Chapter 3

Simple Graphics

Side Effects and Haskell

- 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 effect the real world (printing, controlling a robot, drawing a picture, etc).
- How do we reconcile these two competing properties?
- 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.

Sample Effects or Actions

- State
  - Intuitively, some programs have state that changes over time.
- Exceptions
  - There may be an error instead of a final value
- Non-determinism
  - There may be multiple possible final values
- Communication
  - There may be external influence on the computation
- Perform some Input or Output, as well as a final value.
  - E.g., draw graphics on a display

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 Syntax Details

:: Integer

:: IO Integer

:: IO Char

```
do x <- f 5
   c <- getChar
   putChar c
   return (x + 4)
```

:: Char

:: IO ()
(actions without **“v <- …”** usually have this type)

:: IO Integer
(the type of the last action also determines the type of the entire **do** expression)

When IO Actions are 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`**. If that value has type **`IO a`**, then it will be performed, since it is an action. If it has any other type, its value is simply printed on the display.
- In Hugs, however, you can type any expression to the Hugs prompt. E.g., for **`main`** above, if the expression has type **`IO a`** it will be performed, otherwise its value will be printed on the display.

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 ()
```

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
```

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

I typed this.

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

Main>
```

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 **`Graphic`** 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 (duh…).

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
```

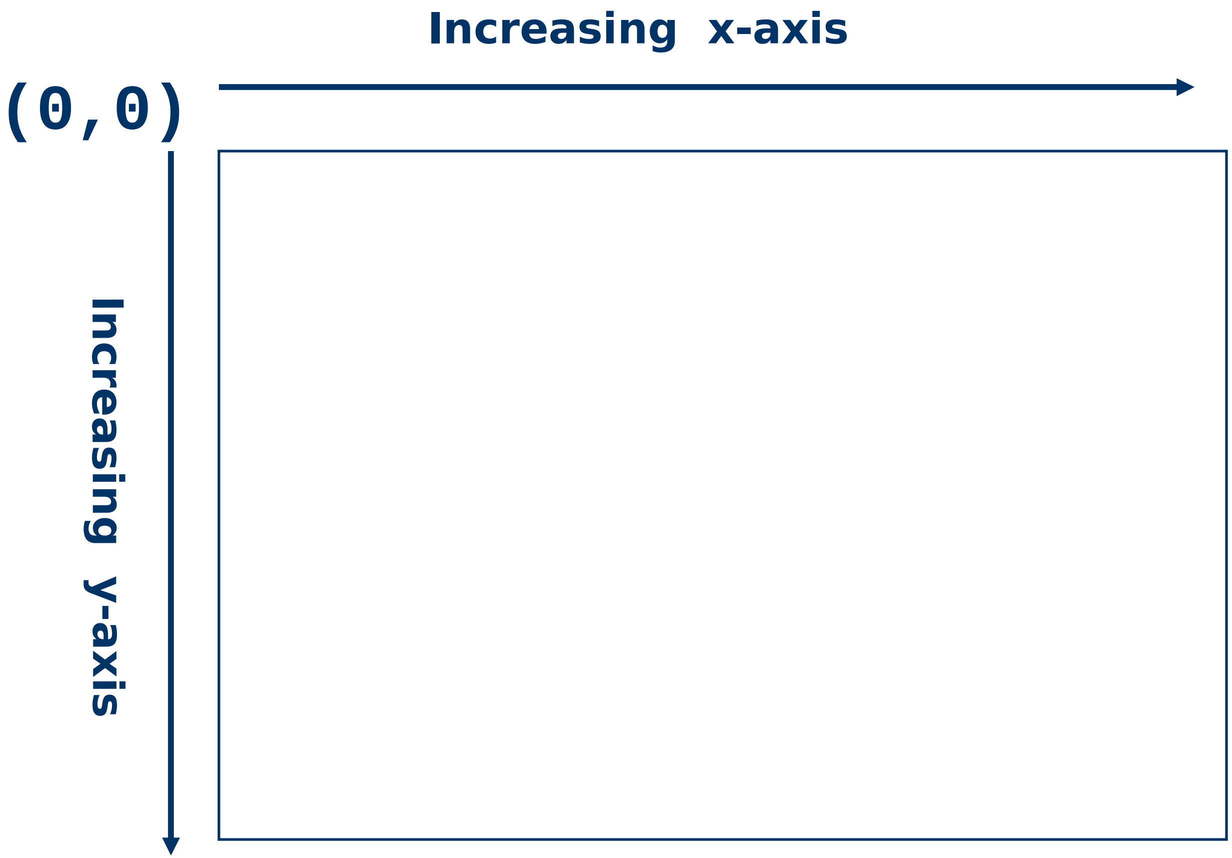
Drawing Primitive Shapes

- 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



Colors

```
withColor :: Color -> Graphic -> Graphic

data Color =
     Black | Blue | Green | Cyan |
     Red | Magenta | Yellow | White
```

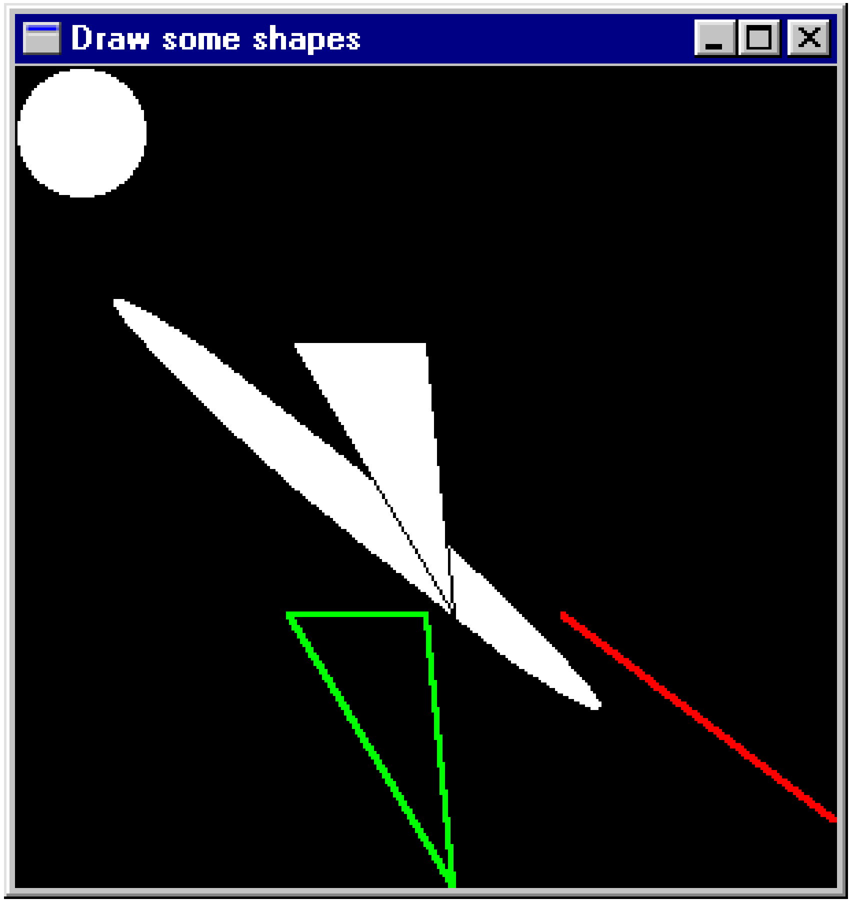
Example Program

```
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

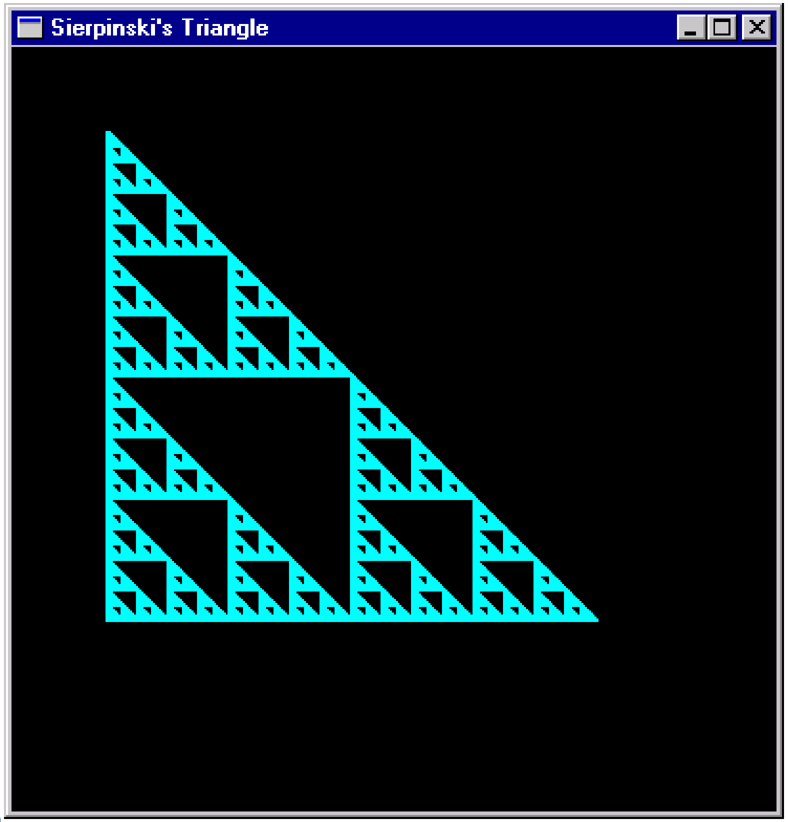
```
; drawInWindow w
    (ellipse (0,0) (50,50))
; drawInWindow w
   (shearEllipse (0,60)
                 (100,120)
                 (150,200))
; drawInWindow w
    (withColor Red
        (line (200,200)

