Functional Programming (CS4011)

Andrew Butterfield    Glenn Strong

Foundations & Methods Group,
Discipline of Software Systems
Trinity College, University of Dublin

{Andrew.Butterfield,Glenn.Strong}@cs.tcd.ie

October 1, 2010
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler

- Configuring and running ghc/ghci

- Syntax Niceties

- Problems to solve

- Do it . . .

- Lab Postamble

- Work Submission
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
Lab 1 : Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
  - Problems to solve
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
  - Problems to solve
- Do it . . .
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi

- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
  - Problems to solve

- Do it . . .

- Lab Postamble
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
  - Problems to solve
- Do it . . .
- Lab Postamble
  - Work Submission
Lab 1: Getting to know GHC

- GHC: The Glasgow (Glorious) Haskell Compiler
  - incorporating the interpreter GHCi
- Lab Preamble
  - Configuring and running ghc/ghci
  - Syntax Niceties
  - Problems to solve
- Do it . . .
- Lab Postamble
  - Work Submission
  - explaining any strange Error messages
Configuring GHC/GHCi

▶ You should be able to find GHCi under the All Programs item in the Start Menu.
Configuring GHC/GHCi

- You should be able to find GHCi under the All Programs item in the Start Menu.
- However, for convenient compiler use, add `C:\ghc\ghc-6.10.4\bin` to your `%PATH%` variable.
Configuring GHC/GHCi

▶ You should be able to find GHCi under the All Programs item in the Start Menu.

▶ However, for convenient compiler use, add C://ghc/ghc-6.10.4//bin to your %PATH% variable
  ▶ Start Menu → Control Panel → System → Advanced
    → Environment Variables
Configuring GHC/GHCi

- You should be able to find GHCi under the All Programs item in the Start Menu.

- However, for convenient compiler use, add `C:\ghc\ghc-6.10.4\bin` to your `%PATH%` variable:
  - [Start Menu → Control Panel → System → Advanced → Environment Variables]
  - Add `C:\ghc\ghc-6.10.4\bin` to front of user PATH variable.
Configuring GHC/GHCi

► You should be able to find GHCi under the All Programs item in the Start Menu.

► However, for convenient compiler use, add `C:\ghc\ghc-6.10.4\bin` to your `%PATH%` variable
  ▶ Start Menu → Control Panel → System → Advanced
  → Environment Variables
  ▶ Add `C:\ghc\ghc-6.10.4\bin`; to front of user PATH variable
  ▶ Save it (“Ok”)


Haskell Syntax Niceties (I)

- Haskell is case-sensitive

  ```haskell
data MyType = MyDataCons1
               | MyDataCons Int Int
  ```

- Ordinary value and function names start with lower-case

  ```haskell
remdups xs = sort (map toUpper names)
  ```

- Functions with identifier names are prefix:

  ```haskell
myfun x y = 2*x + y
  ```

  However, 2-argument identifiers can be used infix-style:

  ```haskell
'(myfun) 2 3
  ```

  (surround with backquotes)

- Functions with symbol names are infix:

  ```haskell
x <+> y = 2*x - y
  ```

  However, can be used infix-style:

  ```haskell
(<+>) 5 7
  ```

  (surround with parentheses)
Haskell Syntax Niceties (I)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case

```haskell
data MyType = MyDataCons1
            | MyDataCons Int Int
```

- Ordinary value and function names start with lower-case

```haskell
remdups xs = sort (map toUpper names)
```

- Functions with identifier names are prefix:

```haskell
myfun x y = 2*x + y
```

- However, 2-argument identifiers can be used infix-style:

```haskell
1 'myfun' 2
```

- Functions with symbol names are infix:

```haskell
x <+> y = 2*x - y
```

- However, can be used infix-style:

```haskell
(<+>) 5 7
```
Haskell Syntax Niceties (I)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case
  ```haskell
data MyType = MyDataCons1
       | MyDataCons Int Int
  ```
- Ordinary value and function names start with lower-case
  ```haskell
remdups xs = sort (map toUpper names)
  ```
Haskell Syntax Niceties (I)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case
  ```haskell
  data MyType = MyDataCons1
    | MyDataCons Int Int
  ```
- Ordinary value and function names start with lower-case
  ```haskell
  remdups xs = sort (map toUpper names)
  ```
- Functions with identifier names are prefix:
  ```haskell
  myfun x y = 2*x + y
  ```
- Functions with symbol names are infix:
  ```haskell
  (x <+> y) = 2*x - y
  ```
Haskell Syntax Niceties (I)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case
  
