Haskell Basics
Contents

1. Jump into Haskell
2. Historical Background of Haskell
3. Lazy, Pure, and Functional Language
4. Using ghc and ghci (more detail)
5. Functions
6. Haskell Scripts
7. Exercises
Using GHC and GHci

From a shell window, the compiler is invoked as

> ghc myfile.hs

> ghci (or as > ghc --interactive)

```
> sqrt (3^2 + 4^2)
5.0
>
```
Demo some basics
Historical Background

1930s:

Alonzo Church develops the *lambda calculus*, a simple but powerful theory of functions. Proved that Peano arithmetic and first-order logic are undecidable.

Sources: [http://www.alpcentauri.info/church.jpg](http://www.alpcentauri.info/church.jpg)
Historical Background

1950s:

John McCarthy develops *Lisp*, the first functional language, with some influences from the lambda calculus, but retaining variable assignments. (Organizes the Dartmouth conference, becomes one of the founders of the field of A.I.)

Historical Background

1960s:

Peter Landin develops ISWIM, the first pure functional language, based strongly on the lambda calculus, with no assignments. (Also his “off-side” rule, which we see in Haskell, Java, …)

Sources: http://www.theguardian.com/technology/2009/sep/22/peter-landin-obituary
Historical Background

1970s:

John Backus develops *FP*, a functional language that emphasizes higher-order functions and reasoning about programs.

Robin Milner and others develop *ML*, the first modern functional language, which introduced type inference and polymorphic types.

Historical Background

1970s - 1980s:

David Turner develops a number of lazy functional languages, culminating in the *Miranda* system.

Sources: [http://www.codemesh.io/codemesh2013](http://www.codemesh.io/codemesh2013)
Historical Background

1987:

An international committee of researchers initiates the development of Haskell, a standard lazy pure functional language.
Historical Background

2003:

The committee publishes the Haskell 98 report, defining a stable version of the language.
Historical Background

Since 2003?

- Next round of standardization has begun: Haskell’ (Haskell prime)
  - a continuous standardization process

- Status in 2016?
  - A widely used and highly influential language for programming language research
  - Reasonably widely used in open-source software
  - Modest commercial use
Haskell is a

Lazy

Pure

Functional Language
“Haskell is a Lazy Pure Functional Language”

A functional language supports the functional programming style where the basic method of computation is application of functions to arguments. For example, in C,

```c
int s = 0;
for (int i=1; i <= 100; ++i) s = s + i;
```

the computation method is variable assignment.

In Haskell,

```haskell
sum [1..100]
```

the computation method is function application.
“Haskell is a Lazy Pure Functional Language”

A *pure* functional language, as with mathematical functions, prohibits side effects (or at least they are confined):

- Immutable data: Instead of altering existing values, altered copies are created and the original is preserved, thus, there’s no destructive assignment:
  
  \[
  a = 1; \ a = 2; \quad -- \ illegal
  \]

- Referential transparency: Expressions yield the same value each time they are invoked; helps reasoning. Such expression can be replaced with its value without changing the behavior of a program, for example,

  \[
  y = f \ x \quad \text{and} \quad g = h \ (f \ x) \ (f \ x)
  \]

  then, replacing the definition of \( g \) with

  \[
  g = h \ (f \ x) \ (f \ x)
  \]

  will get the same result (value).
“Haskell is a **Lazy** Pure Functional Language”

A *lazy* programming language only evaluates arguments when strictly necessary, thus,

1. avoiding unnecessary computation and
2. ensuring that programs terminate whenever possible.

For example, given the definitions

```plaintext
omit x = 0
keep_going x = keep_going (x+1)
```

what is the result of the following expression?

```plaintext
omit (keep_going 1)
```
Features of **Functional** Languages

- Higher-order functions are functions that take other functions as their arguments. E.g.,

  ```
  > map reverse ["abc","def"]
  ["cba","fed"]
  ```

- Purity – prohibits side effects
  (Expressions may result in some actions in addition to return values, such as changing state and I/O; these actions are called side effects.)

- Recursion – the canonical way to iterate in functional languages
A Taste of Haskell

\[
\begin{align*}
f \; [] & \; = \; [] \\
f \; (x:xs) & \; = \; f \; ys \; ++ \; [x] \; ++ \; f \; zs \\
\text{where} & \\
ys & \; = \; [a \mid a \leftarrow xs, \; a \leq x] \\
zs & \; = \; [b \mid b \leftarrow xs, \; b > x]
\end{align*}
\]
Other Characteristics of Haskell

- Statically typed
- Type inference
- Rich type system
- Succinct, expressive syntax yields short programs
- Indentation matters
- Capitalization of names matters
Using GHC and GHCi

- From a shell window, the compiler is invoked as
  > ghc myfile.hs
  > ghci (or as > ghc --interactive)
- For multi-file programs, use --make option
- GHCi operates on an eval-print-loop:

```haskell
> sqrt (3^2 + 4^2)
5.0
>
```
Using GHC and GHCI

- From a shell window, the compiler is invoked as
  
  > ghc myfile.hs

  > ghci (or as > ghc --interactive)

- For multi-file programs, use --make option

- Important: Make your edit-compile-run cycle convenient!

