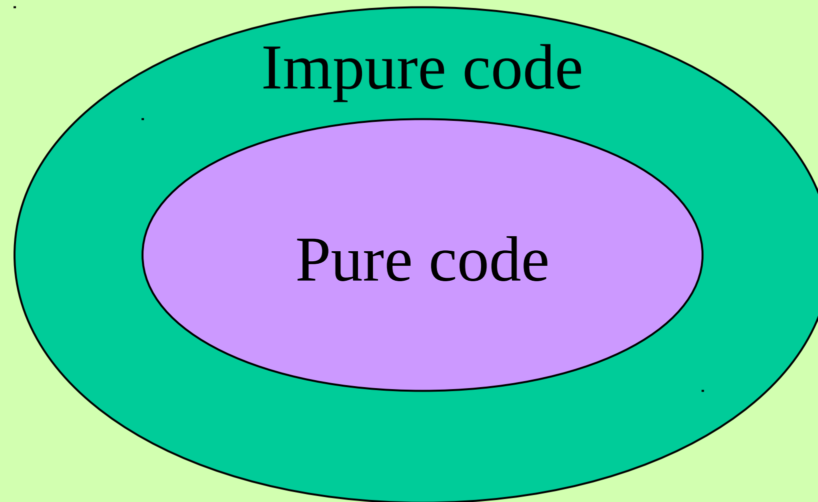
Computer Science 312

Text Files and Random Numbers

Isolate the Pure from the Impure

External world (files, I/O devices, network ports, etc.)



Text File Input with **`readFile`**

Like Python's **`open`** and **`read`** combined:

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

Prelude> readFile "threelines.txt"
"There are three\nlines of text\nin this file."

Prelude> text <- readFile "threelines.txt"

Prelude> putStrLn text
There are three
lines of text
in this file.
```

Bring in all the text in a single string, then extract what you need

Text File Output with **`writeFile`**

Like Python's **`open`** and **`write`** combined:

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

Prelude> let text = "There are three\nlines of text\nin this
file."

Prelude> writeFile "threelines.txt" text
```

Send out the text in a single string

Numbering Lines

```
{-
File: TextUtilities.hs
-}

module TextUtilities (numberLines) where

import Data.String (lines, unlines)

numberLines :: String -> String
numberLines text = unlines (helper (lines text) 1) where
    helper :: [String] -> Int -> [String]
    helper [] _ = []
    helper (x:xs) count =
        (show count ++ " " ++ x) : helper xs (count + 1)
```

Package text processing functions in a separate module

Numbering Lines

```
{-
File: NumberLines.hs
-}

module Main (main) where

import TextUtilities (numberLines)
import TerminalIO (getString)

main :: IO ()
main = do
    inPathName <- getString "Enter the input file name: "

    outPathName <- getString "Enter the output file name: "
    if inPathName == outPathName then do
        putStrLn "Error: files must have different names"
        return ()
    else do
        inputText <- readFile inPathName
        let outputText = numberLines inputText
        writeFile outPathName outputText
```

The main app interacts with the outside world

Random Numbers

- Needed to introduce non-determinism in programs
- Game-playing programs (dice, etc.)
- Guess the number
- Generate sentences, the Doctor program, etc.
- Cannot be a pure function!

Using `MyRandom.randInt`

```
Prelude> :load MyRandom

*MyRandom> randomInt 5    -- Returns a number between 0 and 4
2

*MyRandom> randomInt 5    -- Same argument, different value
0

*MyRandom> randomInt 5
2

*MyRandom> testRandomInt 5 6    -- Check the randomness
4
2
0
4
4
3
```

Strategy for Randomness

- Base the number on the current time, obtained from the system clock
- Import the **`Data.Time.Clock`** module and the **`Data.Time.Format`** module
- Use the functions **`getCurrentTime`** and **`defaultTimeLocale`** to derive a big integer, which will be pseudo-random enough

