Two Dozen Short Lessons in Haskell

a participatory textbook on functional programming

by

Rex Page
School of Computer Science
University of Oklahoma

Rex Page
School of Computer Science
University of Oklahoma
200 Felgar Street — EL114
Norman OK 73019
USA

page@ou.edu

How To Use This Book

The book is spiral bound, to lie flat, so you can put it on a desk or table and write notes in it. You're supposed to work out answers to questions and write them directly in the book. It's a participatory text — a sort of cross between a textbook and a workbook. It doesn't have as many questions as a typical workbook, but it does ask you to interrupt your reading, think about a question, and write an answer directly in the book before proceeding.

You write these answers as you study pages with numbers like 5Q or 27Q. The back of the page will have the same number, but with an A instead of a Q. You will find the answers on these A-numbered pages. Try to work through a full Q-page before looking at the answers.

You will find several kinds of material on Q-pages:

- **commentary** explaining concepts and terms
  Ordinary text, like what you are reading now. No special markings.
- **definitions** of terms, which associate names with values or formulas
  HASKELL DEFINITION • `msg = "Hello World"`
- **commands** telling the Haskell system to make a computation
  HASKELL COMMAND • `reverse msg`
- **responses** from the Haskell system to commands, reporting results of computations
  HASKELL RESPONSE • `"dlroW olleH"`
- **questions** asking you to write in a definition, command, or response that would appropriately complete the surrounding context

¿ *HASKELL DEFINITION* ? *[Here you would write the definition* `msg= "Hello World"`*]*

HASKELL COMMAND • `reverse msg`
HASKELL RESPONSE • `"dlroW olleH"`
HASKELL COMMAND • `msg ++ " Wide Web"`

¿ *HASKELL RESPONSE* ? *[Here you would write the response* `"Hello World Wide Web"`*]*

Commentary explains principles of program design and construction, the form and meaning of elements of Haskell, the programming language of the workbook, and other concepts and fundamentals. You will learn these ideas through reading, looking at examples, thinking, and practice—mostly practice. The idea of the workbook is that you practice by working out answers to the questions that appear in the text, on Q-pages, and checking them against answers, provided on A-pages. You will also develop complete programs on your own, often by applying portions of programs defined in the text in different ways to describe new computations.

Definitions assign meanings to terms. They are written in the syntax of the programming language Haskell. Terms, once defined, can be used in the definitions of other Haskell terms or in commands to the Haskell system. Definitions in the workbook are flagged with a special mark at the beginning of the line: *HASKELL DEFINITION* • Sometimes definitions will be left blank on Q-pages, and flagged with a mark like ordinary definitions, but surrounded by question-marks (¿ *HASKELL DEFINITION* ?) and with a little extra space. These are **questions about definitions**. They are the ones you are supposed to work out on your own. Write your answers on the Q-page, and when you have finished the page, look at the A-page and compare your answers to the ones you see there.

Commands are formulas made up of combinations of terms. The Haskell system must have some way of interpreting these terms, of course. They will be terms that you have defined or terms that are intrinsic in the language—predefined terms, in other words. The Haskell system responds to commands by making the computation specified in the formula (that is, the command) and reporting the results. Like definitions, commands in the workbook are flagged with a special mark at the beginning of the line: *HASKELL COMMAND* • Some commands have been left blank and flagged with the mark ¿ *HASKELL COMMAND* ? These are **questions about commands**. You are supposed to figure out what command would deliver the response that follows it, using the terms that have been defined. Write your answers on the Q-page, and when you have finished the page, compare your answers to those supplied on the A-page.

Responses are the results that the Haskell system delivers when it carries out commands. Responses, too, are flagged in the workbook with a special mark at the beginning of the line: *HASKELL RESPONSE* • Some responses are left blank on Q-pages, and flagged with the special mark ¿ *HASKELL RESPONSE* ? These are **questions about responses**. You should try to work out the response that the Haskell system would deliver to the command that precedes the response-question, considering the terms that have been defined. Write your answers on the Q-page, and when you have finished the page, compare your answers to those supplied on the A-page.

definitions	Programmers provide definitions. Programs are collections of definitions.
commands	People using programs enter commands.
responses	The Haskell system delivers responses by performing computations specified in commands, using definitions provided by programmers.

Here is an example of a question that might appear on a Q-page:

HASKELL COMMAND • `2 + 2`

¿ *HASKELL RESPONSE* ? *[Make a guess about the response and write it here.]*

This question asks you to work out the Haskell system's response to the command `2+2`. You don't know Haskell at this point, so you will have to guess at an answer. This is typical. Most of the time you won't know the answer for certain, but you will know enough to make a good guess.

In this case, Haskell responds with `4`, the sum of `2` and `2`, as you would probably guess. Many numeric operations are predefined in Haskell, intrinsic to the language. The addition operation (+) and a notation for numbers (`2`, for example) are intrinsic: Haskell knows how to interpret "+" and "`2`", so they do not have to be defined in Haskell programs.

Make some kind of guess at an answer for each question, even when you feel like you don't know enough to make a correct answer. Sometimes you will have the right idea, even though you may not get all the details exactly right. By comparing your answers to the correct ones and taking note of the differences, you will gradually learn bits and details about Haskell and about programming principles that will enable you to construct programs entirely on your own.

Here is another question, this time calling for a definition rather than a response:

¿ HASKELL DEFINITION ? *[Guess a definition and write it here.]*

HASKELL COMMAND • x + 2
HASKELL RESPONSE • 5

don't peek — use three-minute rule

Make some kind of stab at an answer to each question and *write it down*. Force yourself. If you don't do this, you may fall into the easy trap of taking a quick peek at part of the answer to give yourself a jump start. This will speed up your reading, but slow down your learning.

Give yourself three minutes to think of an answer. If you think for three minutes and still don't have a good one, write in your best guess, then review your thinking when you turn the page to see the answer.

In this case, the necessary definition is x = 3. You probably had some difficulty guessing this one because you didn't know the form of Haskell definitions. But, you may have realized, after some thought, that the term x needed to be defined; otherwise, it would be hard to make sense of the command x + 2. And, you could tell from the response, 5, that x needed to be 3 to make the formula work out. You might have guessed something like or Set x = 3 or Let x be 3 or x := 3 or some other form of expressing the idea that x should be associated with the number 3. If so, count yourself correct, make note of the particular way this idea is expressed in Haskell, and move on. If not, try to incorporate the idea into the set of things you know about Haskell, and move on..

The important thing is to keep moving on. Eventually you will get better at this.

Sometimes many things will click into place at once, and sometimes your learning will be in little bits at a time. Your greatest frustrations will come when you try to construct programs on your own because programming language systems, Haskell included, are unbelievably intolerant of minor errors. One comma out of place and the whole program is kaput.

This may be the first time in your life you've had to deal with such an extreme level of inflexibility. Unfortunately, you'll just have to get used to it. Computer systems are more tolerant now than they were twenty years ago, and they'll be more tolerant twenty years from now than they are today, but it may be a very long time before they are as tolerant as even the most nit-picky teacher you ever crossed paths with.

It is a good idea to write comments in the workbook about how your answer compared to the correct one—what was right about it and what was wrong. This practice gives you a chance to reflect on your process of reasoning and to improve your understanding of the concepts the workbook talks about.

Haskell Report

Occasionally, you will need to refer to the *Report on the Programming Language Haskell, Verstion 1.3*, by John Peterson and thirteen other authors, available through the Internet. Look for the official definition in the Yale Haskell Project's web site (http://www.cs.yale.edu). The *Report* is a language definition, so it's terse and precise — not fun to read, but useful, and you need to learn how to read this kind of stuff. You will not need it in the beginning, but more and more as you progress.

Warning!

Try to ignore what you have learned about conventional, procedural, programming languages, such as Pascal, C, or Fortran. Most of the concepts you learned about conventional programming will impede your learning the principles of programming in a language like Haskell. Haskell follows an entirely different model of computation. Trying to understand Haskell programs in procedural terms is, at this point, a waste of time and effort—confusing, frustrating, and definitely counter-productive. The time for that is when you take a junior- or senior-level course in programming languages.

For now, start fresh! Think about new things. You will be dealing with equations and formulas, not those step-by-step recipes that you may have learned about before. You will reason as you would if your were solving problems in algebra. That other stuff is more like telling someone how to do long division.

Hello World, Etc. 1

Haskell includes several types of intrinsic data. This chapter makes use of two of them: character strings (sequences of letters, digits, and other characters) and Booleans (True/False data).

HASKELL COMMAND • "Hello World"

¿ HASKELL RESPONSE ?

In a Haskell formula, a sequence of characters enclosed in quotation-marks denotes a data item consisting of the characters between the quotation-marks, in sequence. Such a data item is called a **string**.

For example, "Hello World" denotes the string containing the eleven characters capital-H, lower-case-e, and so on through lower-case-d. That's five letters, a space, and then five more letters. The quotation-marks don't count—they are part of the notation, but not part of the string itself.

A Haskell **command** is a formula, written in the syntax of the Haskell language. When a Haskell command is a string (a particularly simple formula), the Haskell system responds with a message denoting the characters in that string, just as the string would be denoted in a Haskell formula.

Haskell's response to the command "Imagine whirled peas." would be a message consisting of a sequence of characters, beginning with a quotation mark, then capital-I, then lower-case-m, lower-case-a, and so on through lower-case-s, period, and finally a closing quotation mark. That's seven letters, a space, seven more letters, another space, four more letters, and then a period, all enclosed in quotation marks—the twenty-one characters of the string, plus two quotation marks enclosing it.

HASKELL COMMAND • "Imagine whirled peas."
HASKELL RESPONSE • "Imagine whirled peas."

So, now you know how to represent one kind of data, sequences of characters, in a notation that the Haskell system understands, and you know that a data item of this kind is called a string. You might be wondering what you can do with this kind of data. What kinds of computations can Haskell programs describe that use strings?

Haskell's intrinsic definitions include some operations that generate new character strings from old ones. One of these defines a transformation that reverses the order of the characters in a string.

HASKELL COMMAND • reverse "small paws"

¿ HASKELL RESPONSE ?

In this example, the Haskell command is a the string delivered by the transformation reverse, operating on the string "small paws". So, the command reduces to a string, just as before, but this time the command formula describes the string in terms of a data item ("small paws") and a transformation applied to that item (reverse), which produces another string ("swap llams"). It is

character strings — a type of data

Sequences of characters are denoted, in Haskell, by enclosing the sequence in a pair of quotation-marks. Such a sequence can include letters, digits, characters like spaces, punctuation marks, ampersands — basically any character you can type at the keyboard, and even a few more that you'll learn how to denote in Haskell later.

"Ringo"	five-character string, all of which are letters
"@$!#&*#"	seven-character string, none of which are letters

the string delivered by this transformation, in other words the result produced by making the computation specified in the formula, that becomes the Haskell response, and the Haskell system displays that response string in the same form the string would take if it were a command — that is, with the surrounding quotation marks.

HASKELL COMMAND • "swap llams" 6a
HASKELL RESPONSE • "swap llams" 6b

Similarly, the command

HASKELL COMMAND • reverse "aerobatiC" 7

would lead to the response

HASKELL RESPONSE • "Citabrea" 8

Work out the following commands and responses.

HASKELL COMMAND • reverse "too hot to hoot" 9

¿ HASKELL RESPONSE ? 10

¿ HASKELL COMMAND ? 11

HASKELL RESPONSE • "nibor & namtab" 12

¿ HASKELL COMMAND ? 13

HASKELL RESPONSE • "ABLE WAS I ERE I SAW ELBA" 14

use reverse to form these commands

Another intrinsic definition in Haskell permits comparison of strings for equality.

HASKELL COMMAND • "ABLE" == reverse "ELBA" 15

¿ HASKELL RESPONSE ? 16

The command in this example uses a formula that involves two operations, string reversal (reverse) and equality comparison (==). Previous commands have used only one operation (or none), so that makes this one a bit more complex. Combinations of multiple operations in formulas is one way Haskell can express complex computations.

The equality comparison operator reports that two strings are equal when they contain exactly the same characters in exactly the same order. If they are the same, in this sense, the equality comparison operator delivers the value True as its result; otherwise, that is when the strings are different, it delivers the value False. True/False values are not strings, so they are denoted differently — without quotation marks. The value denoted by "True" is a string, but the value denoted by True is not a string. It is another kind of data, known as Boolean data. The quotation marks distinguish one type from the other in Haskell formulas.

> ***operations vs. functions***
>
> In the deepest sense, this textbook uses the terms ***operation*** and ***function*** synonymously. Both terms refer to entities that build new data from old data, performing some transformation along the way (for example, the addition operation takes two numbers and computes their sum). However, the textbook does distinguish between operators and functions in three superficial ways:
>
> 1 function names are made up of letters, or letters and digits in a few cases, while operator symbols contain characters that are neither letters nor digits
>
> 2 the data items that functions transform are called **arguments**, while the data items that operators transform are called **operands**
>
> 3 operators, when they have two operands (which is most of the time), are placed between the operands (as in $a+b$), while functions always precede their arguments (as in sin x).

Here are some examples:

HASKELL COMMAND • "plain" == "plane"

HASKELL RESPONSE • False

HASKELL COMMAND • "WAS" == reverse "SAW"

HASKELL RESPONSE • True

HASKELL COMMAND • "charlie horse" == "Charlie horse"

HASKELL RESPONSE • False

HASKELL COMMAND • "watch for spaces " == "watch for spaces"

HASKELL RESPONSE • False

HASKELL COMMAND • "count   spaces" == "count spaces"

HASKELL RESPONSE • False

As you can see from the examples, equality comparison is case sensitive: upper-case letters are different from lower-case letters and equality comparison delivers the value False when it compares an upper-case letter to a lower-case letter. So, the following comparison delivers the result False, even though the only difference between the strings is that one of the lower-case letters in the first one is capitalized in the second one.

