Haskell

The Craft of Functional Programming

SECOND EDITION

ADDISON-WESLEY

Simon Thompson

Haskell
The Craft of Functional Programming

Second edition

Simon Thompson

Addison-Wesley

An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney
Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

To Alice

and Rory

Contents

Preface

Computer technology changes with frightening speed; the fundamentals, however, remain remarkably static. The architecture of the standard computer is hardly changed from the machines which were built half a century ago, even though their size and power are incomparably different from those of today. In programming, modern ideas like object-orientation have taken decades to move from the research environment into the commercial mainstream. In this light, a functional language like Haskell is a relative youngster, but one with a growing influence to play.

Functional languages are increasingly being used as **components** of larger systems like Fran (Elliott and Hudak 1997), in which Haskell is used to describe reactive graphical animations, which are ultimately rendered in a lower-level language. This inter-operation is done without sacrificing the semantic elegance which characterizes functional languages.

Functional languages provide a **framework** in which the crucial ideas of modern programming are presented in the clearest possible way. This accounts for their widespread use in teaching computing science and also for their influence on the design of other languages. A case in point is the design of G-Java, the generics of which are directly modelled on polymorphism in the Haskell mould.

This book provides a tutorial introduction to functional programming in Haskell. The remainder of the preface begins with a brief explanation of functional programming and the reasons for studying it. This is followed by an explanation of the approach taken in the book and an outline of its contents. Perhaps most importantly for readers of the first edition, the changes in approach and content in this second edition are then discussed. A final section explains different possible routes through the material.

What is functional programming?

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.

An example of the power and generality of the language is the `map` function, which is used to transform every element of a list of objects in a specified way. For example, `map` can be used to double all the numbers in a sequence of numbers or to invert the colours in each picture appearing in a list of pictures.

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 4 5
```

the equations which define the functions involved in the expression are used, so

