

### INTRODUCTION

The standard prelude and the Haskell libraries, Hugs commands , Modules, Hugs Session, Basic types and definitions, Designing and writing programs in Haskell , Recursion , Primitive recursion in practice General forms of recursion , Program testing.

### Introduction

A functional programming language gives a simple model of programming: one value, the result, is computed on the basis of others, the inputs. Because of its simple foundation, a functional language gives the clearest possible view of the central ideas in modern computing, including abstraction (in a function), data abstraction (in an abstract data type), genericity, polymorphism and overloading. This means that a functional language provides not just an ideal introduction to modern programming ideas, but also a useful perspective on more traditional imperative or object-oriented approaches. For example, Haskell gives a direct implementation of data types like trees, whereas in other languages one is forced to describe them by pointer-linked data structures.

Functional programming offers a high-level view of programming, giving its users a variety of features which help them to build elegant yet powerful and general libraries of functions. Central to functional programming is the idea of a function. Which computes a result that depends on the values of its inputs. The elegance of functional programming is a consequence of the way that functions are defined: an equation is used to say what the value of a function is on an arbitrary input.

A simple illustration is the function **addDouble** which adds two integers and doubles their sum. Its definition is

*addDouble*  $x\ y = 2 * (x+y)$

where  $x$  and  $y$  are the inputs and  $2 * (x+y)$  is the result.

The model of functional programming is simple and clean: to work out the value of an expression like

*3+addDouble 7 8*

the equations which define the functions involved in the expression are used. So

*3+addDouble 7 8*

*3+2\*(7+8)*

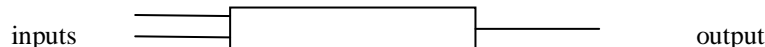
*33*

This is how a computer would work out the value of the expression, but it is also possible to do exactly the same calculation using pencil and paper, making transparent the implementation mechanism.

### What is a function?

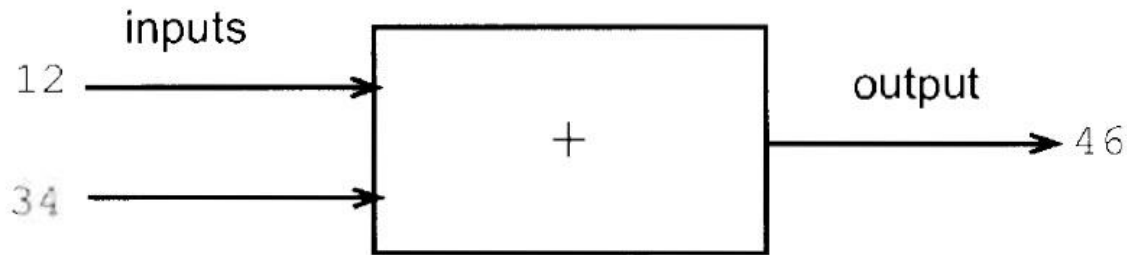
A **function** is something which we can picture as a box with some inputs and an output,

thus:



The function gives an **output** value which depends upon the **input** value(s). We will often use the term **result** for the output, and the terms **arguments** or **parameters** for the inputs.

A simple example of a function is addition,  $+$ , over numbers. Given input values 12 and 34 the corresponding output will be 46.



The process of giving particular inputs to a function is called **function application**, and  $(12 + 34)$  represents the application of the function  $+$  to 12 and 34.

## The Haskell programming language

Haskell (Peyton Jones and Hughes 1998) is the functional programming language which we use in this text. However, many of the topics we cover are of more general interest and apply to other functional languages (as discussed in Chapter 20), and indeed are Lessons for programming in general. Nevertheless, [he book is of most value as a text on functional programming in the Haskell language. Haskell is named after Haskell B. Curry who was one of the pioneers of the  $\lambda$  calculus (lambda calculus), which is a mathematical theory of functions and has been an inspiration to designers of a number of functional languages. Haskell was first Specified in the late 1980s. and has since gone through a number of revisions before reaching its current 'standard' state. There are a variety of implementations of Haskell available; in this text we shall use the **Hugs (1998)** system. We feel that Hugs provides the best environment for the learner, since it is freely available for PC, Unix and Macintosh systems, it is efficient and compact and has a flexible user interface.

Hugs is an interpreter - which means loosely that it evaluates expressions step-by-step as we might on a piece of paper - and so it will be less efficient than a compiler which translates Haskell programs directly into the machine language of a computer. Compiling a language like Haskell allows its programs to run with a speed similar to those written in more conventional languages like C and C++. Details of all the different implementations of Haskell can be found at the Haskell home page,

<http://www.haskell.org/>.

From now on we shall be using the Haskell programming language and the Hugs system as the basis of our exposition of the ideas of functional programming.