HASKELL COMMAND • "mosaic" == "Mosaic"

HASKELL RESPONSE • False

In addition, blanks are characters in their own right: equality comparison doesn't skip them when it compares strings. So, the following comparison delivers the result False, even though the only

difference between the strings is that one has a blank in the fifth character position, and the other omits the blank.

HASKELL COMMAND • "surf ace" == "surface"" 37

HASKELL RESPONSE • False 38

Even a blank at the end of one of the strings, or at the beginning, makes the comparison result False.

HASKELL COMMAND • "end space " == "end space" 39

HASKELL RESPONSE • False 40

The number of blanks matters, too. If one string has two blanks where another has four, the strings are not equal.

HASKELL COMMAND • "ste reo" == "ste   reo" 41

HASKELL RESPONSE • False 42

Remember! Two strings are the same only if both strings contain exactly the same characters in exactly the same relative positions within the strings. All this may seem like a bunch of minor technicalities, but it is the sort of detail you need to pay careful attention to if you want to succeed in the enterprise of software construction.

> ***Boolean — another type of data***
>
> True and False are the symbols Haskell uses to denote logic values, another kind of data (besides strings) that Haskell can deal with. The operation that compares two strings and reports whether or not they are the same (==) delivers a value of this type, which is known as **Boolean** data. A Boolean data item will always be either the value True or the value False.

Work out responses to the following commands.

HASKELL COMMAND • "planet" == "PLANET" 33

¿ HASKELL RESPONSE ? 34

HASKELL COMMAND • "ERE" == "ERE" 27

¿ HASKELL RESPONSE ? 28

HASKELL COMMAND • "Chicago" == reverse "ogacihc" 29

¿ HASKELL RESPONSE ? 30

HASKELL COMMAND • "Chicago" == reverse "ogacihC" 31

¿ HASKELL RESPONSE ? 32

precedence — order of operations in multi-operation formulas

To understand formulas combining more than one operation, one must know which portions of the formula are associated with which operations. Haskell computes formulas by first applying each function to the arguments following it. The values that result from these computations then become operands for the operators in the formula. Parentheses can be used to override (or confirm) this intrinsic order of computation.

HASKELL COMMAND • reverse "ELBA" == "ABLE" 43

means the same thing as

HASKELL COMMAND • (reverse "ELBA") == "ABLE" 44

but does not have the same meaning as

HASKELL COMMAND • reverse ("ABLE" == "ELBA") 45

In Haskell formulas, functions are always grouped with their arguments before operations are grouped with their operands. Operators also have special precedence rules, which will be discussed as the operators are introduced..

Review Questions

1 How does the Haskell system respond to the following command?
HASKELL COMMAND • reverse "Rambutan"
a "Natubmar"
b "tanbuRam"
c "Nambutar"
d natubmaR

2 How about this one?
HASKELL COMMAND • "frame" == reverse "emarf"
a True
a False
b Yes
c assigns emarf, reversed, to frame

3 And this one?
HASKELL COMMAND • "toh oot" == (reverse "too hot")
a True
b False
c Yes
d no response — improper command

4 And, finally, this one?
HASKELL COMMAND • reverse ("too hot" == "to hoot")
a True
b False
c Yes
d no response — improper command

Haskell definitions are written as equations. These equations associate a name on the left-hand-side of the equals-sign with a formula on the right-hand-side. For example, the equation

HASKELL DEFINITION • shortPalindrome = "ERE" 1

associates the name shortPalindrome with the string "ERE". This definition makes the name shortPalindrome equivalent to the string "ERE" in any formula.

So, in the presence of this definition, the command

HASKELL COMMAND • shortPalindrome 2

leads to the response

HASKELL RESPONSE • "ERE" 3.c1

just as the command

HASKELL COMMAND • "ERE" 4
HASKELL RESPONSE • "ERE" 3.c2

would lead to that response.

It's as simple as that! To get used to the idea, practice with it by working through the following questions.

HASKELL DEFINITION • shortPalindrome = "ERE"
HASKELL DEFINITION • longPalindrome = "ABLE WAS I ERE I SAW ELBA"
HASKELL DEFINITION • notPalindrome = "ABLE WAS I ERE I SAW CHICAGO"
HASKELL DEFINITION • squashedPalindrome = "toohottohoot"
HASKELL DEFINITION • spacedPalindrome = "too hot to hoot" 5

HASKELL COMMAND • longPalindrome 6
¿ HASKELL RESPONSE ? 7
HASKELL COMMAND • reverse notPalindrome 8
¿ HASKELL RESPONSE ? 9
HASKELL COMMAND • longPalindrome == reverse longPalindrome 10
¿ HASKELL RESPONSE ? 11
HASKELL COMMAND • notPalindrome == reverse notPalindrome 12
¿ HASKELL RESPONSE ? 13
HASKELL COMMAND • longPalindrome == shortPalindrome 14
¿ HASKELL RESPONSE ? 15
HASKELL COMMAND • reverse squashedPalindrome == squashedPalindrome 16
¿ HASKELL RESPONSE ? 17

HASKELL COMMAND • "ABLE WAS I ERE I SAW ELBA" == spacedPalindrome

¿ HASKELL RESPONSE ?

¿ HASKELL DEFINITION ?

HASKELL COMMAND • defineThisName
HASKELL RESPONSE • "Get this response."

Well, actually it can get a little more complicated.

Definitions may simply attach names to formulas, as in the previous examples. Or, definitions may be parameterized.

A paramerterized definition associates a function name and one or more parameter names with a formula combining the parameters in some way. Other formulas can make use of a parameterized definition by supplying values for its parameters. Those values specialize the formula. That is, they convert it from a generalized formula in which the parameters might represent any value, to a specific formula, in which the parameters are replaced by the supplied values.

For example, the following parameterized definition establishes a function that computes the value True if its parameter is associated with a palindrome, and False if its parameter is not a palindrome.

palindrome

a word or phrase that reads the same backwards as forwards

Normally, punctuation, spaces, and capitalization and the like are ignored in deciding whether or not a phrase is a palindrome. For example, “Madam, I’m Adam” would be regarded as a palindrome. Eventually, you will learn about a Haskell program that recognizes palindromes in this sense — but not in this chapter. In this chapter, only strings that are exactly the same backwards as forwards, without ignoring punctuation, capitalization and the like, will be recognized as palindromes: “toot” is, “Madam” isn’t, at least in this chapter.

HASKELL DEFINITION • isPalindrome phrase = (phrase == reverse phrase)

This defines a function, named isPalindrome, with one parameter, named phrase. The equation that establishes this definition says that an invocation of the function, which will take the form isPalindrome phrase, where phrase stands for a string, means the same thing as the result obtained by comparing the string phrase stands for to its reverse (phrase == reverse phrase). This result is, of course, either True or False (that is, the result is a Boolean value).

HASKELL COMMAND • isPalindrome "ERE"
HASKELL RESPONSE • True
HASKELL COMMAND • isPalindrome "CHICAGO"
HASKELL RESPONSE • False
HASKELL COMMAND • isPalindrome longPalindrome
HASKELL RESPONSE • True

The command isPalindrome longPalindrome, makes use of the definition of longPalindrome that appeared earlier in the chapter. For this to work, both definitions would need to be in the

Haskell script that is active when the command is issued. In this case, the name longPalindrome denotes the string "ABLE WAS I ERE I SAW ELBA", that was established in the definition:

HASKELL DEFINITION • longPalindrome = "ABLE WAS I ERE I SAW ELBA" 30

Continuing to assume that all definitions in this chapter are in effect, answer the following questions.

HASKELL COMMAND • isPalindrome shortPalindrome 31

¿ HASKELL RESPONSE ? 32

HASKELL COMMAND • isPalindrome notPalindrome 33

¿ HASKELL RESPONSE ? 34

HASKELL COMMAND • isPalindrome squashedPalindrome 35

¿ HASKELL RESPONSE ? 36

HASKELL COMMAND • isPalindrome (reverse shortPalindrome) 37

¿ HASKELL RESPONSE ? 38

The command isPalindrome(reverse Palindrome) illustrates, again, the notion of using more than one function in the same formula. The previous example of such a combination used only the intrinsic operations of comparison (==) and reversal (reverse). The present example uses a function established in a definition (isPalindrome) in combination with an intrinsic one (reverse). The formula uses parentheses to group parts of the formula together into subformulas. The parentheses are needed in this case because Haskell’s rules for evaluating formulas require it to associate a function with the argument immediately following it.

By this rule,

isPalindrome reverse shortPalindrome

would mean

(isPalindrome reverse) shortPalindrome

rather than

isPalindrome (reverse shortPalindrome).

The parentheses are necessary to get the intended meaning.

Haskell programs = collections of definitions

Haskell programs are collections of definitions. When you construct software in Haskell, you will be defining the meaning of a collection of terms. Most of these terms will be functions, and you will define these functions as parameterized formulas that say what value the function should deliver.

People using a Haskell program write Haskell commands specifying the computation they want the computer to perform. These commands are formulas written in terms of the functions defined in a program.

Definitions, therefore, form the basis for all software construction in Haskell.

1 How does the Haskell system respond to the following command?

HASKELL DEFINITION • word = reverse "drow"

HASKELL COMMAND • word

a True
b False
c "word"
d "drow"

2 How about this command?

HASKELL DEFINITION • isTrue str = str == "True"

HASKELL COMMAND • isTrue(reverse "Madam, I'm Adam.")

a True
b False
c ".madA m'I ,madaM"
d Type error in application

3 And this one (assuming the definitions in questions 1 and 2 have been made)?

HASKELL COMMAND • isTrue word

a True
b False
c "drow"
d Type error in application

How to Run Haskell Programs 3

To fire up the Haskell system from a Unix or Windows system where it is installed, simply enter the command hugs[1] or click on the Hugs icon.

OPSYS COMMAND • hugs

Once fired up, the Haskell system acts like a general-purpose calculator: you enter commands from the keyboard and the system responds with results on the screen.

> **script**
>
> A collection of definitions written in Haskell is known as a script. Scripts are packaged in files with names ending with the extension .hs (for *Haskell script*) or with .lhs (for *literate Haskell script*). In a literate Haskell script, only lines beginning with the greater-than character (>) contain definitions. All other lines are commentary.

Most of the commands you enter will be formulas, written in Haskell notation, that request certain computations. The entities that these formulas refer to (functions and operators, for example) may be intrinsic in Haskell, in which case they need no definitions (they are predefined), or they may be entities that you or other programmers have defined.

Such definitions are provided in files, and files containing a collection of definitions are called **scripts**. To make a collection of definitions contained in a script available for use in commands, enter a load command. For example, the load command

HASKELL COMMAND • :load myScript.hs — *make definitions in* myScript.hs *available*

would make the definitions in the file myScript.hs available for use in formulas.

The previous chapter defined names such as longPalindrome and isPalindrome. If the file myScript.hs contained these definitions, you could, at this point, use them in commands:

HASKELL COMMAND • longPalindrome 1a.d6

¿ HASKELL RESPONSE ? 1b.d7

HASKELL COMMAND • isPalindrome "ERE" 1c.d24

HASKELL RESPONSE • True 1c.d25

If you want to look at or change the definitions in the file myScript.hs, enter the edit command:

HASKELL COMMAND • :edit — *edit the most recently loaded script*

The edit command will open for editing the file that you most recently loaded. At this point you could change any of the definitions in the script contained in that file. When you terminate the edit session, the Haskell system will be ready to accept new commands and will use definitions currently in the file, which you may have revised.

1. The Haskell system you will be using is called Hugs. It was originally developed by Mark Jones of the University of Nottingham. More recent versions have been implemented by Alastair Reid of the Yale Haskell Project. Hugs stands for the Haskell User's Gofer System. (Gofer is a language similar to Haskell.) Information about Hugs, including installable software for Unix, Windows, and Macintosh systems, is available on the World Wide Web (http://haskell.systemsz.cs.yale.edu/hugs/). Strictly speaking, the things we've been calling Haskell commands are commands to the Hugs system.

For example, if you had redefined the name longPalindrome during the edit session to give it a new value,

HASKELL DEFINITION • longPalindrome = "A man, a plan, a canal. Panama!"

then upon exit from the edit session, the name longPalindrome would have a different value:

HASKELL COMMAND • longPalindrome

¿ HASKELL RESPONSE ?

If you find that you need to use additional definitions that are defined in another script, you can use the also-load command. For example, the also-load command

HASKELL COMMAND • :also yourScript.hs — *add definitions in* yourScript.hs

would add the definitions in the file yourScript.hs to those that were already loaded from the file myScript.hs.

implicit* :load *commands
Commands like :load and :also will be implicit throughout the text. The text will assume that appropriate scripts have been loaded so that formula-commands have access to the definitions they need.

At this point the edit command will open the file yourScript.hs for editing. If you want to edit a different file (myScript.hs, for example), you will need to specify that file as part of the edit command:

HASKELL COMMAND • :edit myScript.hs— *opens file* myscript.hs *for editing*

The definitions in yourScript.hs can define new entities, but they must not attempt to redefine entities already defined in the file myScript.hs. If you want to use new definitions for certain entities, you will need to get rid of the old ones first. You can do this by issuing a load command without specifying a script:

HASKELL COMMAND • :load — *clears all definitions (except intrinsics)*

After issuing a load command without specifying a script, only intrinsic definitions remain available. You will have to enter a new load command if you need to use the definitions from a script.