Defining **MyRandom.randInt**

```
{-
File: MyRandom.hs
-}

module MyRandom where

import Data.Time.Clock
import Data.Time.Format

-- Monadic function returns a random number between
-- 0 and n - 1, inclusive.
randomInt :: Int -> IO Int
randomInt n = do
    time <- getCurrentTime
    return ((`rem` n) $ read $ take 6 $
                              formatTime defaultTimeLocale "%q" time)
```

A Probability Function

```
*MyRandom> probability 0.75
True
*MyRandom> probability 0.75
False
*MyRandom> probability 0.75
True
*MyRandom> probability 0.75
True
*MyRandom> probability 0.25
False
*MyRandom> probability 0.25
False
```

Returns **`True`** a certain percentage of the time

A Probability Function

```
*MyRandom> probability 0.75
True
*MyRandom> probability 0.75
False
*MyRandom> probability 0.75
True
*MyRandom> probability 0.75
True
*MyRandom> probability 0.25
False
*MyRandom> probability 0.25
False
```

```
probability :: Float -> IO Bool
probability probFactor = do
    let probRange = round (100 * probFactor)
    randomValue <- randomInt 100
    return (randomValue > 100 - probRange)
```

Generating Random Sentences

```
Prelude> :load Generator

*Generator> sentence
"the table jumped the field beside a girl"

*Generator> testGenerator 5
a cake dragged a girl below a chair
the girl threw a table to the cake
the chair hit a table to a cake
the boy ate the dog to the chair
a bat threw a dog below a chair
```

A Simple EBNF Grammar

```
sentence = nounPhrase verbPhrase

nounPhrase = article noun

verbPhrase = verb nounPhrase prepositionalPhrase

prepositionalPhrase = preposition nounPhrase
```

A set of replacement rules for deriving sentences in a language

Parts of speech or lexical categories, like **noun** and **verb**, are taken from sets of words in the language

The Vocabulary for Lexical Items

```
nouns :: [String]
nouns = ["bat", "boy", "girl", "dog", "cat", "chair",
"fence", "table", "computer", "cake", "field"]

verbs :: [String]
verbs = ["hit", "threw", "pushed", "ate", "dragged", "jumped"]

prepositions :: [String]
prepositions = ["with", "to", "from", "on", "below",
     "above", "beside"]

articles :: [String]
articles = ["a", "the"]
```

Parts of speech or lexical categories, like **`noun`** and **`verb`**, are taken from sets of words in the language

Like Python's **random.choice**

```
-- Returns a random item from a list.
pickRandom :: [a] -> IO a
pickRandom list = do
    index <- randomInt (length list)
    return (list !! index)
```

Defined along with **`randomInt`** in the **`MyRandom`** module

Each Grammar Rule Gets a Function

```
-- sentence = nounPhrase verbPhrase
sentence :: IO String
sentence = do
    np <- nounPhrase
    vp <- verbPhrase
    return (np ++ " " ++ vp)

-- nounPhrase = article noun
nounPhrase :: IO String
nounPhrase = do
    article <- pickRandom articles
    noun <- pickRandom nouns
    return (article ++ " " ++ noun)

--Etc.
```

Using Probabilities

```
-- verbPhrase = nounPhrase verb [ prepositionalPhrase]
verbPhrase :: IO String
verbPhrase = do
    np <- nounPhrase
    verb <- pickRandom verbs
    let phrase = np ++ " " ++ verb
    ppOkay <- probability 0.25
    if ppOkay then do
        pp = <- prepositionalPhrase
        return (phrase ++ " " ++ pp)
    else
        return phrase
```

Using Probabilities – The Doctor

```
{-
Generates and returns replies to user’s sentences.
The current options are to change persons and prepend
an interrogatory qualifier or just hedge.
-}
reply :: String -> IO String
reply sentence = do
    hedgeOkay <- probability 0.33
    if hedgeOkay then do
        hedge <- pickRandom hedges
        return hedge
    else do
        let newSentence = changePerson sentence
        qualifier <- pickRandom qualifiers
        return (qualifier ++ newSentence ++ "?")
```

For next time

Read the article on Erlang by Joe Armstrong

(in Resources on Sakai)