```
3 + addDouble 4 5
⇝  3 + 2*(4+5)
⇝  ...
⇝  21
```

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.

It is also possible to discuss how the programs behave in general. In the case of `addDouble` we can use the fact that `x+y` and `y+x` are equal for all numbers `x` and `y` to conclude that `addDouble x y` and `addDouble y x` are equal for all `x` and `y`. This idea of proof is much more tractable than those for traditional imperative and object-oriented (OO) languages.

Haskell and Hugs

This text uses the programming language Haskell, which has freely available compilers and interpreters for most types of computer system. Used here is the Hugs interpreter which provides an ideal platform for the learner, with its fast compile cycle, simple interface and free availability for Windows, Unix and Macintosh systems.

Haskell began life in the late 1980s as an intended standard language for lazy functional programming, and since then it has gone through various changes and

modifications. This text is written in Haskell 98, which consolidates work on Haskell thus far, and which is intended to be stable; future extensions will result in Haskell 2 some years down the line, but it is expected that implementations will continue to support Haskell 98 after that point.

While the book covers most aspects of Haskell 98, it is primarily a programming text rather than a language manual. Details of the language and its libraries are contained in the language and library reports available from the Haskell home page,

`http://www.haskell.org/`

Why learn functional programming?

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.

Haskell is not just a good 'teaching language'; it is a practical programming language, supported by having extensions such as interfaces to C functions and component-based programming, for example. Haskell has also been used in a number of real-world projects. More information about these extensions and projects can be found in the concluding chapter.

Who should read this book?

This text is intended as an introduction to functional programming for computer science and other students, principally at university level. It can be used by beginners to computer science, or more experienced students who are learning functional programming for the first time; either group will find the material to be new and challenging.

The book can also be used for self-study by programmers, software engineers and others interested in gaining a grounding in functional programming.

The text is intended to be self-contained, but some elementary knowledge of commands, files and so on is needed to use any of the implementations of Haskell. Some logical notation is introduced in the text; this is explained as it appears. In Chapter 19 it would help to have an understanding of the graphs of the `log`, n^2 and 2^n functions.

The approach taken here

There is a tension in writing about a programming language: one wants to introduce all the aspects of the language as early as possible, yet not to over-burden the reader with too much at once. The first edition of the text introduced ideas 'from the bottom up', which meant that it took more than a hundred pages before any substantial example could be discussed.

The second edition takes a different approach: a case study of 'pictures' introduces a number of crucial ideas informally in the first chapter, revisiting them as the text proceeds. Also, Haskell has a substantial library of built-in functions, particularly over lists, and this edition exploits this, encouraging readers to use these functions before seeing the details of their definitions. This allows readers to progress more quickly, and also accords with practice: most real programs are built using substantial libraries of pre-existing code, and it is therefore valuable experience to work in this way from the start. A section containing details of the other changes in the second edition can be found later in this preface.

Other distinctive features of the approach in the book include the following.

The text gives a thorough treatment of reasoning about functional programs, beginning with reasoning about list-manipulating functions. These are chosen in preference to functions over the natural numbers for two reasons: the results one can prove for lists seem substantially more realistic, and also the structural induction principle for lists seems to be more acceptable to students.

The `Picture` case study is introduced in Chapter 1 and revisited throughout the text; this means that readers see different ways of programming the same function, and so get a chance to reflect on and compare different designs.

Function design – to be done before starting to code – is also emphasized explicitly in Chapters 4 and 11.

There is an emphasis on Haskell as a practical programming language, with an early introduction of modules, as well as a thorough examination of the `do` notation for I/O and other monad-based applications.

Types play a prominent role in the text. Every function or object defined has its type introduced at the same time as its definition. Not only does this provide a check that the definition has the type that its author intended, but also we view types as the single most important piece of documentation for a definition, since a function's type describes precisely how the function is to be used.

A number of case studies are introduced in stages through the book: the picture example noted above, an interactive calculator program, a coding and decoding system based on Huffman codes and a small queue simulation package. These are used to introduce various new ideas and also to show how existing techniques work together.

Support materials on Haskell, including a substantial number of Web links, are included in the concluding chapter. Various appendices contain other backup information including details of the availability of implementations, common Hugs errors and a comparison between functional, imperative and OO programming.

Other support materials appear on the Web site for the book:

`http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`

Outline

The introduction in Chapter 1 covers the basic concepts of functional programming: functions and types, expressions and evaluation, definitions and proof. Some of the more advanced ideas, such as higher-order functions and polymorphism, are previewed here from the perspective of the example of pictures built from characters. Chapter 2 looks at the practicalities of the Hugs implementation of Haskell, loading and running scripts written in traditional and 'literate' styles, and the basics of the module system. It also contains a first exercise using the `Picture` type. These two chapters together cover the foundation on which to build a course on functional programming in Haskell.

Information on how to build simple programs over numbers, characters and Booleans is contained in Chapter 3. The basic lessons are backed up with exercises, as is the case for all chapters from here on. With this basis, Chapter 4 steps back and examines the various strategies which can be used to define functions, and particularly emphasizes the importance of using other functions, either from the system or written by the user. It also discusses the idea of 'divide and conquer', as well as introducing recursion over the natural numbers.

Structured data, in the form of tuples and lists, come in Chapter 5. After introducing the idea of lists, programming over lists is performed using two resources: the list comprehension, which effectively gives the power of `map` and `filter`; and the first-order prelude and library functions. Nearly all the list prelude functions are polymorphic, and so polymorphism is brought in here. Chapter 6 contains various extended examples, and only in Chapter 7 is primitive recursion over lists introduced; a text processing case study provides a more substantial example here.

Chapter 8 introduces reasoning about list-manipulating programs, on the basis of a number of introductory sections giving the appropriate logical background. Guiding principles about how to build inductive proofs are presented, together with a more advanced section on building successful proofs from failed attempts.

Higher-order functions are introduced in Chapters 9 and 10. First functional arguments are examined, and it is shown that functional arguments allow the implementation of many of the 'patterns' of computation identified over lists at the start of the chapter. Chapter 10 covers functions as results, defined both as lambda-expressions and as partial applications; these ideas are examined by revisiting the `Picture` example, as well as through an index case study. This is followed by an interlude – Chapter 11 – which discusses the role of the development life cycle in programming.

Type classes allow functions to be overloaded to mean different things at different types; Chapter 12 covers this topic as well as surveying the various classes built into Haskell. The Haskell type system is somewhat complicated because of the presence of classes, and so Chapter 13 explores the way in which types are checked in Haskell. In general, type checking is a matter of resolving the various constraints put upon the possible type of the function by its definition.

In writing larger programs, it is imperative that users can define types for themselves. Haskell supports this in two ways. Algebraic types like trees are the subject of Chapter 14, which covers all aspects of algebraic types from design and proof to their interaction with type classes, as well as introducing numerous examples of algebraic types in practice. These examples are consolidated in Chapter 15, which contains the case study of coding and decoding of information using a Huffman-style code. The foundations of the approach are outlined before the implementation of the case study. Modules are used to break the design into manageable parts, and the more advanced features of the Haskell module system are introduced at this point.

An abstract data type (ADT) provides access to an implementation through a restricted set of functions. Chapter 16 explores the ADT mechanism of Haskell and gives numerous examples of how it is used to implement queues, sets, relations and so forth, as well as giving the basics of a simulation case study.

Chapter 17 introduces lazy evaluation in Haskell which allows programmers a distinctive style incorporating backtracking and infinite data structures. As an example of backtracking there is a parsing case study, and infinite lists are used to give 'process style' programs as well as a random-number generator.

Haskell programs can perform input and output by means of the `IO` types. Their members – examined in Chapter 18 – represent action-based programs. The programs are most readily written using the `do` notation, which is introduced at the start of the chapter, and illustrated through a series of examples, culminating in an interactive front-end to the calculator. The foundations of the `do` notation lie in monads, which can also be used to do action-based programming of a number of different flavours, some of which are examined in the second half of the chapter.

The text continues with an examination in Chapter 19 of program behaviour, by which we mean the time taken for a program to compute its result, and the space used in that calculation. Chapter 20 concludes by surveying various applications and extensions of Haskell as well as looking at further directions for study. These are backed up with Web and other references.

The appendices cover various background topics. The first examines links with functional and OO programing, and the second gives a glossary of commonly used terms in functional programming. The others include a summary of Haskell operators and Hugs errors, together with help on understanding programs and details of the various implementations of Haskell.

The Haskell code for all the examples in the book, as well as other background materials, can be downloaded from the Web site for the book.

What has changed from the first edition?

The second edition of the book incorporates a number of substantial changes, for a variety of reasons.

`Bottom up' or not?

Most importantly, the philosophy of how to introduce material has changed, and this makes most impact on how lists are handled. The first edition was written with a resolutely 'bottom up' approach, first introducing recursive definitions of monomorphic functions, and only later bringing in the built-in functions of the prelude and the libraries. This edition starts by introducing in Chapter 5 the (first-order) polymorphic list-manipulating functions from the prelude as well as list comprehensions, and only introduces recursive definitions over lists in Chapter 7.

The main reason for this change was the author's (and others') experience that once recursion had been introduced early, it was difficult to get students to move on and use other sorts of definitions; in particular it was difficult to get students to use prelude and library functions in their solutions. This is bad in itself, and gives students only a partial view of the language. Moreover, it rests ill with modern ideas about programming, which emphasize the importance of re-use and putting together solutions to utilize a rich programming environment that provides many of the required building blocks.

Introduction

Another consequence of the first-edition approach was that it took some hundred pages before any substantial examples could be introduced; in this edition there is an example of pictures in Chapter 1 which both forms a more substantial case study and is used to preview the ideas of polymorphism, higher-order functions and type abstraction introduced later in the text. The case study is revisited repeatedly as new material is brought in, showing how the same problems can be solved more effectively with new machinery, as well as illustrating the idea of program verification.

The introduction also sets out more clearly some of the basic concepts of functional programming and Haskell, and a separate Chapter 2 is used to discuss the Hugs system, Haskell scripts and modules and so forth.

Haskell 98

The book now has an emphasis on using the full resources of Haskell 98. Hugs now provides an almost complete implementation of Haskell, and so as far as systems are concerned Hugs is the exclusive subject. In most situations Hugs will probably be the implementation of choice for teaching purposes, and if it is not used, it is only the system descriptions which need to be ignored, as none of the language features described are Hugs-specific.

The treatment of abstract data types uses the Haskell mechanism exclusively, rather than the restricted type synonym mechanism of Hugs which was emphasized in the first edition. The material on I/O now starts with the do notation, treating it as a mini language for describing programs with actions. This is followed by a general introduction to monads, giving an example of monadic computation over trees which again uses the do notation.

