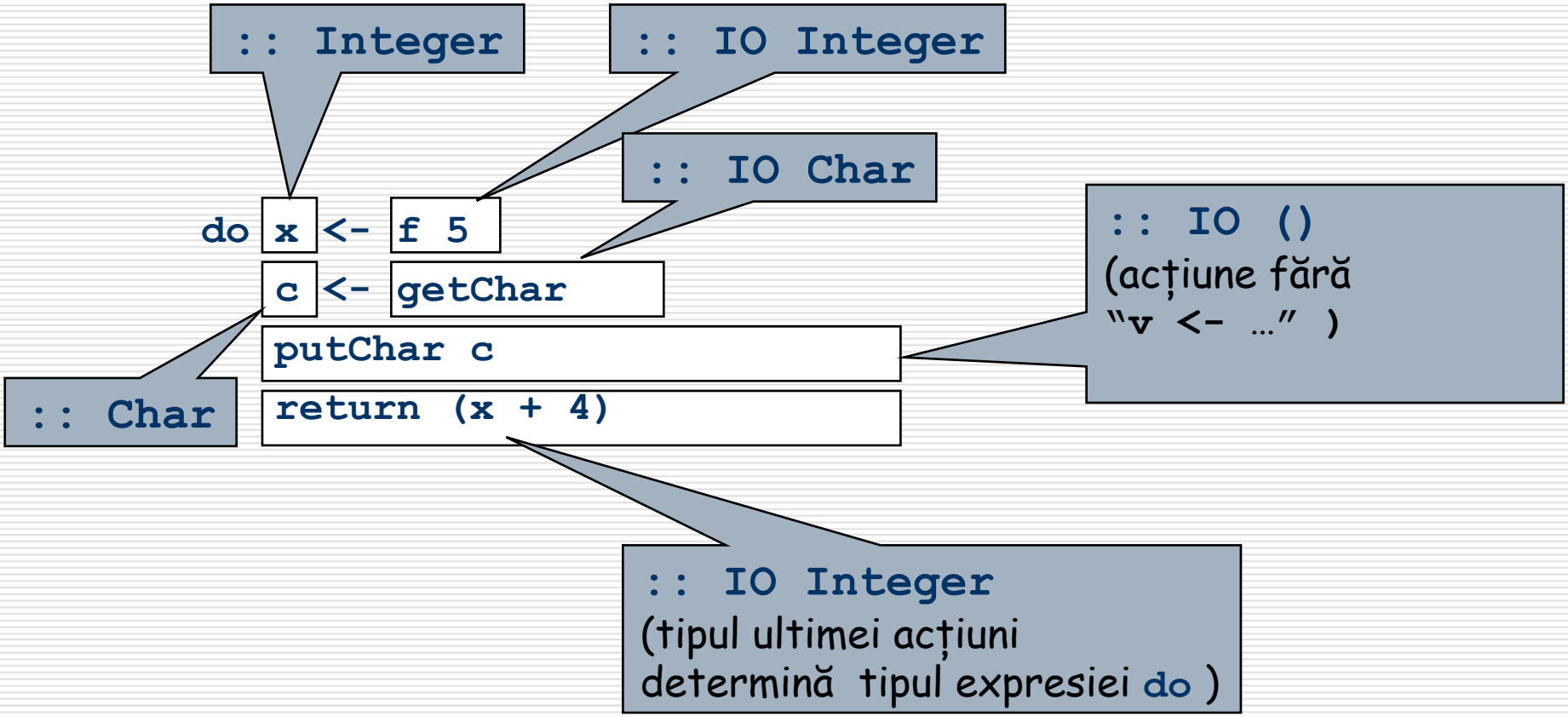
Construcția sintactică do

- Dacă `act` este o acțiune de tip `IO a` atunci putem descrie realizarea acțiunii `act`, returna valoarea corespunzătoare şi eventual adăugarea altor acțiuni, astfel:

```
do val <- act
   ...          -- actiunea urmatoare
   ...          -- actiunea urmatoare
   return x     -- actiunea finală
```

- Toate acțiunile de după `val <- act` pot folosi `val`.
- Funcția `return` are un parametru de tip `a`, devine o acțiune de tip `IO a`, care nu face altceva decât să returneze valoarea respectivă.

do Exemplu


:: Integer
:: IO Integer
:: IO Char
do x <- f 5
c <- getChar
putChar c
return (x + 4)
:: Char
:: IO ()
(acțiune fără
"v <- …")
:: IO Integer
(tipul ultimei acțiuni
determină tipul expresiei do)

Acțiuni IO

- O valoare de tip `IO a` este o acțiune; această valoare va avea efect atunci când va fi executată.