If you want to review the list of all Haskell commands, enter the help command:

HASKELL COMMAND • :?

This will display a list of commands (many of which are not covered in this textbook) and short explanations of what they do.

To exit from the Haskell system, enter the quit command:

HASKELL COMMAND • :quit

This puts your session back under the control of the operating system.

Haskell commands are not part of the Haskell programming language. Haskell scripts contain definitions written in the Haskell programming language, and Haskell commands cause the Haskell system to interpret definitions in scripts to make computations. This is known as the **interactive mode** of working with Haskell programs, and this is how the Hugs Haskell system works.

Some Haskell systems do not support direct interaction of this kind. They require, instead, that the Haskell script specify the interactions that are to take place. This is known as the **batch mode** of operation. Haskell systems that operate in batch mode usually require the person using a Haskell program to first compile it (that is, use a Haskell compiler to translate the script into instructions

directly executable by the computer), then load the instructions generated by the compiler, and finally run the program (that is, ask the computer to carry out the loaded instructions).

The batch-mode, compile/load/run sequence is typical of most programming language systems. The Glasgow Haskell Compiler (http://www.dcs.gla.ac.uk/fp/) and the Chalmers Haskell-B Compiler (http://www.cs.chalmers.se/~augustss/hbc.html) are alternatives to Hugs that use the batch mode of operation. So, you can get some experience with that mode at a later time. For now, it's easier to use the Hugs system's interactive mode.

Review Questions

4 The following command
HASKELL COMMAND • :load script.hs
- a loads script.hs into memory
- b makes definitions in script.hs available for use in commands
- c runs the commands in script.hs and reports results
- d loads new definitions into script.hs, replacing the old ones

5 The following command
HASKELL COMMAND • :also script2.hs
- a loads script.hs into memory
- b adds definitions in script2.hs to those that are available for use in commands
- c runs the commands in script2.hs and reports results
- d tells the Haskell system that the definitions in script2.hs are correct

6 A Haskell system working in interactive mode
- a interprets commands from the keyboard and responds accordingly
- b acts like a general-purpose calculator
- c is easier for novices to use than a batch-mode system
- d all of the above

7 The following command
HASKELL COMMAND • :?
- a initiates a query process to help find out what is running on the computer
- b asks the Haskell system to display the results of its calculations
- c displays a list of commands and explanations
- d all of the above

Computations on Sequences — List Comprehensions 4

Many computations require dealing with sequences of data items. For example, you have seen a formula that reverses the order of a sequence of characters. This formula builds a new string (that is, a new sequence of characters) out of an old one. You have also seen formulas that compare strings. Such a formula delivers a Boolean value (True or False), given a pair of strings to compare. And, you saw a formula that delivered the Boolean value True if a string read the same forwards as backwards. All of these formulas dealt with sequences of characters as whole entities.

Sometimes computations need to deal with individual elements of a sequence rather than on the sequence as a whole. One way to do this in Haskell is through a notation known as **list comprehension**. The notation describes sequences in a manner similar to the way sets are often described in mathematics.

SET NOTATION (MATH) • $\{x \mid x \in \text{chairs}, x \text{ is red}\}$ *— set of red chairs*

HASKELL DEFINITION • madam = "Madam, I'm Adam"

HASKELL COMMAND • [c | c <- madam, c /= ' '] -- *non-blank characters from* madam

HASKELL RESPONSE • "Madam,I'mAdam"

This Haskell command uses list comprehension to describe a sequence of characters coming from the string called madam. The name madam is defined to be the string "Madam, I'm Adam".

In this list comprehension, c stands for a typical character in the new sequence that the list comprehension is describing (just as x stands for a typical element of the set being described in mathematical form). Qualifications that the typical element c must satisfy to be included in the new sequence are specified after the vertical bar (|), which is usually read as "such that."

The qualifier c <- madam, known as a **generator**, indicates that c runs through the sequence of characters in the string called madam, one by one, in the order in which they occur. This is analogous to the phrase $x \in$ chairs in the set description, but not quite the same because in sets, the order of the elements is irrelevant, while in sequences, ordering is an essential part of the concept.

comparison operation	***meaning in string or character comparison***
==	equal to
/=	not equal to
<	less than (alphabetically*)
<=	less than or equal to
>	greater than
>=	greater than or equal to

*sort of

Finally, the qualifier c /= ' ' says that only non-blank characters are to be included in the string being described. This part of the list comprehension is known as a **guard**. It is analogous to the qualifier "x is red" in the mathematical set example; "x is red" indicates what kind of chairs are admissible in the set.

The not-equal-to operation (/=) in the guard is like equality comparison (==), but with a reversed sense: x /= y is True when x is not equal to y and False when x is equal to y.

The blank character is denoted in Haskell by a blank surrounded by apostrophes. Other characters can be denoted in this way: '&' stands for the ampersand character, 'x' for the letter-x, and so on.

> ***characters vs. strings***
>
> An individual character is denoted in a Haskell program by that character, itself, enclosed in apostrophes ('a' denotes letter-a, '8' denotes digit-8, and so on). These data items are not strings. They are a different type of data, a type called Char in formal Haskell lingo.
>
> Strings are sequences of characters and are of a type called (formally) String. A string can be made up of several characters ("abcde"), or only one character ("a"), or even no characters (""). Individual characters, on the other hand, always consist of exactly one character.
>
> Since individual characters are *not* sequences and strings *are* sequences, 'a' is not the same as "a". In fact, the comparison 'a'=="a" doesn't even make sense in Haskell. The Haskell system cannot compare data of different types
>
> In your studies, you will learn a great deal about distinctions between different types of data. Making these distinctions is one of Haskell's most important features as a programming language. This makes it possible for Haskell to check for consistant usage of information throughout a program, and this helps programmers avoid errors common in languages less mindful of data types (the C language, for example). Such errors are very subtle, easy to make, and hard to find.

List comprehensions can describe many computations on sequences in a straightforward way.The command in the preceding example produced a string like the string madam, but without its blanks. This idea can be packaged in a function by parameterizing it with respect to the string to be processed. The result is a function that produces a string without blanks, but otherwise the same as the string supplied as the function's argument .

> ***indentation — offsides rule***
>
> When a definition occupies more then one line, subsequent lines must be indented. The next definition starts with the line that returns to the indentation level of the first line of the current definition. Programmers use indentation to visually bracket conceptual units of their software. Haskell makes use of this visual bracketing, known as the offsides rule, to mark the beginning and ending of definitions. Learn to break lines at major operations and line up comparable elements vertically to display the components of your definitions in a way that brings their interrelationships to the attention of people reading them.

HASKELL DEFINITION • removeBlanks str = [c | c <- str, c /= ' ']

HASKELL DEFINITION • hot = "too hot to hoot" *there is a space between these apostrophes*

HASKELL DEFINITION • napolean = "Able was I ere I saw Elba."

HASKELL DEFINITION • chicago = "Able was I ere I saw Chicago."

HASKELL DEFINITION • maddog = "He goddam mad dog, eh?" 1

HASKELL COMMAND • removeBlanks "Able was I" 2

HASKELL RESPONSE • "AblewasI" 3

HASKELL COMMAND • removeBlanks napolean
¿ HASKELL RESPONSE ?
HASKELL COMMAND • removeBlanks "s p a c e d o u t"
¿ HASKELL RESPONSE ?
HASKELL COMMAND • removeBlanks maddog
¿ HASKELL RESPONSE ?
HASKELL COMMAND • removeBlanks hot
¿ HASKELL RESPONSE ?
HASKELL COMMAND • removeBlanks(reverse chicago)
¿ HASKELL RESPONSE ?
HASKELL COMMAND • removeBlanks hot == reverse(removeBlanks hot)
¿ HASKELL RESPONSE ?
¿ HASKELL DEFINITION ? -- function to remove periods (you write it)
¿ HASKELL DEFINITION ? removePeriods str =
¿ HASKELL DEFINITION ?
HASKELL COMMAND • removePeriods chicago
HASKELL RESPONSE • "Able was I ere I saw Chicago"
HASKELL COMMAND • removeBlanks(removePeriods chicago)
HASKELL RESPONSE • "AblewasIereIsawChicago"
¿ HASKELL DEFINITION ? -- function to remove blanks and periods (you write it)
¿ HASKELL DEFINITION ? removeBlanksAndPeriods str =
¿ HASKELL DEFINITION ?
HASKELL COMMAND • removeBlanksAndPeriods napolean
HASKELL RESPONSE • "AblewasIereIsawElba"

Review Questions

1 The following function delivers
HASKELL DEFINITION • f str = [c | c <- str, c == 'x']
a all the c's from its argument
b an empty string unless its argument has x's in it
c a string like its argument, but with x's in place of c's
d nothing — it contains a type mismatch, so it has no meaning in Haskell

2 The following command delivers
HASKELL DEFINITION • g str = [c | c <- str, c == "x"]
HASKELL COMMAND • g "xerox copy"
a "c"
b "xx"
c "xerox xopy"
d error — g expects its argument to be a sequence of strings, not a sequence of characters

parameterization is abstraction

In the following definition of the name stretchSansBlanks,

HASKELL DEFINITION • stretchSansBlanks = [c | c <- "s t r e t c h", c /= ' ']

the string whose blanks are being removed is specified explicitly: "s t r e t c h". This string is a concrete entity.

On the other hand, in the following definition of the function removeBlanks,

HASKELL DEFINITION • removeBlanks str = [c | c <- str, c /= ' ']

the string whose blanks are being removed is the parameter str. This parameter is an abstract entity that stands in place of a concrete entity to be specified later, in a formula that uses the function. In this way, the abstract form of the formula expresses a general idea that can be applied in many different specific cases. It could as well remove the blanks from "s t r e t c h" as from "s q u a s h" or from any other specific string.

The parameterized formulas that occur in function definitions provide an example of **abstraction**. A parameter is an abstract entity that stands for any concrete value of the appropriate type. Abstraction is one of the most important concepts in computer science. You will encounter it in many different contexts.

3 The following function delivers a string like its argument, but …
HASKELL DEFINITION • h str = [c | c <- reverse str, c < 'n']
a written backwards if it starts with a letter in the first half of the alphabet
b written backwards and without n's
c written backwards and without letters in the first half of the alphabet
d written backwards and without letters in the last half of the alphabet

4 Which of the following equations defines a function that delivers a string like its second argument, but with no letters preceding, alphabetically, the letter specified by its first argument?
A HASKELL DEFINITION • s x str = [c | c <- str, c < x]
B HASKELL DEFINITION • s x str = [c | c <- str, c >= x]
C HASKELL DEFINITION • s abc str = [c | c <- str, c == "abc"]
D HASKELL DEFINITION • s abc str = [c | c <- str, c /= "abc"]

5 In the following definition, the parameter str
HASKELL DEFINITION • f str = [c | c <- str, c == 'x']
a represents the letter x
b represents the letter c
c stands for a sequence of x's
d stands for a string containing a sequence of characters

Function Composition and Currying 5

The use of more than one function in a formula is known as **function composition**. The following formula,

```
HASKELL DEFINITION •   madam = "Madam, I'm Adam."
HASKELL COMMAND •      removePeriods(removeBlanks madam)
```

which removes both periods and blanks from a string called madam, is a composition of the functions removePeriods and removeBlanks. In this composition, the function removePeriods is applied to the string delivered by the function removeBlanks operating on the argument madam.

If there were a third function, say removeCommas, then the following composition

```
HASKELL DEFINITION •   removeCommas str = [c | c <- str, c /= ',']
HASKELL COMMAND •      removeCommas(removePeriods(removeBlanks madam))
```

would apply that function to the string delivered by removePeriods (which in turn operates on the string delivered by removeBlanks operating on madam). This all works well. It applies a simple concept, that of removing a certain character from a string, three times. But, the parentheses are beginning to get thick. They could become bulky to the point of confusion if the idea were extended to put together a command to remove many kinds of punctuation marks.

Fortunately, Haskell provides an operator that alleviates this problem (and lots of other problems that it is too early to discuss at this point). The composition of two functions, f and g, say, can be written as f . g, so that (f . g) x means the same thing as f(g(x)). And, (f . g . h) x means f(g(h(x))). And so on.

Using this operator, the following formula

```
HASKELL COMMAND •      (removeCommas . removePeriods . removeBlanks) madam
```

removes blanks, periods, and commas from madam just like the previous formula for that purpose. This is a little easier to look at because it has fewer parentheses, but it has a more important advantage: it points the way toward a function that removes all punctuation marks.

Of course, one can generalize the preceding formula to a function that will remove blanks, periods, and commas from any string presented to it as an argument. This is done by parameterizing with respect to the string being processed. Try to write this function yourself, using the function composition operator (.) and following the form of the preceding command.

```
¿ HASKELL DEFINITION ?                      -- function to remove blanks, periods, and commas
¿ HASKELL DEFINITION ?   removeBPC str =          -- you write this function
¿ HASKELL DEFINITION ?
HASKELL COMMAND •        removeBPC madam
HASKELL RESPONSE •       "MadamImAdam"
```

Actually, in a formula like (f . g . h) x, the f . g . h portion is a complete formula in its own right. It denotes a function that, when applied to the argument x delivers the value (f . g . h) x. It is

important keep these two things straight: f . g . h is not the same thing as (f . g . h) x. One is a function, and the other is a value delivered by that function.[1]

The fact that a formula like f . g . h is a function in its own right provides a simpler way to write the function that removes blanks, periods, and commas from a string. This function is simply the value delivered by the composition of the three functions that each remove one of the characters. The definition doesn't need to mention the parameter explicitly. The following definition of removeBPC is equivalent to the earlier one (and identical in form, except for the parameter).

```
HASKELL DEFINITION •                     -- function to remove blanks, periods, and commas
HASKELL DEFINITION •   removeBPC = removeCommas . removePeriods . removeBlanks     7
HASKELL COMMAND •      removeBPC madam                                             5.c2
HASKELL RESPONSE •     "MadamImAdam"                                               6.c2
```

The three functions that remove characters from strings all have similar definitions.

```
HASKELL DEFINITION •   removeBlanks str = [c | c <- str, c /= ' ']
HASKELL DEFINITION •   removePeriods str = [c | c <- str, c /= '.']
HASKELL DEFINITION •   removeCommas str = [c | c <- str, c /= ',']                 8
```

The only difference in the formulas defining these functions is the character on the right-hand-side of the not-equals operation in the guards.

