CS 457/557 Functional Programming

Lecture 3

IO Actions; Graphics

Can we be imperative?

- All the programs we have seen so far have no “side-effects.” That is, programs are executed only for their **values**.

- But sometimes we want our programs to affect the real world (reading, printing, drawing a picture, controlling a robot, etc).

- Yet, IO operations (and other “effectful” operations) don't mix well with Haskell's lazy evaluation, because evaluation order is very complicated and hard to predict.

    - How can we reconcile purity and utility?

Example: Using the Trace facility

- Hugs has a built-in facility for wrapping an expression with a string that is to be printed whenever the expression is evaluated.

```
trace ::  String -> a -> a

f x = trace "goodbye\n" (x+1)
a = f (trace "hello\n" 1)

g x = x + trace "goodbye\n" 1
b = g (trace "hello\n" 1)
```

- Even if order of evaluation is not an issue, being able to do IO would violate “computation by calculation” paradigm, e.g.

```
c = x + x
  where x = trace "hi!\n" 1
```

versus

```
c = (trace "hi!\n" 1) + (trace "hi!\n" 1)
```

IO Actions

- In Haskell, “pure values” are separated from “worldly actions”, in two ways:
  - **Types**: An expression with type `IO a` has possible **actions** associated with its execution, while returning a value of type `a`.
  - **Syntax**: The `do` syntax **performs** an action, and (using layout) allows one to **sequence** several actions.
- Example: code to read a character, echo it, and return Boolean value indicating if it was a newline

```
do c <- getChar
   putChar c
   return (c == '\n')
```

Some Predefined IO Actions

```
-- get one character from keyboard
getChar :: IO Char

-- write one character to terminal
putChar :: Char -> IO()

-- get a whole line from keyboard
getLine :: IO String

-- read a file as a String
readFile :: FilePath -> IO String

-- write a String to a file
writeFile :: FilePath -> String -> IO ()
```