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

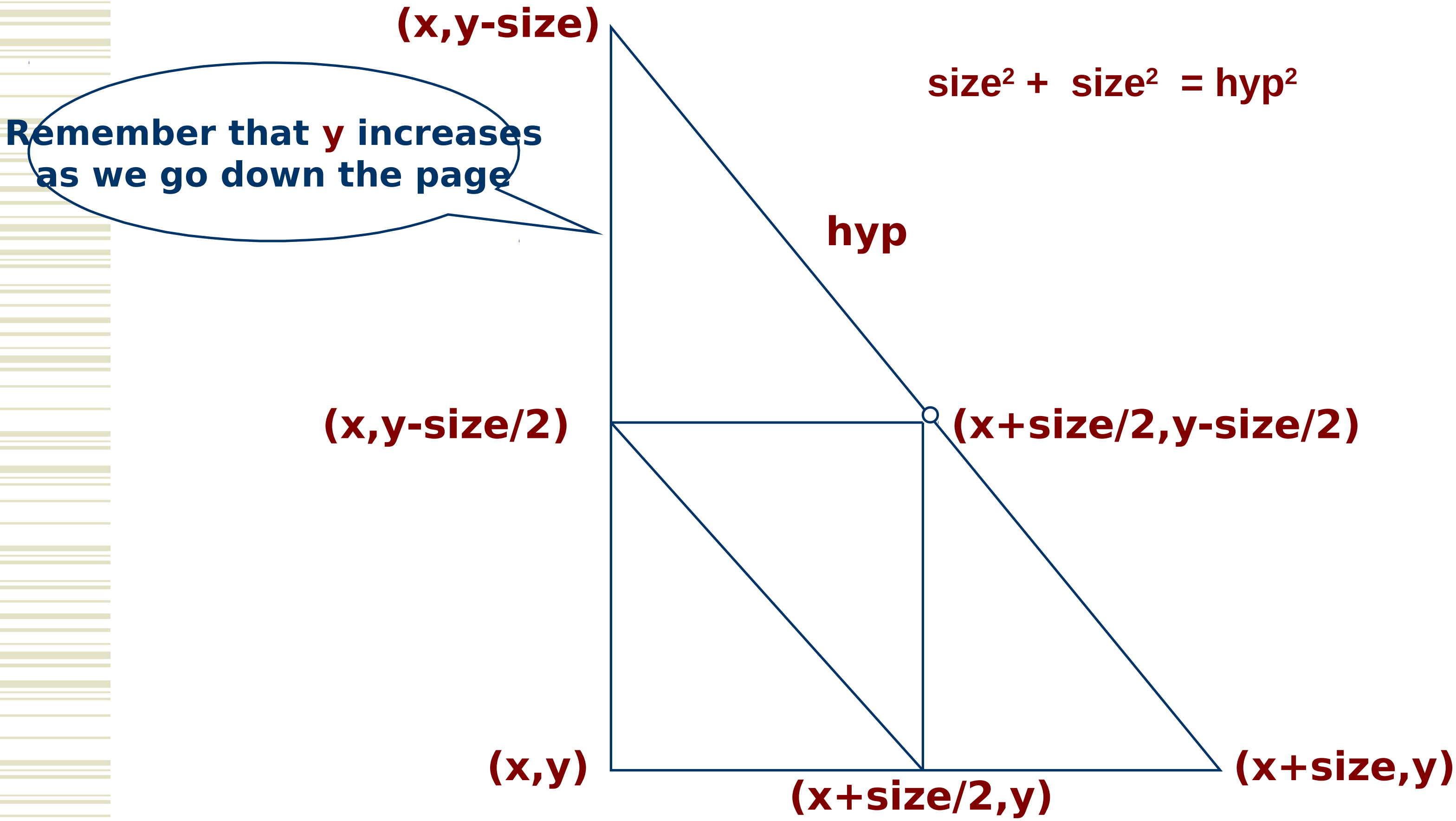


More Complex Programs

- 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.