## Expressions and evaluation

In our first years at school we learn to **evaluate** an **expression** like  $(7 - 3) * 2$  expression value

$$(7 - 3) * 2 \xrightarrow{\text{Evaluation}} 8$$

to give the **value** 8. This expression is built up from symbols for numbers and for functions over those numbers: subtraction  $-$  and multiplication  $*$ ; the value of the expression is a number. This process of evaluation is automated in an electronic calculator.

## Definitions

A functional program in Haskell consists of a number of **definitions**. A Haskell definition associates a **name** (or **identifier**) with a value of a particular **type**. In the simplest case a definition will have the form

```
name :: type
name = expression
```

as in the example

```
size :: Int
size = 12+13
```

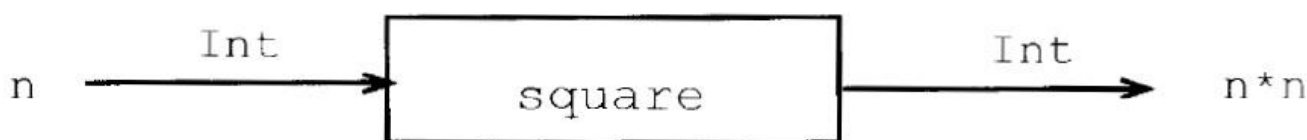
Which associates the name on the left-hand side, **size**, with the value of the expression on the right-hand side, 25, a value whose type is **Int**, the type of whole numbers or integers. The symbol '::' should be read as 'is of type', so the first line of the last definition reads '**size** is of type **Int**'. Note also that names for functions and other values begin with a small letter, while type names begin with a capital letter.

## Function definitions

We can also define functions, and we consider some simple examples now. To square an integer we can say

```
square :: Int -> Int
square n = n*n
```

where diagrammatically the definition is represented by

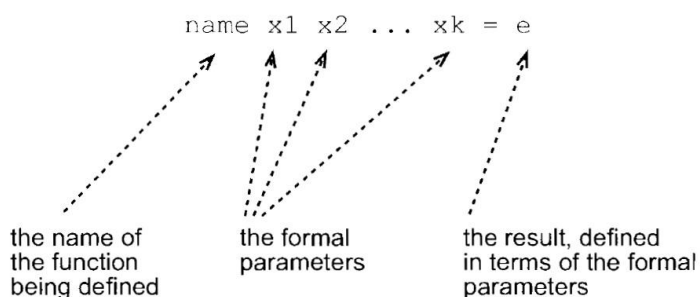


The first line of the Haskell definition of square declares the type of the thing being defined: this states that square is a function - signified by the arrow `->` - which has a single argument of type `Int` (appearing before the arrow) and which returns a result of type `Int` (coming after the arrow). The second line gives the definition of the function: the equation states that when square is applied to an unknown or variable `n`, then the result is `n*n`. How should we read an equation like this? Because `n` is an arbitrary, or unknown value, it means that the equation holds *whatever the value of `n`*, so that it will hold whatever integer expression we put in the place of `n`, having the consequence that, for instance

```
square 5 = 5*5
and
square (2+4) = (2+4)*(2+4)
```

This is the way that the equation is used in evaluating an expression which uses square. If we are required to evaluate square applied to the expression `e`, we replace the application `square e` with the corresponding right-hand side, `e*e`.


In general a simple function definition will take the form




The variables used on the left-hand side of an equation defining a function are called the formal parameters because they stand for arbitrary values of the parameters (or actual parameters, as they are sometimes known). We will only use 'formal' and 'actual' in the text when we need to draw a distinction between the two; in most cases it will be obvious which is meant when 'parameter' is used.

Accompanying the definition of the function is a declaration of its type. This will take the following form, where we use the function `scale` over pictures for illustration:


`scale :: Picture -> Int -> Picture`



the function  
name



the types of  
the arguments



the type of  
the result

## First Haskell program

A first Haskell program or **script**, which consists of the numerical examples. As well as definitions, a script will contain comments.

```
{-#####
```

```
    FirstScript.hs
```

```
    Simon Thompson, June 1998
```

```
    The purpose of this script is
```

- to illustrate some simple definitions over integers (Int);
- to give a first example of a script.

```
#####-}
```

```
--      The value size is an integer (Int), defined to be
--      the sum of twelve and thirteen.
```

```
size :: Int
size = 12+13
```

```
--      The function to square an integer.
```

```
square :: Int -> Int
square n = n*n
```

```
--      The function to double an integer.
```

```
double :: Int -> Int
double n = 2*n
```

```
--      An example using double, square and size.
```

```
example :: Int
example = double (size - square (2+2))
```

A **comment** in a script is a piece of information of value to a human reader rather than to a computer. It might contain an informal explanation about how a function works how it should or should not be used, and so forth. Comments are indicated in two ways. The symbol '--' begins a comment which occupies the part of the line to the right of the symbol. Comments can also be enclosed by the symbols '{-' and '-}'. These comments can be of arbitrary length, spanning more than one line, as well as enclosing other comments; they are therefore called **nested comments**.