By parameterizing the formula with respect to that character, one can construct a function that could, potentially (given appropriate arguments) remove any character (period, comma, semicolon, apostrophe, . . . whatever) from a string. The function would then have two arguments, the first representing the character to be removed and the second representing the string to remove characters from.

```
¿ HASKELL DEFINITION ?                  -- function to remove character chr from string str
¿ HASKELL DEFINITION ?   remove chr str =        -- you write this function
¿ HASKELL DEFINITION ?                                                              9
HASKELL COMMAND •        remove ' ' madam -- remove ' '  works just like  removeBlanks     10
HASKELL RESPONSE •       "Madam,I'mAdam."                                                   11
HASKELL COMMAND •        remove ',' madam -- remove ',' works just like removeCommas       12
HASKELL RESPONSE •       "Madam I'm Adam."                                                  13
HASKELL COMMAND •        remove ',' (remove ' ' madam)  -- remove blanks, then commas      14
HASKELL RESPONSE •       "MadamI'mAdam"                                                     15
HASKELL COMMAND •        (remove ',' . remove ' ') madam  -- using curried references      16
HASKELL RESPONSE •       "MadamI'mAdam."                                                    17
```

This new function, remove, is generalizes functions like removeBlanks and removeCommas. That is what parameterization of a formulas does: it makes the formula apply to a more general class of problems. When a function invocation provides arguments to a function, the arguments

1. To put the same idea in simpler terms, reverse and reverse "Chicago" are not the same thing: reverse is a function that operates on one string and delivers another. On the other hand reverse "Chicago" is *not* a function. It is a string, namely the string "ogacihC". This is another case where you must keep the types straight: reverse and reverse "Chicago" are different types of things, which implies that they can't be the same thing.

select a particular special case of the class of problems the function's parameterized formula can apply to. The arguments turn the generalized formula back into one of the special cases that the parameterization generalized.

As you can see in the preceding examples, the formula remove ', ' madam behaves in exactly the same way as the formula removeCommas madam. The function remove, has two arguments. The first argument specifies what character to remove and the second is the string whose commas (or whatever character is specified) are to be removed. On the other hand, the function removeCommas has only one argument: the string whose commas are to be removed. The formula

remove ','

in which the second argument (the string to be processed) is omitted, but in which a specific value for the first argument (the character to be removed from the string) is supplied, is an example of a **curried invocation**[1] to a function.

Curried invocations are functions in their own right. If you look at the formula defining the function remove,

```
HASKELL DEFINITION •    remove chr str = [c | c <- str, c /= chr]
```

and you specialize it by putting a comma-character where the first argument, chr, appears in the formula, then you get the formula used to define the function removeCommas:

removeCommas str	is defined by the formula	[c | c <- str, c /= ',']
remove ',' str	is defined by the same formula	[c | c <- str, c /= ',']

So, the function denoted by the curried invocation remove ',' delivers the same results as the function removeCommas. It has to, because the two functions are defined by the same formulas.

Since these curried invocations are functions, one can use them in composition. Previously a function called removeBPC, the function defined earlier in a formula composing three functions together,

```
HASKELL DEFINITION •                        -- function to remove blanks, periods, and commas
HASKELL DEFINITION •    removeBPC = removeCommas . removePeriods . removeBlanks
```

can be defined equivalently by composing three different curried invocations to the function remove:

```
HASKELL DEFINITION •                        -- function to remove blanks, periods, and commas
HASKELL DEFINITION •    removeBPC = remove ',' . remove '.' . remove ' '
```

The two definitions are equivalent. But, the one using curried invocations to remove, instead of the three specialized functions, is more compact. One formula defines remove, and the definition of removeBPC uses this formula in three different ways. This saves writing three separate formulas for the specialized functions, removeBlanks, removePeriods, and removeCommas.

1. After Haskell B. Curry, a prominent logician who, in the first half of this century developed many of the theoretical foundations on which programming languages like Haskell are based. Yes, the language Haskell was named after Professor Curry.

1 Given the following definitions of f and g, the following Haskell command delivers

```
HASKELL DEFINITION •   f  str  =  [ c | c <- str, c == 'x' ]
HASKELL DEFINITION •   g  str  =  [ c | c <- reverse str, c < 'n' ]
HASKELL COMMAND •      f(g  "A man, a plan, a canal. Panama!")
```

a syntax error, unexpected parenthesis
b the empty string
c syntax error, type conflict in operands
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

2 Given the above definitions of f and g, and the following definition of teddy, the following command delivers

```
HASKELL DEFINITION •   teddy = "A man, a plan, a canal. Panama!"
HASKELL COMMAND •      (f . g)  teddy
```

a syntax error, unexpected parenthesis
b the empty string
c syntax error, type conflict in operands
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

3 Given the above definitions of f, g, and teddy, the following Haskell command delivers

```
HASKELL COMMAND •      (f . g) teddy == f(g  teddy)
```

a syntax error, unexpected parenthesis
b the empty string
c True
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

4 What would be the answer to the preceding question if, in the definitions of f and g, the sense of all the comparisons had been reversed (not equals instead of equals, less-than instead of greater-than-or-equal, etc.)?
a syntax error, unexpected parenthesis
b the empty string
c True
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

5 If equals is a function that requires two arguments, then equals 'x' is a function that requires
a no arguments
b one argument
c two arguments
d three arguments

Patterns of Computation — Composition, Folding, and Mapping 6

Function composition provides a way to build a composite function that has the effect of applying several individual functions in sequence. An example you have already seen involves removing various characters from a string by applying, one after another, functions that each specialize in removing a particular character.

HASKELL COMMAND • (remove ',' . remove '.' . remove ' ') "Madam, I'm Adam."
HASKELL RESPONSE • "MadamI'mAdam"

To carry out the preceding command, Haskell constructs a composite function from three individual functions. The composite function successively removes blanks (remove ' '), then removes periods (remove '.'), and finally removes commas (remove ',') from a string supplied as an argument ("Madam, I'm Adam") to the composite function.

patterns of computation

Composition of functions, which applies a succession of transformations to supplied data, is the most common pattern of computation. It occurs in almost every program. Folding, which reduces a sequence of values to a single value by combining adjacent pairs, and mapping, which applies the same transformation to each value in a sequence also find frequent use in software. You will learn about these patterns in this lesson. A fourth common pattern, iteration, which you will learn to use later, applies the same transformation repeatedly to its own delivered values, building a sequence of successively more refined iterates. These patterns of computation probably account for over 90% of all the computation performed. It pays to be fluent with them.

The composite function (remove ',' . remove '.' . remove ' ') processes its argument by applying the blank removal function to it. The string delivered by that function is passed along as the argument for the next function in the composite, the period removal function. Finally, the result delivered by the period removal function is passed along to the comma removal function, and the result delivered by the comma removal function becomes the result delivered by the composite function.

The composition operation (.) has two operands, one on the left and one on the right. Both operands are functions. Call them f and g, for purposes of this discussion, and suppose that f and g transform arguments of a particular type into results that have the same type. Call it type t, to make it easier to talk about. Then the function delivered by their composition, f . g, also transforms arguments of type t into results of type t.

You can work this out by looking at the meaning of the formula (f . g) x. This formula means the same thing as the formula f(g(x)). Since g requires its argument to have type t, the formula f(g(x)) will make sense only if x has type t. The function g operates on x and delivers a value of type t. This value is passed along to the function f, which takes arguments of type t and delivers values of type t. The result that f delivers, then, has type t, which shows that f(g(x)) has type t. Since (f . g) x means the same thing as f(g(x)), (f . g) x must also have type t. Therefore, the function f . g transforms arguments of type t into results of type t.

To carry this a step further, if there is a third function, h, that transforms arguments of type t into results of type t, it makes sense to compose all three functions (f . g . h), and so on for any number of functions dealing with data of type t in this way. Function composition provides a way to build an assembly-line operation from a sequence of functions.

An argument to be processed by such an assembly line first passes through the first function in the assembly line (which is the rightmost function in the composition), and the result that the rightmost function delivers is passed along to the next function in the assembly line, and so on down the line until the final result pops out the other end. The assembly line comes from having several functions arranged in a sequence and inserting the composition operator between each adjacent pair of functions in the sequences:

forming an assembly line from functions f, g, *and* h: f . g . h

There is an intrinsic function, foldr1, in Haskell that inserts a given operation between adjacent elements in a given sequence. (Actually, foldr1 is not the only intrinsic function in Haskell that does operator insertion, but it is a good place to start.) Inserting an operation between elements in this way "folds" the elements of the sequence into a single value of the same type as the elements of the sequence.

foldr1 ***(intrinsic function)***

foldr1 (⊕) $[x_1, x_2, ..., x_n]$ == $x_1 \oplus x_2 \oplus \ldots \oplus x_n$

where
⊕ is an operation such that $x \oplus y$ delivers another value of the same type as x and y

pronounced: fold-R-one (not folder-one)
$n \geq 1$ *required*

groups from right: $x_1 \oplus (x_2 \oplus \ldots \oplus (x_{n-1} \oplus x_n)\ldots)$
(matters only if ⊕ is not associat

Here is an example of such a folding process: If pre is a function that chooses, from two letters supplied as arguments, the one that precedes the other in the alphabet (pre 'p' 'q' is 'p' and pre 'u' 'm' is 'm'). Then foldr1 pre string delivers the letter from string that is earliest in the alphabet. Using x ‘pre‘ y to stand for pre x y, the following is a step-by-step accounting of the reduction of the formula foldr1 pre "waffle" to the result it delivers:

```
foldr1 pre "waffle"= 'w' ‘pre‘ 'a' ‘pre‘ 'f' ‘pre‘ 'f' ‘pre‘ 'l' ‘pre‘ 'e'
= 'w' ‘pre‘ 'a' ‘pre‘ 'f' ‘pre‘ 'f' ‘pre‘ 'e'
= 'w' ‘pre‘ 'a' ‘pre‘ 'f' ‘pre‘ 'e'
= 'w' ‘pre‘ 'a' ‘pre‘ 'e'
= 'w' ‘pre‘ 'a'
= 'a'
```

functions as operators

f x y *operator form (backquotes)*
function form SAME MEANING x ‘f‘ y

Getting back to the assembly line example, folding can be used with the composition operator to build an assembly line from a sequence of functions:

foldr1 (.) [f, g, h] means f . g . h

This example of folding uses two new bits of notation. One of these is the matter of enclosing the composition operator in parentheses in the reference to the function foldr1. These parentheses are necessary to make the operation into a separate package that foldr1 can use as an argument. If the parentheses were not present, the formula would denote an invocation of the composition operator with foldr1 as the left-hand argument and [f, g, h] as the right-hand argument, and that wouldn't make sense.

The other new notation is a way to specify sequences. Up to now, all of the sequences in the workbook were strings, and the notation for those sequences consisted of a sequence of characters enclosed in quotation marks.

A sequence in Haskell can contain any type of elements, and the usual way to denote a sequence of elements is to list them, separated by commas and enclosed in square brackets: [f, g, h] denotes the sequence containing the elements f, g, and h.

operators as arguments

foldr1 (.) [f, g, h]

function composition as an operator

function composition as an argument

foldr1 . [f, g, h]

formula makes no sense —wrong type of operands

Operators cannot be used as arguments or operands, at least not directly, but functions can be used as arguments or operands (as you've seen in formulas using function composition).

Fortunately, operations and functions are equivalent entities, and Haskell provides a way to convert one form to the other: an operator-symbol enclosed in parentheses becomes a function. The function-version of the operation has the same number of arguments as the operator has operands.

(.) f g means f . g

function form *operator form*

converting operators to functions

sequences (also known as lists)

NOTATION

[*element*$_1$, *element*$_2$, ... *element*$_n$]

MEANING

a sequence containing the listed elements

COMMENTS

- elements all must have same type
- sequence may contain any number of elements, including none
- sequences are commonly called "lists"

EXAMPLES

['a', 'b', 'c']— longhand for "abc", a sequence of three characters
[remove ',' , remove '.' , remove ' ']— sequence of three functions
["alpha", "beta", "gamma", "delta"]— sequence of four strings

The previous chapter defined a function called removeBPC as a composition of three functions:

```
HASKELL DEFINITION •                  -- function to remove blanks, periods, and commas
HASKELL DEFINITION •   removeBPC = remove ',' . remove '.' . remove ' '
```

Try to define removeBPC with a new formula that uses the function foldr1.

¿ HASKELL DEFINITION ? 3

Using this folding operation and list comprehension together, one can design a solution to the problem of removing all the punctuation marks from a string. It requires, however, a slight twist on the notation for list comprehension.

In the notation for sets in mathematics that inspired the list comprehension notation, transformations are sometimes applied to the typical element. In the following example, the typical element is squared, and it is the squares of the typical elements that comprise the set.