- În Haskell, o valoare program este valoarea variabilei `main` din modulul `Main`. Dacă această valoare este de tip `IO a`, atunci va fi executată ca o acțiune. Dacă este de alt tip atunci valoarea sa va fi afişată.

Acțiuni IO predefinite

- Introducerea unui caracter de la tastatură

```
getChar :: IO Char
```

- Scrie un caracter la terminal

```
putChar :: Char -> IO ()
```

- Introducerea unei linii de la tastatură

```
getLine :: IO String
```

- Citirea unui fişier ca şi String

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

- Scriere String în fişier

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

Acțiuni recursive

- Acțiunea `getLine` poate fi definită recursiv din acțiuni mai simple:

```
getLine :: IO String
getLine =  do c <- getChar          -- citeste un caracter
    if c == '\n'                    -- daca este newline
       then return []               -- return sirul vid
       else do l <- getLine         -- recursiv restul liniei
               return (c:l)         -- return intreaga linie
```

Acțiuni recursive

- Acțiunea `putStr` poate fi definită recursiv din acțiuni mai simple:

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

putStr[]      = return ()
putStr (x:xs) = do putChar x
                   putStr xs

putStrLn:: String -> IO ()

putStrLn xs   = do putStr xs
                   putChar '\n'
```

Exemplul 1: Lungimea unui șir citit

- Acțiunea de citire a unui șir de la tastatură și afișarea lungimii sale:

```
strlen:: IO()
strlen = do putStr "Introdu un sir: "
            xs <- getLine
            putStr " Sirul are "
            putStr(show(length(xs))
            putStrLn " caractere"

Main> strlen
Introdu un sir: abracadabra
Sirul are 11 caractere
```

Exemplul 2: Suma unor intregi

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)


sumInts :: IO Int
sumInts = do n <- getInt
             if n==0
                then return 0
                else
                  do m <- sumInts
                     return (n+m)
suma::IO ()
suma = do x <- sumInts
          putStrLn("Suma este: " ++ show x )
```

```
Main> suma
0
Suma este: 0


Main> suma
354
24
100
0
Suma este: 478
```

Exemplul 3: Comanda Unix wc

- Programul Unix wc (word count) citeşte un fişier şi determină numărul de caractere, cuvinte şi linii pe care le afişează
- Citirea fişierului este o acțiune, restul este un calcul.
- Strategia:
  - Se defineşte o funcție care determină numărul de caractere, cuvinte şi linii într-un string.
    - Numărul de linii = numărul de '\n'
    - Numărul de cuvinte ~= numărul de ' ' plus numărul de '\t'
  - Se sefineşte o actiune care citeşte un fişier într-un string, aplică funcția, apoi printează rezultatul.

Implementarea

```
wcf :: (Int,Int,Int) -> String -> (Int,Int,Int)
wcf (cc,w,lc) []           = (cc,w,lc)
wcf (cc,w,lc) (' '  : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\t' : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\n' : xs) = wcf (cc+1,w+1,lc+1) xs
wcf (cc,w,lc) (x : xs)    = wcf (cc+1,w,lc) xs

wc :: IO ()
wc = do name     <- getLine
        contents <- readFile name
        let (cc,w,lc) = wcf (0,0,0) contents
        putStrLn ("Fisierul: " ++ name ++ " are ")
        putStrLn (show cc ++ " caractere ")
        putStrLn (show w  ++ " cuvinte ")
        putStrLn (show lc ++ " linii ")
```

Exemplu execuție

```
Main> wc
wcount.hs
Fisierul: wcount.hs are
579 caractere
166 cuvinte
16 linii
```