## Using Hugs

Hugs is a Haskell implementation which runs on both PCs (under Windows 95 and NT) and Unix systems, including Linux. It is freely available via the Haskell home page,

<http://www.haskell.org/hugs/>

This is a source of much material on Haskell and its implementation.

## Hugs Commands

Hugs commands begin with a colon, ':'. A summary of the main commands follows.

Command	Description
:load parrot Or :l parrot	Load the Haskell file parrot. hs or parrot. lhs. The file extension .hs or .lhs does not need to be included in the filename.
:reload	Repeat the last load command
:edit filename.hs	Edit the file first . lhs in the default editor. Note that the file extension. hs or . lhs is needed in this case. See the following section for more information on editing.
:type exp	Give the type of the expression exp. For example, the result of typing: type size+2 is Int.
: info name	Give information about the thing named name.
: find name	Open the editor on the file containing the definition of name.
:quit	Quit the system.
:?	Give a list of the Hugs commands.
! com	Escape to perform the Unix or DOS command com.

## Editing Scripts

Hugs can be connected to a 'default' text editor, so that Hugs commands such as :edit and :find use this editor. This may well be determined by your local set-up. The 'default' default editor on Unix is vi; on Windows systems edit or notepad might be used. Using the Hugs :edit command causes the editor to be invoked on the appropriate file. When the editor is quit, the updated file is loaded automatically. However, it is Using Hugs 25 more convenient to keep the editor running in a separate window and to reload the file by writing the updated file from the editor (without quitting it), and then reloading the file in Hugs using : reload or :reload filename.

## The standard prelude and the Haskell libraries

Haskell has various built-in types, such as integers and lists and functions over those types, including the arithmetic functions and the list functions `map` and `++`. Definitions of these are contained in a file, the standard prelude, `Prelude.hs`. When Haskell is used, the default is to load the standard prelude.

### Modules

A typical piece of computer software will contain thousands of lines of program text. To make this manageable, we need to split it into smaller components, which we call modules. A module has a name and will contain a collection of Haskell definitions.

To introduce a module called **Ant** we begin the program text in the file thus:

*module Ant where*

A module may also import definitions from other modules. The module **Bee** will import the definitions in **Ant** by including an **import** statement, thus:

*module Bee where*

*import Ant*

*...*

The import statement means that we can use all the definitions in **Ant** when making definitions in **Bee**.

## Basic types and definitions

Haskell contains a variety of numerical types. We have already seen the `Int` type in use; we shall cover this and also the type `Float` of **floating-point** fractional numbers. Often in programming we want to make a choice of values, according to whether or not a particular **condition** holds. Such conditions include tests of whether one number is greater than another; whether two values are equal, and so on. The results of these tests - True if the condition holds and False if it fails - are called the **Boolean** values, after the nineteenth-century logician George Boole, and they form the Haskell type `Bool`. The Booleans and how they are used to give choices in function Definitions by means of **guards**. Finally, we look at the type of characters - individual letters, digits, spaces and so forth - which are given by the Haskell type `Char`.

### The Booleans: `Bool`

The Boolean values `True` and `False` represent the results of tests, which might, for instance, compare two numbers for equality, or check whether the first is smaller than the second. The Boolean type in Haskell is called `Bool`. The Boolean operators provided in the language are:

`&&` *and*

`//` *or*

`not` *not*

Because `Bool` contains only two values, we can define the meaning of boolean operators by **truth tables** which show the result of applying the operator to each possible combination of arguments. For instance, the third line of the first table says that the value of `False && True` is `False` and that the value of `False || True` is `True`.

$t_1$	$t_2$	$t_1 \ \&\& \ t_2$	$t_1 \    \ t_2$	$t_1$	$\text{not } t_1$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	F	T		
F	F	F	F		

Booleans can be the arguments to or the results of functions. We now look at some examples. 'Exclusive or' is the function which returns True exactly when one but not both of its arguments have the value True; it is like the 'or' of a restaurant menu: you may have vegetarian moussaka or fish as your main course, but not both! The 'built-in or', `||`, is 'inclusive' because it returns True if either one or both of its arguments are True.

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

## The integers: Int

The Haskell type `Int` contains the integers. The integers are the whole numbers, used for counting; they are written thus: The `Int` type represents integers in a fixed amount of space, and so can only represent a finite range of integers. The value `maxBound` gives the greatest value in the type, which happens to be 2147483647. For the majority of integer calculations these fixed size numbers are suitable, but if larger numbers are required we may use the `Integer` type, which can accurately represent whole numbers of any size. We do arithmetic on integers using the following operators and functions; the operations we discuss here also apply to the `Integer` type.

<code>+</code>	The sum of two integers.
<code>*</code>	The product of two integers.
<code>^</code>	Raise to the power; <code>2^3</code> is 8.
<code>-</code>	The difference of two integers, when infix: <code>a-b</code> ; the integer of opposite sign, when prefix: <code>-a</code> .
<code>div</code>	Whole number division; for example <code>div 14 3</code> is 4. This can also be written <code>14 'div' 3</code> .
<code>mod</code>	The remainder from whole number division; for example <code>mod 14 3</code> (or <code>14 'mod' 3</code> ) is 2.
<code>abs</code>	The absolute value of an integer; remove the sign.
<code>negate</code>	The function to change the sign of an integer.