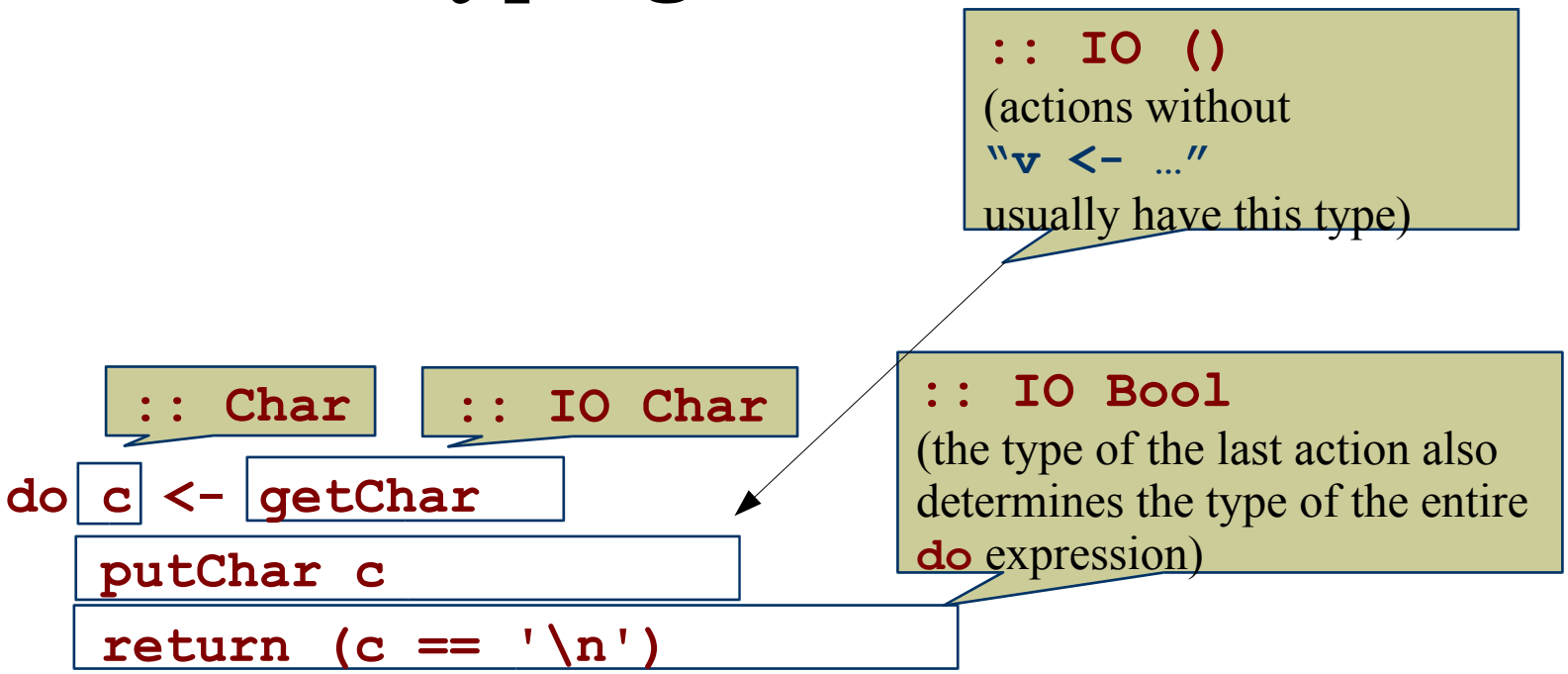
The `do` Syntax

- Let `act` be an action with type `IO a`.
- Then we can perform `act`, retrieve its return value, and sequence it with other actions, by using the `do` syntax:

```
do val <- act
   ...          -- the next action
   ...          -- the action following that
   return x     -- final action may return a value
```

- Note that all actions following `val <- act` can use the variable `val`.
- The function `return` takes a value of type `a`, and turns it into an action of type `IO a`, which does nothing but return the value.

`do` Typing Details



When are IO Actions Performed?

- A value of type `IO a` is an action, but it is still a value: it will only have an effect **when it is performed**.
- In Haskell, a program's value is the value of the variable `main` in the module `Main`. That value must have type `IO a`. The associated action will be performed when the whole program is run.
- In Hugs, however, you can type any expression to the Hugs prompt. If the expression has type `IO a` it will be performed; otherwise its value will be printed on the display.
- There is **no** other way to perform an action (well, almost).

Recursive Actions

`getLine` can be defined recursively in terms of simpler actions:

```
getLine :: IO String
getLine =
 do c <- getChar          -- get a character
    if c == '\n'          -- if it's a newline
       then return ""     -- then return empty string
       else do l <- getLine -- otherwise get rest of
                            -- line recursively,
               return (c:l)  -- and return entire line
```

Actions are just values

- Actions are just like other (first-class) values: they can be passed, returned, stored, etc.
- For example, it can be handy to build lists of actions, e.g.

```
putCharList :: String -> [IO ()]
putCharList cs = [putChar c | c <- cs]
```

- There's a library function to convert this to a single action

```
sequence_ :: [IO a] -> IO ()

putStr :: String -> IO ()
putStr  s = sequence_ (putCharList s)
```

- Remember, actions are only executed at top level, e.g.

```
main = putStr “abc”
```

Example: Unix `wc` Command

- The unix **`wc`** (word count) program reads a file and then prints out counts of characters, words, and lines.
- Reading the file is an action, but computing the information is a pure computation.
- Strategy:
  - Define a pure function that counts the number of characters, words, and lines in a string.
    - » number of lines = number of '\n'
    - » number of words ~= number of ' ' plus number of '\t'
  - Define an action that reads a file into a string, applies the above function, and then prints out the result.

Implementation

```
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 ("The file: " ++ name ++ "has ")
        putStrLn (show cc ++ " characters ")
        putStrLn (show w  ++ " words ")
        putStrLn (show lc ++ " lines ")
```

Example Run

```
Main> wc
elegantProse.txt
The file: elegantProse.txt has
   2970 characters
   1249 words
   141 lines

Main>
```

I typed this.

Graphics Actions

- Graphics windows are traditionally programmed using commands; i.e. actions.
- Some graphics actions relate to opening up a graphics window, closing it, etc.
- Others are associated with drawing lines, circles, text, etc.