SET NOTATION (MATH) • $\{ x^2 \mid x \in \text{Integers} \}$ *— set of squares of integers*

List comprehensions permit functions to be applied to the typical element in the list comprehension, just as in the notation for sets. Using this idea, you can write a list comprehension that changes all the letters in a string to lower case. The function applied to the typical element in this list comprehension will be an intrinsic function called toLower that delivers the lower case version of the character supplied as its argument. The instrinsic function operates on individual characters, but by using list comprehension, you can apply it to all the characters in a string.

***mapping** — making a new sequence from an old one by applying a function to each of its elements*

[f x | x <- xs] *pronounced "exes"*

same function to each element

***known as "mapping" the function* f *onto the sequence* xs**

toLower *(function in* Char *library)*

toLower :: Char -> Char

double-colon reads "has type" *argument type* *result type*

toLower 'A' == 'a'
toLower 'B' == 'b'
etc.

toLower *delivers a copy of its argument if its argument is not a capital letter*

import Char(toLower)

access to toLower

The function toLower converts its argument, which must be a character (not a string), to a lower case letter if its argument is a letter.[1] If its argument isn't a letter, toLower simply delivers a value that is exactly the same as its argument. So, for example, toLower 'E' is 'e', toLower('n') is 'n', and toLower('+') is '+' .

This function for converting from capital letters to lower-case ones can be used in a list comprehension to convert all the capital letters in a string to lower-case, leaving all the other characters in the string as they were:

```
HASKELL COMMAND •   [toLower c | c <- "Madam, I'm Adam." ]   4
HASKELL RESPONSE •  "madam, i'm adam."                        5
```

mapping toLower *onto the sequence* "Madam, I'm Adam."

By parameterizing the preceding formula with respect to the string whose letters are being capitalized, define a function to convert all the capital letters in a string to lower case, leaving all other characters in the string (that is, characters that aren't capital letters) unchanged.

1. The function toLower resides in a library. Library functions are like intrinsic functions except that you must include an import directive in any script that uses them. The name of the library that toLower resides in is Char. To use toLower, include the directive import Char(toLower) in the script that uses it.You will learn more about import directives later, when you learn about modules.

```
¿ HASKELL DEFINITION ?                          -- convert all capital letters in a string to lower case
¿ HASKELL DEFINITION ?   import Char(toLower)  -- get access to toLower function
¿ HASKELL DEFINITION ?   allLowerCase str =
¿ HASKELL DEFINITION ?
  HASKELL COMMAND •      allLowerCase "Madam, I'm Adam."
  HASKELL RESPONSE •     "madam, i'm adam."
```

A sequence consisting of the three functions, remove ',', remove '.', and remove ' ' can also be constructed using this notation.

```
HASKELL COMMAND •                   -- formula for [remove ',', remove '.', remove ' ']
HASKELL COMMAND •   [ remove c | c <- ",. "]
```

mapping remove *onto the sequence ",. "* *There is a blank here.*

This provides a new way to build the composition of these three functions; that is, yet another way to write the function removeBPC:

```
HASKELL DEFINITION •   removeBPC = foldr1 (.) [remove c | c <- ",. "]
```

There is a blank here.

By adding characters to the string in the generator (c <- ",. "), a function to remove all punctuation marks from a phrase can be written:

```
¿ HASKELL DEFINITION ?   removePunctuation =          -- you write it
¿ HASKELL DEFINITION ?
  HASKELL COMMAND •      removePunctuation "Madam, I'm Adam."
  HASKELL RESPONSE •     "MadamImAdam"
```

You need to know a special trick to include a quotation mark in a string, as required in the definition of removePunctuation. The problem is that if you try to put a quotation mark in the string, that quotation mark will terminate the string.

The solution is to use **escape mode** within the string. When a backslash character (\) appears in a string, Haskell interprets the next character in the string literally, as itself, and not as a special part of Haskell syntax. The backslash does not become part of the string, but simply acts as a mechanism to temporarily turn off the usual rules for interpreting characters.

The same trick can be used to include the backslash character itself into a string: "\\" is a string of only one character, not two. The first backslash acts as a signal to go into escape mode, and the second backslash is the character that goes into the string. Escape mode works in specifying individual characters, too: '\'' denotes the apostrophe character, for example, and '\\' denotes the backslash character.

The functions removePunctuation and allLowerCase can be composed in a formula to reduce a string to a form that will be useful in writing a function to decide whether a phrase is palindromic in the usual sense, which involves ignoring whether or not letters are capitals or lower case and ignoring blanks, periods, and other punctuation. In this sense, the phrase "Madam, I'm Adam." is

embedding a quotation mark in a string

WRONG WAY

```
[ remove c | c <- ",. ;:"?!()" ]
```

This quotation mark terminates the string.

The remaining part of the intended string gets left out.

RIGHT WAY

```
[ remove c | c <- ",. ;:\"?!()" ]
```

*Backslash "escapes" normal mode and is **not** part of string.*

In escape-mode, Haskell interprets quotation mark after backslash as part of string, not as terminator.

regarded as palindromic. The characters in the phrase do not read the same backwards as forwards, but they do spell out the same phrase if punctuation and capitalization are ignored.

Use the functions removePunctuation and allLowerCase from this chapter and the function isPalindrome from the previous chapter to write a function that delivers the value True if its argument is a palindromic phrase (ignoring punctuation and capitalization) and whose value is False, otherwise.

```
¿ HASKELL DEFINITION ?   isPalindromic =     -- you write this definition
¿ HASKELL DEFINITION ?
  HASKELL COMMAND •      isPalindromic "Able was I ere I saw Elba."
  HASKELL RESPONSE •     True
  HASKELL COMMAND •      isPalindromic "Able was I ere I saw Chicago."
  HASKELL RESPONSE •     False
```

***Boolean type:* Bool**

Haskell uses the name Bool for the type consisting of the values True and False.

The definition of the function isPalindromic uses function composition in a more general way than previous formulas. In previous formulas, both functions involved in the composition delivered values of the same type as their arguments. In this case, one of the functions (isPalindrome) delivers a value of type Bool, but has an argument of type String. Yet, the composition makes sense because the value delivered to isPalindrome in the composition comes from removePunctuation, and removePunctuation delivers values of type String, which is the proper type for arguments to isPalindrome.

The crucial point is that the right-hand operand of the function composition operation must be a function that delivers a value that has an appropriate type to become an argument for the left-hand operand. With this restriction, function composition can be applied with any pair of functions as its operands.

other correct answers

Either of the following definitions would also be correct.

HASKELL DEFINITION •

Definitions appear on A-page.

Review Questions

1 Suppose that post is a function that, given two letters, chooses the one that follows the other in the alphabet (post 'x' 'y' is 'y'; post 'u' 'p' is 'u'). Then the formula foldr1 post string delivers
- a the letter from string that comes earliest in the alphabet
- b the letter from string that comes latest in the alphabet
- c the first letter from string
- d the last letter from string

2 Suppose that && is an operator that, given two Boolean values, delivers True if both are True and False otherwise. Then the formula foldr1 (&&) [False, True, False, True, True] delivers the value
- a True
- b False
- c Maybe
- d Nothing — the formula doesn't make sense

3 In the formula foldr1 f [a, b, c, d]
- a a, b, c, and d must have the same type
- b f must deliver a value of the same type as its arguments
- c f must be a function that requires two arguments
- d all of the above

4 If f is a function that requires two arguments, then foldr1 f is a function that requires
- a no arguments
- b one argument
- c two arguments
- d three arguments

5 The second argument of foldr1 must be
- a a sequence
- b a function
- c a sequence of functions
- d a function of sequences

6 If next is a function that, given a letter, delivers the next letter of the alphabet, then the mapping process in the formula [next c | c <- "hal"] delivers the string
- a "lah"
- b "ibm"
- c "alm"
- d "lha"

7 The string "is \"hot\" now"
- a has four quotation marks in it
- b has exactly two spaces and two back-slashes
- c has 12 characters, including exactly two spaces
- d has 14 characters, including exactly two spaces

Types 7

Haskell programs can deal with many different types of data. You already know about three types of data: string, Boolean, and character. And you have learned that the Haskell system keeps track of types to make sure formulas use data consistently.

Up to now, the discussion of types has been informal, but even so, you may have found it tedious at times. In this chapter the discussion gets more formal and probably more tedious. This would be necessary at some point, in any case, because types are kind of a fetish in the Haskell world. Fortunately, it is also desirable. The idea of types is one of the most important concepts in computer science. It is a fundamental, organizing influence in software construction. It will pay you to try to apply these ideas, even when you are writing software in a language not as mindful of types as Haskell.

nonsense!

```
HASKELL COMMAND •   reverse 'x'
HASKELL RESPONSE •  ERROR: Type error in application
HASKELL RESPONSE •  ***      : reverse 'x'
HASKELL RESPONSE •  *** term         : 'x'
HASKELL RESPONSE •  *** type         : Char
HASKELL RESPONSE •  *** does not match : [a]
```

The formula reverse 'x' makes no sense because the function reverse requires its argument to be a string. But the erroneous formula supplies an individual character, not a string, as an argument of reverse. The response is some startling gobbledygook that tries to explain why the Haskell system can't make sense of the command. You will need to understand types to make sense of error reports like this.

The Haskell response says "Type error in application." You probably have a vague notion of what a type error is, but what is an application? An **application** is a formula, or part of a formula, that applies a function to an argument. In this case the application is reverse 'x', and it is displayed on the line following the "Type error in application." message. So far, so good.

Next, the report displays the term with the erroneous type ('x') and states its type (Char). Char is the formal name of the data type that the textbook has been referring to informally as "individual character." From now on, it's Char.

names of types

Char	individual character ('x')
String	sequence of Char ("abc") synonym of [Char]
Bool	Boolean (True, False)

all type names in Haskell begin with capital letters

Now comes the really mysterious part:

*** does not match: [a]

Up to now, you have seen only part of the story about the type that the function reverse expects its argument to be — reverse has been applied only to arguments of type String, which are sequences of characters. However, reverse can handle any argument that is a sequence, regardless of the type of the elements of the sequence.

In the message "does not match: [a]", the "[a]" part specifies the type that the term 'x' does not

match. The type "[a]" represents not just a single type, but a collection of types. Namely, it represents all types of sequences. The "a" in the notation stands for any type and is known as a **type variable**. The brackets in the notation indicate that the type is a sequence. Altogether, "[a]" indicates a sequence type in which the elements have type a, where a could be any type.

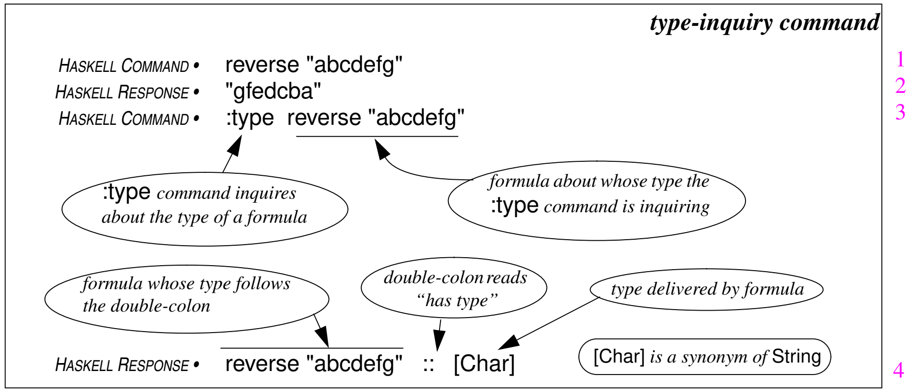
sequence types

[Char]	['a', 'b', 'c'] *same as* "abc"
[Bool]	[True, False, True]
[[Char]]	*same type as* [String]
	[['a', 'b'], ['d', 'e', 'f']] *same as* ["ab", "def"]

a type specifier of the form [a] *(that is, a type enclosed in brackets) indicates a sequence type — the elements of the sequence have type* a

If the type a were Char, then the type [a] would mean [Char]. The type String is a synonym for sequence of characters, which is what [Char] means (Haskell often uses the [Char] form to denote this type in its error reports). Therefore, one of the types that reverse can accept as an argument is String. But it could also accept arguments of type [Bool], or [String] (equivalently [[Char]], indicating a type that is a sequence, each of whose elements is a sequence with elements of type Char), or any other type.

In summary, the error report says that the argument supplied to reverse has type Char, that reverse expects an argument of type [a], and that Char is not among the types indicated by [a]. In other words, Char is not a sequence. You already knew that. Now you know how to interpret the error report from the Haskell system.



You can ask the Haskell system to tell you what type a formula delivers by using the type-inquiry command, which is simply the symbol :type (starts with a colon, like :load — all Haskell commands that aren't formulas start with a colon) followed by the name or formula whose type you'd like to know. Haskell will respond with the type of the name or the type of the value that the formula denotes.

```
HASKELL COMMAND •    reverse [True, False, False]       5
¿ HASKELL RESPONSE ?                                    6
HASKELL COMMAND •    :type [True, False, False]         7
¿ HASKELL RESPONSE ?                                    8
```

HASKELL COMMAND • :type ["aardvark", "boar", "cheeta"]

¿ HASKELL RESPONSE ?

HASKELL COMMAND • reverse ["aardvark", "boar", "cheeta"]

¿ HASKELL RESPONSE ?

HASKELL COMMAND • :type reverse ["aardvark", "boar", "cheeta"]

¿ HASKELL RESPONSE ?