Finally, functions in the text are given the same names as they have in the prelude or libraries, which was not always the case in the first edition. Type variables are the

customary `a`, `b`, ... and list variables are named `xs`, `ys` and so on.

Recursion, types and proof

As hinted earlier, recursion is given less emphasis than before.

The material on type checking now takes the approach of looking more explicitly at the constraints put upon types by definitions, and emphasizes this through a sequence of examples. This replaces an approach which stated typing rules but said less about their application in practice.

Students have made the point that proof over lists seems more realistic and indeed easier to understand than proof over the natural numbers. For that reason, proof over lists is introduced in Chapter 8 rather than earlier. This has the advantage that practical examples can be brought in right from the start, and the material on proof is linked with the pictures case study.

Problem solving and patterns of definition

Because of a concern for 'getting students started' in solving problems, there is an attempt to talk more explicitly about strategies for programming, reorganizing and introducing new material in Chapters 4 and 11; this material owes much to Polya's problem-solving approach in mathematics. There is also explicit discussion about various 'patterns of definition' of programs in Section 9.1.

Conclusion and appendices

The new edition contains a concluding chapter which looks to further resources, both printed and on the Web, as well as discussing possible directions for functional programming.

Some material from the appendices has been incorporated into the conclusion, while the appendix that discusses links with other paradigms says rather more about links with OO ideas. The other appendices have been updated, while the one that dealt with 'some useful functions' has been absorbed into the body of the text.

To the reader

This introduction to functional programming in Haskell is designed to be read from start to finish, but some material can be omitted, or read in a different order.

The material is presented in an order that the author finds natural, and while this also reflects some of the logical dependencies between parts of the subject, some material later in the text can be read earlier than it appears. Specifically, the introductions to I/O in the first four sections of Chapter 18 and to algebraic types in the early sections on Chapter 14 can be tackled at any point after reading Chapter 7. Local definitions, given by `where` and `let`, are introduced in Chapter 6; they can be covered at any point after Chapter 3.

It is always an option to cover only a subset of the topics, and this can be achieved by stopping before the end; the rest of this section discusses in more detail other ways of trimming the material.

There is a thread on program verification which begins with Chapter 8 and continues in Sections 10.9, 14.7, 16.10 and 17.9; this thread is optional. Similarly, Chapter 19 gives a self-contained treatment of program time and space behaviour which is also optional.

Some material is more technical, and can be omitted on (at least the) first reading. This is signalled explicitly in the text, and is contained in Sections 8.7 and part of Section 13.2.

Finally, it is possible to omit some of the examples and case studies. For example, Sections 6.2 and 6.4 are extended sets of exercises which need not be covered; the text processing (Section 7.6) and indexing (Section 10.8) can also be omitted – their roles are to provide reinforcement and to show the system used on rather larger examples. In the later chapters, the examples in Sections 14.6 and 16.7–16.9 and in Chapter 17 can be skipped, but paring too many examples will run the risk of losing some motivating material.

Chapter 15 introduces modules in the first two sections; the remainder is the Huffman coding case study, which is optional. Finally, distributed through the final chapters are the calculator and simulation case studies. These are again optional, but omission of the calculator case study will remove an important illustration of parsing and algebraic and abstract data types.

Acknowledgements

For feedback on the first edition, I am grateful particularly to Ham Richards, Bjorn von Sydow and Kevin Hammond and indeed to all those who have pointed out errata in that text. In reading drafts of the second edition, thanks to Tim Hopkins and Peter Kenny as well as the anonymous referees.

Emma Mitchell and Michael Strang of Addison-Wesley have supported this second edition from its inception; thanks very much to them.

Particular thanks to Jane for devotion beyond the call of duty in reading and commenting very helpfully on the first two chapters, as well as for her support over the last year while I have been writing this edition. Finally, thanks to Alice and Rory who have more than readily shared our home PC with Haskell.

Simon Thompson
Canterbury, January 1999

Chapter 1

Introducing functional programming

This chapter sets the scene for our exposition of functional programming in Haskell. The chapter has three aims.

We want to introduce the main ideas underpinning functional programming. We explain what it means to be a function and a type. We examine what it means to find the value of an expression, and how to write an evaluation line-by-line. We look at how to define a function, and also what it means to prove that a function behaves in a particular way.

We want to illustrate these ideas by means of a realistic example; we use the example of pictures to do this.

Finally, we want to give a preview of some of the more powerful and distinctive ideas in functional programming. This allows us to illustrate how it differs from other approaches like object-oriented programming, and also to show why we consider functional programming to be of central importance to anyone learning computing science. As we proceed with this informal overview we will give pointers to later chapters of the book where we explain these ideas more rigorously and in more detail.

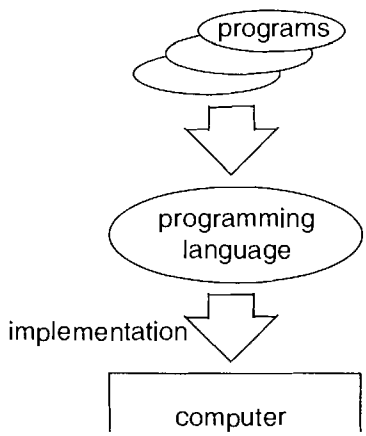
Computers and modelling

In the last fifty years computers have moved from being enormous, expensive, scarce, slow and unreliable to being small, cheap, common, fast and (relatively!) dependable. The first computers were 'stand-alone' machines, but now computers can also play different roles, being organized together into networks, or being embedded in domestic machines like cars and washing machines, as well as appearing in personal computers (PCs), organizers and so on.

Despite this, the fundamentals of computers have changed very little in this period: the purpose of a computer is to manipulate symbolic information. This information can represent a simple situation, such as the items bought in a supermarket shopping trip, or more complicated ones, like the weather system above Europe. Given this information, we are required to perform tasks like calculating the total cost of a supermarket trip, or producing a 24-hour weather forecast for southern England.

How are these tasks achieved? We need to write a description of how the information is manipulated. This is called a **program** and it is written in a **programming language**. A programming language is a formal, artificial language used to give instructions to a computer. In other words the language is used to write the **software** which controls the behaviour of the **hardware**. While the structure of computers has remained very similar since their inception, the ways in which they are programmed have developed substantially. Initially programs were written using instructions which controlled the hardware directly, whereas modern programming languages aim to work closer to the level of the problem – a 'high' level – rather than at the 'low' or machine level.

The programming language is made to work on a computer by an **implementation**, which is itself a program and which runs programs written in the higher-level language on the computer in question.

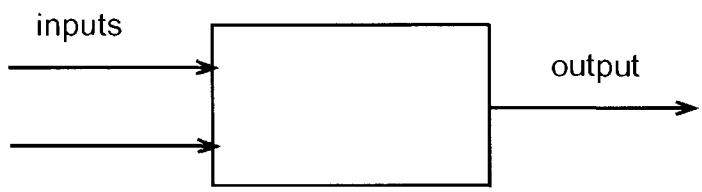


Our task in this text is programming, so we shall be occupied with the upper half of the diagram above, and not the details of implementation (which are discussed in Peyton Jones 1987; Peyton Jones and Lester 1992).

Our subject here is **functional** programming, which is one of a number of different programming styles or **paradigms**; others include object-oriented (OO), structured and logic programming. How can there be different paradigms, and how do they differ? One very fruitful way of looking at programming is that it is the task of **modelling** situations – either real-world or imaginary – within a computer. Each programming paradigm will provide us with different tools for building these models; these different tools allow us – or force us – to think about situations in different ways. A functional programmer will concentrate on the relationships between values, while an OO programmer will concentrate on the objects, say. Before we can say anything more about functional programming we need to examine what it means to be a function.

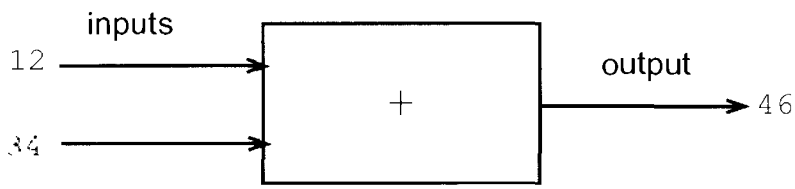
What is a function?

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`.

Addition is a mathematical example, but there are also functions in many other situations; examples of these include

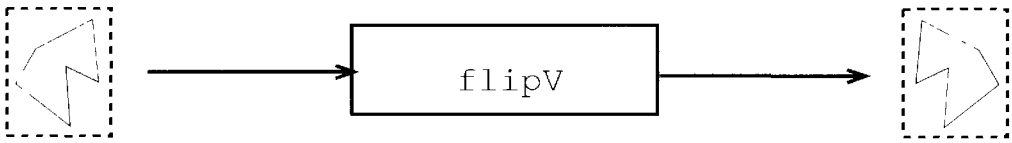
- a function giving the distance by road (*output*) between two cities (*inputs*);
- a supermarket check-out program, which calculates the bill (*output*) from a list of bar codes scanned in (*input*); and
- a process controller, which controls valves in a chemical plant. Its inputs are the information from sensors, and its output the signals sent to the valve actuators.

We mentioned earlier that different paradigms are characterized by the different tools which they provide for modelling: in a functional programming language functions will be the central component of our models. We shall see this in our running example of pictures, which we look at now.

Pictures and functions

In this chapter, and indeed throughout the book, we will look at an example of two-dimensional pictures, and their representation within a computer system. At this stage we simply want to make the point that many common relationships between pictures are modelled by functions; in the remainder of this section we consider a series of examples of this.

Reflection in a vertical mirror will relate two pictures, and we can model this by a function `flipV`:

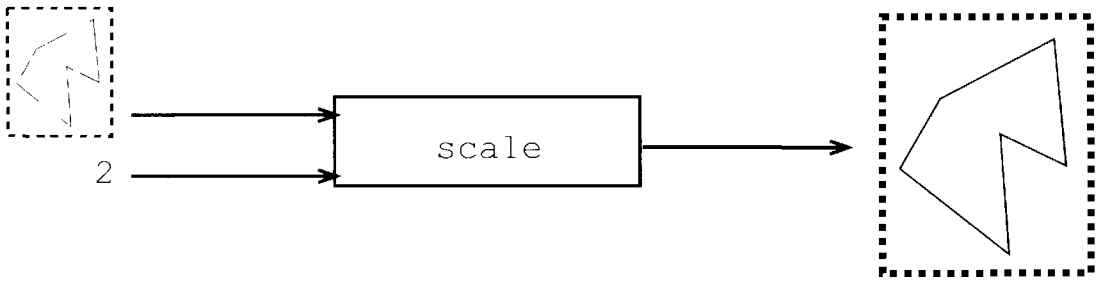


where we have illustrated the effect of this reflection on the 'horse' image

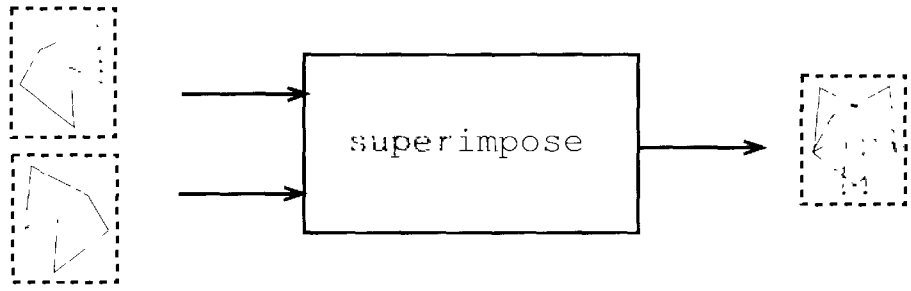
In a similar way we have a function `flipH` to represent flipping in a horizontal mirror.

Another function models the inversion of the colours in a (monochrome) image

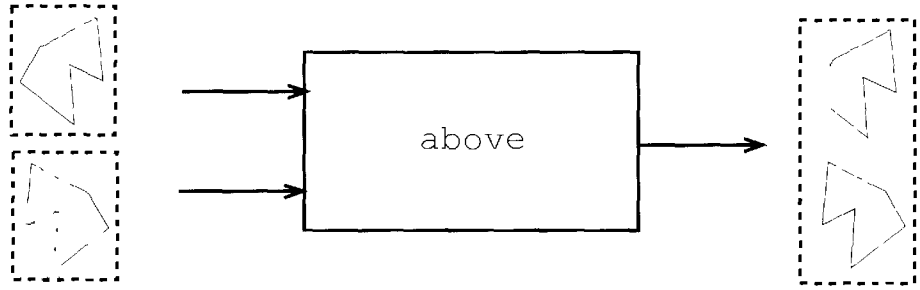
Some functions will take two arguments, among them a function to scale images,



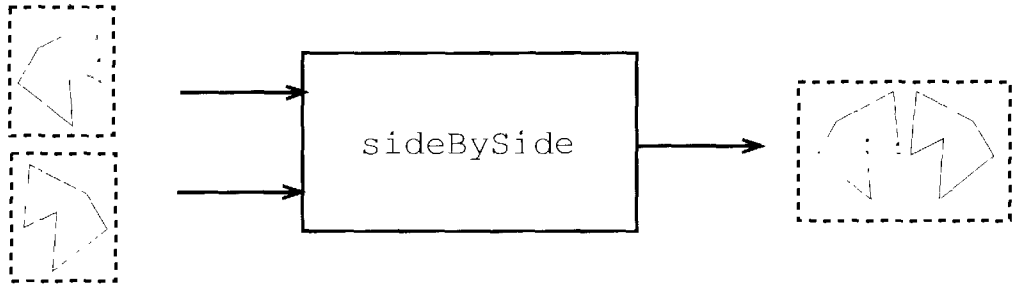
a function describing the superimposition of two images,



a function to put one picture above another,



and a function to place two pictures side by side.



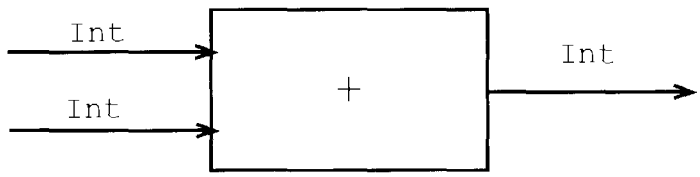
We have now seen what it means to be a function, as well as some examples of functions. Before we explain functional programming, we first have to look at another idea, that of a 'type'.

Types

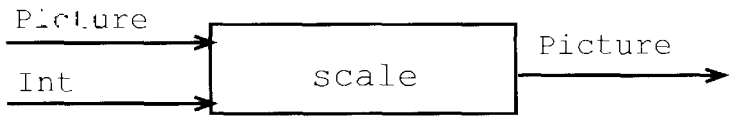
The functions which we use in functional programs will involve all sorts of different kinds of value: the addition function `+` will combine two numbers to give another number; `flipV` will transform a picture into a picture; `scale` will take a picture and a number and return a picture, and so on.

A **type** is a collection of values, such as numbers or pictures, grouped together because although they are different – `2` is not the same as `567` – they are the same *sort* of thing, in that we can apply the same functions to them. It is reasonable to find the larger of two numbers, but not to compare a number and a picture, for instance.

If we look at the addition function, `+`, it only makes sense to add two numbers but not two pictures, say. This is an example of the fact that the functions we have been talking about themselves have a type, and indeed we can illustrate this diagrammatically, thus:



The diagram indicates that + takes two whole numbers (or Integers) as arguments and gives a whole number as a result. In a similar way, we can label the `scale` function



to indicate that its first argument is a `Picture` and its second is an `Int`, with its result being a `Picture`.

We have now explained two of the central ideas in functional programming: a type is a collection of values, like the whole numbers or integers; a function is an operation which takes one or more arguments to produce a result. The two concepts are linked: functions will operate over particular types: a function to scale a picture will take two arguments, one of type `Picture` and the other of type `Int`.

In modelling a problem situation, a type can represent a concept like 'picture', while a function will represent one of the ways that such objects can be manipulated, such as placing one picture above another. We shall return to the discussion of types in Section 1.11.

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, the 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 λ 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

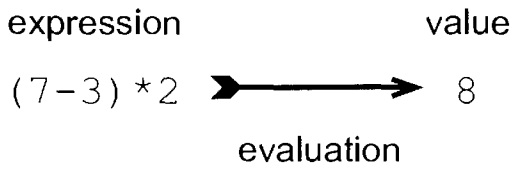
implementations of Haskell can be found in Appendix E and at the Haskell home page, `http://www.haskell.org/`.

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. Details of how to obtain Hugs are in Appendix E. All the programs and examples used in the text can be downloaded from the Web page for this book,

`http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`

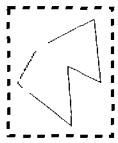
Expressions and evaluation

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

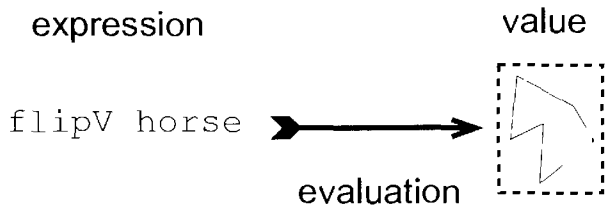


to give the **value** 8. This expression is built up from symbols for numbers and for functions over those numbers: subtraction - and multiplication *; the value of the expression is a number. This process of evaluation is automated in an electronic calculator.

In functional programming we do exactly the same: we evaluate expressions to give values, but in those expressions we use functions which model our particular problem. For example, in modelling pictures we will want to evaluate expressions whose values are pictures. If the picture



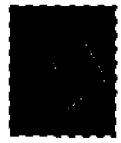
is called `horse`, then we can form an expression by applying the function `flipV` to the horse. This function application is written by putting the function followed by its argument(s), thus: `flipV horse` and then evaluation will give



A more complicated expression is