Acțiuni grafice

- Ferestrele grafice sunt realizate prin comenzi specifice – acțiuni grafice
- Acțiuni grafice:
  - Deschiderea unei ferestre grafice
  - Închiderea unei ferestre grafice
  - Desenarea de linii, cercuri etc.
  - Includere text
- Utilizarea bibliotecii grafice **Graphics.HGL** :

```
import Graphics.HGL
```

Operatori grafici

- Deschiderea unei ferestre ce are un nume(string) şi o dimensiune:

```
openWindow :: String -> Point -> IO Window
```

- Afişearea unei valori de tip **Graphic** într-o fereastră:

```
drawInWindow :: Window -> Graphic -> IO ()
```

- Aşteată tastarea unei key şi returnează caracterul corespunzător:

```
getKey :: Window -> IO Char
```

- Închide fereastra:

```
closeWindow :: Window -> IO ()
```

Programul "Hello World"

```
import Graphics.HGL
main0 =
 runGraphics $
    do w <- openWindow "Prima fereastra" (400,400)
       drawInWindow w (text (100,200) "Hello world!")
       k <- getKey w
       closeWindow w


Main> main0
```

Rezultat execuție `Main> main0`

main1

```
spaceClose :: Window -> IO ()
spaceClose w =
    do k <- getKey w
       if k == ' ' then closeWindow w
                   else spaceClose w
main1 =
  runGraphics $
    do w <- openWindow "A doua fereastra" (300,300)
       drawInWindow w (text (100,200) "Hello Again!")
       spaceClose w
```