Review Questions

1 The type of [[True, False], [True, True, True], [False]] is
- a mostly True
- b ambiguous
- c [Bool]
- d [[Bool]]

2 The type of ["alpha", "beta", "gamma"] is
- a [[Char]]
- b [String]
- c both of the above
- d none of the above

3 The type of [["alpha", "beta", "gamma"], ["psi", "omega"]] is
- a [[String]]
- b [[Char]]
- c [String]
- d [String, String]

4 The formula foldr1 (.) [f, g, h, k] delivers
- a a string
- b a function
- c a sequence
- d nothing — it's an error because k can't be a function

5 Which of the following is a sequence whose elements are sequences
- a ["from", "the", "right"]
- b [["Beyond", "Rangoon"], ["Belle", "du", "Jour"]]
- c [['a'], ['b'], ['c']]
- d all of the above

6 The type of the formula foldr1 (.) [f, g, h, k] is
- a String -> String
- b the same as the type of f
- c the same as the type of the composition operator
- d [a] -> [b]

7 If the type of f is String -> String, then the type of [f x | x <- xs] is
- a String -> String
- b String
- c [String]
- d none of the above

8 The Haskell entity [["Beyond", "Rangoon"], ["Belle", "du", "Jour"]] has type
- a [[String]]
- b [[[Char]]]
- c both of the above
- d none of the above

Function Types, Classes, and Polymorphism 8

Functions have types, too. Their types are characterized by the types of their arguments and results. In this sense, the type of a function is an ordered sequence of types.

For example, the function reverse takes arguments of type [a] and delivers values of type [a], where [a] denotes the sequence type, a generalized type that includes type String (sequences of elements of type Char — another notation for the type is [Char]), Boolean sequences (sequences with elements of type Bool, the type denoted by [Bool]), sequences of strings (denoted [[Char]] or, equivalently, [String]), and so on. A common way to say this is that reverse transforms values of type [a] to other values of type [a]. Haskell denotes the type of such a function by the formula [a] -> [a].

```
HASKELL COMMAND •    :type reverse
HASKELL RESPONSE •   reverse :: [a] -> [a]
```

> ***polymorphism***
>
> Polymorphism is the ability to assume different forms. Functions like reverse are called **polymorphic** because they can operate on many different types of arguments. This notion plays an important role in modern software development. The use of polymorphism reduces the size of software by reusing definitions in multiple contexts. It also makes software more easily adaptable as requirements evolve.

Functions defined in scripts also have types. For example, the function isPalindromic was defined earlier in the workbook:

```
HASKELL DEFINITION •   isPalindromic =
HASKELL DEFINITION •     isPalindrome . removePunctuation . allLowerCase
HASKELL COMMAND •      :type isPalindromic
HASKELL RESPONSE •     isPalindromic :: [Char] -> Bool
```

This function transforms strings to Boolean values, so its type is [Char]->Bool. This is a more restrictive type than the type of reverse. The reason for the restriction is that isPalindromic applies the function allLowerCase to its argument, and allLowerCase requires its argument to be a String.

```
HASKELL DEFINITION •   allLowerCase str = [toLower c | c <- str]
HASKELL COMMAND •      :type allLowerCase
HASKELL RESPONSE •     allLowerCase :: [Char] -> [Char]
```

An argument supplied to allLowerCase must be a String because it applies the intrinsic function toLower to each element of its argument, and toLower transforms from type Char to type Char.

```
HASKELL COMMAND •      :type toLower
¿ HASKELL RESPONSE ?
```


To continue the computation defined in isPalindromic, the function removePunctuation, like allLowerCase, transforms strings to strings, and finally the function isPalindrome transforms strings delivered by removePunctuation to Boolean values. That would make [Char]->Bool the type of isPalindrome, right?

Not quite! Things get more complicated at this point because isPalindrome, like reverse, can handle more than one type of argument.

```
HASKELL DEFINITION •   isPalindrome phrase  =  (phrase == reverse phrase)
HASKELL COMMAND •      :type isPalindrome
HASKELL RESPONSE •     isPalindrome :: Eq a => [a] -> Bool
```

Whoops! There's the new complication. The [a] -> Bool part is OK. That means that the isPalindrome function transforms from sequences to Boolean values. But where did that other part come from: Eq a => ?

Eq is the name of a class. A **class** is a collection of types that share a collection of functions and/or operations. The class Eq, which is known as the **equality class**, is the set of types on which equality comparison is defined. In other words, if it is possible to use the operation of equality comparison (==) to compare two items of a particular type, then that type is in the class Eq.

The "Eq a =>" portion of the response to the inquiry about the type isPalindrome is a restriction on the type a. It says that the type a must be in the class Eq in order for [a] -> Bool to be a proper type for isPalindrome.

The type for the function reverse, which is [a] -> [a], has no restrictions; reverse can operate on sequences of any kind. But an argument of isPalindrome must be a sequence whose elements can be compared for equality. This makes sense because isPalindrome compares its argument to the reverse of its argument, and two sequences are equal only if their elements are equal. So, to make its computation, isPalindrome will have to compare elements of its argument sequence to other values of the same type. So, the restriction to equality types is necessary.

When isPalindrome was first written, the intention was to apply it only to strings, but the definition turned out to be more general than that. It can be applied to many kinds of sequences. Arguments of type [Bool] (sequences of Boolean values) or [String] (sequences of String values) would be OK for isPalindrome because Boolean values can be compared for equality and so can strings. A sequence of type [Char->Char], however, would not work as an argument for isPalindrome because the equality comparison operation (==) is not able to compare values of type Char->Char, that is functions transforming characters to characters.[1] So, isPalindrome is polymorphic, but not quite as polymorphic as reverse.

Operators are conceptually equivalent to functions and have types as well. The equality operator (==) transforms pairs of comparable items into Boolean values. When a function has more than one argument, the Haskell notation for its type has more than one arrow:

1. There is a mathematical concept of equality between functions, but the equality comparison operator (==) is not able to compare functions defined in Haskell. There is a good reason for this: it is not possible to describe a computation that compares functions for equality. It can be done for certain small classes of functions, but the computation simply cannot be specified in the general case. The notion of incomputabilty is an important concept in the theory of computation, one of the central fields of computer science.

parentheses make this the function-version of the operator ==

```
HASKELL COMMAND •   :type (==)
HASKELL RESPONSE •  Eq a => a -> a -> Bool
```

The type of the equality operator is denoted in Haskell as a->a->Bool, where a must be in the class Eq (types comparable by ==). This indicates that the equality operator, viewed as a function, takes two arguments, which must have the same type, and delivers a Boolean value.

Take another look at the function remove, defined previously:

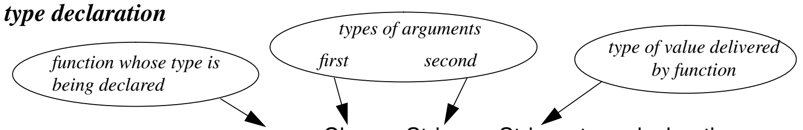
```
HASKELL DEFINITION •                   -- function to remove character chr from string str
HASKELL DEFINITION •   remove chr str = [c | c <- str, c /= chr]
HASKELL COMMAND •      :type remove
HASKELL RESPONSE •     remove :: Eq a => a -> [a] -> [a]
```

The function remove has two arguments, so its type has two arrows. Its first argument can be any type in the class Eq, and its other argument must then be a sequence whose elements have the same type as its first argument. It delivers a value of the same type as its second argument.

The function was originally designed to remove the character specified in its first argument from a string supplied as its second argument and to deliver as its value a copy of the supplied string with all instances of the specified character deleted. That is, the type of the function that the person who defined it had in mind was Char->[Char]->[Char], a special case of a->[a]->[a].

The Haskell system deduces the types of functions defined in scripts. The type it comes up with in this deductive process is the most general type that is consistent with the definition. The designer of a function can force the type of the function to be more specific by including a **type declaration** with the definition of the function in the script.

type declaration




```
HASKELL DEFINITION •   remove :: Char -> String -> String-- type declaration
HASKELL DEFINITION •   remove chr str = [c | c <- str, c /= chr]
```
31

A type declaration may confirm or restrict the type of a function.

The Haskell system will issue an error report if the type declaration is neither the same as the type that Haskell deduces for the function or a special case of that type.

It is good practice to include type declarations because it forces you to formulate a framework of consistency among the types you are using in a program.

The type declaration must be consistent with the type that the Haskell system deduces for the function, but may be a special case of the deduced type. Sometimes, because of ambiguities in the types of the basic elements of a script, the Haskell system will not be able to deduce the type of a function defined in the script. In such cases, and you will see some of these in the next chapter, you must include a type declaration.

It is a good practice to include type declarations with all definitions because it forces you to

understand the types you are using in your program. This understanding will help you keep your concepts straight and make it more likely that you are constructing a correct program. If the Haskell system deduces a type that is incompatible with a declaration, it will report the inconsistency and the type it deduced. This information will help you figure out what is wrong with your formulas.

Review Questions

1 Polymorphic functions
- a change the types of their arguments
- b combine data of different types
- c can operate on many types of arguments
- d gradually change shape as the computation proceeds

2 The function toUpper takes a letter of the alphabet (a value of type Char) and delivers the upper-case version of the letter. What is the type of toUpper?
- a polymorphic
- b Char -> Char
- c lower -> upper
- d cannot be determined from the information given

3 A value of type [a] is
- a a sequence with elements of several different types
- b a sequence with some of its elements omitted
- c a sequence whose elements are also sequences
- d a sequence whose elements are all of type a

4 A function of type [a] -> [[a]] could
- a transform a character into a string
- b deliver a substring of a given string
- c deliver a string like its argument, but with the characters in a different order
- d transform a string into a sequence of substrings

5 Suppose that for any type a in the class Ord, pairs of values of type a can be compared using the operator <. A function of type Ord a => [a] -> [a] could
- a rearrange the elements of a sequence into increasing order
- b deliver a subsequence of a given sequence
- c both of the above
- d none of the above

6 Suppose Ord is the class described in the preceding question. What is the type of the operator <.
- a Ord a => a -> a -> Bool
- b Ord a => a -> Bool
- c Ord a => a -> Char
- d Ord a => a -> [Char]

7 The equality class
- a includes all Haskell types
- b is what makes functions possible
- c is what makes comparison possible
- d excludes function types

8 A function with the type `Eq a => a -> Bool`
- a requires an argument with the name `a`
- b delivers `True` on arguments of type `a`
- c is polymorphic
- d must be equal to `a`

9 If the type of `f` has three arrows in it, then the type of `f x` has
- a one arrow in it
- b two arrows in it
- c three arrows in it
- d four arrows in it

10 A polymorphic function
- a has more than one argument
- b has only one argument
- c may deliver values of different types in different formulas
- d can morph many things at once

Types of Curried Forms and Higher Order Functions 9

Curried forms of function invocations supply, as you know, less than the full complement of arguments for a function. The formula used to define the function removePunctuation, for example, used a list of curried invocations of remove.

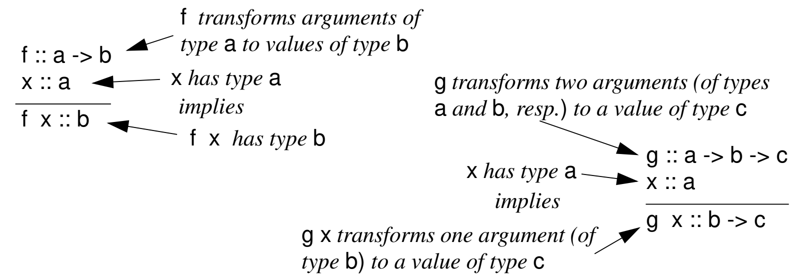
HASKELL DEFINITION • `removePunctuation = foldr1 (.) [remove c | c <- ",. ;:'\"?!()"]` 32
HASKELL COMMAND • `:type remove` 35.30c2
HASKELL RESPONSE • `remove :: Eq a => a -> [a] -> [a]` 36.29c2
HASKELL COMMAND • `:type remove '?'` 33
¿ HASKELL RESPONSE ? 34

The curried invocation remove '?' supplies one argument to the two-argument function remove. The first argument of remove can be of any type (any type in the equality class, that is), as you can see in its type specification. The argument supplied in this invocation has type Char, which is in the equality class. So far so good.

The type of remove indicates that its second argument must be a sequence type, and the elements of the sequence must have the same type as its first argument. This implies that when the first argument has type Char, the second argument must have type [Char]. The value that result delivers has the same type as its second argument, as the type of remove shows.

Therefore, the type of the curried invocation, remove '?', must be [Char]->[Char]. That is, the function remove '?' has type [Char]->[Char].

finding the type of a curried invocation



Now that you know the type of a curried invocation like remove '?', what do you think would be the type of the sequence of such invocations that are part of the formula for the function remove-Punctuation?

HASKELL COMMAND • `:type [remove c | c <- " ,.;:'\"?!()" ]` 37
¿ HASKELL RESPONSE ? 38

This is a sequence whose elements are functions. Remember! The elements of a sequence can have any type, as long as all the elements of the sequence have the same type.

Another part of the formula for the function removePunctuation is the composition operator expressed in the form of a function: (.). The type of this one gets a little complicated because both of its arguments are functions and it delivers a function as its value. The functions supplied as arguments can transform any type to any other type, but they must be compatible for composition: the right-hand operand must be a function that delivers a value of a type that the left-hand operand can use as an argument.

> ***higher-order functions***
>
> *Functions that have an argument that is, itself, a function are called higher-order functions. Functions that deliver a value that is, itself, a function are also called higher-order functions. The composition operator is a higher-order function that does both.*

When the operands are compatible in this way, then the result of the composition will be a function that transforms the domain of the right-hand operand into the range of the left-hand operand. These hints may help you work out the type of the composition operator.