“Hello World” program using Graphics Library

This imports a library, **SOEGraphics**, which contains many functions


```
import SOEGraphics
main0 =
 runGraphics (
    do w <- openWindow "First window" (300,300)
       drawInWindow w (text (100,200) "hello world")
       k <- getKey w
       closeWindow w
 )
```

Graphics Operators

- `openWindow :: String -> Point -> IO Window`
  - Opens a titled window of a particular size.
- `drawInWindow :: Window -> Graphic -> IO ()`
  - Displays a `Draw ()` value in a given window.
  - Note that the return type is `IO ()`.
- `getKey :: Window -> IO Char`
  - Waits until a key is pressed and then returns the character associated with the key.
- `closeWindow :: Window -> IO ()`
  - Closes the window.
- `runGraphics :: IO () -> IO ()`
  - Required "wrapper" around graphics operations to init/close graphics system.

Mixing Graphics IO with Terminal IO

```
spaceClose :: Window -> IO ()
spaceClose w =
    do k <- getKey w
       if k == ' ' then closeWindow w
                   else spaceClose w

main1 =
  runGraphics (
    do w <- openWindow "Second Program" (300,300)
       drawInWindow w (text (100,200) "Hello Again")
       spaceClose w
    )
```

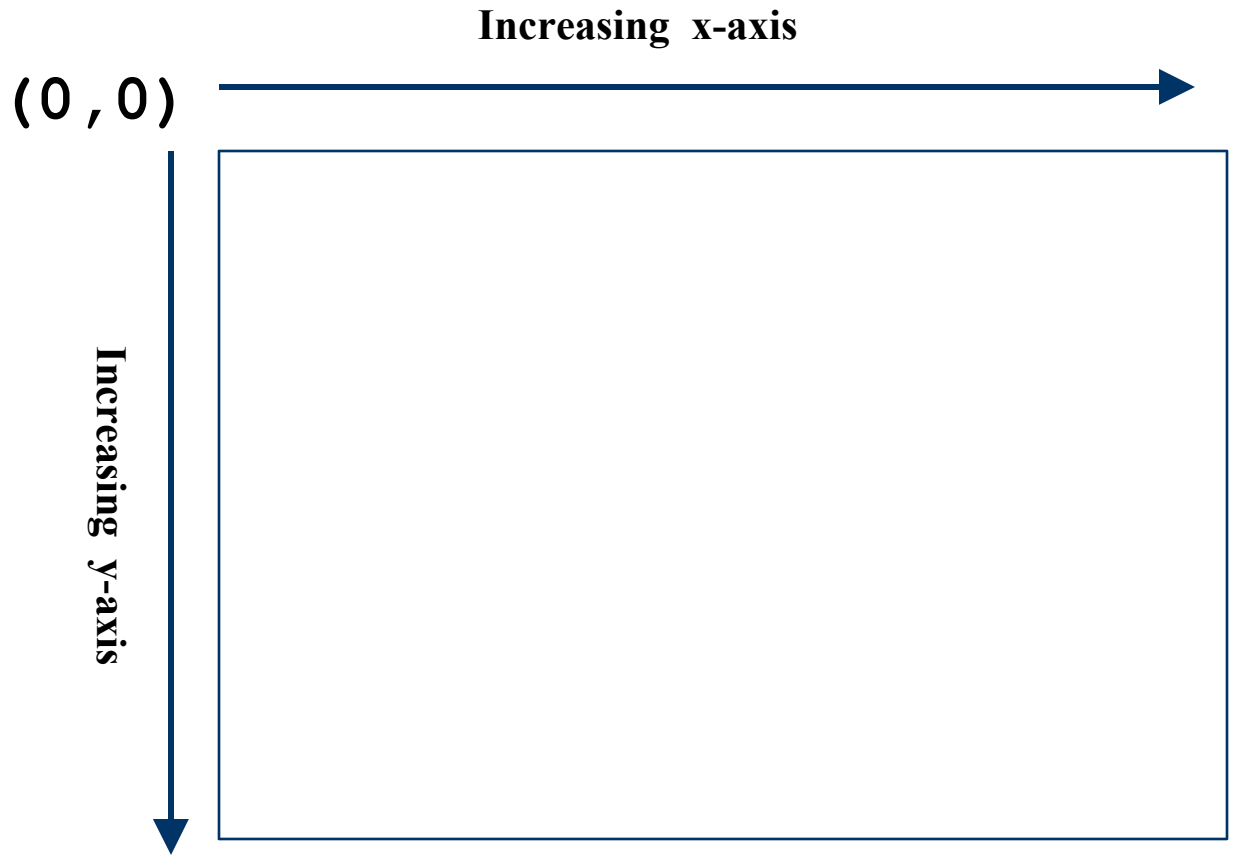
Drawing Primitive Shapes

- The Graphics libraries contain simple actions for drawing a few primitive shapes.

