

# Haskell Workshop

## Writing your first Haskell programm

Nicolas Audinet

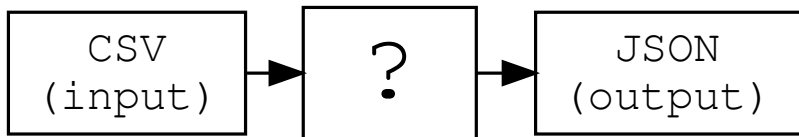
RELEX Solutions Dev Day

2019-04-12



# The Goal

Develop a complete Haskell program that converts CSV data to JSON.



# The Data

```
1 "Constituency", "Party", "Sex", "Average age"
2 "Helsinki constituency", "KOK", "Men", 47.6
3 "Helsinki constituency", "KOK", "Women", 49.4
4 "Helsinki constituency", "SDP", "Men", 48.0
5 "Helsinki constituency", "SDP", "Women", 46.1
6 "Helsinki constituency", "PS", "Men", 46.9
7 "Helsinki constituency", "PS", "Women", 43.1
8 "Helsinki constituency", "KESK", "Men", 44.0
9 "Helsinki constituency", "KESK", "Women", 46.8
10 "Helsinki constituency", "VAS", "Men", 40.3
11 "Helsinki constituency", "VAS", "Women", 37.0
12 "Helsinki constituency", "VIHR", "Men", 41.8
13 "Helsinki constituency", "VIHR", "Women", 40.5
14 "Helsinki constituency", "RKP", "Men", 39.7
15 "Helsinki constituency", "RKP", "Women", 42.3
16 "Helsinki constituency", "KD", "Men", 42.8
17 "Helsinki constituency", "KD", "Women", 47.1
```

# The Plan

---

- 1) Haskell Development Environment
- 2) Hello World and some Syntax
- 3) The Data Model
- 4) Data Transformation
- 5) Connecting to the Outside
- 6) Conclusion

# The Haskell Development Environment

# Setting up your environment

- Go to the Github repo for the workshop (<https://github.com/relex/haskell-workshop>)
- Clone it locally
- Go to workshop1/Exercise.md
- Complete “Set up tooling” section

# Stack

- One of Haskell's build tools.
- To build a project: `stack build`
- To access the REPL: `stack repl`

```

~/D/haskell-workshop > presentation ... workshop1/slides > stack repl
Note: Enabling Nix integration, as it is required under NixOS
Note: Enabling Nix integration, as it is required under NixOS
Building all executables for `workshop1' once. After a successful build of all of them, only speci
ed executables will be rebuilt.
workshop1-0.1.0.0: initial-build-steps (exe)
The following GHC options are incompatible with GHCi and have not been passed to it: -threaded
Configuring GHCi with the following packages: workshop1
Using main module: 1. Package `workshop1' component exe:workshop1-exe with main-is file: /home/nic/D
ocuments/haskell-workshop/workshop1/src/Main.hs
GHCi, version 8.4.4: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main
           ( /home/nic/Documents/haskell-workshop/workshop1/src/Main.hs, in
terpreted )
OK, one module loaded.
Loaded GHCi configuration from /run/user/1000/haskell-stack-ghci/3a980dcd/ghci-script
*Main>

```

# Hello Syntax



# Playing with the REPL

- Useful tool to try out parts of your program
- Extensively used in the workshop

# Hello World

- Open src/Main.hs
- Run in the REPL by typing `:main`

```
main :: IO ()  
main = putStrLn "Hello World!"
```

- `main` is special:
  - Present in every Haskell executable
  - No function arguments
  - Entry point for all IO action
- Change the text and run it in the REPL

# Functions and Types

- The primary way of defining computation is with **functions**
- **Types** describe the inputs and outputs of functions
- Types are **enforced** by the compiler

```
identity :: Int -> Int
```

```
identity x = x
```

```
hello :: String -> String
```

```
hello name = "Hello, " ++ name
```

```
addInt :: Int -> Int -> Int
```

```
addInt x y = x + y
```

Add these ^ to a file and try them out!