HASKELL COMMAND • :type (.)

¿ HASKELL RESPONSE ?

In the definition of removePunctuation, composition is being used to compose curried invocations of remove. These curried invocation have the type [Char] -> [Char]. When two such functions are composed, the result is another function of the same type (because the domain and range of the operands are the same). So, the composition taking place in a formula such as

```
remove ',' . remove '.'
```

has type

```
([Char]->[Char]) -> ([Char]->[Char]) -> ([Char]->[Char])
```

This is a special case of the polymorphic type of the composition operator. Actually, the last pair of parentheses in this type specification are redundant because the arrow notation is right-associative (see box). Haskell would omit the redundant parentheses and denote this type as

```
([Char]->[Char]) -> ([Char]->[Char]) -> [Char]->[Char]
```

> ***arrow (->) is right-associative***
>
> To make the arrow notation (->) for function types compatible with curried forms of function invocations, the arrow associates to the right. That is, a -> b -> c is interpreted as a -> (b -> c) automatically, even if the parentheses are missing. In reporting types, the Haskell system omits redundant parentheses.

The function foldr1 is another higher-order function. Its first argument is a function of two arguments, both of the same type, that delivers values that also have that type. The second argument of foldr1 is a sequence in which adjacent elements are pairs of potential arguments of the function that is the first argument of foldr1. The function foldr1 treats the function (its first argument) as if it were an operator and inserts that operator between each pair of adjacent elements of the sequence (its second argument), combining all of the elements of the sequence into one value whose type must be the type of elements the sequence.

> **foldr1**
>
> foldr1 op [w, x, y, z]
>
> means
>
> w ‘op‘ x ‘op‘ y ‘op‘ z
>
> where ‘op‘ denotes the operator equivalent to the two-argument function op. More precisely, it means w ‘op‘ (x ‘op‘ (y ‘op‘ z)). The "r" in foldr1 means that the operation is to be carried out by associating the operands in the sequence from the right.

HASKELL COMMAND • :type foldr1 41

¿ HASKELL RESPONSE ? 42

The definition of removePunctuation supplies, as a second argument to foldr1, a sequence of functions that transform [Char] to [Char]. That is, the data type of the elements of the sequence is [Char]->[Char]. This means that the first argument must be a function that takes two arguments of type [Char]->[Char] and delivers a value of that same type. It also means that the value that this application of foldr1 will deliver will have type [Char]->[Char].

That is, in this particular case, foldr1 is being used in a context in which its type is

```
([Char]->[Char]->[Char]) -> [ [Char] ] -> [Char]
```

This is just one of the specific types that the polymorphic function foldr1 can take on.

Review Questions

1 Suppose functions f and g have types Char -> String and String -> [String], respectively. Then their composition g . f has type
   a Char -> String
   b Char -> String -> [String]
   c Char -> [String]
   d [[String]]

2 Suppose the type of a function f is f :: String -> String -> Bool. Then the type of f "x" is
   a Bool
   b String
   c String -> Bool
   d Nothing — f "x" is not a proper formula

3 Suppose the type of a function f is f :: Char -> String -> [String] -> [[String]]. Then f ’x’ and f ’x’ "y" have, respectively, types
   a [String] -> [[String]] and [[String]]
   b Char -> String -> [String] and Char -> String
   c String -> [String] -> [[String]] and [String] -> [[String]]
   d Nothing — f ’x’ is not a proper formula

4 Because the arrow notation is right associative, the type a -> b -> c has the same meaning as
a (a -> b) -> c
b a -> (b -> c)
c (a -> b -> c)
d a -> b (-> c)

5 The composition operator has type (.) :: (a -> b) -> (c -> a) -> (c -> b). Another way to express this type is
a (.) :: a -> b -> (c -> a) -> (c -> b)
b (.) :: (a -> b) -> (c -> a) -> c -> b
c (.) :: a -> b -> c -> a -> (c -> b)
d (.) :: a -> b -> c -> a -> c -> b

6 The composition operator has type (.) :: (a -> b) -> (c -> a) -> (c -> b). Another way to express this type is
a (.) :: (b -> c) -> (a -> b) -> (a -> c)
b (.) :: (a -> c) -> (b -> a) -> (b -> c)
c (.) :: (c -> a) -> (b -> c) -> (b -> a)
d all of the above

7 A function whose type is (a -> b) -> c must be
a lower order
b middle order
c higher order
d impossible to define

8 If a function f has type f :: a -> a, then the formulas f 'x' and f True have, respectively, types
a Char and Bool
b [Char] and [Bool]
c Char -> Char and Bool -> Bool
d cannot be determined from given information

Private Definitions — the where-clause 10

Functions, once defined in a script, can be used in formulas that occur anywhere in the script. Sometimes one wants to define a function or variable that will only be used in formulas that occur in a single definition and not in other definitions in the script. This avoids cluttering the name space with functions needed only in the context of a single definition.

The concept of private variables versus public variables provides a way to encapsulate portions of a program, hiding internal details from other parts of the program. **Encapsulation** is one of the most important ideas in software engineering. Without it, the development of large software systems is virtually impossible.

variables
Names used in Haskell programs are sometimes called variables, like names used in mathematical equations. It's a bit of a misnomer, since their values, once defined, don't vary. The same is true in mathematical equations: there is only one value for x in the equation $x + 5 = 10$.

encapsulation
isolating components of software so that modifying internal details in one component will have no affect on other components of the software

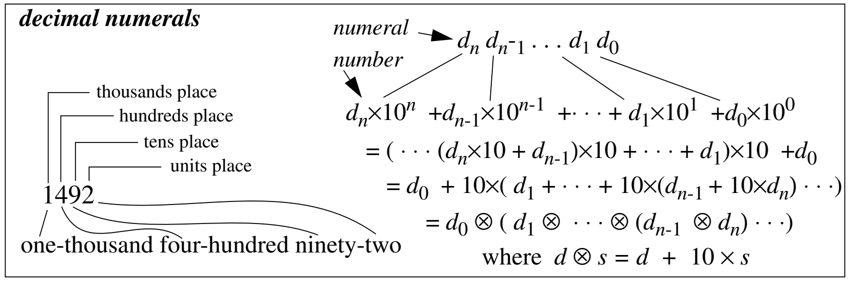
The examples of the next few chapters have to do with different ways to represent information, from numbers to encrypted text. **Information representation** is another central theme of computer science. For this reason, the examples themselves are as important to your education as the programming methods that you will be learning along the way.

The first example discusses some methods for representing numbers. These methods apply only to non-negative numbers without fractional parts — integers from zero up, in other words — but the ideas carry over to the representation of other kinds of numbers and even to other kinds of information. In fact, a subsequent example will use these number representation ideas to do encryption and decryption of text.

Numbers are denoted by numerals. Numerals and numbers are not the same things: one is a symbol for the other. For example, 87 is a numeral often used to denote the number four score and seven, but LXXXVII (Roman), 57 (hexadecimal), / \+t (Chinese), and 1010111 (binary) are other numerals also in common use to denote the same quantity that the decimal numeral 87 represents.

The Haskell language uses decimal numerals to denote numbers, but the Haskell system uses it own internal mechanisms, which it does not reveal to the outside world, to represent in its calculations the numbers that these numerals denote.

A decimal numeral uses base ten, positional notation to represent a number. The number that a decimal numeral represents, is the sum of a collection of multiples of powers of ten. Each position in the numeral represents a different power of ten: the rightmost digit position is the units place (ten to the power zero); the next position is the tens place (ten to the power one); next the hundreds place (ten to the power two); and so on. The digits of the numeral in each position specify which multiple of the power of ten represented by that position to include in the collection of numbers to add up.



If the digits in a decimal numeral are $d_n\, d_{n-1} \ldots d_1\, d_0$, where d_i is the digit in the 10^i position, then the number the numeral denotes is the following sum:

$$d_n{\times}10^n \;\; +d_{n-1}{\times}10^{n-1} \;\; + \cdots + d_1{\times}10^1 \;\; +d_0{\times}10^0$$

A direct way to compute this sum would be to compute the powers of ten involved, multiply these quantities by the appropriate coefficients (the d_i values), then add up the results. This approach requires a lot of arithmetic: n-1 multiplications for the power 10^n, n-2 multiplications for 10^{n-1}, and so on, then n+1 more multiplications to handle the coefficients, and finally n additions.

A more efficient way to compute the sum is to look at it in a new form by factoring out the tens. The new formula, a special case of a method known as Horner's rule for evaluating polynomials, doesn't require any direct exponentiation — just multiplication and addition, and only n of each. This leads to the Horner formula:

$$d_0 \;\; + 10{\times}(\, d_1 + \cdots + 10{\times}(d_{n-1} + 10{\times}d_n) \cdots) \qquad \textit{Horner Formula}$$

There are n stages in the Horner formula, and each stage requires a multiplication by ten of the value delivered by the previous stage, and then the addition of a coefficient value. This pair of operations can be viewed as a package. The following equation defines a symbol (⊗) for that package of two operations:

$$d \otimes s = d \;\; + 10 \times s$$

With this new notation, the following formula is exactly equivalent to the Horner formula.

$$d_0 \otimes (\, d_1 \otimes \;\; \cdots \otimes (d_{n-1} \;\; \otimes d_n) \cdots) \qquad \textit{Horner Formula using } \otimes$$

In this form, the Horner formula is just the d_i coefficients combined by inserting the ⊗ operation between each adjacent pair of them and grouping the sequence of operations with parentheses from the right. This is exactly what the foldr1 function of Haskell does: it inserts an operator between each adjacent pair of elements in a sequence and then groups from the right with parentheses (that's what the "r" stands for in foldr1 — "from the right"). So, foldr1 should be useful in expressing the Horner formula in Haskell notation.

Using the ⊗-version of the Horner formula as a guide, try to use foldr1 to define a Haskell function to compute the value expressed by the Horner formula. The argument of the function will be

a sequence of numbers $[d_0, d_1, \ldots, d_{n-1}, d_n]$ of a new type, Integer, that can act as operands in addition (+) and multiplication (∗) operations. The formula in the function will use foldr1 and will also use a function called multAdd, defined below, which is a Haskell version of the circle-cross operator (⊗), defined above.

Integral *types* — Integer *and* Int

Haskell uses ordinary decimal numerals to denote integral numbers. They may be positive or negative; negative ones are preceded by a minus sign. There are two kinds of integral numbers in Haskell: Integer and Int. Integer numbers behave like mathematical integers in arithmetic operations: addition (+), subtraction (-), multiplication (∗), quotient-or-next-smaller-integer (‘div‘), clock-remainder (n∗(m ‘div‘ n) + m ‘mod‘ n == m), and exponentiation (^) deliver Integer values from Integer operands. Numbers of type Int behave in the same way, except that they have a limited range (about 10 decimal digits).

0	*nada*	14110	*altitude of Pike's Peak*
23	*Jordan's number*	-280	*altitude of Death Valley*
23 ‘mod‘ 12	*"film at" number*	-3	*Sarazan's number*
55 ‘div‘ 5	*"film at" again*	7 ‘mod‘ (-5)	*Sarazan's number again*
59 ‘div‘ 5	*and again*	5 ‘div‘ (-2)	*Sarazan yet again*

- The operands of ordinary division (/) are not Integral numbers in Haskell.
- Because the minus sign is used to denote both subtraction and the sign of Integer numbers, negative integers sometimes need to be enclosed in parentheses:
  mod 7 (-5) *not* mod 7 -5*, which would mean* (mod 7) - 5*, which is nonsense*
- Context determines whether a particular numeral in a Haskell script denotes an Integer or an Int.

```
¿ HASKELL DEFINITION ?   multAdd d s = d + 10*s
¿ HASKELL DEFINITION ?   horner10 ds =                    -- you define horner10
¿ HASKELL DEFINITION ?                                                          1
HASKELL COMMAND •        horner10 [1, 2, 3, 4]                                  2
HASKELL RESPONSE •       4321                                                   3
```

The multAdd function is tailored specifically for use in the definition of horner10. It is not likely to be needed outside the context of that definition. For this reason, it would be better to make the definition of multAdd private, to be used only by horner10.

Haskell provides a notation, known as the **where-clause**, for defining names that remain unknown outside a particular context. Names defined in a where-clause are for the private use of the definition containing the where-clause. They will be unknown elsewhere.

A where-clause appears as an indented subsection of a higher-level definition. The indentation is Haskell's way of marking subsections — it is a bracketing method known as the **offsides rule**. A where-clause may contain any number of private definitions. The end of a where-clause occurs when the level of indentation moves back to the left of the level established by the keyword **where** that begins the where clause.

where-clause— for defining private terms

A where-clause makes it possible to define terms for private use entirely within the confines of another definition.