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

- From these we will build much more complex drawing programs.

Coordinate System



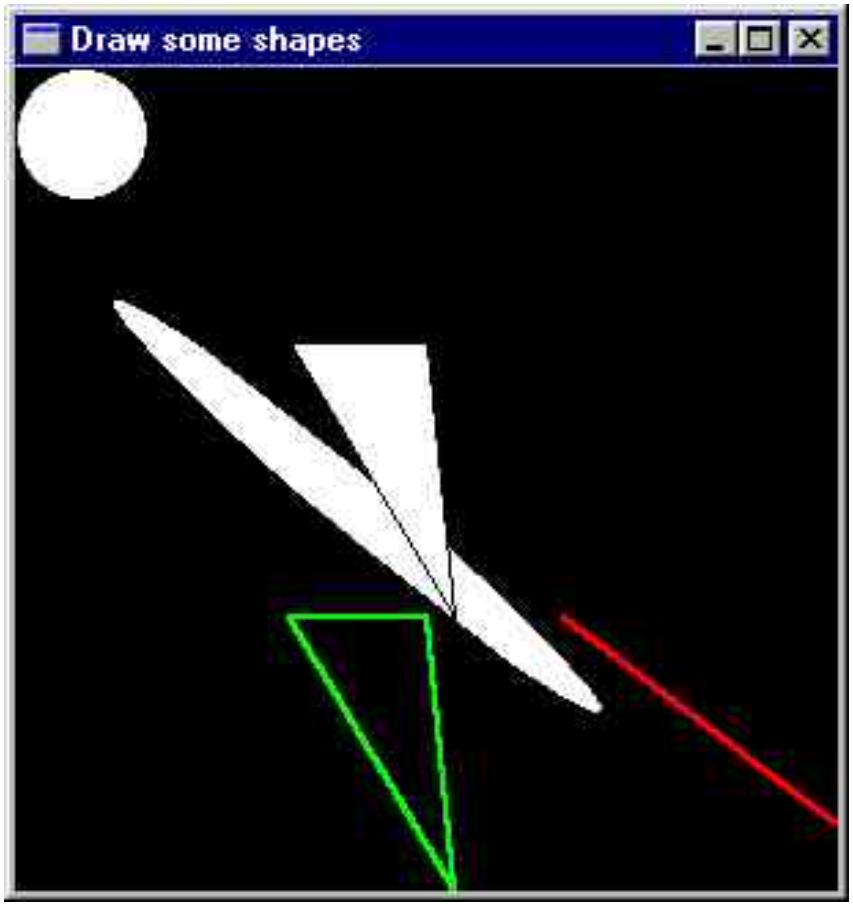
Example Program

```
main2 =
 runGraphics (
   do w <- openWindow "Draw some shapes" (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)]))
      spaceClose w
 )
```