## Relational operators

There are ordering and inequality relations over the integers, as there are over all basic types. These functions take two integers as input and return a Bool, that is either True or False. The relations are

<code>&gt;</code>	greater than (and not equal to)
<code>&gt;=</code>	greater than or equal to
<code>==</code>	equal to
<code>/=</code>	not equal to
<code>&lt;=</code>	less than or equal to
<code>&lt;</code>	less than (and not equal to)

A simple example using these definitions is a function to test whether three Ints are equal.

```
threeEqual :: Int -> Int -> Int -> Bool  
threeEqual m n p = (m==n) && (n==p)
```

## Overloading

Both integers and Booleans can be compared for equality, and the same symbol `==` is used for both these operations, even though they are different. Indeed, `==` will be used for equality over any type `t` for which we are able to define an equality operator. This means that `(==)` will have the type

```
Int -> Int -> Bool  
Bool -> Bool -> Bool  
and indeed t -> t -> Bool if the type t carries an equality.
```

Using the same symbol or name for different operations is called overloading.

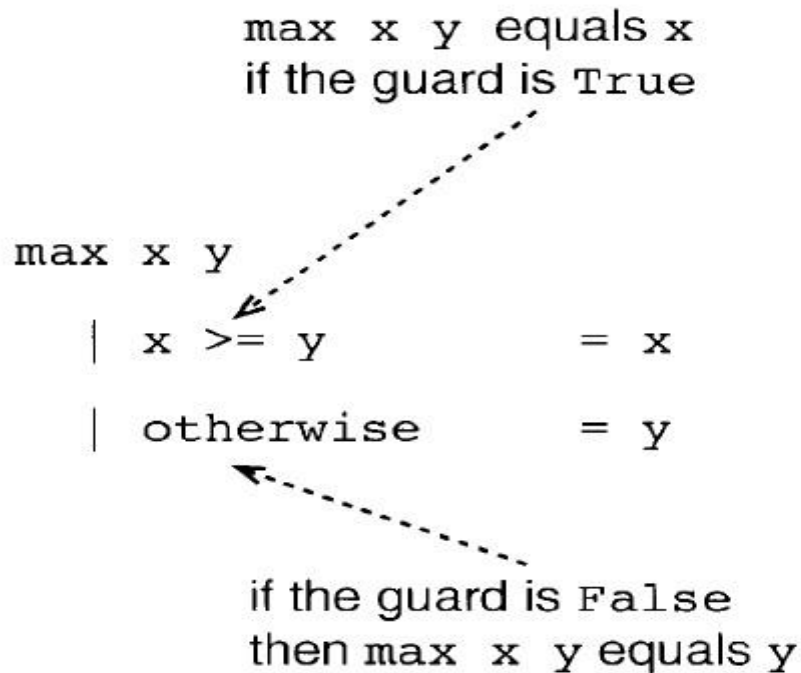
## Guards

Here we explore how conditions or guards are used to give alternatives in the definitions of functions. A guard is a Boolean expression, and these expressions are used to express various cases in the definition of a function. We take as a running example in this section functions which compare integers for size, and start by looking at the example of the function to return the maximum value of two integers. When the two numbers are the same then we call their common value the maximum.

```
max :: Int -> Int -> Int  
max x y  
/ x >= y = x  
/ otherwise = y
```

How do we read a definition like this, which appears in the Haskell prelude'?





In general, if the first guard (here  $x \geq y$ ) is True then the corresponding value is the result (x in this case). On the other hand, if the first guard is False, then we look at the second, and so on. An otherwise guard will hold whatever the arguments, so that in the case of max the result is x if  $x \geq y$  and y otherwise, that is in the case that  $y > x$ .

## Conditional expressions

Guards are conditions which distinguish between different cases in definitions of functions. We can also write general conditional expressions by means of the if.. . then.. . else construct of Haskell. The value of if condition then m else n is m if the condition is True and is n if the condition is False, so that the expression

if False then 3 else 4 has the value 4, and in general

*if  $x \geq y$  then x else y*

will be the maximum of x and y. This shows that we can write max in a different way thus:

```
max :: Int -> Int -> Int
max x y
= if x >= y then x else y
```

We tend to use the guard form either than this, but we will see examples below where the use of if . . . then . . . else . . . is more natural.