  ```haskell
  data MyType = MyDataCons1
               | MyDataCons Int Int
  ```

- Ordinary value and function names start with lower-case
  
  ```haskell
  remdups xs = sort (map toUpper names)
  ```

- Functions with identifier names are prefix:
  
  ```haskell
  myfun x y = 2*x + y
  ```

  - However, 2-argument identifiers can be used infix-style:
    
    ```haskell
    'myfun' 2
    ```

    (surround with back-quotes)
Haskell Syntax Niceties (1)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case
  ```haskell
data MyType = MyDataCons1
             | MyDataCons Int Int
  ```
- Ordinary value and function names start with lower-case
  ```haskell
remdups xs = sort (map toUpper names)
  ```
- Functions with identifier names are prefix:
  ```haskell
myfun x y = 2*x + y
  ```
  - However, 2-argument identifiers can be used infix-style:
    ```haskell
1 'myfun' 2
(surround with back-quotes)
  ```
- Functions with symbol names are infix:
  ```haskell
x <+> y = 2*x - y
  ```
Haskell Syntax Niceties (I)

- Haskell is case-sensitive
- Type and Data Constructors start with upper-case
  
  ```haskell
data MyType = MyDataCons1
            | MyDataCons Int Int
  ```

- Ordinary value and function names start with lower-case
  
  ```haskell
remdups xs = sort (map toUpper names)
  ```

- Functions with identifier names are prefix:
  
  ```haskell
myfun x y = 2*x + y
  ```
  - However, 2-argument identifiers can be used infix-style:
    
    ```haskell
1 'myfun' 2
   (surround with back-quotes)
    ```

- Functions with symbol names are infix:
  
  ```haskell
x <+> y = 2*x - y
  ```
  - However, can be used infix-style: `(<<+>>)) 5 7`
    (surround with parentheses)
Some Haskell syntax specifies lists like
{ item1 ; item2 ; item3 ; ... ; itemn }

The layout (or "off-side") rule takes effect whenever the open brace is omitted. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).

For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted).
Haskell Syntax Niceties (II — Layout Rule)

- Some Haskell syntax specifies lists like
  
  \{ item1 ; item2 ; item3 ; ... ; itemn \}

- In some cases (after keywords where, let, do, of), we can drop {,} and ;.
Haskell Syntax Niceties (II — Layout Rule)

▶ Some Haskell syntax specifies lists like
   { item1 ; item2 ; item3 ; ... ; itemn }
▶ In some cases (after keywords where, let, do, of), we can drop {,} and ;.
▶ The layout (or "off-side") rule takes effect whenever the open brace is omitted.
Haskell Syntax Niceties (II — Layout Rule)

- Some Haskell syntax specifies lists like
  
  ```haskell
  { item1 ; item2 ; item3 ; ... ; itemn }
  ```

- In some cases (after keywords `where`, `let`, `do`, `of`), we can drop `{,}` and `;`.

- The layout (or "off-side") rule takes effect whenever the open brace is omitted.
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
Some Haskell syntax specifies lists like
{ item1 ; item2 ; item3 ; ... ; itemn }

In some cases (after keywords where, let, do, of), we can drop {,} and ;.

The layout (or "off-side") rule takes effect whenever the open brace is omitted.

- When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
- For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
Haskell Syntax Niceties (II — Layout Rule)

- Some Haskell syntax specifies lists like
  \{ item1 ; item2 ; item3 ; ... ; itemn \}
- In some cases (after keywords \where, \let, \do, \of), we can drop \{,\} and ;.
- The layout (or "off-side") rule takes effect whenever the open brace is omitted.
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
  - For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
  - if it is indented the same amount, then a new item begins (a semicolon is inserted);
Some Haskell syntax specifies lists like
{ item1 ; item2 ; item3 ; ... ; itemn }
In some cases (after keywords where, let, do, of), we can drop {,} and ;.
The layout (or "off-side") rule takes effect whenever the open brace is omitted.
When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
if it is indented the same amount, then a new item begins (a semicolon is inserted);
and if it is indented less, then the layout list ends (a close brace is inserted).
Lab Tasks

