PROGRAMMING IN HASKELL

Chapter 2 - First Steps

The Hugs System

- Hugs is an implementation of Haskell 98, and is the most widely used Haskell system;

- The interactive nature of Hugs makes it well suited for teaching and prototyping purposes;

- Hugs is available on the web from:

  `www.haskell.org/hugs`

Starting Hugs

On a Unix system, Hugs can be started from the % prompt by simply typing <u>hugs</u>:

```
% hugs

__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   ||                         _________________________________________

>
```

The Hugs > prompt means that the Hugs 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]
```

- 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
```

- Append two lists:

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

- Reverse a list:

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

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

- 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 Hugs.

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 Hugs with the new script:

```
% hugs 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]
```

Leaving Hugs open, return to the editor, add the following two definitions, and resave:

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

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

Note:

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

Hugs does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

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

> factorial 10
3628800

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

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
```

means

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

implicit grouping

explicit grouping

Useful Hugs Commands

Command	Meaning
:load *name*	load script *name*
:reload	reload current script
:edit *name*	edit script *name*
:edit	edit current script
:type *expr*	show type of *expr*
:?	show all commands
:quit	quit Hugs

Exercises

(1) Try out slides 2-8 and 14-17 using Hugs.

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

```
N = a 'div' length xs
    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.

(4) Can you think of another possible definition?

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

PROGRAMMING IN HASKELL

Chapter 3 - Types and Classes

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False        True
```

Type Errors

Applying a function to one or more arguments of the wrong type is called a <u>type error</u>.

```
> 1 + False
Error
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e <u>has type</u> t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called <u>type inference</u>.

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In Hugs, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

`Bool`	-	logical values
`Char`	-	single characters
`String`	-	strings of characters
`Int`	-	fixed-precision integers
`Integer`	-	arbitrary-precision integers
`Float`	-	floating-point numbers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

Note:

- The type of a list says nothing about its length:

```
[False,True]       :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Note:

- The type of a tuple encodes its size:

```
(False,True)       :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

Function Types

A <u>function</u> is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

In general:

$t1 \to t2$ is the type of functions that map values of type t1 to values to type t2.

Note:

- The arrow $\rightarrow$ is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add         :: (Int,Int) → Int
add (x,y)    = x+y

zeroto      :: Int → [Int]
zeroto n     = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning <u>functions as results</u>:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function <u>add' x</u>. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add′ produce the same final result, but add takes its two arguments at the same time, whereas add′ takes them one at a time:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called <u>curried</u> functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult       :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function <u>mult x</u>, which in turn takes an integer y and returns a function <u>mult x y</u>, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow $\rightarrow$ associates to the right.

```
Int → Int → Int → Int
```

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int)).

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```

Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

for any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

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

a = Bool

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

Overloaded Functions

A polymorphic function is called <u>overloaded</u> if its type contains one or more class constraints.

```
sum :: Num a ⇒ [a] → a
```

for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> sum [1,2,3]
6

> sum [1.1,2.2,3.3]
6.6

> sum ['a','b','c']
ERROR
```

a = Int

a = Float

Char is not a numeric type

- Haskell has a number of type classes, including:

  `Num` - Numeric types

  `Eq` - Equality types

  `Ord` - Ordered types

- For example:

```
(+)  :: Num a ⇒ a → a → a

(==) :: Eq a  ⇒ a → a → Bool

(<)  :: Ord a ⇒ a → a → Bool
```

Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;

- Within a script, it is good practice to state the type of every new function defined;

- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Exercises

(1) What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) What are the types of the following functions?

```
second xs     = head (tail xs)

swap (x,y)    = (y,x)

pair x y      = (x,y)

double x      = x*2

palindrome xs = reverse xs == xs

twice f x     = f (f x)
```

(3) Check your answers using Hugs.

PROGRAMMING IN HASKELL

Chapter 4 - Defining Functions

Conditional Expressions

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

```
abs  :: Int → Int
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  :: Int → Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must <u>always</u> have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using <u>guarded equations</u>.

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

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

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

Note:

- 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       :: Bool → Bool
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

```
(&&)                 :: Bool → Bool → Bool
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
```

Note:

- The underscore symbol _ is a wildcard pattern that matches any argument value.

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

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

- Patterns may not repeat 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          :: [a] → a
head (x:_)     = x

tail          :: [a] → [a]
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 []
Error
```

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

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

Integer Patterns

As in mathematics, functions on integers can be defined using n+k patterns, where n is an integer variable and k>0 is an integer constant.

```
pred        :: Int → Int
pred (n+1) = n
```

pred maps any positive integer to its predecessor.

Note:

- n+k patterns only match integers $\geq$ k.

```
> pred 0
Error
```

- n+k patterns must be parenthesised, because application has priority over +. For example, the following definition gives an error:

```
pred n+1 = n
```

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

$$\lambda x \;\rightarrow\; x+x$$

the nameless function that takes a number x and returns the result x+x.

Note:

- The symbol λ is the Greek letter <u>lambda</u>, and is typed at the keyboard as a backslash \.

- In mathematics, nameless functions are usually denoted using the $\mapsto$ symbol, as in $x \mapsto x+x$.

- In Haskell, the use of the λ symbol for nameless functions comes from the <u>lambda calculus</u>, the theory of functions on which Haskell is based.

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using <u>currying</u>.

For example:

```
add x y = x+y
```

means

```
add = λx → (λy → x+y)
```

Lambda expressions are also useful when defining functions that return functions as results.

For example:

```
const     :: a → b → a
const x _ = x
```

is more naturally defined by

```
const   :: a → (b → a)
const x = λ_ → x
```

Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

For example:

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Sections

An operator written <u>between</u> its two arguments can be converted into a curried function written <u>before</u> its two arguments by using parentheses.

For example:

```
> 1+2
3

> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
3

> (+2) 1
3
```

In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called <u>sections</u>.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

`(1+)` - successor function

`(1/)` - reciprocation function

`(*2)` - doubling function

`(/2)` - halving function

Exercises

(1) Consider a function <u>safetail</u> that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function null :: [a] $\rightarrow$ Bool can be used to test if a list is empty.

(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:

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

(4) Do the same for the following version:

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