## The characters: Char

People and computers communicate using keyboard input and screen output. Which are based on sequences of characters, that is letters, digits and 'special' characters like space, tab, newline and end-of-file. Haskell contains a built-in type of characters, called Char. Literal characters are written inside single quotes, thus 'd' is the Haskell representative of the character d. Similarly '3' is the character three. Some special characters are represented as follows

Tab	<code>'\t'</code>
Newline	<code>'\n'</code>
backslash (\)	<code>'\\'</code>
single quote (')	<code>'\''</code>
double quote (")	<code>'\"'</code>

There is a standard coding for characters as integers, called the ASCII coding. The capital letters 'A' to 'Z' have the sequence of codes from 65 to 90, and the small letters 'a' to 'z' the codes 97 to 122. The character with code 34, for example, can be written '\34', and '\9' and '\97' have the same meaning. ASCII has recently been extended to the Unicode standard, which contains characters from fonts other than English. There are conversion functions between characters and their numerical codes which Convert an integer into a character, and vice versa.

## Floating-point numbers: Float

In calculating we also want to use numbers with fractional parts, which are represented in Haskell by the floating-point numbers which make up the type Float. We do not use Float heavily in what follows, and so this section can be omitted on first reading and used as reference material to be consulted when necessary. Internal to the Haskell system there is a fixed amount of space allocated to representing each Float. This has the effect that not all fractions can be represented by floating point numbers, and arithmetic over them will not be always be exact. It is possible to use the type of double-precision floating-point numbers, Double for greater precision. or for full-precision fractions built from Integer there is the type Rational.

<code>+</code>	<code>Float -&gt; Float -&gt; Float</code>	Add, subtract, multiply.
<code>-</code>	<code>Float -&gt; Float -&gt; Float</code>	Fractional division.
<code>*</code>	<code>Float -&gt; Int -&gt; Float</code>	Exponentiation $x^n = x^n$ for a natural number $n$ .
<code>/</code>	<code>Float -&gt; Float -&gt; Float</code>	Exponentiation $x**y = x^y$ .
<code>^</code>	<code>Float -&gt; Float -&gt; Bool</code>	Equality and ordering operations.
<code>**</code>	<code>Float -&gt; Float -&gt; Bool</code>	Absolute value.
<code>==,/=,&lt;,&gt;,&lt;=,&gt;=</code>	<code>Float -&gt; Float</code>	The inverse of cosine, sine and tangent.
<code>abs</code>	<code>Float -&gt; Float</code>	Convert a fraction to an integer by rounding up, down, or to the closest integer.
<code>acos,asin</code>	<code>Float -&gt; Float</code>	Cosine, sine and tangent.
<code>atan</code>	<code>Float -&gt; Float</code>	Powers of $e$ .
<code>ceiling</code>	<code>Float -&gt; Int</code>	Convert an Int to a Float.
<code>floor</code>	<code>Float -&gt; Float</code>	Logarithm to base $e$ .
<code>round</code>	<code>Float -&gt; Float</code>	Logarithm to arbitrary base, provided as first argument.
<code>cos,sin</code>	<code>Float -&gt; Float</code>	
<code>tan</code>	<code>Float -&gt; Float</code>	
<code>exp</code>	<code>Float -&gt; Float</code>	
<code>fromInt</code>	<code>Int -&gt; Float</code>	
<code>log</code>	<code>Float -&gt; Float</code>	
<code>logBase</code>	<code>Float -&gt; Float -&gt; Float</code>	

## Names in Haskell

We have seen a variety of uses of names in definitions and expressions. In a definition like

```
addTwo :: Int -> Int -> Int
addTwo first second = first+second
```

The names or identifiers `Int`, `addTwo`, `first` and `second` are used to name a type, a function and two variables. Identifiers in Haskell must begin with a letter - small or capital - which is followed by an optional sequence of letters, digits, underscores `'_'` and single quotes.

The names used in definitions of values must begin with a small letter, as must variables and type variables, which are introduced later. On the other hand, capital letters are used to begin type names, such as `Int`; constructors, such as `True` and `False`; module names and also the names of type classes, which we shall encounter below.

An attempt to give a function a name which begins with a capital letter, such as

```
Fun x = x+l
```

gives the error message 'Undefined constructor function "Fun1"'. There are some restrictions on how identifiers can be chosen. There is a small collection of reserved words which cannot be used; these are

```
case class data default deriving do else if import in infix infix1 infixr
instance let module newtype of then type where
```

The special identifiers `as`, `qualified` and `hiding` have special meanings in certain contexts but can be used as ordinary identifiers. By convention, when we give names built up from more than one word, we capitalize the first letters of the second and subsequent words, as in `maxThree`.

The same identifier can be used to name both a function and a variable, or both a type and a type constructor; we recommend strongly that this is not done, as it can only lead to confusion. If we want to redefine a name that is already defined in the prelude or one of the libraries we have to hide that name on import.

Haskell is built on top of the Unicode character description standard, which allows symbols from fonts other than those in the ASCII standard. These symbols can be used in identifiers and the like, and Unicode characters - which are described by a 16-bit sequence - can be input to Haskell in the form `\uhhhh` where each of the `h` is a hexadecimal (4 bit) digit. In this text we use the ASCII subset of Unicode exclusively.