▶ Using your favourite *plain-text* text-editor, create a file called `<username>-lab01.hs`
where `<username>` is replaced by your *TCD* username.

▶ Enter solutions to the exercises on later slides

▶ Run them using GHCi

▶ start GHCi (Start Menu)

▶ from prompt enter :
l `<path—to—your—file>`

▶ If no errors, then try to evaluate some examples

▶ If errors, call me — I want to talk about these at the end
then edit the source file, and enter :r at the prompt to reload.

▶ Email me your final version by the start of next week's lab
*(12noon, Fri. Oct 8th)*
Lab Tasks

- Using your favourite plain-text text-editor, create a file called `<username>-lab01.hs` where `<username>` is replaced by your TCD username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
- Start GHCi (Start Menu)
- From prompt enter `:l <path to your file>`
- If no errors, then try to evaluate some examples
- If errors, call me — I want to talk about these at the end
- Then edit the source file, and enter `:r` at the prompt to reload.
- Email me your final version by the start of next week's lab (12noon, Fri. Oct 8th)
Lab Tasks

- Using your favourite *plain-text* text-editor, create a file called `<username>-lab01.hs` where `<username>` is replaced by your *TCD* username.
- Enter solutions to the exercises on later slides
- Run them using GHCi

▶ Email me your final version by the start of next week's lab (12noon, Fri. Oct 8th)
Lab Tasks

- Using your favourite *plain-text* text-editor, create a file called `<username>-lab01.hs` where `<username>` is replaced by your *TCD* username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
  - start GHCi (Start Menu)
- Email me your final version by the start of next week's lab (12noon, Fri. Oct 8th)
Lab Tasks

- Using your favourite *plain-text* text-editor, create a file called `<username>-lab01.hs`
  where `<username>` is replaced by your TCD username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
  - start GHCi (Start Menu)
  - from prompt enter `:l <path-to-your-file>`
Lab Tasks

- Using your favourite *plain-text* text-editor, create a file called `<username>-lab01.hs`
  where `<username>` is replaced by your *TCD* username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
  - start GHCi (Start Menu)
  - from prompt enter :l `<path-to-your-file>`
  - If no errors, then try to evaluate some examples
- Email me your final version by the start of next week's lab *(12noon, Fri. Oct 8th)*
Lab Tasks

- Using your favourite plain-text text-editor, create a file called `<username>-lab01.hs` where `<username>` is replaced by your TCD username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
  - start GHCi (Start Menu)
  - from prompt enter :l `<path-to-your-file>`
  - If no errors, then try to evaluate some examples
  - If errors, call me — I want to talk about these at the end then edit the source file, and enter :r at the prompt to reload.
- Email me your final version by the start of next week's lab (12noon, Fri. Oct 8th)
Lab Tasks

- Using your favourite plain-text text-editor, create a file called `<username>-lab01.hs` where `<username>` is replaced by your TCD username.
- Enter solutions to the exercises on later slides
- Run them using GHCi
  - start GHCi (Start Menu)
  - from prompt enter :l `<path – to – your – file>`
  - If no errors, then try to evaluate some examples
  - If errors, call me — I want to talk about these at the end then edit the source file, and enter :r at the prompt to reload.
- Email me your final version by the start of next week’s lab (12noon, Fri. Oct 8th)
What Can I do, if I’m stuck?

▶ Class webpage
https://www.cs.tcd.ie/
    research_groups/fmg/courses/CS4011/
(bookmark now !)

▶ Haskell Libraries
(http://haskell.org/onlinereport/ — see Part II)

▶ Hoogle Search Engine
http://haskell.org/hoogle/
(can search by name.
(can search by type).
What Can I do, if I’m stuck?

- Class webpage
  https://www.cs.tcd.ie/
  research_groups/fmg/courses/CS4011/
  (bookmark now !)

- Haskell Libraries
  (http://haskell.org/onlinereport/ — see Part II)
What Can I do, if I’m stuck?

- Class webpage
  https://www.cs.tcd.ie/research_groups/fmg/courses/CS4011/
  (bookmark now !)

- Haskell Libraries
  (http://haskell.org/onlinereport/ — see Part II)