```
invertColour (flipV horse)
```

the effect of which is to give a horse reflected in a vertical mirror – `flipV horse` as shown above – and then to invert the colours in the picture to give



To recap, in functional programming, we compute by evaluating expressions which use functions in our area of interest. We can see an implementation of a functional language as something like an electronic calculator: we supply an expression, and the system evaluates the expression to give its value. The task of the programmer is to write the functions which model the problem area.

A **functional program** is made up of a series of **definitions** of functions and other values. We will look at how to write these definitions now.

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.

Suppose that we are supplied with the definitions of `horse` and the various functions over `Picture` mentioned earlier – we will discuss in detail how to download these and use them in a program in Chapter 2 – we can then write definitions which use these operations over pictures. For example, we can say

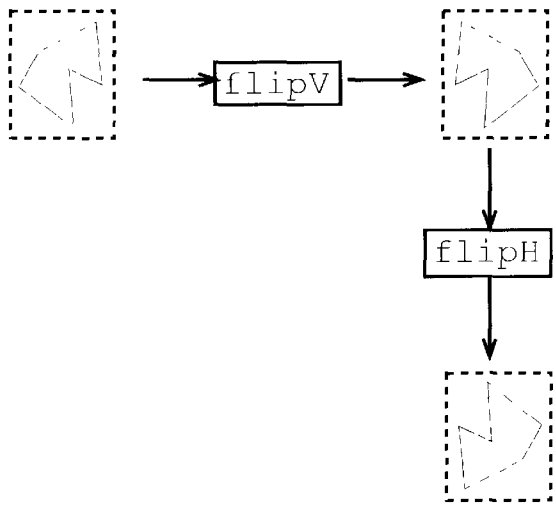
```
blackHorse :: Picture
blackHorse = invertColour horse
```

so that the `Picture` associated with `blackHorse` is obtained by applying the function `invertColour` to the `horse`, thus giving

Another example is the definition