# A First Function

- A function that converts (some) integers to words
- Pattern matching

```
intToWord :: Int -> String
intToWord 1 = "one"
intToWord 2 = "two"
intToWord 3 = "three"
intToWord _ = "dunno"
```

- `_` means “for every other value”
- Try and write `wordToInt :: String -> Int`

# Polymorphic Types

- Some functions have the same behaviour for values of different types
  - E.g. the `identity` function
- Can generalise functions by using **type variables**

```
identity :: a -> a  
identity x = x
```

- Can also **constrain** the type variables

```
add :: Num a => a -> a -> a  
add x y = x + y
```

- These constraints can be defined using **type classes**

# Composition

- Using the output of one function as the input of another function:  
**“chaining functions together”**
- Use the dot operator for composition
- $h(x) = g(f(x)) \Rightarrow g \circ f$

```
f :: Int -> String
```

```
f x = intToWord x
```

```
g :: String -> Int
```

```
g x = wordToInt x
```

```
h :: Int -> Int
```

```
h = g . f
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(.) f g x = g (f x)
```

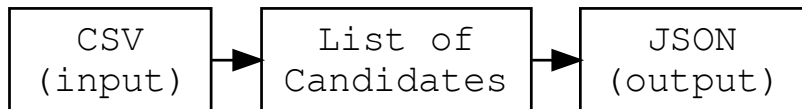
# The Data Model

# Goal

- Create a data structure that accurately models the data
- Introduce Maybe
- Introduce Lists

## Approach

- Define a model for a single row of data (an average candidate)
- Apply this model to all the rows





# A Single Candidate

```
1 "Constituency", "Party", "Sex", "Average age"
2 "Helsinki constituency", "KOK", "Men", 47.6
3 "Helsinki constituency", "KOK", "Women", 49.4
4 "Helsinki constituency", "SDP", "Men", 48.0
5 "Helsinki constituency", "SDP", "Women", 46.1
6 "Helsinki constituency", "PS", "Men", 46.9
7 "Helsinki constituency", "PS", "Women", 43.1
8 "Helsinki constituency", "KESK", "Men", 44.0
```

- A candidate has 4 attributes:
  - constituency
  - party
  - sex
  - average age
- Need a way of grouping these together

# Data Types

A simple data type:

```
data Person = MakePerson
  { name :: String
  , age  :: Int
  } deriving Show
```

- Person is the type being defined
- MakePerson is the constructor function
  - Usually the same as the type
- name, age are fields with associated types
- deriving Show allows the use of the show function
  - show converts structure to a string

# Working with Data Types

- Creating a new data type:

```
MakePerson :: String -> Int -> Person
```

- Accessing the data type:

```
name :: Person -> String
```

```
age :: Person -> Int
```

```
*Main> let neo = MakePerson "Neo (The One)" 34
*Main> neo
MakePerson {name = "Neo (The One)", age = 34}
*Main> name neo
"Neo (The One)"
*Main> age neo
34
*Main> █
```

# The Maybe Type

- What happens if we cannot parse a candidate?
- Need some way to represent an invalid row:

```
data Maybe a = Nothing | Just a
```

- Maybe is the type being defined
- a is a **type variable** (could be any type)
- Nothing, Just are **both** constructors
  - Nothing :: Maybe a
  - Just :: a -> Maybe a

# The List Type

- How do we represent a list of candidates?
- Use singly-linked lists:
  - `[]` is the empty list
  - `:` lets you append values to the front
  - `[a,b,c] == (a : b : c : [])`

```
*Main> (1 : 2 : 3 : [])  
[1,2,3]  
*Main> 10 : [1,2,3]  
[10,1,2,3]  
*Main> [1 .. 10]  
[1,2,3,4,5,6,7,8,9,10]  
*Main> ("I'm" : "in" : "a" : "list" : [])  
["I'm","in","a","list"]  
*Main> █
```

# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
```

```
{ ...
```

```
} deriving Show
```

```
"Constituency", "Party", "Sex", "Average age"
```

```
"Helsinki constituency", "KOK", "Men", 47.6
```

# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
  { constituency :: String
  , party        :: String
  , sex          :: String
  , averageAge   :: Double
  } deriving Show
```

# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
  { constituency :: String
  , party        :: String
  , sex          :: String
  , averageAge   :: Double
  } deriving Show
```

- Add a notion of failure



# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
  { constituency :: String
  , party        :: String
  , sex          :: String
  , averageAge   :: Double
  } deriving Show
```

- Add a notion of failure

```
type DataModel = Maybe Candidate
```

# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
  { constituency :: String
  , party        :: String
  , sex          :: String
  , averageAge   :: Double
  } deriving Show
```

- Add a notion of failure

```
type DataModel = Maybe Candidate
```

- Create a list of candidates

# The Data Model

- Define our basic candidate type

```
data Candidate = Candidate
  { constituency :: String
  , party        :: String
  , sex          :: String
  , averageAge   :: Double
  } deriving Show
```

- Add a notion of failure

```
type DataModel = Maybe Candidate
```

- Create a list of candidates

```
type DataModel = [Maybe Candidate]
```

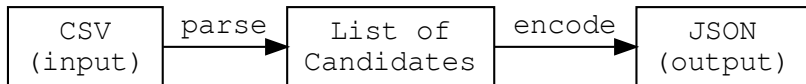
# Data Transformation

# Goal

- Write some functions to:
  - Convert CSV into data model
  - Convert data model into JSON
- Demonstrate power of Generic
- Introduce `map`

## Approach

- Write a function `parseLine :: CSVLine -> Maybe Candidate`
- Write a function `parseFile :: CSV -> DataModel`
- Specialise a function `encode :: DataModel -> JSONByteString`
- Apply these to every row



## Writing `parseLine :: CSVLine -> Maybe Candidate`

```
type CSVLine = String -- type alias
```

```
parseLine :: CSVLine -> Maybe Candidate
```

```
parseLine line = makeCandidate undefined
```

```
makeCandidate :: [String] -> Maybe Candidate
```

```
makeCandidate fields = undefined
```

```
fromMaybeAge
```

```
  :: (Double -> Candidate)
```

```
  -> Maybe Double
```

```
  -> Maybe Candidate
```

```
fromMaybeAge partialCandidate maybeAge = undefined
```

### Hints:

- `Data.List.Split.splitOn` splits a list on the specified character
- `readMaybe` decodes a double from a string (may fail)

# Writing `parseLine :: CSVLine -> Maybe Candidate`

Our solution:

```

parseLine :: CSVLine -> Maybe Candidate
parseLine line = makeCandidate (splitOn ', ' line)

makeCandidate :: [String] -> Maybe Candidate
makeCandidate [c, p, s, a] =
    fromMaybeAge (Candidate c p s) (readMaybe a)
makeCandidate _ = Nothing

fromMaybeAge
    :: (Double -> Candidate)
    -> Maybe Double
    -> Maybe Candidate
fromMaybeAge partialCandidate Nothing = Nothing
fromMaybeAge partialCandidate (Just a) = Just (partialCandidate a)

```

# Stripping Quotes

```
stripQ :: String -> String
stripQ = undefined
  where
    leftStrip :: String -> String
    leftStrip ('''' : xs) = xs
    leftStrip xs          = xs

    rightStrip :: String -> String
    rightStrip = undefined
```

## Hints:

- Use leftStrip and reverse to implement rightStrip
- Use leftStrip and rightStrip to implement stripQ
- reverse reverses items in a list
- where used to define local functions



# Stripping Quotes

Our solution:

```
stripQ :: String -> String
stripQ = leftStrip . rightStrip
  where
    leftStrip :: String -> String
    leftStrip ('"' : xs) = xs
    leftStrip xs         = xs

    rightStrip :: String -> String
    rightStrip = reverse . leftStrip . reverse
```

- Point-free style: composition FTW!

# Improving parseLine

- Using stripQ to improve makeCandidate

```
...  
  
makeCandidate :: [String] -> Maybe Candidate  
makeCandidate [c, p, s, a] =  
    fromMaybeAge (Candidate c' p' s') (readMaybe a)  
    where  
        c' = stripQ c  
        p' = stripQ p  
        s' = stripQ s  
makeCandidate _ = Nothing  
  
