nedap
technology that matters

Functional programming

Part three!

nedap
technology that matters

IO in Haskell

- What is the problem?
- How does Haskell solve this?
- The ‘type’ data IO a = …
- Expressions of this type are called ‘actions’

Some basic actions

- ```
  getChar :: IO Char
  getChar = …
  ```
- ```
  putChar :: Char -> IO ()
  putChar c = …
  ```
- ```
  return :: a -> IO a
  return v = …
  ```
  ```

Sequencing(1)

- Sequencing can be used to combine actions
- The form of sequencing typically is:

```
do v1 <- a1
   v2 <- a2
   …
   vn <- an
   return (f v1 v2 … vn)
```

- vi <- ai is called a generator
- The result of a generator can be discarded by doing either _ <- ai or just ai

Sequencing(2)

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
```

Derived primitives(1)

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
               return n
             else
               do xs <- getLine
                  return (x:xs)
```

Derived primitives(2)

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Derived primitives(3)

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```