```
rotateHorse :: Picture
rotateHorse = flipH (flipV horse)
```

and we can picture the evaluation of the right-hand side like this



assuming the function `flipH` has the effect of reflecting a `Picture` in a horizontal mirror. The effect of these two reflections is to rotate the picture through 180°.

In Section 1.6 we explained that the Hugs system works rather like a calculator in evaluating expressions. How will it evaluate an expression like

```
size - 17
```

for instance? Using the definition of `size` given earlier, we can replace the left-hand side – `size` – with the corresponding right-hand side – `12+13`; this gives us the expression

```
(12+13) - 17
```

and so by doing some arithmetic we can conclude that the value of the expression is 8.

The definitions we have seen so far are simply of constant values; we now turn our attention to how functions are defined.

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

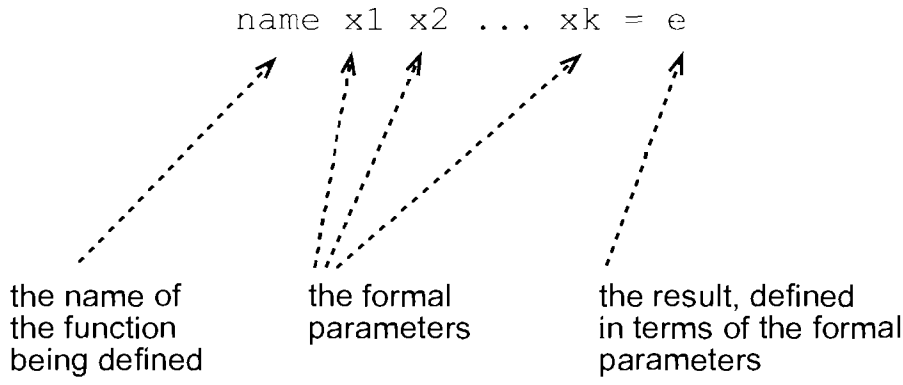
```
square 5 = 5*5
```

and

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

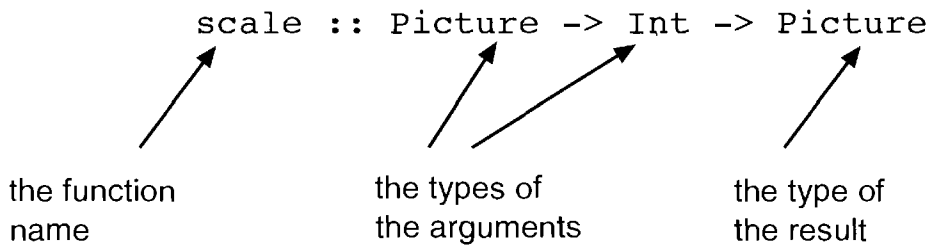
This is the way that the equation is used in evaluating an expression which uses `square`. If we are required to evaluate `square` applied to the expression `e`, we replace the application `square e` with the corresponding right-hand side, `e*e`.

In general a simple function definition will take the form

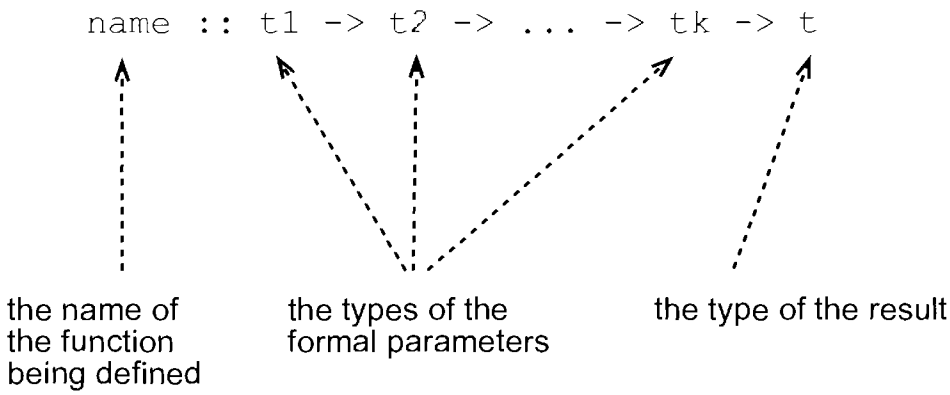


The variables used on the left-hand side of an equation defining a function are called the **formal parameters** because they stand for arbitrary values of the parameters (or **actual** parameters, as they are sometimes known). We will only use 'formal' and 'actual' in the text when we need to draw a 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:



In the general case we have



The definition of `rotateHorse` in Section 1.7 suggests a general definition of a rotation function. To rotate *any* picture we can perform the two reflections, and so we define

```
rotate :: Picture -> Picture
rotate pic = flipH (flipV pic)
```

We can read the equation thus:

> To `rotate` a picture `pic`, we first apply `flipV` to form `(flipV pic)`; we then reflect this in a horizontal mirror to give `flipH (flipV pic)`.

Given this definition, we can replace the definition of `rotateHorse` by

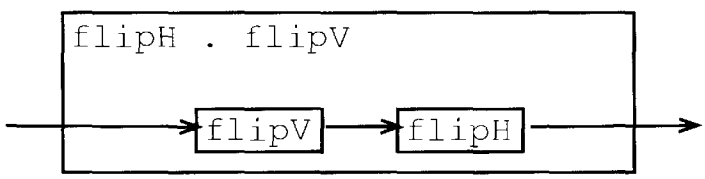
```
rotateHorse :: Picture
rotateHorse = rotate horse
```

which states that `rotateHorse` is the result of applying the function `rotate` to the picture `horse`.

The pattern of definition of `rotate` – 'apply one function, and then apply another to the result' – is so common that Haskell gives a way of combining functions directly in this way. We define

```
rotate :: Picture -> Picture
rotate = flipH . flipV
```

The '.' in the definition signifies **function composition**, in which the output of one function becomes the input of another. In pictures,



we see the creation of a new function by connecting together the input and output of two given functions: obviously this suggests many other ways of connecting together functions, many of which we will look at in the chapters to come.

The direct combination of functions gives us the first example of the power of functional programming: we are able to combine functions using an operator like '.' just as we can combine numbers using '+'. We use the term '**operator**' here rather

than 'function' since '.' is written between its arguments rather than before them; we discuss operators in more detail in Section 3.7.

The direct combination of functions by means of the operator '.' which we have seen here is not possible in other programming paradigms, or at least it would be an 'advanced' aspect of the language, rather than appearing on page 11 of an introductory text.

Type abstraction

Before moving on, we point out another crucial issue which we will explore later in the book. We have just given definitions of