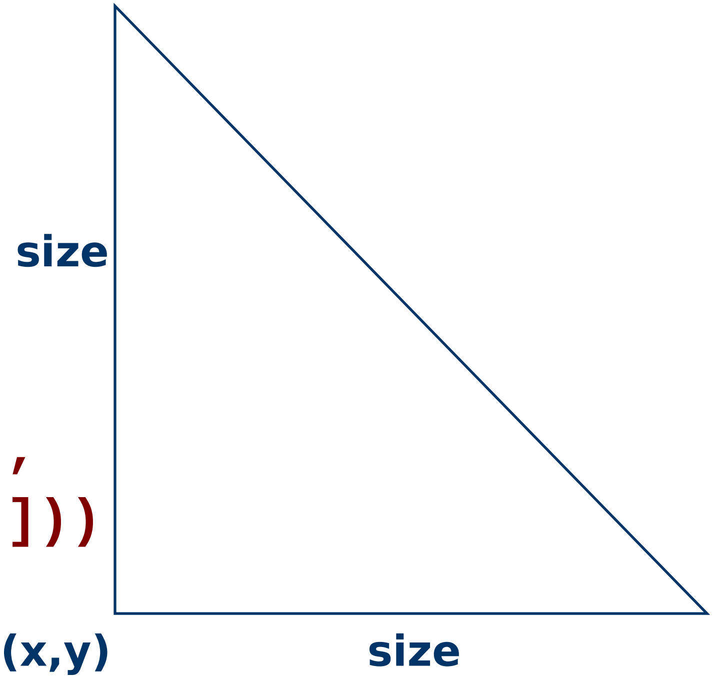
Geometry of One Triangle



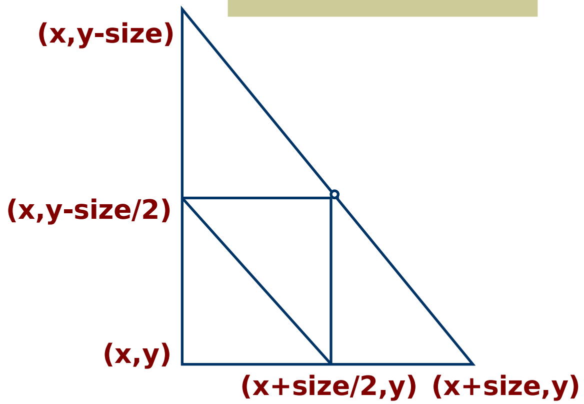
Draw 1 Triangle

```
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
```