CSE399: Advanced Programming

Handout 2

Input and Output Actions

I/O in Haskell

Q: Of course, most programs don't just calculate values: they have effects on the world — displaying text or graphics, reading or writing the file system and network...

How does this square with Haskell's value-oriented, calculational style of computation?

I/O in Haskell

A: Haskell provides a special kind of value, called an action, that describes an effect on the world.

Pure actions, which just do something and have no interesting "result," are values of type `IO ()`.

For example, the `putStr` function takes a string and yields an action describing the act of displaying this string on `stdout`.

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

I/O in Haskell

To actually **perform** an action, we make it the value of the special name `main`.

```
    main :: IO ()
    main = putStr "Hello world\n"
```

I/O in Haskell

```
~/current/advprog/common/lectures> hugs hello.hs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: November 2003  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Main> main
Hello world

Main>
```

The Batch Interpreter

The command

```
runhugs file.hs
```

will load the file `file.hs` into hugs and perform the action bound to the top-level name `main`.

```
~/current/advprog/common/lectures> runhugs hello.hs
Hello world
~/current/advprog/common/lectures>
```

Actions

Actions are descriptions of effects on the world. Simply writing an action does not, by itself, cause anything to happen.

```haskell
hellos :: [IO ()]
hellos = [putStr "Hello somebody\n",
          putStr "Hello world\n",
          putStr "Hello universe\n"]
main = head (tail hellos)
```

I/O in Haskell

```
~/current/advprog/common/lectures> runhugs hellos.hs
Hello world
~/current/advprog/common/lectures>
```

Combining Actions

The infix operator `>>` takes two actions `a` and `b` and yields an action that describes the effect of executing `a` and `b` in sequence.

```
    hello1 :: IO ()
    hello1 = putStr "hello " >> putStr "world\n"
```

“Do” Notation

To avoid writing `>>` all the time, Haskell provides special syntax for sequencing actions:

```
hello2 = do putStr "hello "
            putStr "world\n"
```

In general, if `act1`, `act2`, ..., `actn` are actions, then

```
do act1
   act2
   ...
   actn
```

is an action that represents performing them in sequence.

Note the use of the “layout convention” here: the first action begins right after the `do` and the others are laid out vertically beneath it.

Input Actions

Some actions have an effect on the world and yield a result. For example,

```
    getLine :: IO String
```

is an action that, when executed, consumes the next line from the standard input and returns it.

“Do” Notation for Input Actions

The do syntax provides a way to bind the result of an action to a variable so that it can be referred to later.

```
main =
  do putStr "Please type a line...\n"
     s <- getLine
     putStr "You typed ’"
     putStr s
     putStr "’\n"
```

“Do” Notation for Input Actions

```
~/current/advprog/common/lectures> runhugs lec2a.lhs
Please type a line...
hello there
You typed ’hello there’
```

Graphics

Graphics in Haskell

The module SOEGraphics provides a number of actions for drawing things on the screen.

```
    openWindow :: Title -> Size -> IO Window

    type Title = String
    type Size = (Int,Int)
```

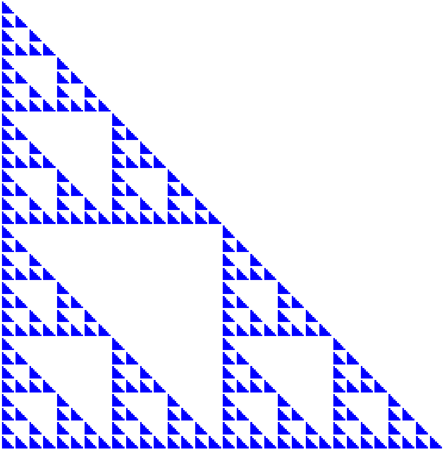
A Complete Graphics Program

```
import SOEGraphics

g = do w <- openWindow
              "My First Graphics Program" (300,300)
       drawInWindow w
          (text (100,200) "Hello Graphics World")
       drawInWindow w
          (withColor Red (ellipse (0,0) (100,150)))
       k <- getKey w
       closeWindow w

main = runGraphics g
```

Sierpinski's Triangle



Sierpinski's Triangle

```
fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size
  = drawInWindow w (withColor Blue
      (polygon [(x,y),(x+size,y),(x,y-size),(x,y)]))
```

Sierpinski’s Triangle

```
sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size
  = if size <= 8
    then fillTri w x y size
    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
```

Sierpinski's Triangle

```
g = do w <- openWindow
              "Sierpinski's Triangle" (400,400)
       sierpinskiTri w 50 300 256
       k <- getKey w
       closeWindow w

main = runGraphics g
```