The Result

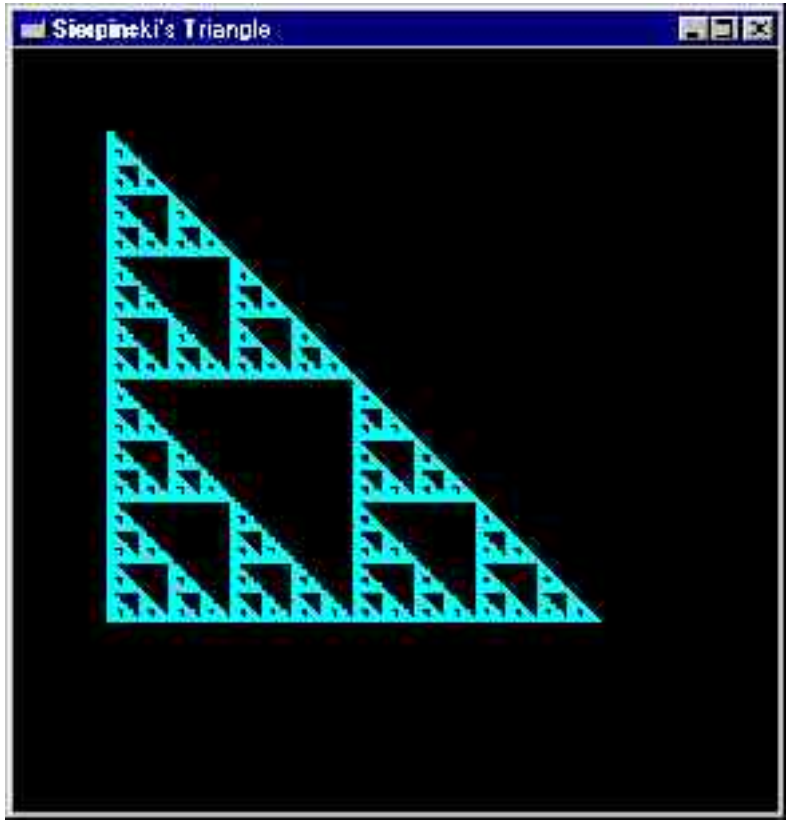
```
drawInWindow w
    (ellipse (0,0) (50,50))
drawInWindow w
  (shearEllipse (0,60)
                (100,120)
                (150,200))
drawInWindow w
    (withColor Red
        (line (200,200)

              (299,275)))
drawInWindow w
  (polygon [(100,100),
            (150,100),
            (160,200)])
drawInWindow w
  (withColor Green
    (polyline
      [(100,200),(150,200),
       (160,299),(100,200)]))
```