- Hoogle Search Engine http://haskell.org/hoogle/
What Can I do, if I’m stuck?

- Class webpage
  https://www.cs.tcd.ie/
  research_groups/fmg/courses/CS4011/
  (bookmark now !)

- Haskell Libraries
  (http://haskell.org/onlinereport/ — see Part II)

- Hoogle Search Engine http://haskell.org/hoogle/
  - can search by name.
What Can I do, if I’m stuck?

- Class webpage
  https://www.cs.tcd.ie/research_groups/fmg/courses/CS4011/
  (bookmark now !)

- Haskell Libraries
  (http://haskell.org/onlinereport/ — see Part II)

- Hoogle Search Engine http://haskell.org/hoogle/
  ▶ can search by name.
  ▶ can search by type.
What Can I do, if I’m stuck?

- Class webpage
  https://www.cs.tcd.ie/
  research_groups/fmg/courses/CS4011/
  (bookmark now !)

- Haskell Libraries
  (http://haskell.org/onlinereport/ — see Part II)

- Hoogle Search Engine http://haskell.org/hoogle/
  - can search by name.
  - can search by type.

- Ask me!
Problems to Solve (I)

1. A function `is_even` such that `is_even n` will be true when `n` is divisible by 2, and false otherwise.
Problems to Solve (I)

1. A function `is_even` such that `is_even n` will be true when `n` is divisible by 2, and false otherwise.

2. A function `unixname` which removes all vowels from a string (so `unixname "the house"` would be "th hs").
1. A function `is_even` such that `is_even n` will be true when `n` is divisible by 2, and false otherwise.

2. A function `unixname` which removes all vowels from a string (so `unixname "the house"` would be "th hs").

3. A function `censor` which will replace all vowels in a string with the letter `x` (so, `censor "the house"` would be thx hxxsx).
Problems to Solve (I)

1. A function `is_even` such that `is_even n` will be true when `n` is divisible by 2, and false otherwise.
2. A function `unixname` which removes all vowels from a string (so `unixname "the house"` would be "th hs").
3. A function `censor` which will replace all vowels in a string with the letter `x` (so, `censor "the house"` would be `thx hxxsx`).
4. A function `copies` such that `copies 3 "hello"` will compute `["hello", "hello", "hello"]`
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Test your version using the following features of GHCi:
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Test your version using the following features of GHCi:
  - The special name `it` refers to the result of the last successful expression evaluation
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Test your version using the following features of GHCi:
  - The special name it refers to the result of the last successful expression evaluation
  - You can give a name to an evaluation using a let-construct:
    ```ghci
    > let n = length [1,2,3]
    > n
    3
    ```
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Test your version using the following features of GHCi:
  - The special name `it` refers to the result of the last successful expression evaluation
  - You can give a name to an evaluation using a `let`-construct:
    ```haskell
    > let n = length [1,2,3]
    > n
    3
    ```
  - You can ask GHCi for information about a name:
    ```haskell
    > :i is_even
    is_even :: Integral a => a -> Bool
    ```
Problems to Solve (II)

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Test your version using the following features of GHCi:
  - The special name `it` refers to the result of the last successful expression evaluation
  - You can give a name to an evaluation using a `let`-construct:
    ```hs
    > let n = length [1,2,3]
    > n
    3
    ```
  - You can ask GHCi for information about a name:
    ```hs
    > :i is_even
    is_even :: Integral a => a -> Bool
    ```
- Use Alt-Prnt Scrn to get a snapshot of your GHCi window. Paste the image into Paint.
  Save as `<username>-lab01.png`
Expression Calculator

```haskell
data Expression = Val Float
  | Add Expression Expression
  | Multiply Expression Expression
  | Subtract Expression Expression
  | Divide Expression Expression
deriving Show

eval :: Expression -> Float
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Multiply x y) = eval x * eval y
eval (Subtract x y) = eval x - eval y
eval (Divide x y) = eval x / eval y
```