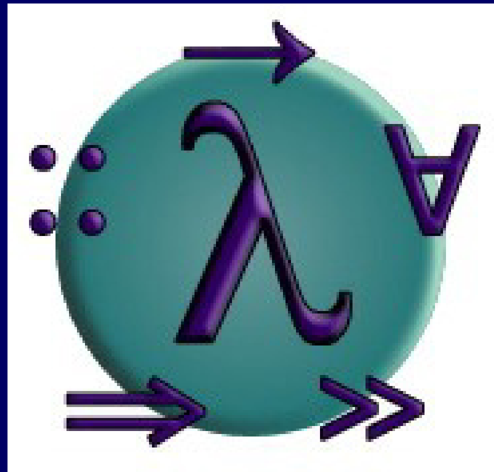
PROGRAMMING IN HASKELL



Part 1 - Introduction

Starting GHCi

On a Unix system, GHCi can be started from the % prompt by simply typing ghci:

```
% ghci

> GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

The GHCi > prompt means that the ghci system is ready to evaluate an expression.

For example:

```
> 2+3*4
14

> (2+3)*4
20

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

The Standard Prelude

The library file Prelude.hs provides a large number of standard 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:

```
> 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 nth 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]
```

Haskell Scripts

- As well as the functions in the standard prelude, you can also define your own functions;

- 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. This is not mandatory, but is useful for identification purposes.

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:

```
double x    = x + x

quadruple x = double (double x)
```

Leaving the editor open, in another window start up ghci with the new script:

```
% ghci test.hs
```

Now both Prelude.hs and test.hs are loaded, and functions from both scripts can be used:

```
> quadruple 10
40

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

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`

Naming Requirements

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

`myFun` `fun1` `arg_2` `x’`

- By convention, list arguments usually have an <u>s</u> suffix on their name. For example:

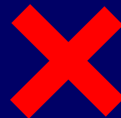
`xs` `ns` `nss`

The Layout Rule

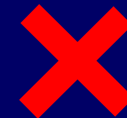
In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10

b = 20

c = 30
```

```
a = 10

 b = 20

c = 30
```

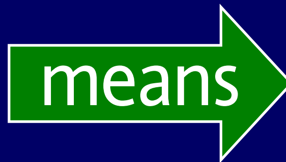


```
 a = 10

b = 20

 c = 30
```



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

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```




```
a = b + c
     where
       {b = 1;
        c = 2}
d = a * 2
```

implicit grouping

explicit grouping

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

```
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

- In Haskell, conditional expressions must always have an else branch.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

As previously, but using guarded equations.

Guarded equations make definitions involving multiple conditions easier to read:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

- The catch-all condition <u>otherwise</u> is defined in the prelude by **otherwise = True**.

Pattern Matching

Many functions have a particularly clear definition using <u>pattern matching</u> on their arguments.

```
not False = True
not True  = False
```

`not` maps **`False`** to **`True`**, and **`True`** to **`False`**.

Functions can often be defined in many different ways using pattern matching. For example

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is **`False`**:

```
True  && b = b
False && _ = False
```

- The underscore symbol _ is a <u>wildcard</u> pattern that matches any argument value.

- Patterns are matched <u>in order</u>. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```

- Patterns may not <u>repeat</u> variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (`:`) called "cons" that adds an element to the start of a list.

```
[1,2,3,4]
```

Means **1:(2:(3:(4:[])))**.

Functions on lists can be defined using x:xs patterns.

```
head (x:_)  = x

tail (_:xs) = xs
```

head and **tail** map any non-empty list to its first and remaining elements.

Note:

- **x:xs** patterns only match non-empty lists:

```
> head []
*** Exception: Prelude.head:
empty list
```

- **x:xs** patterns must be parenthesised, because application has priority over (:). For example, the following definition gives an error:

```
head x:_ = x
```