```
blackHorse :: Picture
rotate     :: Picture -> Picture
```

which **use** the type `Picture` and some functions already defined over it, namely `flipH` and `flipV`. We were able to write the definitions of `blackHorse` and `rotate` *without knowing anything about* the details of the type `Picture` or about how the 'flip' functions work over pictures, save for the fact that they behave as we have described.

Treating the type `Picture` in this way is called **type abstraction**: as users of the type we don't need to concern ourselves with how the type is defined. The advantage of this is that the definitions we give apply *however* pictures are modelled. We might choose to model them in different ways in different situations; whatever the case, the function composition `flipH . flipV` will rotate a picture through 180°. Chapter 16 discusses this in more detail, and explains the Haskell mechanism to support type abstraction. In the next section we preview other important features of Haskell.

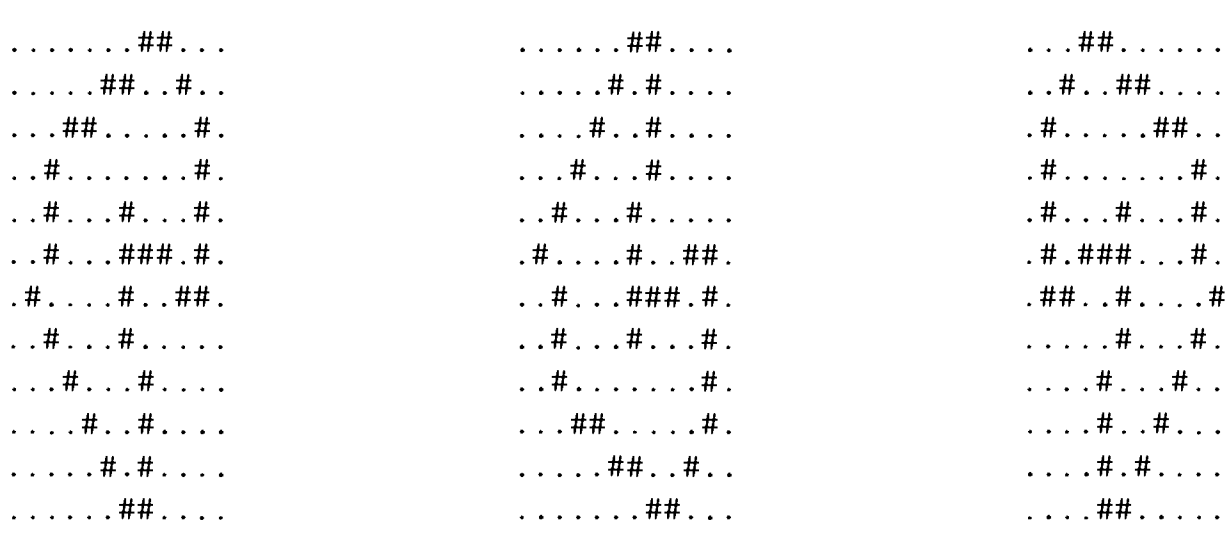
1.9 Looking forward: a model of pictures

We include this section in the first chapter of the book for two reasons. To start with, we want to describe one straightforward way in which `Pictures` can be modelled in Haskell. Secondly, we want to provide an informal preview of a number of aspects of Haskell which make it a powerful and distinctive programming tool. As we go along we will indicate the parts of the book where we expand on the topics first introduced here.

Our model consists of two-dimensional, monochrome pictures built from **characters**. Characters are the individual letters, digits, spaces and so forth which can be typed at the computer keyboard and which can also be shown on a computer screen. In Haskell the characters are given by the built-in type `Char`.

This model has the advantage that it is straightforward to view these pictures on a computer terminal window (or if we are using Windows, in the Hugs window). On the other hand, there are other more sophisticated models; details of these can be found at the Web site for the book, mentioned on page 7.

Our version of the `horse` picture, and the same picture flipped in horizontal and vertical mirrors will be



`horse` `flipH horse` `flipV horse`

where we use dots to show the white parts of the pictures.

How are the pictures built from characters? In our model we think of a picture as being made up of a **list** of lines, that is a collection of lines coming one after another in order. Each line can be seen in a similar way as a list of characters. Because we often deal with collections of things when programming, lists are built into Haskell. More specifically, given any type – like characters or lines – Haskell contains a type of lists of that type, and so in particular we can model pictures as we have already explained, using lists of characters to represent lines, and lists of lines to represent pictures.

With this model of `Pictures`, we can begin to think about how to model functions over pictures. A first definition comes easily; to reflect a picture in a horizontal mirror each line is unchanged, but the order of the lines is reversed: in other words we reverse the list of lines:

```
flipH = reverse
```

where `reverse` is a built-in function to reverse the order of items in a list. How do we reflect a picture in a vertical mirror? The ordering of the lines is not affected, but instead *each line is to be reversed.* We can write

```
flipV = map reverse
```

since `map` is the Haskell function which applies a function to each of the items in a list, individually. In the definitions of `flipH` and `flipV` we can begin to see the power and elegance of functional programming in Haskell.

We have used `reverse` to reverse a list of lines in `flipH` and to reverse each line in `flipV`: this is because the same definition of the function `reverse` can be used over *every* type of list. This is an example of polymorphism, or generic programming, which is examined in detail in Section 5.7.

In defining `flipV` we see the function `map` applied to its argument `reverse`, *which is itself a function.* This makes `map` a very general function, as it can have any desired action on the elements of the list, specified by the function which is its argument. This is the topic of Chapter 9.

Finally, the *result* of applying `map` to `reverse` is itself a function. This covered in Chapter 10.

The last two facts show that functions are 'first-class citizens' and can be handled in exactly the same way as any other sort of object like numbers or pictures. The combination of this with polymorphism means that in a functional language we can write very general functions like `reverse` and `map`, which can be applied in a multitude of different situations.

The examples we have looked at here are not out of the ordinary. We can see that other functions over pictures have similarly simple definitions. We place one picture `above` another simply by joining together the two lists of lines to make one list. This is done by the built-in operator ++, which joins together two lists:[1]

```
above = (++)
```

To place two pictures `sideBySide` we have to join corresponding lines together, thus

```
.......##...   ++   ......##....
.....##..#..   ++   .....#.#....
...##.....#.   ++   ....#..#....
..#.......#.   ++   ...#...#....
..#...#...#.   ++   ..#...#.....
..#...###.#.   ++   .#....#..##.
.#....#..##.   ++   ..#...###.#.
..#...#.....   ++   ..#...#...#.
...#...#....   ++   ..#.......#.
....#..#....   ++   ...##.....#.
.....#.#....   ++   .....##..#..
......##....   ++   .......##...
```

and this is defined using the function `zipWith`. This function is defined to 'zip together' corresponding elements of two lists using – in this case – the operator ++.

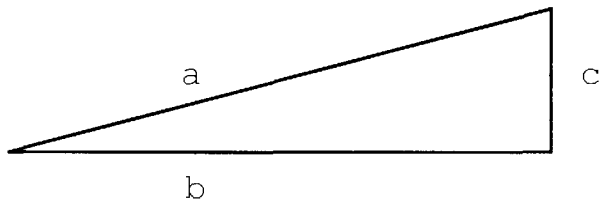
```
sideBySide = zipWith (++)
```

The function `superimpose` is a rather more complicated application of `zipWith`, and also we can define `invertColour` using `map`. We shall return to these examples in Chapter 10.

1.10 Proof

In this section we explore another characteristic aspect of functional programming: proof. A proof is a logical or mathematical argument to show that something holds *in all circumstances*. For example, given any particular right-angled triangle

[1] The operator ++ is surrounded by parentheses `(...)` in this definition so that it is interpreted as a function; we say more about this in Section 3.7.

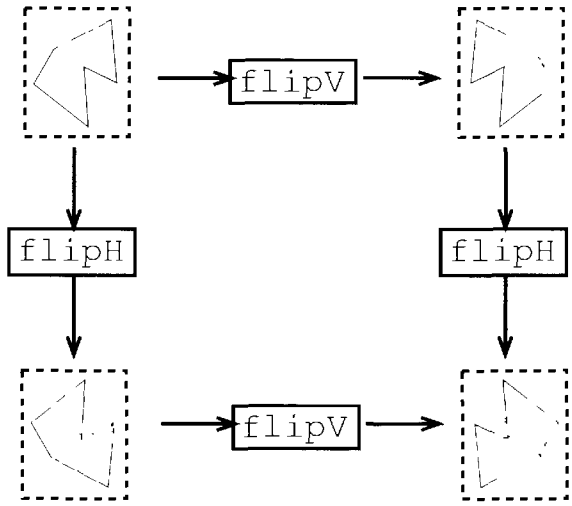


we can check whether or not $a^2=b^2+c^2$ holds. In each case we check, this formula will hold, but this is not in itself enough to show that the formula holds for all `a`, `b` and `c`. A proof, on the other hand, is a general argument which establishes that $a^2=b^2+c^2$ holds whatever right-angled triangle we choose.

How is proof relevant to functional programming? To answer this we will take an example over the `Picture` type to illustrate what can be done. We saw in Section 1.8 that we can define