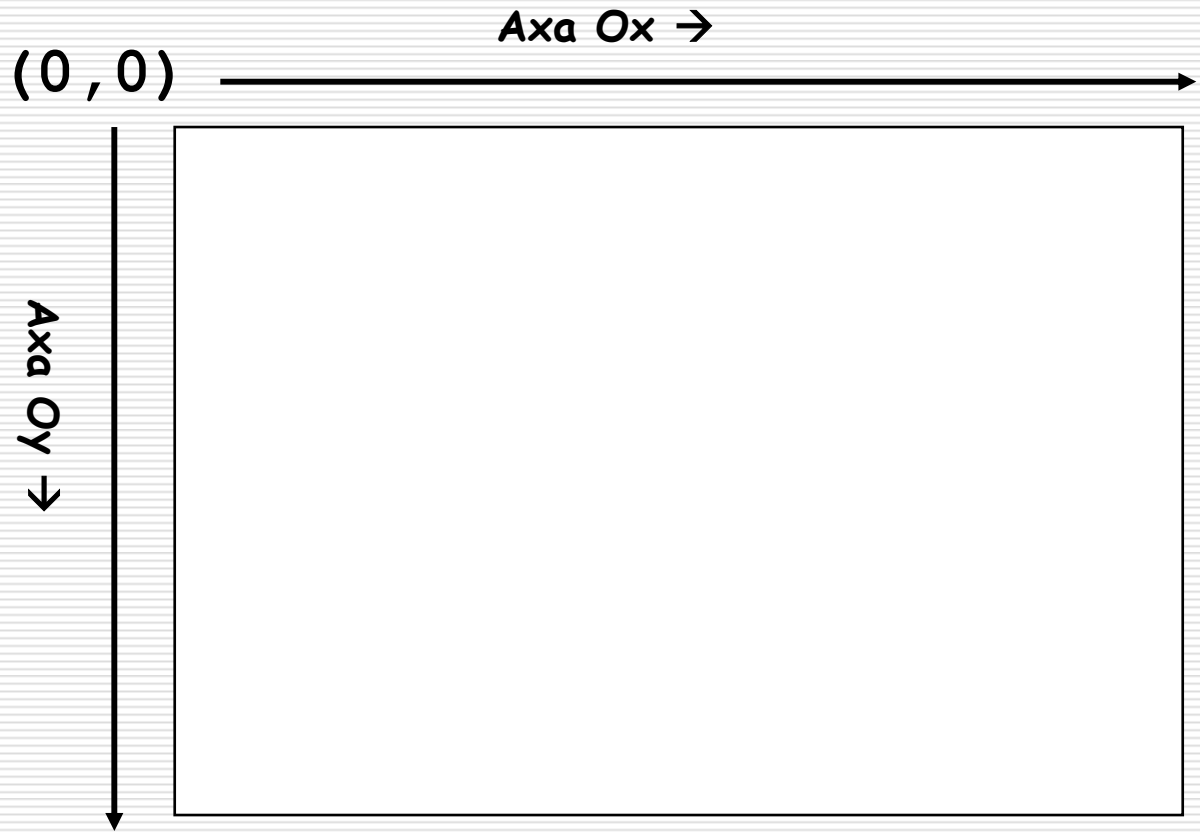
Rezultat execuție `Main> main1`

Desenare figuri

- Biblioteca grafică conține acțiuni pentru desenarea unor figuri:

```
ellipse      :: Point -> Point -> Graphic
shearEllipse :: Point -> Point -> Point -> Graphic
line         :: Point -> Point -> Graphic
polygon      :: [Point] -> Graphic
polyline     :: [Point] -> Graphic
```

Sistemul de coordonate



Culori

- Tipul de dată `Color`

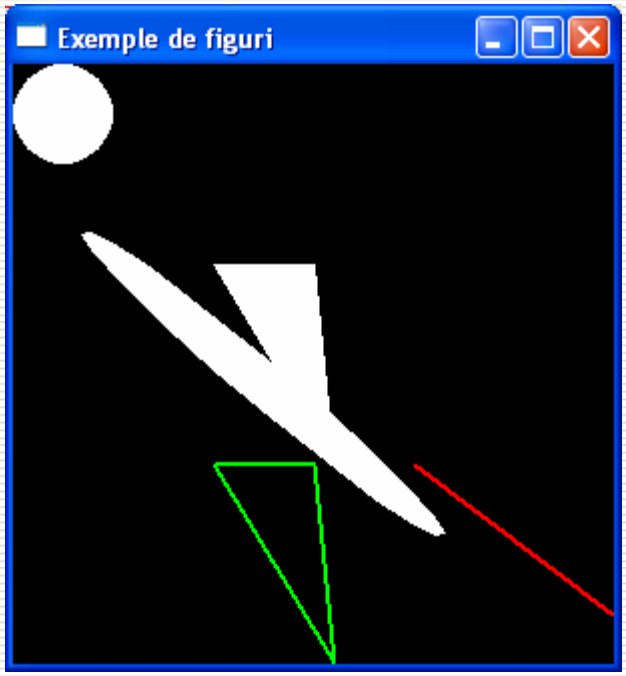
```
data Color =  Black | Blue | Green | Cyan |
              Red | Magenta | Yellow | White
       deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)

colorList  :: [(Color, RGB)]
colorTable :: Array Color RGB
withColor :: Color -> Graphic -> Graphic
```

Programul

```
main2 =
 runGraphics $
   do w <- openWindow "Exemple de figuri" (300,300)
      drawInWindow w (ellipse (0,0) (50,50))
      drawInWindow w
           (shearEllipse (0,60) (100,120) (150,200))
      drawInWindow w
           (withColor Red (line (200,200) (299,275)))
      drawInWindow w
           (polygon [(100,100),(150,100),(160,200)])
      drawInWindow w
           (withColor Green
                (polyline [(100,200),(150,200),
                           (160,299),(100,200)]))
      k <- getKey w
      closeWindow w
```