- I’m going to use vim

- Some people use Emacs and “haskell-mode”

  https://github.com/serras/emacs-haskell-tutorial/blob/master/tutorial.md
Using GHCi

- **Useful basic GHCi commands:**
  - `:?` Help! Show all commands
  - `:load test` Open file test.hs or test.lhs
  - `:reload` Reload the previously loaded file
  - `:main a1 a2` Invoke main with command line args a1 a2
  - `:!` Execute a shell command
  - `:edit name` Edit script name
  - `:edit` Edit current script
  - `:type expr` Show type of expr
  - `:quit` Quit GHCi

- Commands can be abbreviated. E.g., `:r` is `:reload`
- At startup, the definitions of the “Standard Prelude” are loaded
- Hint: GHCi executes commands from `$HOME/.ghci`, then from `./.ghci` at startup
Demo some basics with lists
The Standard Prelude

Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

-- Select the first element of a list:

```haskell
> head [1,2,3,4,5]
1
```

-- Remove the first element from a list:

```haskell
> tail [1,2,3,4,5]
[2,3,4,5]
```
-- Select the nth element of a list:

\[
> \ [1,2,3,4,5] \ !! \ 2 \\
\]	3

-- Select the first n elements of a list:

\[
> \ \text{take} \ 3 \ [1,2,3,4,5] \\
\] [1,2,3]

-- Remove the first n elements from a list:

\[
> \ \text{drop} \ 3 \ [1,2,3,4,5] \\
\] [4,5]

-- Append two lists:

\[
> \ [1,2,3] \ ++ \ [4,5] \\
\] [1,2,3,4,5]
-- Reverse a list:

```plaintext
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

-- Calculate the length of a list:

```plaintext
> length [1,2,3,4,5]
5
```

-- Calculate the sum of a list of numbers:

```plaintext
> sum [1,2,3,4,5]
15
```

-- Calculate the product of a list of numbers:

```plaintext
> product [1,2,3,4,5]
120
```
Demo some basics with functions
Functions (1)

- Function and parameter names must start with a lower case letter, e.g., `myFun1`, `arg_x`, `personName`, etc.
  - By convention, list arguments usually have an `s` suffix on their name, e.g., `xs`, `ns`, `nss`

- Functions are defined as equations:
  - `square x = x * x`
  - `add x y = x + y`

- Once defined, apply the function to arguments:
  - `> square 7`
  - `49`
  - `> add 2 3`
  - `5`
  - In C, these calls would be `square(7)`; and `add(2,3)`;

- Parentheses are often needed in Haskell too
  - `> add (square 2) (add 2 3)`
  - `9`
Functions (2)

- Function application has the highest precedence
  \[ \text{square } 2 + 3 \text{ means } (\text{square } 2) + 3 \text{ not } \text{square } (2+3) \]
- Function call associates to the left and is by pattern matching (first one to match is used)
- Function application operator $\$\text{ has the lowest precedence and is used to rid of parentheses.} \]
  \[ \text{sum } ([1..5] ++ [6..10]) \rightarrow \text{sum } $ [1..5] ++ [6..10] \]
- Combinations of most symbols are allowed as function
  \[ x \ #@%&^-+-@#$%^ y = "\text{What on earth?}" \]
  Another (more reasonable) example:
  \[ x \ +/\ y = (x+y, x-y) \]
  \[ > 10 \ +/\ 1 \]
  \[ (11,9) \]
Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

\[ f(a,b) + c d \]

Apply the function \( f \) to \( a \) and \( b \), and add the result to the product of \( c \) and \( d \)

In Haskell, function application is denoted using space, and multiplication is denoted using \(*\).

\[ f \ a \ b + c*d \]

As previously, but in Haskell syntax
Examples

Mathematics

\[ f(x) \]
\[ f(x, y) \]
\[ f(g(x)) \]
\[ f(x, g(y)) \]
\[ f(x)g(y) \]

Haskell

\[ f \ x \]
\[ f \ x \ y \]
\[ f \ (g \ x) \]
\[ f \ x \ (g \ y) \]
\[ f \ x * \ g \ y \]
Evaluating Functions (1)

Think of evaluating functions as substitution and reduction

\[
\text{add } x \ y = x + y; \quad \text{square } x = x * x
\]

\[
\text{add (square 2) (add 2 3)}
\]
add \( x \ y = x + y \); \hspace{1em} \text{square } x = x \times x \\
add \ (\text{square } 2) \ (\add 2 3)

-- apply square:
add \( (2 \times 2) \ (\add 2 3) \)
-- apply *:
add 4 \ (\add 2 3) 
-- apply inner add:
add 4 \ (2 + 3) 
-- apply +:
add 4 5 
-- apply add 
4 + 5 
-- apply + 
9
Evaluating Functions (2)

- There are many possible orders to evaluate a function

\[
\text{head} \left(1 : (\text{reverse } [2, 3, 4, 5])\right) \quad \text{head} \left(1 : (\text{reverse } [2, 3, 4, 5])\right)
\]

-- apply reverse

-- ... many steps omitted here

\[
\text{head} \left(1 : [5, 4, 3, 2]\right)
\]

-- apply head

1

- Was the same output a fluke?
Evaluating Functions (2)

- There are many possible orders to evaluate a function
- In a pure functional language, evaluation order does not affect the value of the computation
- It can, however, affect the amount of computation and whether the computation terminates or not (or fails with a run-time error)
- Haskell evaluates a function’s argument lazily
  
  “Call-by-need” - only apply a function if its value is needed, and “memoize” what’s already been evaluated
A Taste of Haskell

\[
\begin{align*}
f \; [] & \; = \; [] \\
f \; (x:xs) & \; = \; f \; ys \; ++ \; [x] \; ++ \; f \; zs \\
\text{where} \\
ys & \; = \; [a \mid a \leftarrow xs, \; a \; \leq \; x] \\
zs & \; = \; [b \mid b \leftarrow xs, \; b \; > \; x]
\end{align*}
\]
Haskell Scripts

A Haskell program consists of one or more scripts.
A script is a text file comprising a sequence of definitions, where new functions are defined.
By convention, Haskell scripts usually have a .hs suffix on their filename. This is useful for identification purposes.
Loading new script causes new definitions to be in scope:

Prelude> :l test.hs
[1 of 1] Compiling Main      ( test.hs, interpreted )
Ok, modules loaded: Main.
(Main>
My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi:

Start an editor, type in the following two function definitions, and save the script as test.hs:

```haskell
double x = x + x
quadruple x = double (double x)
```

In another window start up GHCi with the new script:

```
% ghci test.hs
```

Now both the standard library and the file test.hs are loaded:

```
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```
Leaving GHCi open, return to the editor, add the following definitions, and resave:

```
factorial n = product [1..n]
average ns = sum ns `div` length ns
```

**Note:**
- `div` is enclosed in back quotes, not forward ones
- `x `f` y` is syntactic sugar for `f x y`
- Any function with two (or more args) can be used as an infix operator (enclosed in back quotes)
- Any infix operator can be used as a function (enclosed in parentheses), e.g., `(+) 10 20`
- GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
288> :r
Reading file "test.hs"

> factorial 10
3600

> average [1,2,3,4,5]
3
```
The Layout Rule

- Layout of a script determines the structure of definitions
- Commonly use layouts instead of braces and semicolons (which are still allowed and can be mixed with layout)
- Each definition must begin in precisely the same column:

```
\begin{align*}
    a &= 10 \\
    b &= 20 \\
    c &= 30 \\
    a &= 10 \quad \text{where} \quad b = 1 \\
    \hspace{1cm} c = 2 \\
    d &= a \times 2 \\
\end{align*}
```

- Implicit grouping by layout
- Explicit grouping by braces and semicolons
Exercises

(1) Try out the codes in slides 15-24 and 30-32 using GHCi.

(2) Fix the syntax errors in the program below, and test your solution using GHCi.

\[ N = a \div \text{length} \ xs \]
\[ \text{where} \]
\[ a = 10 \]
\[ xs = [1,2,3,4,5] \]

\[ n = a \ `\div` \text{length} \ xs \]
\[ \text{where} \]
\[ a = 10 \]
\[ xs = [1,2,3,4,5] \]
(3) Show how the library function last that selects the last element of a list can be defined using the functions introduced in this lecture.

\[
\text{last } xs = \ldots
\]

(4) Can you think of another possible definition?

\[
\text{last } xs = \ldots
\]

(5) Similarly, show how the library function init that removes the last element from a list can be defined in two different ways.

\[
\text{init } xs = \ldots
\]

\[
\text{init } xs = \ldots
\]