```
rotate = flipH . flipV
```

but it is interesting to observe that if we reverse the order in which the flip functions are applied then the composition has the same effect, as illustrated here:



Now, we can express this property as a simple equation between functions:

```
flipH . flipV = flipV . flipH                                    (flipProp)
```

Moreover, we can look at our implementations of `flipV` and `flipH` and give a logical **proof** that these functions have the property labelled `(flipProp)` above. The crux of the argument is that the two functions operate independently:

> the function `flipV` affects each line but leaves the lines in the same order while
>
> the function `flipH` leaves each line unaffected, while reversing the order of the list of lines.

Because the two functions affect different aspects of the list it is immaterial which is applied first, since the overall effect of applying the two in either case is to

> reverse each line and reverse the order of the list of lines.

Proof is possible for most programming languages, but it is substantially easier for functional languages than for any other paradigm. Proof of program properties will be a theme in this text, and we start by exploring proof for list-processing functions in Chapter 8.

What benefit is there in having a proof of a property like `(flipProp)`? It give us *certainty* that our functions have a particular property. Contrast this with the usual approach in computing where we **test** the value of a function at a selection of places; this only gives us the assurance that the function has the property we seek at the test points, and in principle tells us nothing about the function in other circumstances. There are safety-critical situations in which it is highly desirable to be sure that a program behaves properly, and proof has a role here. We are not, however, advocating that testing is unimportant – merely that testing and proof have complementary roles to play in software development.

More specifically, `(flipProp)` means that we can be sure that however we apply the functions `flipH . flipV` and `flipV . flipH` they will have the same effect. We could therefore **transform** a program using `flipH . flipV` into one using the functions composed in the reverse order, `flipV . flipH`, and be certain that the new program will have exactly the same effect as the old. Ideas like this can be used to good effect within implementations of languages, and also in developing programs themselves, as we shall see in Section 10.9.

1.11 Types and functional programming

What is the role of types in functional programming? Giving a type to a function first of all gives us crucial information about how it is to be used. If we know that

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

we know two things immediately.

First, `scale` has two arguments, the first being a `Picture` and the second an `Int`; this means that `scale` can be applied to `horse` and `3`.

The result of applying `scale` to this `Picture` and `Int` will be a `Picture`.

The type thus does two things. First, it expresses a **constraint** on how the function `scale` is applied: it must be applied to a `Picture` and an `Int`. Second, the type tells us what the result is if the function is correctly applied: in this case the result is a `Picture`.

Giving types to functions and other things not only tells us how they can be used; it is also possible to check automatically that functions are being used in the right way and this process – which is called **type checking** – takes place in Haskell. If we use an expression like

```
scale horse horse
```

we will be told that we have made an error in applying `scale` to two pictures when a picture and a number are what was expected. Moreover, this can be done without knowing the *values* of `scale` or `horse` – all that we need to know to perform the check

is the *types* of the things concerned. Thus, **type errors** like these are caught before programs are used or expressions are evaluated.

It is remarkable how many errors, due either to mistyping or to misunderstanding the problem, are made by both novice and experienced programmers. A type system therefore helps the user to write correct programs, and to avoid a large proportion of programming pitfalls, both obvious and subtle.

Calculation and evaluation

We have explained that Hugs can be seen as a general calculator, using the functions and other things defined in a functional program. When we evaluate an expression like

```
23 - (double (3+1))                                    (‡)
```

we need to use the definition of the function:

```
double :: Int -> Int
double n = 2*n                                         (dbl)
```

This we do by replacing the unknown `n` in the definition `(dbl)` by the expression `(3+1)`, giving

```
double (3+1) = 2*(3+1)
```

Now we can replace `double (3+1)` by `2*(3+1)` in (‡), and evaluation can continue.

One of the distinctive aspects of functional programming is that such a simple 'calculator' model effectively describes computation with a functional program. Because the model is so straightforward, we can perform evaluations in a **step-by-step** manner; in this text we call these step-by-step evaluations **calculations**. As an example, we now show the calculation of the expression with which we began the discussion.

```
23 - (double (3+1))
⇝  23 - (2*(3+1))                                  using (dbl)
⇝  23 - (2*4)                                        arithmetic
⇝  23 - 8                                            arithmetic
⇝  15                                                arithmetic
```

where we have used '⇝ ' to indicate a step of the calculation, and on each line we indicate at the right-hand margin how we have reached that line. For instance, the second line of the calculation:

```
⇝  23 - (2*(3+1))                                  using (dbl)
```

says that we have reached here using the definition of the `double` function, `(dbl)`.

In writing a calculation it is sometimes useful to underline the part of the expression which gets modified in transition to the next line. This is, as it were, where we need to focus our attention in reading the calculation. The calculation above will have underlining added thus:

```
23 - (double (3+1))
⇝  23 - (2*(3+1))          using (dbl)
⇝  23 - (2*4)              arithmetic
⇝  23 - 8                  arithmetic
⇝  15                      arithmetic
```

In what is to come, when we introduce a new feature of Haskell we shall show how it fits into this line-by-line model of evaluation. This has the advantage that we can then explore new ideas by writing down calculations which involve these new concepts.

mmary

As we said at the start, this chapter has three aims. We wanted to introduce some of the fundamental ideas of functional programming; to illustrate them with the example of pictures, and also to give a flavour of what it is that is distinctive about functional programming. To sum up the definitions we have seen,

a function is something which transforms its inputs to an output;

a type is a collection of objects of similar sort, such as whole numbers (integers) or pictures;

every object has a clearly defined type, and we state this type on making a definition;

functions defined in a program are used in writing expressions to be evaluated by the implementation; and

the values of expressions can be found by performing calculation by hand, or by using the Hugs interpreter.

In the remainder of the book we will explore different ways of defining new types and functions, as well as following up the topics of polymorphism, functions as arguments and results, data abstraction and proof which we have touched upon in an informal way here.

Chapter 2

Getting started with Haskell and Hugs

Chapter 1 introduced the foundations of functional programming in Haskell. We are now ready to use the Hugs system to do some practical programming, and the principal purpose of this chapter is to give an introduction to Hugs.

In beginning to program we will also learn the basics of the Haskell module system, under which programs can be written in multiple, interdependent files, and which can use the `built-in' functions in the prelude and libraries.

Our programming examples will concentrate on using the `Picture` example introduced in Chapter 1 as well as some simple numerical examples. In support of this we will look at how to download the programs and other background materials for the book, as well as how to obtain Hugs.

We conclude by briefly surveying the kinds of error message that can result from typing something incorrect into Hugs.

2.1 A first Haskell program

We begin the chapter by giving a first Haskell program or **script**, which consists of the numerical examples of Chapter 1. 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))
```

Figure 2.1 An example of a traditional script.

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.

There are two different styles of Haskell script, which reflect two different philosophies of programming.

Traditionally, everything in a program file is interpreted as program text, *except* where it is explicitly indicated that something is a comment. This is the style of

```
############################################################

  FirstLiterate.lhs

  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 literate 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))
```

Figure 2.2 An example of a literate script.

`FirstScript.hs`, in Figure 2.1. Scripts of this style are stored in files with an **extension** '`.hs`'.

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**.