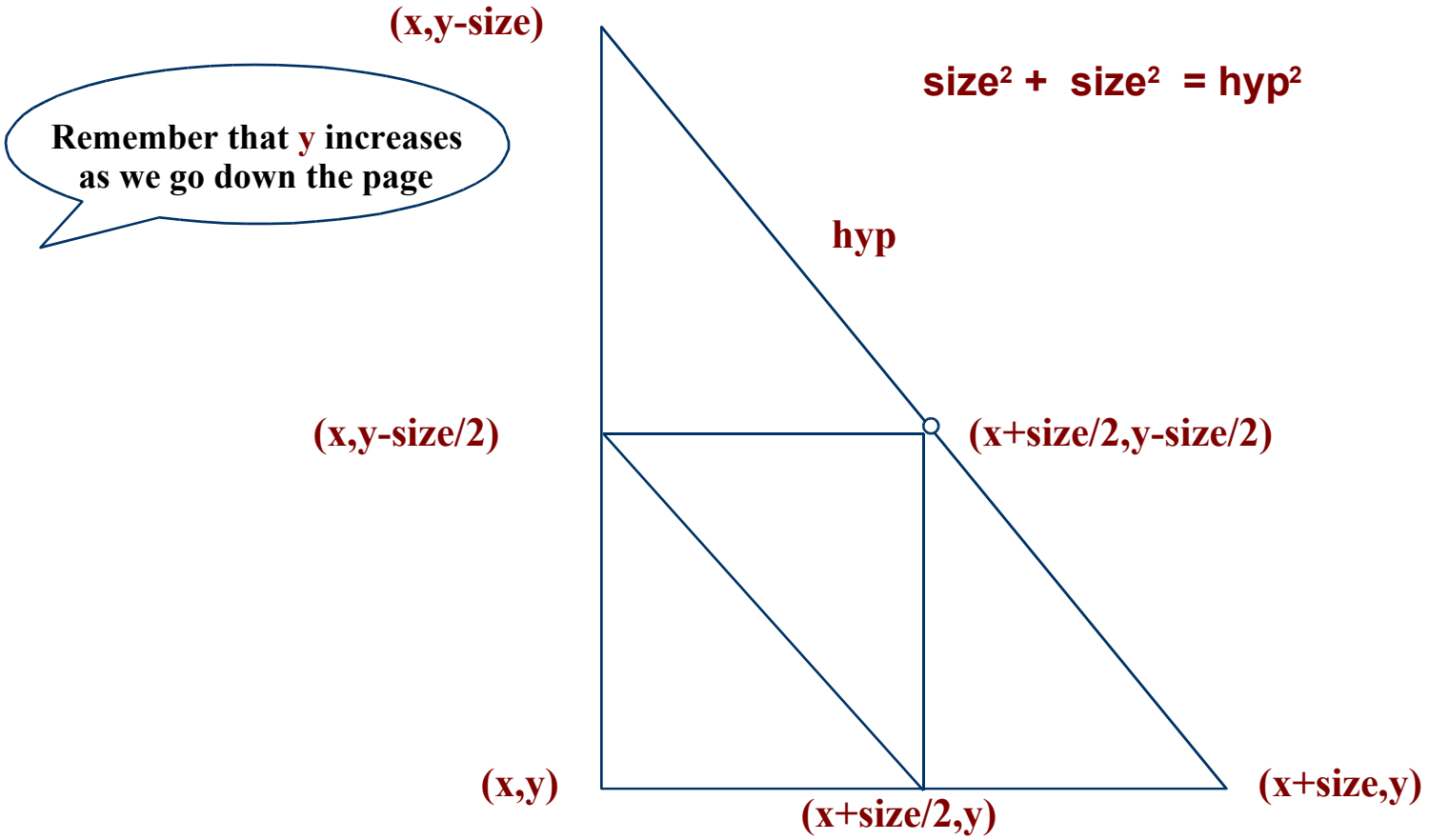


More Complex Programs

- We'd like to build bigger programs from these small pieces.
- For example:
  - Sierpinski's Triangle – a *fractal* consisting of repeated drawing of a triangle at successively smaller sizes.
- As before, a key idea is separating pure computation from graphics actions.



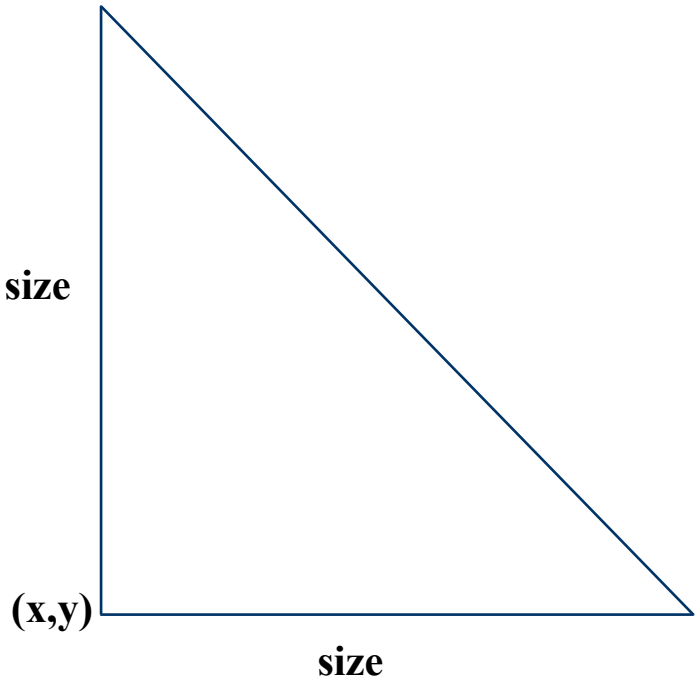
Geometry of One Triangle



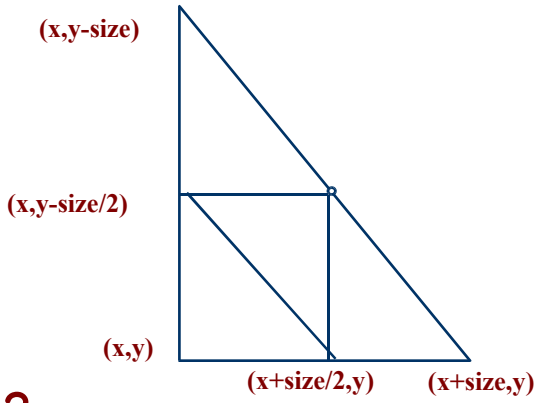
Draw 1 Triangle

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

minSize = 8
```



Sierpinski's Triangle




```
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
 )
```

Questions?

- Whats the largest triangle **`sierpinskiTri`** ever draws?
- How do the big triangles appear?