2 - First Steps

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell, and comprises a compiler and an interpreter;

- The interactive nature of the interpreter makes it well suited for teaching and prototyping;

- GHC is freely available from:

```
www.haskell.org/platform
```

Starting GHCi

The interpreter can be started from the terminal command prompt by simply typing **`ghci`**:

```
$ ghci

GHCi, version X: http://www.haskell.org/ghc/  :? for help

Prelude>
```

The interpreter is now ready to evaluate expressions

Can be used as a desktop calculator:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0

> mod 4 2
0
```

The Standard Prelude

Haskell comes with a large number of standard library functions, in particular useful functions on ***lists***

- Select the first element of a list:

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

- Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Select the n-th element of a list:

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

- Select the first n elements of a list:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```


- Remove the first n elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

- Calculate the length of a list:

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

- Calculate the sum of a list of numbers:

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


- Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

- Reverse a list:

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

- Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

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.

Moreover, function application is assumed to have ***higher priority*** than all other operators.

```
f a + b
```

Means (f a) + b, rather than f (a + b).

Examples

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

Haskell Scripts

- New functions are defined within a ***script***, a text file comprising a sequence of definitions
- By convention, `Haskell` scripts usually have a ***.hs*** suffix on their filename
- Example, the script/file **`test.hs`**

```
double x = x + x

quadruple x = double (double x)
```

To start up `GHCi` with the new script:

```
$ ghci test.hs
```

Both the standard library and the file **`test.hs`** are loaded, and functions from both can be used:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Exercise: write `factorial` with no recursive call

```
factorial n = product [1..n]
```

A ***reload*** command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800
```

Exercise: write a function which computes the `average` of the elements of a list

```
average ns = div (sum ns) (length ns)
```

```
average ns = sum ns `div` length ns
```

- `div` is enclosed in **back** quotes, not forward
- x `f` y is just ***syntactic sugar*** for f x y

Useful GHCi Commands

Command	Meaning
:load name	load script name
:reload	reload current script
:set editor name	set editor to name
:edit name	edit script name
:edit	edit current script
:type expr	show type of expr
:?	show all commands
:quit	quit GHCi

Naming Requirements

- Function and argument names must begin with a lower-case letter:

```
myFun
```

```
fun1
```

```
arg_2
```

```
x'
```

- By convention, list arguments usually have an s suffix on their name:

```
xs
```

```
ns
```

```
nss
```

The Layout Rule

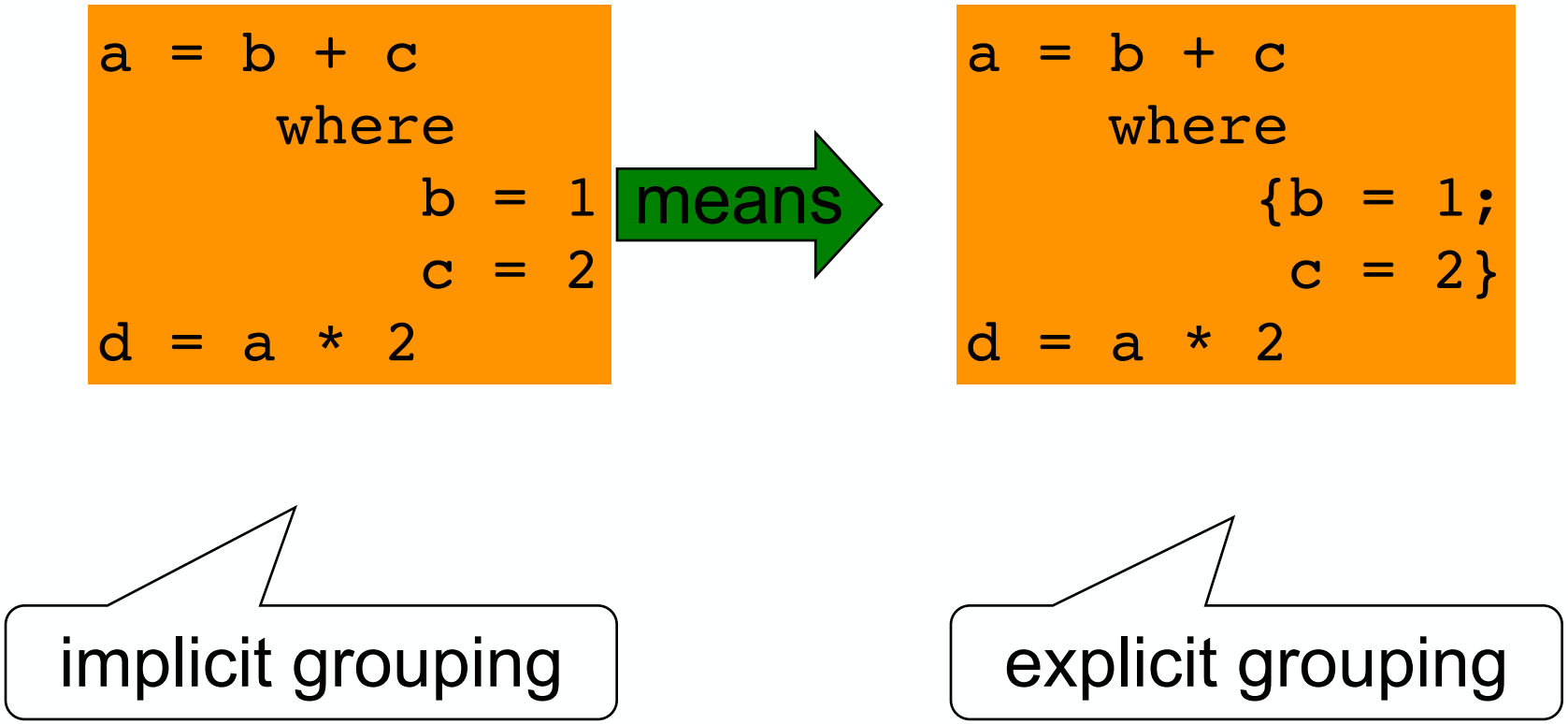
In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10
b = 20
c = 30
```

```
a = 10
  b = 20
c = 30
```

```
 a = 10
b = 20
  c = 30
```

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.



Scripts can contain one line comments

```
-- Factorial of a positive integer
factorial n = product [1..n]
```

as well as nested comments which could span on multiple lines

```
{-
double x = x + x

quadruple x = double (double x)
-}
```