## Operators

The Haskell language contains various operators, like `+`, `++` and so on. Operators are infix functions, so that they are written between their arguments, rather than before them, as is the case for ordinary functions.

In principle it is possible to write all applications of an operator with enclosing parentheses, thus but expressions rapidly become difficult to read. Instead two extra properties of operators allow us to write expressions uncluttered by parentheses.

## Associativity

If we wish to add the three numbers 4, 8 and 99 we can write either `4+(8+99)` or `(4+8)+99`. The result is the same whichever we write, a property we call the associativity of addition. Because of this, we can write

`4+8+99`

for the sum, unambiguously. Not every operator is associative, however; what happens when we write

`4-2-1`

for instance? The two different ways of inserting parentheses give

$(4-2)-1 = 2-1 = 1$	<i>left associative</i>
$4-(2-1) = 4-1 = 3$	<i>right associative</i>

In Haskell each non-associative operator is classified as either left or right associative. If left associative, any double occurrences of the operator will be parenthesized to the left; if right associative, to the right. The choice is arbitrary, but follows custom as much as possible, and in particular `'-'` is taken to be left associative.

## Designing and writing Programs

One theme which we want to emphasize in this book is how we can **design** programs to be written in Haskell. Design is used to mean many different things in computing: the **way** that **we** want to think of **it** is like this:

### Definition

Design is the stage before we start writing detailed Haskell code. In this section we will concentrate on looking at examples, and on talking about the different ways we can try to define functions, but we will also try to give some general advice about how to start writing a program. These are set out as questions we can ask ourselves when we are stuck with a programming problem.

### Sample Codes in Haskell

```
main = putStrLn "Hello World"
Hello World
```

```
Prelude> 3 * 5
15
Prelude> 4 ^ 2 - 1
15
Prelude> (1 - 5)^(3 * 2 - 4)
16
Prelude> "Hello"
"Hello"
Prelude> "Hello" ++ ", Haskell"
"Hello, Haskell"
Prelude> succ 5
6
Prelude> truncate 6.59
6
Prelude> round 6.59
7
Prelude> sqrt 2
1.4142135623730951
Prelude> not (5 < 3)
True
Prelude> gcd 21 14
7
Prelude> putStrLn "Hello, Haskell"
Hello, Haskell
Prelude> putStr "No newline"
No newline
```

```
Prelude> print (5 + 4)
9
Prelude> print (1 < 2)
True
Prelude> do { putStr "2 + 2 = " ; print (2 + 2) }
2 + 2 = 4
Prelude> do { putStrLn "ABCDE" ; putStrLn "12345" }
ABCDE
12345
Prelude> do { n <- readLn ; print (n^2) }
4
16
Prelude> 5 :: Int
5
Prelude> 5 :: Double
5.0
Prelude> 5 :: Char
'5'
Prelude> :t True
True :: Bool
Prelude> :t 'X'
'X' :: Char
Prelude> :t "Hello, Haskell"
"Hello, Haskell" :: [Char]
Prelude> :t 42
42 :: (Num t) => t
Prelude> :t 42.0
42.0 :: (Fractional t) => t
Prelude> :t gcd 15 20
gcd 15 20 :: (Integral t) => t
Prelude> :t "a"
15+(5*5)-40
```

### **test.hs**

```
main = do putStrLn "What is 2 + 2?"
        x <- readLn
        if x == 4
            then putStrLn "You're right!"
            else putStrLn "You're wrong!"
```

## **test1.hs**

```
main = do
    let var1 = 2
    let var2 = 3
    putStrLn "The addition of the two numbers is:"
    print(var1 + var2)
```

### **Output:**

```
Prelude > main
The addition of the two numbers is : 5
```

## **Sequence or Range is a special operator in Haskell. It is denoted by "(..)".**

```
main :: IO()
main = do
    print [1..10]
```

### **Output:**

```
Prelude > [1 2 3 4 5 6 7 8 9 10]
```

## **add function**

```
add :: Integer -> Integer -> Integer --function declaration
add x y = x + y                      --function definition
```

```
main = do
    putStrLn "The addition of the two numbers is:"
    print(add 2 5) --calling a function
```

## **Pattern Matching**

Pattern Matching is process of matching specific type of expressions.

It is nothing but a technique to simplify your code

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
main = do
    putStrLn "The factorial of 5 is:"
    print (fact 5)
```

## Guards

A guard is a concept that is very similar to pattern matching. In pattern matching, we usually match one or more expressions, but we use guards to test some property of an expression.

```
fact :: Integer -> Integer
fact n | n == 0 = 1
      | n /= 0 = n * fact (n-1)
main = do
  putStrLn "The factorial of 5 is:"
  print (fact 5)
```

## Where Clause

Where is a keyword or inbuilt function that can be used at runtime to generate a desired output. It can be very helpful when function calculation becomes complex.