The alternative, **literate** approach is to make *everything* in the file commentary on the program, and explicitly to signal the program text in some way. A literate version of the script is given in Figure 2.2, where it can be seen that the program text is on lines beginnning with '>', and is separated from the rest of the text in the file by blank lines. Literate scripts are stored in '`.lhs`' files.

The two approaches emphasize different aspects of programming. The traditional gives primacy to the program, while the literate approach emphasizes that there is more to programming than simply making the right definitions. Design decisions need to be explained, conditions on using functions and so on need to be written down – this is of benefit both for other users of a program and indeed for ourselves if we re-visit a program we have written some time ago, and hope to modify or extend it. We could see this book itself as an extended 'literate script', since commentary here is interspersed by programs which appear in `typewriter font` on lines of their own. Typewriter font is also used for URLs and proofs in later chapters.

Downloading the programs

All the programs defined in the book, together with other support material and general Haskell and functional programming links, can be found at the Web site for the book,

```
http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/
```

The scripts we define are given in literate form.

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/
```

which is a source of much material on Haskell and its implementations. Further information about downloading and installing Hugs may be found in Appendix E.

In this text we describe the terminal-style interface to Hugs, illustrated in Figure 2.3, because this is common to both Windows and Unix. Experienced PC users should have little difficulty in using the Winhugs system, which gives a Windows-style interface to the Hugs commands, once they have understood how Hugs itself works.

Starting Hugs

To start Hugs on Unix, type hugs to the prompt; to launch Hugs using a particular file, type hugs followed by the name of the file in question, as in

```
hugs FirstLiterate
```

Figure 2.3 A Hugs session on Windows.

On a Windows system, Hugs is launched by choosing it from the appropriate place on the Start menu; to launch Hugs on a particular file, double-click the icon for the file in question.[1]

Haskell scripts carry the extension `.hs` or `.lhs` (for literate scripts); only such files can be loaded, and their extensions can be omitted when they are loaded either when Hugs is launched or by a `:load` command within Hugs.

Evaluating expressions in Hugs

As we said in Section 1.6, the Hugs interpreter will evaluate expressions typed at the prompt. We see in Figure 2.3 the evaluation of `size` to 25, `example` to 18 and two more complex expressions, thus

```
Main> double 32 - square (size - double 3)
-297
Main> double 320 - square (size - double 6)
471
Main>
```

where we have indicated the machine output by using a slanted font; user input appears in unslanted form. The **prompt** here, *`Main>`*, will be explained in Section 2.4 below.

As can be seen from the examples, we can evaluate expressions which use the definitions in the current script. In this case it is `FirstLiterate.lhs` (or `FirstScript.hs`).

[1] This assumes that the appropriate registry entries have been made; we work here with the standard installation of Hugs as discussed in Appendix E.

One of the advantages of the Hugs interface is that it is easy to experiment with functions, trying different evaluations simply by typing the expressions at the keyboard. If we want to evaluate a complex expression, it might be sensible to add it to the program, as in the definition

```
test :: Int
test = double 320 - square (size - double 6)
```

All that we then need to do is to type `test` to the *`Main>`* prompt.

Hugs commands

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

`:load 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 first.lhs`	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`.

All the '`:`' commands can be shortened to their initial letter, giving `:l parrot` and so forth. Details of other commands can be found in the comprehensive on-line Hugs documentation which can be read using a Web browser. On a standard Windows installation it is to be found at

`C:\hugs\docs\manual-html\manual_contents.html`

but in general you will need to consult locally to find its location on the system which you are using.

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. Details of how to `:set` values such as the default editor are discussed in Appendix E.

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

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`.

In this way the editor is still open on the file should it need further modification.

We now give some introductory exercises for using Hugs on the first example programs.

A first Hugs session

Task 1

Load the file `FirstLiterate.lhs` into Hugs, and evaluate the following expressions

```
square size
square
double (square 2)
$$
square (double 2)
23 - double (3+1)
23 - double 3+1
$$ + 34
13 `div` 5
13 `mod` 5
```

On the basis of this can you work out the purpose of $$?

Task 2

Use the Hugs command `:type` to tell you the type of each of these, apart from $$.

Task 3

What is the effect of typing each of the following?

```
double square
2 double
```

Try to give an explanation of the results that you obtain.

Task 4

Edit the file `FirstLiterate.lhs` to include definitions of functions from integers to integers which behave as follows.

The function should double its input and square the result of that.

The function should square its input and double the result of that.

Your solution should include declarations of the types of the functions.

The standard prelude and the Haskell libraries

We saw in Chapter 1 that 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, and this can be seen in Figure 2.3 in the line

```
Reading file: "C:\HUGS\lib\Prelude.hs";
```

which precedes the processing of the file `FirstLiterate.lhs` on which Hugs was invoked.

As Haskell has developed over the last decade, the prelude has also grown. In order to make the prelude smaller, and to free up some of the names used in it, many of the definitions have been moved into **standard libraries**, which can be included when they are needed. We shall say more about these libraries as we discuss particular parts of the language.

As well as the standard libraries, the Hugs distribution includes various contributed libraries which support concurrency, functional animations and so forth. Again, we will mention these as we go along. Other Haskell systems also come with contributed libraries, but all systems support the standard libraries.

In order to use the libraries we need to know something about Haskell modules, which we turn to now.

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. In dealing with modules in this text we adopt the conventions that

there is exactly one module per file;

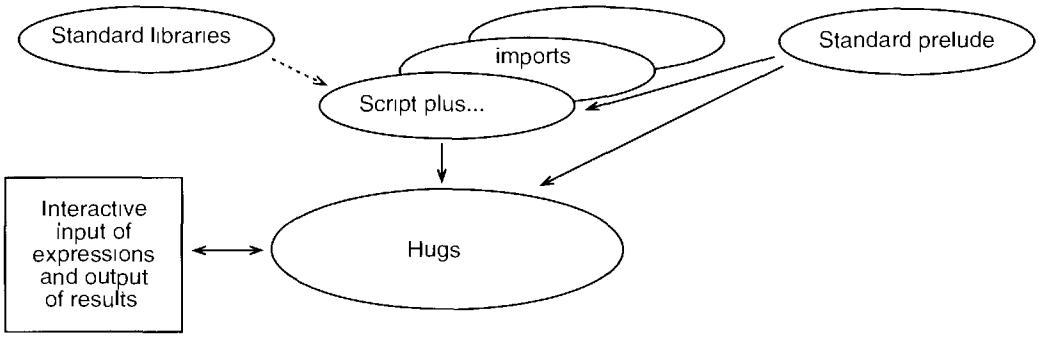
the file Blah.hs or Blah.lhs contains the module Blah.

The module mechanism supports the libraries we discussed in Section 2.3, but we can also use it to include code written by ourselves or someone else.

The module mechanism allows us to control how definitions are imported and also which definitions are made available or **exported** by a module for use by other modules. We look at this in more depth in Chapter 15, where we also ask how modules are best used to support the design of software systems.

We are now in a position to explain why the Hugs prompt appears as *Main>*. The prompt shows the name of the top-level module currently loaded in Hugs, and in the absence of a name for the module it is called the 'Main' module, discussed in Chapter 15.

In the light of what we have seen so far, we can picture a Hugs session thus:



The current script will have access to the standard prelude, and to those modules which it imports; these might include modules from the standard libraries, which are found in the same directory as the standard prelude. The user interacts with Hugs, providing expressions to evaluate and other commands and receiving the results of the evaluations.

The next section revisits the picture example of Chapter 1, which is used to give a practical illustration of modules.

A second example: Pictures

The running example in Chapter 1 was of pictures, and in Figure 2.4 we show parts of a script implementing pictures. We have omitted some of the definitions, replacing them with ellipses '...'. The module here is called Pictures, and can be downloaded from the Web page for this text, mentioned on page 22. This module is imported into another module by the statement

```
import Pictures
```

The only new aspect to the example here is the function

```
printPicture :: Picture -> IO ()
```