...
```

# The map function

- How do we apply `parseLine` to every row?
- Use `map`, a **higher-order function**
  - Takes two inputs: a function and a list.
  - Applies the function to every item in the list

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

- Defined using **recursion**

# Writing `parseFile :: CSV -> DataModel`

- We are ready to write the full parser function:

```
type CSV = String
type DataModel = [Maybe Candidate]

parseFile :: CSV -> DataModel
parseFile = undefined
```

- Try and use point-free style

# Writing `parseFile :: CSV -> DataModel`

Our solution:

```
parseFile :: CSV -> DataModel
parseFile = map parseLine . tail . lines
```

# Specializing encode :: DataModel -> JSONByteString

```
{-# LANGUAGE DeriveGeneric #-}  
main Main where  
  
import           GHC.Generics  
import qualified Data.Aeson           as JSON  
import qualified Data.ByteString.Lazy as B  
  
data Candidate = Candidate  
  { ...  
  } deriving (Show, Generic)  
  
instance Aeson.ToJSON Candidate
```

- Language extensions expand the Haskell language
- Data.Aeson provides encoding to and from JSON
- Data.ByteString.Lazy provides efficient binary strings

# Specializing encode :: DataModel -> JSONByteString

```
type JSONByteString = ByteString
```

- encode outputs efficient binary strings
- Try it out in the REPL!

```
> :set -XDeriveGeneric
> import qualified Data.Aeson as JSON
> import qualified Data.ByteString.Lazy as BL
> let candidate = Candidate "Helsinki" "KOK" "Women" 49.4
> B.putStr (JSON.encode candidate)
{"constituency":"Helsinki","averageAge":49.4,"party":"KOK","sex":"Women"}
```

# Connecting to the Outside

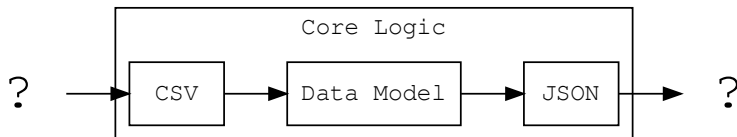


# Goal

- Write a complete program!
- Introduce IO and do notation

## Approach

- Introduce purity and impurity
- Write main function

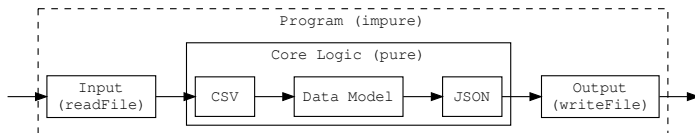


# Purity in Haskell

- All functions we have written in the previous section are **pure**
- A pure function **always** returns the same **output** when given the same **input**
- This behaviour is **enforced by the compiler**
- Why?
  - Code is easier to refactor and reason about
  - Interactions with the outside world are more explicit
  - Easy to test
- Pure functions cannot interact with the world
- So how do we communicate with the outside world?

# IO

- Communication with the outside handled through IO
- Impurity is **always** explicitly encoded in the types



# Do notation

```
askName :: IO ()
askName = do
  putStrLn "Enter your name:"
  name <- getLine
  let reversedName = reverse name
  putStrLn ("Your name reversed: " ++ reversedName)
```

- `putStrLn` prints a string
- `getLine` gets a line of input from the user
- Do blocks are a special Haskell syntax impure programming
  - Always start with the `do` keyword
  - Functions executed top-to-bottom
  - Return values stored in local variables using `<-`
  - `let` used for pure computation

# Read / Write

```
convert :: String -> IO ()  
convert filename = do  
    undefined
```

- `readFile` returns the contents of a file as a `String`
- `B.writeFile` writes a `ByteString` to a file

# Read / Write

## Our Solution:

```
convert :: String -> IO
convert filename = do
  csv <- readFile filename
  let json = JSON.encode (parseFile csv)
  B.writeFile "output.json" json
```

# The Complete App

- Get the filename from the arguments:

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    [filename] -> convert filename
    _          -> putStrLn "Call only with filename"
```

- And we have a complete Haskell app!! :D

# Conclusion



# Conclusion



- Types are useful to keep track of functions
- Abstraction makes things easier to think about
- Split problem into data model and data transformation
- Keep core logic pure, minimize IO and impurity

# Challenges

---

# Further Reading

---