```
roots :: (Float, Float, Float) -> (Float, Float)
roots (a,b,c) = (x1, x2) where
  x1 = e + sqrt d / (2 * a)
  x2 = e - sqrt d / (2 * a)
  d = b * b - 4 * a * c
  e = - b / (2 * a)
main = do
  putStrLn "The roots of our Polynomial equation are:"
  print (roots(1,-8,6))
```

## **Recursion**

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, sorting elements etc.

But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop.

Getting started: a story about factorials

Suppose that someone tells us that the factorial of a natural number is the product of all natural numbers from one up to (and including) that number, so that, for instance

$$fac\ 6 = 1*2*3*4*5*6$$

Suppose we are also asked to write down a table of factorials, where we take the factorial of zero to be one.



We begin thus

*n fac n*

0 1

1 1

2  $1*2 = 2$

3  $1*2*3 = 6$

4  $1*2*3*4 = 24$

but we notice that we are repeating a lot of multiplication in doing this. In working out

$1*2*3*4$

we see that we are repeating the multiplication of  $1*2*3$  before multiplying the result by 4

and this suggests that we can produce the table in a different way, by saying how to start

*fac 0 = 1 (fac . 1)*

which starts the table thus

*n fac n*

0 1

and then by saying how to go from one line to the next

*fac n = fac (n-1) \* n*

since this gives us the lines

*n fac n*

0 1

1  $1*1 = 1$

2  $1*2 = 2$

3  $2*3 = 6$

4  $6*4 = 24$

and so on.

What is the moral of this story? We started off describing the table in one way, but came to see that all we needed was the information in (fac. 1) and (fac. 2).

(fac -1) tells us the first line of the table, and (fac. 2) tells us how to get from one line of the table to the next.

The table is just a written form of the factorial function, so we can see that (f ac. 1) and (f ac. 2) actually describe the **function** to calculate the factorial, and putting them together we get

*fac :: Int -> Int*

*fac n*

*/ n==0 = 1*

*/ n>0 = fac (n-1) \* n*

A definition like this is called **recursive** because we actually use *f ac* in describing *f ac* itself. Put this way it may sound paradoxical: after all, how can we describe something in terms of itself? But, the sorry we have just told shows that the definition is perfectly sensible, since it gives a starting point: the value off *ac* at 0. And a way of going from the value off *ac* at a particular point, *f ac (n-1)*. to the value off *ac* on the next line. Namely *fac n*. These recursive rules will give a value to *f ac n* whatever the (positive) value *n* has - we just have to write out *n* lines of the table. as it were.

## Recursion and calculation

The story in the previous section described how the definition of factorial

```
fac : Int -> Int
fac n
/ n==0 = 1
/ n>0 = fac (n-1) * n
```

can be seen as generating the table of factorials, starting from fac 0 and working up to fac 1, fac 2 and so forth, up to any value we wish. We can also read the definition in a calculation way, and see recursion justified in another way. Take the example of fac 4

```
fac 4
fac 3 * 4
```

so that (fac. 2) replaces one goal - fac 4 - with a simpler goal - finding fac 3 (and multiplying it by 4). Continuing to use (fac.2), we have

```
fac 4
fac 3 * 4
(fac 2 * 3) * 4
((fac 1 * 2) * 3) * 4 ((fac 0 * 1) * 2) * 3 * 4
```

Now, we have got down to the simplest case (or **basecase**), which is solved by (fac. 1). In the calculation we have worked from the goal back down to the base case, using the **recursion step** (fac. 2). We can again see that we get the result we want. because the recursion step takes us from *n* more complicated case to a simpler one, and we have given a value for the simplest case (zero, here) which we will eventually reach. We have now seen in the case of fac two explanations for why recursion works. The **bottom-up** explanation says that the fac equations can be seen to generate the values of fac one-by-one from the base case at zero.

A **top-down** view starts with a goal to be evaluated, and shows how the equations simplify this until we hit the base case. The two views here are related, Once we can think of the top-down explanation generating a table too, but in this case the table is generated as it is needed. Starting with the goal of fac 4 we require the lines for 0 to 3 also. Technically, we call the form of recursion we have seen here **primitive recursion**. We will describe it more formally in the next section, where we examine how to start to find recursive definitions. Before we do that, we discuss another aspect of the fac function as defined here.

## Primitive recursion in practice

This section examines how primitive recursion is used in practice by examining a number of examples. The pattern of primitive recursion says that we can define a function from the natural numbers 0, 1, . . . by giving the value at zero, and by explaining how to go from the value at n-1 to the value at n. We can give a **template** for this

```
fun n
/ n==0    = . . . .
/ n>0     = . . . fun (n-1) . . . .
```

where we have to supply the two right-hand sides. How can we decide whether a function can be defined in this way? Just as we did earlier in the chapter, we frame a question which summarizes the essential property we need for primitive recursion to apply.

## General forms of recursion

a recursive definition of a function such as *fac* would give the value of *fac* *n* using the value *fac* (***n-1***). We saw there that *fac* (*n-1*) is *simpler* in being closer to the base case *fac* **0**. As long as we preserve this property of becoming simpler, different patterns of recursion are possible and we look at some of them in this section. These more general forms of recursion are called **general recursion**.

## Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

## Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

## Example Recursion Function Scripts

### Maximum

The maximum function takes a list of things that can be ordered (e.g. instances of the *Ord* typeclass) and returns the biggest of them.