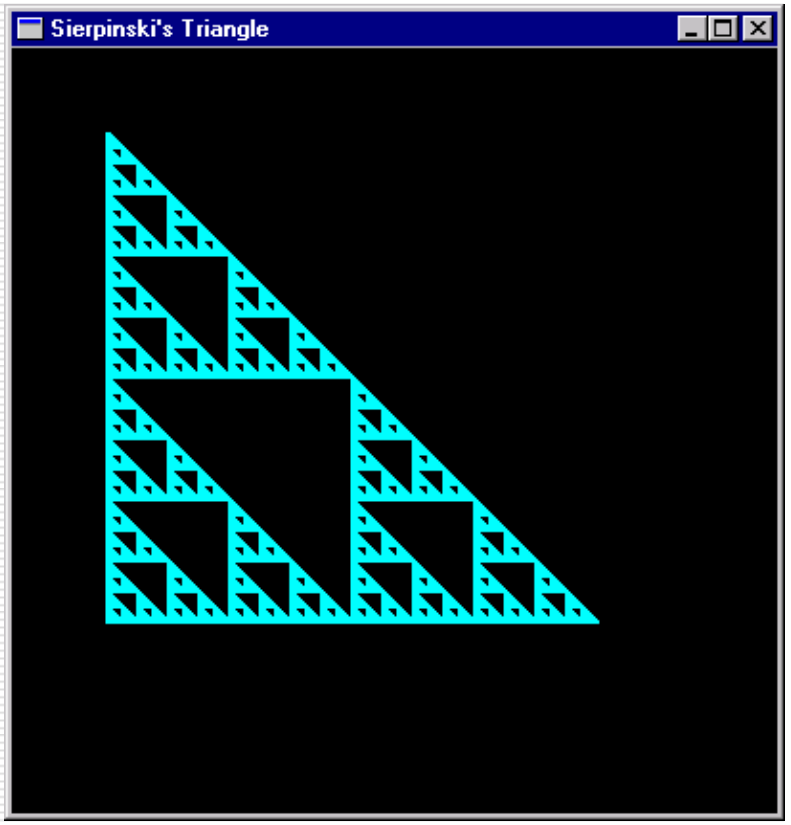
Rezultatul execuției main2

```
; drawInWindow w
    (ellipse (0,0) (50,50))
; drawInWindow w
   (shearEllipse (0,60)
                 (100,120)
                 (150,200))
; drawInWindow w
    (withColor Red
        (line (200,200)
              (299,275)))
; drawInWindow w
   (polygon [(100,100),
             (150,100),
             (160,200)])
; drawInWindow w
   (withColor Green
     (polyline
       [(100,200),(150,200),
        (160,299),(100,200)]))
```