- The keyword where, indented below the definition requiring private terms, begins the where-clause, and the clause ends when the indentation level returns to the previous point.
- The where-clause can use any terms it defines at any point in the clause.
- A where-clause within a function definition can refer to the formal parameters of the function.

```
HASKELL DEFINITION •   sumOfLastTwoDigits x = d1 + d0
HASKELL DEFINITION •     where
HASKELL DEFINITION •     d0 = x ‘mod‘ 10
HASKELL DEFINITION •     d1 = shift ‘mod‘ 10
HASKELL DEFINITION •     shift = x ‘div‘ 10
```

uses parameter x

uses private variable shift *defined here*

offsides rule — a bracketing mechanism

```
HASKELL DEFINITION •   inches yds ft ins =
HASKELL DEFINITION •     insFromFt (ft + ftFromYds yds) + ins
HASKELL DEFINITION •   feet mis yds = fm + fy
HASKELL DEFINITION •     where
HASKELL DEFINITION •     fm = ftFromMiles mis
HASKELL DEFINITION •     fy = ftFromYds yds
HASKELL DEFINITION •   insFromFt ft = 12∗ft
```

indentation marks extent of current definition

return to previous indentation level
- *ends previous definition*
- *starts new definition*

The following new definition of horner10 uses a where-clause to encapsulate the definition of the multAdd function.

```
¿ HASKELL DEFINITION ?   horner10 ds =                    -- you define it again
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?     where
¿ HASKELL DEFINITION ?     multAdd  d  s = d + 10*s
```

One of the goals of this chapter was to put together a function to transform lists of digits representing decimal numerals into the numbers those numerals denote. The function horner10 essentially does that, except for a kind of quirk: the numeral is written with its digits reversed (with the

units place on the left (first in the sequence), then the tens place, and so on. The following function accepts a list of digits in the usual order (units place last), and delivers the number those digits represent when the sequence is interpreted as a decimal numeral.

```
¿ HASKELL DEFINITION ?   integerFromDecimalNumeral ds =          --you define it
¿ HASKELL DEFINITION ?
   HASKELL COMMAND •     integerFromDecimalNumeral [1,4,9,2]
   HASKELL RESPONSE •    1492
```

The decimal numeral representing a particular integer is not unique. It is always possible to put any number of leading zeros on the front of the decimal numeral without affecting the value the numeral represents: [1, 4, 9, 2], [0, 1, 4, 9, 2], and [0, 0, 1, 4, 9, 2] all represent the integer 1492. Similarly, you can always add or remove any number of leading zeros in a numeral without changing the integer the numeral represents.

How about zero itself? The numerals [0], [0, 0], and [0, 0, 0] all represent zero. You can include as many zero digits as you like in the numeral. Or you can remove any number: [0, 0, 0], [0, 0], [0], [ ]. Whoops! Ran out of digits. How about that empty sequence?

For reasons having to do with how the function will be used in subsequent chapters, it is important for the function integerFromDecimalNumeral to be able to deal with the case when the sequence of digits representing the decimal numeral is empty. As the above analysis shows, it is reasonable to interpret an empty sequence of digits as one of the alternative decimal numerals for zero.

As written, integerFromDecimalNumeral will fail if its argument is the empty sequence:

```
   HASKELL COMMAND •     integerFromDecimalNumeral [ ]
   HASKELL RESPONSE •    Program error: {foldr1 (v706 {dict}) []}
```

comparison operations on* Integral *types

Integral values can be compared for equality and for order:

• equal to	$x == y$
• not equal to	$x /= y$
• less than	$x < y$
• greater than	$x > y$
• less than or equal to	$x <= y$
• greater than or equal to	$x >= y$

Relationship of the class Integral to other classes

- in the equality class (Eq)
- in a class called Ord, the class for which the operations less-than, greater-than, less-than-or-equal-to, and greater-than-or-equal-to (plus two operations derived from these: max and min), are applicable
- in a hierarchy of numeric classes that relate different kinds of numbers, classes that you will learn about later

The error message is pretty much undecipherable, but it does indicate a problem with foldr1. The problem is that foldr1 expects its the sequence to be non-empty (that's what the "1" stands for in foldr1 — "at least one element"). It doesn't know what to do if the sequence has no elements.

However, Haskell provides another intrinsic function foldr, that acts like foldr1, but can also handle the empty sequence. The first argument of foldr is a function of two arguments. Like foldr1, foldr views this function as an operator that it places between adjacent pairs of elements of a sequence, which is supplied as its last argument.

But, foldr has three arguments (unlike foldr1, which as only two arguments). The second argument of foldr is a value to serve as the right-hand operand in the rightmost application of the operation. In case the sequence (third argument) is empty, it is this value (second argument) that foldr delivers as its result.

foldr

foldr op z [x_1, x_2, ..., x_n] =
    x_1 ‘op‘ (x_2 ‘op‘ (... ‘op‘ (x_n ‘op‘ z) …))

foldr op z [w, x, y] = w ‘op‘ (x ‘op‘ (y ‘op‘ z))

foldr op z [ ] = z

HASKELL IDENTITY • foldr1 op xs = foldr op (last xs) (init xs)
  where
    last xs *is the last element in the sequence* xs
    init xs *is the sequence* xs *without its last element*

In fact, foldr delivers the same value that foldr1 would have delivered when supplied with the same operator as its first argument and, for its other argument, a combination of the second and third arguments of foldr (namely, a sequence just like the third argument of foldr, but with the second argument of foldr inserted at the end). This makes it possible for foldr to deliver a value even when the sequence is empty.

To work out what an invocation of foldr means, augment the sequence supplied as its third argument by inserting its second argument at the end the sequence, then put the operator supplied as its first argument between each adjacent pair of elements in the augmented sequence. For example, foldr could be used to define a function to find the sum of a sequence of numbers:

```
HASKELL DEFINITION •  total xs= foldr (+) 0 xs
HASKELL COMMAND •     total [12, 3, 5, 1, 4]
HASKELL RESPONSE •    25
HASKELL COMMAND •     total [100, -50, 20]
¿ HASKELL RESPONSE ?
```

total [12, 3, 5, 1, 4] =
12 + (3 +(5 + (1 + (4 + 0))))

The function horner10 can be defined using foldr instead of foldr1 by supplying zero as the second argument. This makes integerFromDecimalNumeral work properly when the numeral is empty. The computation is subtly different: the rightmost multAdd operation in the foldr construction will be d+10∗0, where d is the last high-order digit of the numeral (that is, the coefficient of the highest power of ten in the sum that the numeral represents). Since 10∗0 is zero, this extra multAdd step doesn't change the result.

No change needs to be made in the definition of integerFromDecimalNumeral. Its definition depends on horner10. Once horner10 is corrected, integerFromDecimalNumeral computes the desired results.


```
¿ HASKELL DEFINITION ?  horner10 ds =          -- you define (0 on empty argument)
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?
HASKELL COMMAND •       horner10 [1, 2, 3, 4]
HASKELL RESPONSE •      4321
HASKELL COMMAND •       horner10 [ ]
HASKELL RESPONSE •      0
HASKELL COMMAND •       integerFromDecimalNumeral [ ]
HASKELL RESPONSE •      0
```

8

Review Questions

1 Integral numbers are denoted in Haskell by
  a decimal numerals enclosed in quotation marks
  b decimal numerals — no quotation marks involved
  c decimal numerals with or without quotation marks
  d values of type String

2 The Haskell system, when interpreting scripts, represents numbers as
  a decimal numerals
  b binary numerals
  c hexadecimal numerals
  d however it likes — none of your business anyway

3 Encapsulation
  a prevents name clashes
  b prevents one software component from messing with the internal details of another
  c is one of the most important concepts in software engineering
  d all of the above

4 A where-clause in Haskell defines variables that
  a will be accessible from all where-clauses
  b take the name of the where-clause as a prefix
  c cannot be accessed outside the definition containing the clause
  d have strange names

5 In Haskell, the beginning and end of a definition is determined by
  a begin and end statements
  b matched sets of curly braces { }
  c indentation
  d however it likes — none of your business anyway

6 The intrinsic function foldr
  a is more generally applicable than foldr1
  b takes more arguments than foldr1
  c can accommodate an empty sequence as its last argument
  d all of the above

7 What value does the following command deliver?

```
HASKELL DEFINITION •  ct  xs= foldr addOne 0  xs
HASKELL DEFINITION •      where
```

HASKELL DEFINITION • addOne x sum = 1 + sum
HASKELL COMMAND • ct [1, 2, 3, 4]

a 10, by computing 1+(2+(3+(4+0)))
b 4, by computing 1+(1+(1+(1+0)))
c 5, by computing 1+(1+(1+(1+1)))
d nothing — ct is not properly defined

Tuples 11

Consider the reverse of the problem of computing the number that a sequence of digits represents. Suppose, instead, you would like to compute the sequence of digits in the decimal numeral that denotes a given number. In one case, you start with a sequence of digits and compute a number, and in the other case you start with a number and compute a sequence of digits.

making functions into operators

function syntax	op arg1 arg2
operator syntax	arg1 ‘op‘ arg2

backquote—looks like backwards slanting apostrophe on keyboard

- these are equivalent notations when op has two arguments
- op may have a defined fixity (left-, right-, or non-associative) and precedence that affects grouping

The units digit of the decimal numeral for a non-negative number is simply the remainder when the number is divided by ten. The clock-remainder function, mentioned in the previous chapter, can be used to extract this digit:

HASKELL DEFINITION • unitsDigit x = x `mod` 10 1
HASKELL COMMAND • unitsDigit 1215 2
¿ HASKELL RESPONSE ? 3

The trick to getting the tens digit of a number is to first drop the units digit, then extract the units digit of what's left:

HASKELL DEFINITION • tensDigit x = d1
HASKELL DEFINITION • where
HASKELL DEFINITION • xSansLastDigit = x `div` 10
HASKELL DEFINITION • d1 = xSansLastDigit `mod` 10 4
HASKELL COMMAND • tensDigit 1789 5
¿ HASKELL RESPONSE ? 6

It often happens that the ‘div‘ and ‘mod‘ operators need to be used together, as in the calculation of the tens digit of a numeral. For this reason, Haskell includes an operator called ‘divMod‘ that delivers both the ‘div‘ part and the ‘mod‘ part in the division of two Integers. The ‘divMod‘ operator returns this pair of numbers in a Haskell structure known as a **tuple**.

patterns must match exactly

If a tuple of variables appears on the left side in a definition, the value on the right must be a tuple with the same number of components. A definition is an equation. If one side of the equation has one form (say a two-tuple) and the other side has a different form (say a three-tuple), it can't really be an equation, can it?

A tuple in Haskell is an aggregate of two or more individual components. The components may be of any type, and different items of a tuple may have different types. Tuples are denoted in Haskell scripts by a list of the components, separated by commas, with the entire list enclosed in parentheses.

Equations that define variables as tuples can use a single name for the whole tuple, or they can use **tuple patterns** to give a name to each component. When a tuple pattern is defined, the first variable gets the value of the first component of the tuple in the formula on the right hand side, and the second variable gets the value of the second component.

The ‘divMod‘ operator computes the quotient and remainder of two integral operands. Its left operand acts as the dividend and the right operand acts as the divisor. It delivers the quotient and remainder in the form of a tuple with two components:

x ‘divMod‘ d = (x ‘div‘ d, x ‘mod‘ d)

tuples

("Rodney Bottoms", 2. True) :: (String, Integer, Bool)
(6,1) :: (Integer, Integer) — *result of* 19 ‘divMod‘ 3

- *must have at least two components*
- *components may be of different types*
- *components separated by commas*
- *parentheses delimit tuple*
- *type of tuple looks like a tuple, but with types as components*

The quotient x ‘div‘ d is the next integer smaller than the exact ratio of the x to d, or the exact ratio itself if x divided by d is an integer. (This is what you'd expect if both arguments are positive. If one is negative, then it is one less than you might expect.) The remainder x ‘mod‘ d is chosen to make the following relationship True.

d∗(x ‘div‘ d) + (x ‘mod‘ d) == x

One can extract the hundreds digit of a numeral through an additional iteration of the idea used in extracting the units and tens digits. It amounts to successive applications of the ‘divMod‘ operator. All of this could be done with the ‘div‘ and ‘mod‘ operators separately, of course, but since the operations are used together, the ‘divMod‘ operator is more convenient.

```
HASKELL DEFINITION •   hundredsDigit  x  = d2
HASKELL DEFINITION •     where
HASKELL DEFINITION •     (xSansLastDigit, d0)  =  x `divMod` 10
HASKELL DEFINITION •     (xSansLast2Digits, d1) =  xSansLastDigit `divMod` 10
HASKELL DEFINITION •     (xSansLast3Digits, d2) = xSansLast2Digits `divMod` 10
HASKELL COMMAND •      hundredsDigit 1517
¿ HASKELL RESPONSE ?
```

The definition

(xSansLastDigit, d0) = x ‘divMod‘ 10

defines both the variable xSansLastDigit (defining its value to be the first component of the tuple delivered by x ‘divMod‘ 10) and the variable d0 (defining its value to be the second component of the tuple delivered by x ‘divMod‘ 10). The other definitions in the above where-clause also use tuple patterns to define the two variables that are components in the tuple patterns.

Review Questions

1 The type of the tuple ("X Windows System", 11, "GUI") is
a (String, Integer, Char)
b (String, Integer, String)
c (X Windows System, Eleven, GUI)
d (Integer, String, Bool)

2 After the following definition, the variables x and y are, respectively,
HASKELL DEFINITION • (x, y) = (24, "XXIV")
a both of type Integer
b both of type String
c an Integer and a String
d undefined — can't define two variables at once

3 After the following definition, the variable x is
HASKELL DEFINITION • x = (True, True, "2")
a twice True
b a tuple with two components and a spare, if needed
c a tuple with three components
d undefined — can't define a variable to have more than one value

4 After the following definition, the variable x is
HASKELL DEFINITION • x = 16 ‘divMod‘ 12
a 1 + 4
b 16 ÷ 4
c 1 × 12 + 4
d (1, 4)

5 The formula divMod x 12 == x ‘divMod‘ 12 is
a (x ‘div‘ 12, x ‘mod‘ 12)
b (True, True)
c True if x is not zero
d True, no matter what Integer x is

6 In a definition of a tuple
a both components must be integers
b the tuple being defined and its definition must have the same number of components
c surplus components on either side of the equation are ignored
d all of the above