```
maximum :: (Ord a) => [a] -> a
maximum [] = error "maximum of empty list"
maximum [x] = x
maximum (x:xs)
  | x > maxTail = x
  | otherwise = maxTail
  where maxTail = maximum xs
```

We use a *where* binding to define *maxTail* as the maximum of the rest of the list.

### Replicate

The replicate function takes an *Int* and some element and returns a list that has several repetitions of the same element. For instance, *replicate* 3 5 returns [5,5,5].

```
replicate :: (Num i, Ord i) => i -> a -> [a]
replicate n x
  | n <= 0 = []
  | otherwise = x:replicate' (n-1) x
```

## Take

It takes a certain number of elements from a list. For instance, take 3 [5,4,3,2,1] will return [5,4,3].

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
  | n <= 0  = []
take' _ []  = []
take' n (x:xs) = x : take' (n-1) xs
```

## Reverse

The reverse function simply reverses a list. An empty list reversed equals the empty list itself.

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

## Repeat

The repeat takes an element and returns an infinite list that just has that element. A recursive implementation of that is really easy.

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

## Elem

**The elem function** takes an element and a list and sees if that element is in the list.

```
elem :: (Eq a) => a -> [a] -> Bool
elem a [] = False
elem a (x:xs)
  | a == x    = True
  | otherwise = a `elem` xs
```

## Program Testing

Just because a program is accepted by the Haskell system, it does not mean that it necessarily does what it should. How can we be sure that a program behaves as it is intended to? One option, test aimed is to prove in some way that it behaves correctly. Proof is, however, an expensive business, and we can get a good deal of assurance that our programs behave correctly by testing the program on selected inputs. The art of testing is then to choose the inputs to be as comprehensive as possible. That is, we want to test data to represent all the different 'kinds' of input that can be presented to the function. How might we choose test data? There are two possible approaches. We could simply be told the specification of the function, and devise test data according to that.

This is called black box testing, as we cannot see into the box which contains the function. On the other hand, in devising white box tests we can use the form of the function definition itself to guide our choice of test data. We will explore these two in turn, by addressing the example of the function which is to return the maximum of three integers,

```
maxThree :: Int -> Int -> Int -> Int
```

## Black box testing

How can we make a rational choice of test data for a function, rather than simply picking (supposedly) random numbers out of the air? What we need to do is try to partition the inputs into different testing groups where we expect the function to behave in a similar way for all the values in a given group. In picking the test data we then want to make sure that we choose at least one representative from each group. We should also pay particular attention to any special cases, which will occur on the 'boundaries' of the groups. If we have groups of positive and negative numbers, then we should pay particular attention to the zero case, for instance. What are the testing groups for the example of **maxThreed**? There is not a single right answer to this, but we can think about what is likely to be relevant to the problem and what is likely to be irrelevant. In the case of **maxThree** it is reasonable to think that the size or sign of the integers will not be relevant: what will determine the result is their relative ordering.

We can make a first subdivision this way

all three values different;

all three values the same;

two items equal, the third different. In fact, this represents two cases

- two values equal to the maximum, one other;

- one value equal to the maximum, two others.

We can then pick a set of test data thus

6 4 1

6 6 6

2 6 6

2 2 6

If we test our definition with these data then we see that the program gives the right results. So too does the following program:

```
mysteryMax : Int -> Int -> Int -> Int
```

```
mysteryMax x y z
```

```
/ x > y & x > z = X
```

```
/ y > x & y > z = Y
```

```
/ otherwise = z
```

so should we conclude that **mysteryMax** computes the maximum of the three inputs'?

If we do, we are wrong, for we have that This is an important example: it tells us that **testing alone cannot assure us that a function is correct**. How might we have spotted this error in designing our test data? We could have said that not only did we need to consider the groups above, but that we should have looked at all the different possible orderings of the data, giving

all three values different: six different orderings;

all three values the same: one ordering;

two items equal, the third different. In each of the two cases we consider three orderings.

The final case generates the test data 6 6 2 which find the error.

We mentioned special cases earlier: we could see this case of two equal to the maximum in this way. Clearly the author of **mysteryMax** was thinking about the general case of three different values, so we can see the example as underlining the importance of looking at special cases.

## White box testing

In writing white box test data we will be guided by the principles which apply to black box testing. but we can also use the form of the program to help us choose data. **If** we have a function containing guards, we should supply data for each case in the definition. We should also pay attention to 'boundary conditions' by testing the equality case when a guard uses  $\geq$  or  $>$ , for example.

If a function uses recursion we should test the zero case, the one case and the general case. In the example of `mysteryMax` we should be guided to the data 6 6 2 since the first two inputs are at the boundaries of the guards.