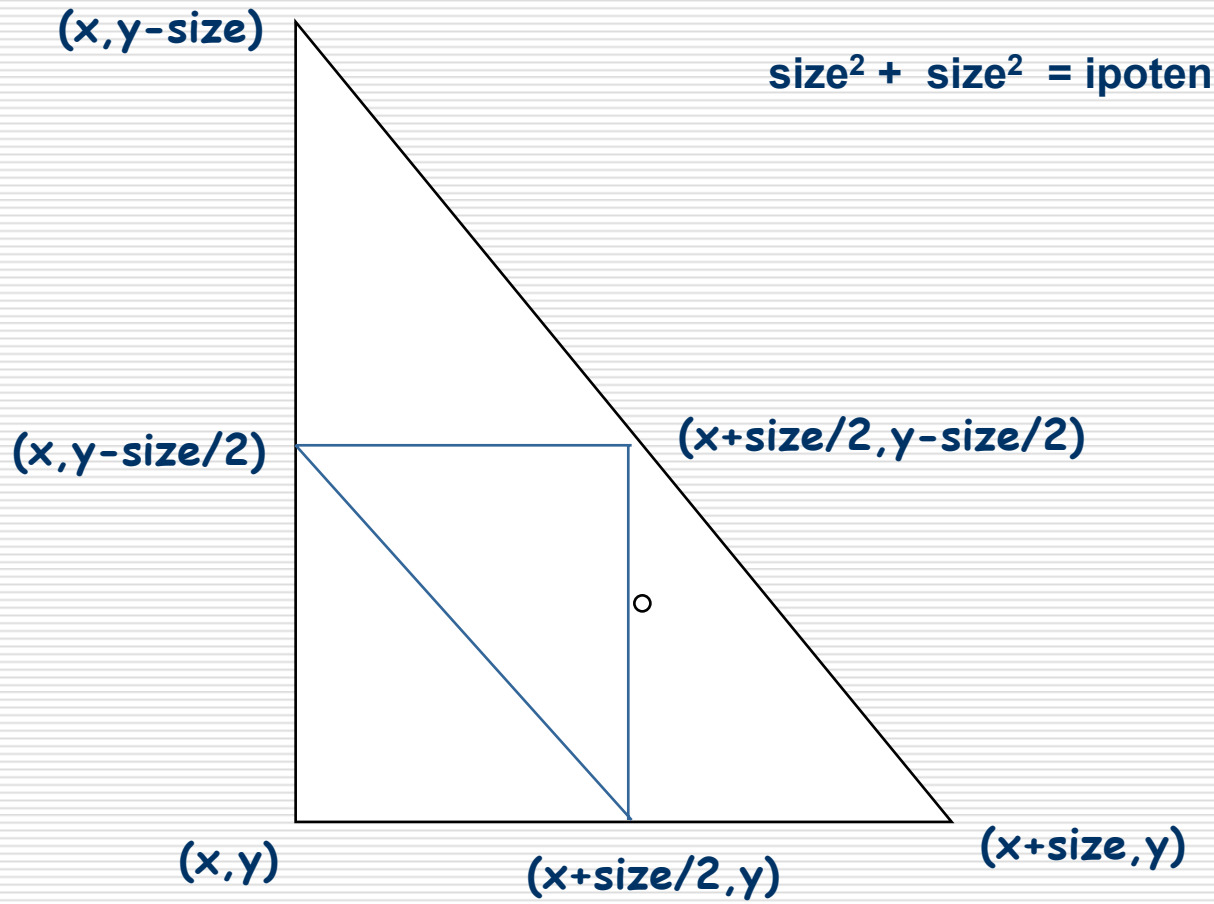


Studiu de caz: Triunghiul lui Sierpinski

- Triunghiul lui Sierpinski – un fractal constând din triunghiuri desenate unul în altul

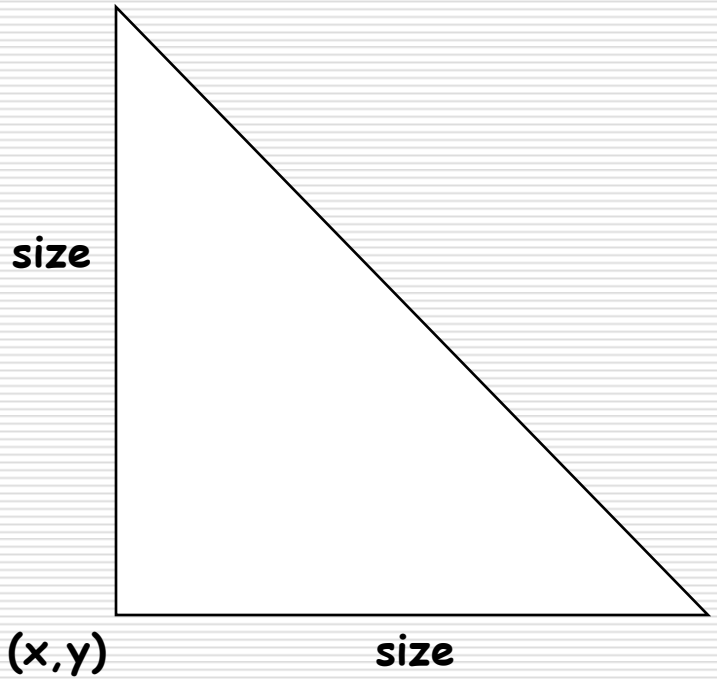


Geometria unui triunghi



Desenarea unui triunghi

```
fillTri x y size w =
    drawInWindow w
            (withColor Blue
             (polygon [(x,y),
                       (x+size,y),
                       (x,y-size)]))

minSize = 8
```



Triunghiul lui Sierpinski

```
sierpinskiTri w x y size =
  if size <= minSize
     then fillTri x y size w
     else let size2 = size `div` 2
          in do sierpinskiTri w x y size2
                sierpinskiTri w x (y-size2) size2
                sierpinskiTri w (x+size2) y size2
main3 =
  runGraphics $
    do w <- openWindow "Sierpinski's Tri" (400,400)
       sierpinskiTri w 50 300 256
       spaceClose w
```